Local Lexing

Steven Obua

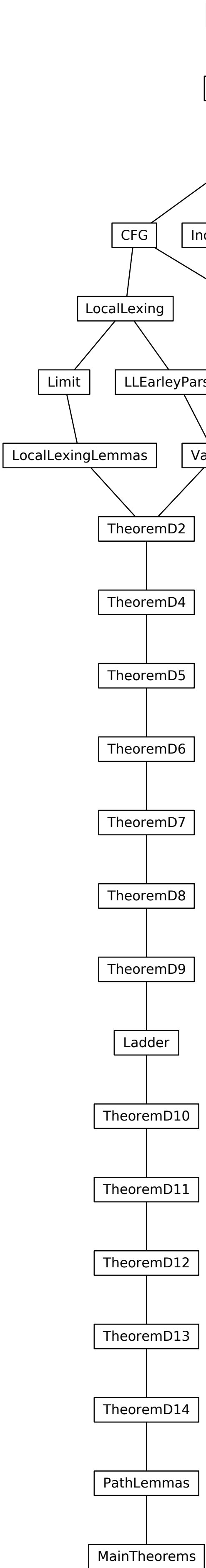
March 17, 2025

Abstract

This formalisation accompanies the paper Local Lexing[1], which introduces a novel parsing concept of the same name. The paper also gives a high-level algorithm for local lexing as an extension of Earley's algorithm. This formalisation proves the algorithm to be correct with respect to its local lexing semantics. As a special case, this formalisation thus also contains a proof of the correctness of Earley's algorithm. The paper contains a short outline of how this formalisation is organised.

Contents

[1] https://arxiv.org/abs/1702.03277



[Pure]
[Tools]
[HOL]
CFG
InductRules
ListTools
LocalLexing
Derivations
Limit
LLEarleyParsing
LocalLexingLemmas
Validity
TheoremD2
TheoremD4
TheoremD5
TheoremD6
TheoremD7
TheoremD8
TheoremD9
Ladder
TheoremD10
TheoremD11
TheoremD12
TheoremD13
TheoremD14
PathLemmas
MainTheorems

theory *CFG*
imports *Main*
begin

typedecl *symbol*

type-synonym *rule* = *symbol* × *symbol list*

type-synonym *sentence* = *symbol list*

locale *CFG* =
  fixes $\mathfrak{N}$ :: *symbol set*
  fixes $\mathfrak{T}$ :: *symbol set*
  fixes $\mathfrak{R}$ :: *rule set*
  fixes $\mathfrak{S}$ :: *symbol*
  assumes *disjunct-symbols*: $\mathfrak{N} \cap \mathfrak{T} = \{\}$
  assumes *startsymbol-dom*: $\mathfrak{S} \in \mathfrak{N}$
  assumes *validRules*: $\forall\ (N, \alpha) \in \mathfrak{R}.\ N \in \mathfrak{N} \wedge (\forall\ s \in set\ \alpha.\ s \in \mathfrak{N} \cup \mathfrak{T})$
begin

definition *is-terminal* :: *symbol* ⇒ *bool*
where
  is-terminal s = $(s \in \mathfrak{T})$

definition *is-nonterminal* :: *symbol* ⇒ *bool*
where
  is-nonterminal s = $(s \in \mathfrak{N})$

lemma *is-nonterminal-startsymbol*:*is-nonterminal* $\mathfrak{S}$
  by (*simp add*: *is-nonterminal-def startsymbol-dom*)

definition *is-symbol* :: *symbol* ⇒ *bool*
where
  is-symbol s = (*is-terminal s* ∨ *is-nonterminal s*)

definition *is-sentence* :: *sentence* ⇒ *bool*
where
  is-sentence s = *list-all is-symbol s*

definition *is-word* :: *sentence* ⇒ *bool*
where
  is-word s = *list-all is-terminal s*

definition *derives1* :: *sentence* ⇒ *sentence* ⇒ *bool*
where
  derives1 u v =
    $(\exists\ x\ y\ N\ \alpha.$
      $u = x\ @\ [N]\ @\ y$
    $\wedge\ v = x\ @\ \alpha\ @\ y$

∧ *is-sentence* x
∧ *is-sentence* y
∧ $(N, \alpha) \in \mathfrak{R})$

definition *derivations1* :: (*sentence* × *sentence*) *set*
where
derivations1 = { (u,v) | u v. *derives1* u v }

definition *derivations* :: (*sentence* × *sentence*) *set*
where
derivations = *derivations1*^*

definition *derives* :: *sentence* ⇒ *sentence* ⇒ *bool*
where
derives u v = ((u, v) ∈ *derivations*)

definition *is-derivation* :: *sentence* ⇒ *bool*
where
is-derivation u = *derives* [$\mathfrak{S}$] u

definition $\mathcal{L}$ :: *sentence set*
where
$\mathcal{L}$ = { v | v. *is-word* v ∧ *is-derivation* v}

definition $\mathcal{L}_P$ :: *sentence set*
where
$\mathcal{L}_P$ = { u | u v. *is-word* u ∧ *is-derivation* (u@v) }

end

end
theory *LocalLexing*
imports *CFG*
begin

typedecl *character*

type-synonym *lexer* = *character list* ⇒ *nat* ⇒ *nat set*

type-synonym *token* = *symbol* × *character list*

type-synonym *tokens* = *token list*

definition *terminal-of-token* :: *token* ⇒ *symbol*
where
terminal-of-token t = *fst* t

definition *terminals* :: *tokens* ⇒ *sentence*
where

terminals ts = *map terminal-of-token ts*

definition *chars-of-token* :: *token* ⇒ *character list*
where
chars-of-token t = *snd t*

fun *chars* :: *tokens* ⇒ *character list*
where
chars [] = []
| *chars* (*t#ts*) = (*chars-of-token t*) @ (*chars ts*)

fun *charslength* :: *tokens* ⇒ *nat*
where
charslength cs = *length* (*chars cs*)

definition *is-lexer* :: *lexer* ⇒ *bool*
where
is-lexer lexer =
(∀ *D p l*. (*p* ≤ *length D* ∧ *l* ∈ *lexer D p* ⟶ *p* + *l* ≤ *length D*) ∧
(*p* > *length D* ⟶ *lexer D p* = {}))

type-synonym *selector* = *token set* ⇒ *token set* ⇒ *token set*

definition *is-selector* :: *selector* ⇒ *bool*
where
is-selector sel = (∀ *A B*. *A* ⊆ *B* ⟶ (*A* ⊆ *sel A B* ∧ *sel A B* ⊆ *B*))

fun *by-length* :: *nat* ⇒ *tokens set* ⇒ *tokens set*
where
by-length l tss = { *ts* . *ts* ∈ *tss* ∧ *length* (*chars ts*) = *l* }

fun *funpower* :: (′*a* ⇒ ′*a*) ⇒ *nat* ⇒ (′*a* ⇒ ′*a*)
where
funpower f 0 x = *x*
| *funpower f* (*Suc n*) *x* = *f* (*funpower f n x*)

definition *natUnion* :: (*nat* ⇒ ′*a set*) ⇒ ′*a set*
where
natUnion f = ⋃ { *f n* | *n*. *True* }

definition *limit* :: (′*a set* ⇒ ′*a set*) ⇒ ′*a set* ⇒ ′*a set*
where
limit f x = *natUnion* (λ *n*. *funpower f n x*)

locale *LocalLexing* = *CFG* +
fixes *Lex* :: *symbol* ⇒ *lexer*
fixes *Sel* :: *selector*
assumes *Lex-is-lexer*: ∀ *t* ∈ $\mathfrak{T}$. *is-lexer* (*Lex t*)
assumes *Sel-is-selector*: *is-selector Sel*

fixes *Doc* :: *character list*
begin

definition *admissible* :: *tokens* $\Rightarrow$ *bool*
where
admissible ts = (*terminals ts* $\in \mathcal{L}_P$)

definition *Append* :: *token set* $\Rightarrow$ *nat* $\Rightarrow$ *tokens set* $\Rightarrow$ *tokens set*
where
Append Z k P = *P* $\cup$
{ *p* @ [*t*] | *p t*. *p* $\in$ *by-length k P* $\wedge$ *t* $\in$ *Z* $\wedge$ *admissible* (*p* @ [*t*])}

definition $\mathcal{X}$ :: *nat* $\Rightarrow$ *token set*
where
$\mathcal{X}$ *k* = {(*t*, ω) | *t l* ω. *t* $\in \mathfrak{T}$ $\wedge$ *l* $\in$ *Lex t Doc k* $\wedge$ ω = *take l* (*drop k Doc*)}

definition $\mathcal{W}$ :: *tokens set* $\Rightarrow$ *nat* $\Rightarrow$ *token set*
where
$\mathcal{W}$ *P k* = { *u*. *u* $\in \mathcal{X}$ *k* $\wedge$ ($\exists$ *p* $\in$ *by-length k P*. *admissible* (*p*@[*u*])) }

definition $\mathcal{Y}$ :: *token set* $\Rightarrow$ *tokens set* $\Rightarrow$ *nat* $\Rightarrow$ *token set*
where
$\mathcal{Y}$ *T P k* = *Sel T* ($\mathcal{W}$ *P k*)

fun $\mathcal{P}$ :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *tokens set*
and $\mathcal{Q}$ :: *nat* $\Rightarrow$ *tokens set*
and $\mathcal{Z}$ :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *token set*
where
$\mathcal{P}$ *0 0* = {[]}
| $\mathcal{P}$ *k* (*Suc u*) = *limit* (*Append* ($\mathcal{Z}$ *k* (*Suc u*)) *k*) ($\mathcal{P}$ *k u*)
| $\mathcal{P}$ (*Suc k*) *0* = $\mathcal{Q}$ *k*
| $\mathcal{Z}$ *k 0* = {}
| $\mathcal{Z}$ *k* (*Suc u*) = $\mathcal{Y}$ ($\mathcal{Z}$ *k u*) ($\mathcal{P}$ *k u*) *k*
| $\mathcal{Q}$ *k* = *natUnion* ($\mathcal{P}$ *k*)

definition $\mathfrak{P}$ :: *tokens set*
where
$\mathfrak{P}$ = $\mathcal{Q}$ (*length Doc*)

definition *ll* :: *tokens set*
where
ll = { *p* . *p* $\in \mathfrak{P}$ $\wedge$ *charslength p* = *length Doc* $\wedge$ *terminals p* $\in \mathcal{L}$ }

end

end
theory *LLEarleyParsing*
imports *LocalLexing*
begin

datatype *item* =
  Item
    (*item-rule*: *rule*)
    (*item-dot* : *nat*)
    (*item-origin* : *nat*)
    (*item-end* : *nat*)

type-synonym *items* = *item set*

definition *item-nonterminal* :: *item* ⇒ *symbol*
where
  item-nonterminal x = *fst* (*item-rule x*)

definition *item-rhs* :: *item* ⇒ *sentence*
where
  item-rhs x = *snd* (*item-rule x*)

definition *item-α* :: *item* ⇒ *sentence*
where
  item-α x = *take* (*item-dot x*) (*item-rhs x*)

definition *item-β* :: *item* ⇒ *sentence*
where
  item-β x = *drop* (*item-dot x*) (*item-rhs x*)

definition *init-item* :: *rule* ⇒ *nat* ⇒ *item*
where
  init-item r k = *Item r 0 k k*

definition *is-complete* :: *item* ⇒ *bool*
where
  is-complete x = (*item-dot x* ≥ *length* (*item-rhs x*))

definition *next-symbol* :: *item* ⇒ *symbol option*
where
  next-symbol x = (*if is-complete x then None else Some* ((*item-rhs x*) ! (*item-dot x*)))

definition *inc-item* :: *item* ⇒ *nat* ⇒ *item*
where
  inc-item x k = *Item* (*item-rule x*) (*item-dot x* + *1*) (*item-origin x*) *k*

definition *bin* :: *items* ⇒ *nat* ⇒ *items*
where
  bin I k = { *x* . *x* ∈ *I* ∧ *item-end x* = *k* }

context *LocalLexing* **begin**

definition *Init* :: *items*
where
$Init = \{\ init\text{-}item\ r\ 0\ |\ r.\ r \in \mathfrak{R} \wedge fst\ r = \mathfrak{S}\ \}$

definition *Predict* :: *nat* ⇒ *items* ⇒ *items*
where
$Predict\ k\ I = I\ \cup$
$\{\ init\text{-}item\ r\ k\ |\ r\ x.\ r \in \mathfrak{R} \wedge x \in bin\ I\ k\ \wedge$
$next\text{-}symbol\ x = Some(fst\ r)\ \}$

definition *Complete* :: *nat* ⇒ *items* ⇒ *items*
where
$Complete\ k\ I = I \cup \{\ inc\text{-}item\ x\ k\ |\ x\ y.$
$x \in bin\ I\ (item\text{-}origin\ y) \wedge y \in bin\ I\ k \wedge is\text{-}complete\ y\ \wedge$
$next\text{-}symbol\ x = Some\ (item\text{-}nonterminal\ y)\ \}$

definition *TokensAt* :: *nat* ⇒ *items* ⇒ *token set*
where
$TokensAt\ k\ I = \{\ (t,\ s)\ |\ t\ s\ x\ l.\ x \in bin\ I\ k\ \wedge$
$next\text{-}symbol\ x = Some\ t \wedge is\text{-}terminal\ t\ \wedge$
$l \in Lex\ t\ Doc\ k \wedge s = take\ l\ (drop\ k\ Doc)\ \}$

definition *Tokens* :: *nat* ⇒ *token set* ⇒ *items* ⇒ *token set*
where
$Tokens\ k\ T\ I = Sel\ T\ (TokensAt\ k\ I)$

definition *Scan* :: *token set* ⇒ *nat* ⇒ *items* ⇒ *items*
where
$Scan\ T\ k\ I = I\ \cup$
$\{\ inc\text{-}item\ x\ (k + length\ c)\ |\ x\ t\ c.\ x \in bin\ I\ k \wedge (t,\ c) \in T\ \wedge$
$next\text{-}symbol\ x = Some\ t\ \}$

definition π :: *nat* ⇒ *token set* ⇒ *items* ⇒ *items*
where
$\pi\ k\ T\ I =$
$limit\ (\lambda\ I.\ Scan\ T\ k\ (Complete\ k\ (Predict\ k\ I)))\ I$

fun $\mathcal{J}$ :: *nat* ⇒ *nat* ⇒ *items*
and $\mathcal{I}$ :: *nat* ⇒ *items*
and $\mathcal{T}$ :: *nat* ⇒ *nat* ⇒ *token set*
where
$\mathcal{J}\ 0\ 0 = \pi\ 0\ \{\}\ Init$
$|\ \mathcal{J}\ k\ (Suc\ u) = \pi\ k\ (\mathcal{T}\ k\ (Suc\ u))\ (\mathcal{J}\ k\ u)$
$|\ \mathcal{J}\ (Suc\ k)\ 0 = \pi\ (Suc\ k)\ \{\}\ (\mathcal{I}\ k)$
$|\ \mathcal{T}\ k\ 0 = \{\}$
$|\ \mathcal{T}\ k\ (Suc\ u) = Tokens\ k\ (\mathcal{T}\ k\ u)\ (\mathcal{J}\ k\ u)$
$|\ \mathcal{I}\ k = natUnion\ (\mathcal{J}\ k)$

definition $\mathfrak{I}$ :: *items*

where
$\mathfrak{I} = \mathcal{I}$ (*length Doc*)

definition *is-finished* :: *item* ⇒ *bool* **where**
is-finished x = (*item-nonterminal x* = $\mathfrak{S}$ ∧ *item-origin x* = *0* ∧ *item-end x* = *length Doc* ∧
is-complete x)

definition *earley-recognised* :: *bool*
where
earley-recognised = (∃ *x* ∈ $\mathfrak{I}$. *is-finished x*)

end

end
theory *Limit*
imports *LocalLexing*
begin

definition *setmonotone* :: (*'a set* ⇒ *'a set*) ⇒ *bool*
where
setmonotone f = (∀ *X*. *X* ⊆ *f X*)

lemma *setmonotone-funpower*: *setmonotone f* ⟹ *setmonotone* (*funpower f n*)
by (*induct n*, *auto simp add*: *setmonotone-def*)

lemma *subset-setmonotone*: *setmonotone f* ⟹ *X* ⊆ *f X*
by (*simp add*: *setmonotone-def*)

lemma *elem-setmonotone*: *setmonotone f* ⟹ *x* ∈ *X* ⟹ *x* ∈ *f X*
by (*auto simp add*: *setmonotone-def*)

lemma *elem-natUnion*: (∀ *n*. *x* ∈ *f n*) ⟹ *x* ∈ *natUnion f*
by (*auto simp add*: *natUnion-def*)

lemma *subset-natUnion*: (∀ *n*. *X* ⊆ *f n*) ⟹ *X* ⊆ *natUnion f*
by (*auto simp add*: *natUnion-def*)

lemma *setmonotone-limit*:
assumes *fmono*: *setmonotone f*
shows *setmonotone* (*limit f*)
proof −
show *setmonotone* (*limit f*)
apply (*auto simp add*: *setmonotone-def limit-def*)
apply (*rule elem-natUnion*, *auto*)
apply (*rule elem-setmonotone*[*OF setmonotone-funpower*])
by (*auto simp add*: *fmono*)
qed

lemma[*simp*]: *funpower id n* = *id*
  by (*rule ext*, *induct n*, *simp-all*)

lemma[*simp*]: *limit id* = *id*
  by (*rule ext*, *auto simp add*: *limit-def natUnion-def*)

lemma *natUnion-decompose*[*consumes 1*, *case-names Decompose*]:
  assumes *p*: $p \in natUnion\ S$
  assumes *decompose*: $\bigwedge n\ p.\ p \in S\ n \Longrightarrow P\ p$
  shows *P p*
proof −
  from *p* **have** $\exists\ n.\ p \in S\ n$
    by (*auto simp add*: *natUnion-def*)
  then obtain *n* **where** $p \in S\ n$ **by** *blast*
  from *decompose*[*OF this*] **show** *?thesis* **.**
qed

lemma *limit-induct*[*consumes 1*, *case-names Init Iterate*]:
  assumes *p*: $(p :: 'a) \in limit\ f\ X$
  assumes *init*: $\bigwedge p.\ p \in X \Longrightarrow P\ p$
  assumes *iterate*: $\bigwedge p\ Y.\ (\bigwedge q\ .\ q \in Y \Longrightarrow P\ q) \Longrightarrow p \in f\ Y \Longrightarrow P\ p$
  shows *P p*
proof −
  from *p* **have** *p-in-natUnion*: $p \in natUnion\ (\lambda\ n.\ funpower\ f\ n\ X)$
    by (*simp add*: *limit-def*)
  {
    fix *p* :: *'a*
    fix *n* :: *nat*
    have $p \in funpower\ f\ n\ X \Longrightarrow P\ p$
    proof (*induct n arbitrary*: *p*)
      case *0* **thus** *?case* **using** *init*[*OF 0*[*simplified*]] **by** *simp*
    next
      case (*Suc n*) **show** *?case*
        using *iterate*[*OF Suc*(*1*) *Suc*(*2*)[*simplified*], *simplified*] **by** *simp*
    qed
  }
  with *p-in-natUnion* **show** *?thesis*
    by (*induct rule*: *natUnion-decompose*)
qed

definition *chain* :: $(nat \Rightarrow 'a\ set) \Rightarrow bool$
where
  $chain\ C = (\forall\ i.\ C\ i \subseteq C\ (i + 1))$

definition *continuous* :: $('a\ set \Rightarrow 'b\ set) \Rightarrow bool$
where
  $continuous\ f = (\forall\ C.\ chain\ C \longrightarrow (chain\ (f\ o\ C) \wedge f\ (natUnion\ C) = natUnion\ (f\ o\ C)))$

```
lemma continuous-apply:
  continuous f ⟹ chain C ⟹ f (natUnion C) = natUnion (f o C)
by (simp add: continuous-def)

lemma continuous-imp-mono:
  assumes continuous: continuous f
  shows mono f
proof −
  {
    fix A :: 'a set
    fix B :: 'a set
    assume sub: A ⊆ B
    let ?C = λ (i::nat). if (i = 0) then A else B
    have chain ?C by (simp add: chain-def sub)
    then have fC: chain (f o ?C) using continuous continuous-def by blast
    then have f (?C 0) ⊆ f (?C (0 + 1))
    proof −
      have ⋀f n. ¬ chain f ∨ (f n::'b set) ⊆ f (Suc n)
        by (metis Suc-eq-plus1 chain-def)
      then show ?thesis using fC by fastforce
    qed
    then have f A ⊆ f B by auto
  }
  then show mono f by (simp add: monoI)
qed

lemma mono-maps-chain-to-chain:
  assumes f: mono f
  assumes C: chain C
  shows chain (f o C)
by (metis C comp-def f chain-def mono-def)

lemma natUnion-upperbound:
  (⋀ n. f n ⊆ G) ⟹ (natUnion f) ⊆ G
by (auto simp add: natUnion-def)

lemma funpower-upperbound:
  (⋀ I. I ⊆ G ⟹ f I ⊆ G) ⟹ I ⊆ G ⟹ funpower f n I ⊆ G
proof (induct n)
  case 0 thus ?case by simp
next
  case (Suc n) thus ?case by simp
qed

lemma limit-upperbound:
  (⋀ I. I ⊆ G ⟹ f I ⊆ G) ⟹ I ⊆ G ⟹ limit f I ⊆ G
by (simp add: funpower-upperbound limit-def natUnion-upperbound)

lemma elem-limit-simp: x ∈ limit f X = (∃ n. x ∈ funpower f n X)
```

```
by (auto simp add: limit-def natUnion-def)

definition pointwise :: ('a set ⇒ 'b set) ⇒ bool where
  pointwise f = (∀ X. f X = ⋃ { f {x} | x. x ∈ X})

lemma pointwise-simp:
  assumes f: pointwise f
  shows f X = ⋃ { f {x} | x. x ∈ X}
proof −
  from f have ∀ X. f X = ⋃ { f {x} | x. x ∈ X}
    by (rule iffD1[OF pointwise-def[where f=f]])
  then show ?thesis by blast
qed

lemma natUnion-elem: x ∈ f n ⟹ x ∈ natUnion f
using natUnion-def by fastforce

lemma limit-elem: x ∈ funpower f n X ⟹ x ∈ limit f X
by (simp add: limit-def natUnion-elem)

lemma limit-step-pointwise:
  assumes x: x ∈ limit f X
  assumes f: pointwise f
  assumes y: y ∈ f {x}
  shows y ∈ limit f X
proof −
  from x have ∃ n. x ∈ funpower f n X
    by (simp add: elem-limit-simp)
  then obtain n where n: x ∈ funpower f n X by blast
  have y ∈ funpower f (Suc n) X
    apply simp
    apply (subst pointwise-simp[OF f])
    using y n by auto
  then show y ∈ limit f X by (meson limit-elem)
qed

definition pointbase :: ('a set ⇒ 'b set) ⇒ 'a set ⇒ 'b set where
  pointbase F I = ⋃ { F X | X. finite X ∧ X ⊆ I }

definition pointbased :: ('a set ⇒ 'b set) ⇒ bool where
  pointbased f = (∃ F. f = pointbase F)

lemma pointwise-implies-pointbased:
  assumes pointwise: pointwise f
  shows pointbased f
proof −
  let ?F = λ X. f X
  {
    fix I :: 'a set
```

```
    fix x :: 'b
    have lr: x ∈ pointbase ?F I ⟹ x ∈ f I
    proof −
      assume x: x ∈ pointbase ?F I
      have ∃ X. x ∈ f X ∧ X ⊆ I
        proof −
          have x ∈ ⋃{f A |A. finite A ∧ A ⊆ I}
            by (metis pointbase-def x)
          then show ?thesis
            by blast
        qed
      then obtain X where X:x ∈ f X ∧ X ⊆ I by blast
      have ∃ y. y ∈ I ∧ x ∈ f {y}
        using X apply (simp add: pointwise-simp[OF pointwise, where X=X])
        by blast
      then show x ∈ f I
        apply (simp add: pointwise-simp[OF pointwise, where X=I])
        by blast
    qed
    have rl: x ∈ f I ⟹ x ∈ pointbase ?F I
    proof −
      assume x: x ∈ f I
      have ∃ y. y ∈ I ∧ x ∈ f {y}
        using x apply (simp add: pointwise-simp[OF pointwise, where X=I])
        by blast
      then obtain y where y ∈ I ∧ x ∈ f {y} by blast
      then have ∃ X. x ∈ f X ∧ finite X ∧ X ⊆ I by blast
      then show x ∈ pointbase f I
        apply (simp add: pointbase-def)
        by blast
    qed
    note lr rl
  }
  then have ⋀ I. pointbase f I = f I by blast
  then have pointbase f = f by blast
  then show ?thesis by (metis pointbased-def)
qed

lemma pointbase-is-mono:
  mono (pointbase f)
proof −
  {
    fix A :: 'a set
    fix B :: 'a set
    assume subset: A ⊆ B
    have (pointbase f) A ⊆ (pointbase f) B
      apply (simp add: pointbase-def)
      using subset by fastforce
  }
```

```
  then show ?thesis by (simp add: mono-def)
qed

lemma chain-implies-mono: chain C ⟹ mono C
by (simp add: chain-def mono-iff-le-Suc)

lemma chain-cover-witness: finite X ⟹ chain C ⟹ X ⊆ natUnion C ⟹ ∃ n.
X ⊆ C n
proof (induct rule: finite.induct)
  case emptyI thus ?case by blast
next
  case (insertI X x)
  then have X ⊆ natUnion C by simp
  with insertI have ∃ n. X ⊆ C n by blast
  then obtain n where n: X ⊆ C n by blast
  have x: x ∈ natUnion C using insertI.prems(2) by blast
  then have ∃ m. x ∈ C m
  proof –
    have x ∈ ⋃{A. ∃ n. A = C n} by (metis x natUnion-def)
    then show ?thesis by blast
  qed
  then obtain m where m: x ∈ C m by blast
  have mono-C: ⋀ i j. i ≤ j ⟹ C i ⊆ C j
    using chain-implies-mono insertI(3) mono-def by blast
  show ?case
    apply (rule-tac x=max n m in exI)
    apply auto
    apply (meson contra-subsetD m max.cobounded2 mono-C)
    by (metis max-def mono-C n subsetCE)
qed

lemma pointbase-is-continuous:
  continuous (pointbase f)
proof –
  {
    fix C :: nat ⇒ 'a set
    assume C: chain C
    have mono: chain ((pointbase f) o C)
      by (simp add: C mono-maps-chain-to-chain pointbase-is-mono)
    have subset1: natUnion ((pointbase f) o C) ⊆ (pointbase f) (natUnion C)
    proof (auto)
      fix y :: 'b
      assume y ∈ natUnion ((pointbase f) o C)
      then show y ∈ (pointbase f) (natUnion C)
      proof (induct rule: natUnion-decompose)
        case (Decompose n p)
         thus ?case by (metis comp-apply contra-subsetD mono-def natUnion-elem
            pointbase-is-mono subsetI)
      qed
```

```
    qed
    have subset2: (pointbase f) (natUnion C) ⊆ natUnion ((pointbase f) o C)
    proof (auto)
      fix y :: 'b
      assume y: y ∈ (pointbase f) (natUnion C)
      have ∃ X. finite X ∧ X ⊆ natUnion C ∧ y ∈ f X
      proof −
        have y ∈ ⋃{f A |A. finite A ∧ A ⊆ natUnion C}
          by (metis y pointbase-def)
        then show ?thesis by blast
      qed
      then obtain X where X: finite X ∧ X ⊆ natUnion C ∧ y ∈ f X by blast
      then have ∃ n. X ⊆ C n using chain-cover-witness C by blast
      then obtain n where X-sub-C: X ⊆ C n by blast
      show y ∈ natUnion ((pointbase f) o C)
        apply (rule-tac natUnion-elem[where n=n])
        proof −
          have y ∈ ⋃{f A |A. finite A ∧ A ⊆ C n}
          using X X-sub-C by blast
          then show y ∈ (pointbase f ∘ C) n by (simp add: pointbase-def)
      qed
    qed
    note mono subset1 subset2
  }
  then show ?thesis by (simp add: continuous-def subset-antisym)
qed

lemma pointbased-implies-continuous:
  pointbased f ⟹ continuous f
  using pointbase-is-continuous pointbased-def by force

lemma setmonotone-implies-chain-funpower:
  assumes setmonotone: setmonotone f
  shows chain (λ n. funpower f n I)
by (simp add: chain-def setmonotone subset-setmonotone)

lemma natUnion-subset: (⋀ n. ∃ m. f n ⊆ g m) ⟹ natUnion f ⊆ natUnion g
  by (meson natUnion-elem natUnion-upperbound subset-iff)

lemma natUnion-eq[case-names Subset Superset]:
  (⋀ n. ∃ m. f n ⊆ g m) ⟹ (⋀ n. ∃ m. g n ⊆ f m) ⟹ natUnion f = natUnion
g
by (simp add: natUnion-subset subset-antisym)

lemma natUnion-shift[symmetric]:
  assumes chain: chain C
  shows natUnion C = natUnion (λ n. C (n + m))
proof (induct rule: natUnion-eq)
  case (Subset n)
```

```
    show ?case using chain chain-implies-mono le-add1 mono-def by blast
next
  case (Superset n)
    show ?case by blast
qed

definition regular :: ('a set ⇒ 'a set) ⇒ bool
where
  regular f = (setmonotone f ∧ continuous f)

lemma regular-fixpoint:
  assumes regular: regular f
  shows f (limit f I) = limit f I
proof −
  have setmonotone: setmonotone f using regular regular-def by blast
  have continuous: continuous f using regular regular-def by blast

  let ?C = λ n. funpower f n I
  have chain: chain ?C
    by (simp add: setmonotone setmonotone-implies-chain-funpower)
  have f (limit f I) = f (natUnion ?C)
    using limit-def by metis
  also have f (natUnion ?C) = natUnion (f o ?C)
    by (metis continuous continuous-def chain)
  also have natUnion (f o ?C) = natUnion (λ n. f(funpower f n I))
    by (meson comp-apply)
  also have natUnion (λ n. f(funpower f n I)) = natUnion (λ n. ?C (n + 1))
    by simp
  also have natUnion (λ n. ?C(n + 1)) = natUnion ?C
    apply (subst natUnion-shift)
    using chain by (blast+)
  finally show ?thesis by (simp add: limit-def)
qed

lemma fix-is-fix-of-limit:
  assumes fixpoint: f I = I
  shows limit f I = I
proof −
  have funpower: ⋀ n. funpower f n I = I
  proof −
    fix n :: nat
    from fixpoint show funpower f n I = I
      by (induct n, auto)
  qed
  show ?thesis by (simp add: limit-def funpower natUnion-def)
qed

lemma limit-is-idempotent: regular f ⟹ limit f (limit f I) = limit f I
by (simp add: fix-is-fix-of-limit regular-fixpoint)
```

definition *mk-regular1* :: ('b ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'a ⇒ 'a) ⇒ 'a set ⇒ 'a set
where
  mk-regular1 P F I = *I* ∪ { *F q x* | *q x*. *x* ∈ *I* ∧ *P q x* }

definition *mk-regular2* :: ('b ⇒ 'a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'a ⇒ 'a ⇒ 'a) ⇒ 'a set
⇒ 'a set **where**
  mk-regular2 P F I = *I* ∪ { *F q x y* | *q x y*. *x* ∈ *I* ∧ *y* ∈ *I* ∧ *P q x y* }

lemma *setmonotone-mk-regular1*: *setmonotone* (*mk-regular1 P F*)
by (*simp add*: *mk-regular1-def setmonotone-def*)

lemma *setmonotone-mk-regular2*: *setmonotone* (*mk-regular2 P F*)
by (*simp add*: *mk-regular2-def setmonotone-def*)

lemma *pointbased-mk-regular1*: *pointbased* (*mk-regular1 P F*)
proof −
  let *?f* = λ *X*. *X* ∪ { *F q x* | *q x*. *x* ∈ *X* ∧ *P q x* }
  {
    fix *I* :: 'a set
    have *1*: *pointbase ?f I* ⊆ *mk-regular1 P F I*
      by (*auto simp add*: *pointbase-def mk-regular1-def*)
    have *2*: *mk-regular1 P F I* ⊆ *pointbase ?f I*
      apply (*simp add*: *pointbase-def mk-regular1-def*)
      apply *blast*
      done
    from *1 2* **have** *pointbase ?f I* = *mk-regular1 P F I* **by** *blast*
  }
  then show *?thesis*
    apply (*subst pointbased-def*)
    apply (*rule-tac x=?f* **in** *exI*)
    by *blast*
qed

lemma *pointbased-mk-regular2*: *pointbased* (*mk-regular2 P F*)
proof −
  let *?f* = λ *X*. *X* ∪ { *F q x y* | *q x y*. *x* ∈ *X* ∧ *y* ∈ *X* ∧ *P q x y* }
  {
    fix *I* :: 'a set
    have *1*: *pointbase ?f I* ⊆ *mk-regular2 P F I*
      by (*auto simp add*: *pointbase-def mk-regular2-def*)
    have *2*: *mk-regular2 P F I* ⊆ *pointbase ?f I*
      apply (*auto simp add*: *pointbase-def mk-regular2-def*)
      apply *blast*
      proof −
        fix *q x y*
        assume *x*: *x* ∈ *I*
        assume *y*: *y* ∈ *I*
        assume *P*: *P q x y*

```
      let ?X = {x, y}
      let ?A = ?X ∪ {F q x y |q x y. x ∈ ?X ∧ y ∈ ?X ∧ P q x y}
      show ∃ A. (∃ X. A = X ∪ {F q x y |q x y. x ∈ X ∧ y ∈ X ∧ P q x y} ∧
        finite X ∧ X ⊆ I) ∧ F q x y ∈ A
        apply (rule-tac x=?A in exI)
        apply (rule conjI)
        apply (rule-tac x=?X in exI)
        apply (auto simp add: x y)[1]
        using x y P by blast
    qed
    from 1 2 have pointbase ?f I = mk-regular2 P F I by blast
  }
  then show ?thesis
    apply (subst pointbased-def)
    apply (rule-tac x=?f in exI)
    by blast
qed

lemma regular1:regular (mk-regular1 P F)
by (simp add: pointbased-implies-continuous pointbased-mk-regular1 regular-def
  setmonotone-mk-regular1)

lemma regular2: regular (mk-regular2 P F)
by (simp add: pointbased-implies-continuous pointbased-mk-regular2 regular-def
  setmonotone-mk-regular2)

lemma continuous-comp:
  assumes f: continuous f
  assumes g: continuous g
  shows continuous (g o f)
by (metis (no-types, lifting) comp-assoc comp-def continuous-def f g)

lemma setmonotone-comp:
  assumes f: setmonotone f
  assumes g: setmonotone g
  shows setmonotone (g o f)
by (metis (mono-tags, lifting) comp-def f g rev-subsetD setmonotone-def subsetI)

lemma regular-comp:
  assumes f: regular f
  assumes g: regular g
  shows regular (g o f)
using continuous-comp f g regular-def setmonotone-comp by blast

lemma setmonotone-id[simp]: setmonotone id
  by (simp add: id-def setmonotone-def)

lemma continuous-id[simp]: continuous id
  by (simp add: continuous-def)
```

```
lemma regular-id[simp]: regular id
  by (simp add: regular-def)

lemma regular-funpower: regular f ⟹ regular (funpower f n)
proof (induct n)
  case 0 thus ?case by (simp add: id-def[symmetric])
next
  case (Suc n)
  have funpower: funpower f (Suc n) = f o (funpower f n)
    apply (rule ext)
    by simp
  with Suc show ?case
    by (auto simp only: funpower regular-comp)
qed

lemma mono-id[simp]: mono id
  by (simp add: mono-def id-def)

lemma mono-funpower:
  assumes mono: mono f
  shows mono (funpower f n)
proof (induct n)
  case 0 thus ?case by (simp add: id-def[symmetric])
next
  case (Suc n)
  show ?case by (simp add: Suc.hyps mono monoD monoI)
qed

lemma mono-limit:
  assumes mono: mono f
  shows mono (limit f)
proof(auto simp add: mono-def limit-def)
  fix A :: 'a set
  fix B :: 'a set
  fix x
  assume subset: A ⊆ B
  assume x ∈ natUnion (λn. funpower f n A)
  then have ∃ n. x ∈ funpower f n A using elem-limit-simp limit-def by fastforce

  then obtain n where n: x ∈ funpower f n A by blast
  then have mono: mono (funpower f n) by (simp add: mono mono-funpower)
  then have x ∈ funpower f n B by (meson contra-subsetD monoD n subset)
  then show x ∈ natUnion (λn. funpower f n B) by (simp add: natUnion-elem)
qed

lemma continuous-funpower:
  assumes continuous: continuous f
  shows continuous (funpower f n)
```

```
proof (induct n)
  case 0 thus ?case by (simp add: id-def[symmetric])
next
  case (Suc n)
  have mono: mono (funpower f (Suc n))
    by (simp add: continuous continuous-imp-mono mono-funpower)
  have chain: ∀ C. chain C ⟶ chain ((funpower f (Suc n)) o C)
    by (simp del: funpower.simps add: mono mono-maps-chain-to-chain)
  have limit: ⋀ C. chain C ⟹ (funpower f (Suc n)) (natUnion C) =
      natUnion ((funpower f (Suc n)) o C)
      apply simp
      apply (subst continuous-apply[OF Suc])
      apply simp
      apply (subst continuous-apply[OF continuous])
      apply (simp add: Suc.hyps continuous-imp-mono mono-maps-chain-to-chain)
      apply (rule arg-cong[where f=natUnion])
      apply (rule ext)
      by simp
  from chain limit show ?case using continuous-def by blast
qed

lemma natUnion-swap:
  natUnion (λ i. natUnion (λ j. f i j)) = natUnion (λ j. natUnion (λ i. f i j))
by (metis (no-types, lifting) natUnion-elem natUnion-upperbound subsetI subset-antisym)

lemma continuous-limit:
  assumes continuous: continuous f
  shows continuous (limit f)
proof –
  have mono: mono (limit f)
    by (simp add: continuous continuous-imp-mono mono-limit)
  have chain: ⋀ C. chain C ⟹ chain ((limit f) o C)
    by (simp add: mono mono-maps-chain-to-chain)
  have ⋀ C. chain C ⟹ (limit f) (natUnion C) = natUnion ((limit f) o C)
  proof –
    fix C :: nat ⇒ 'a set
    assume chain-C: chain C
    have contpower: ⋀ n. continuous (funpower f n)
      by (simp add: continuous continuous-funpower)
    have comp: ⋀ n F. F o C = (λ i. F (C i))
      by auto
    have (limit f) (natUnion C) = natUnion (λ n. funpower f n (natUnion C))
      by (simp add: limit-def)
    also have natUnion (λ n. funpower f n (natUnion C)) =
        natUnion (λ n. natUnion ((funpower f n) o C))
      apply (subst continuous-apply[OF contpower])
      apply (simp add: chain-C)+
      done
    also have natUnion (λ n. natUnion ((funpower f n) o C)) =
```

natUnion (*λ n. natUnion* (*λ i. funpower f n* (*C i*)))
apply (*subst comp*)
apply *blast*
done
also have *natUnion* (*λ n. natUnion* (*λ i. funpower f n* (*C i*))) =
natUnion (*λ i. natUnion* (*λ n. funpower f n* (*C i*)))
apply (*subst natUnion-swap*)
apply *blast*
done
also have *natUnion* (*λ i. natUnion* (*λ n. funpower f n* (*C i*))) =
(*natUnion* (*λ i. limit f* (*C i*)))
apply (*simp add*: *limit-def*)
done
also have *natUnion* (*λ i. limit f* (*C i*)) = *natUnion* ((*limit f*) *o C*)
apply (*subst comp*)
apply *simp*
done
finally show (*limit f*) (*natUnion C*) = *natUnion* ((*limit f*) *o C*) **by** *blast*
qed
with *chain* **show** *?thesis* **by** (*simp add*: *continuous-def*)
qed

lemma *regular-limit*: *regular f* ⟹ *regular* (*limit f*)
by (*simp add*: *continuous-limit regular-def setmonotone-limit*)

lemma *regular-implies-mono*: *regular f* ⟹ *mono f*
by (*simp add*: *continuous-imp-mono regular-def*)

lemma *regular-implies-setmonotone*: *regular f* ⟹ *setmonotone f*
by (*simp add*: *regular-def*)

lemma *regular-implies-continuous*: *regular f* ⟹ *continuous f*
by (*simp add*: *regular-def*)

end
theory *LocalLexingLemmas*
imports *LocalLexing Limit*
begin

context *LocalLexing* **begin**

lemma[*simp*]: *setmonotone* (*Append Z k*) **by** (*auto simp add*: *Append-def setmonotone-def*)

lemma *subset-𝒫Suc*: 𝒫 *k u* ⊆ 𝒫 *k* (*Suc u*)
by (*simp add*: *subset-setmonotone*[*OF setmonotone-limit*])

lemma *subset-fSuc-strict*:
assumes *f*: ⋀ *u. f u* ⊆ *f* (*Suc u*)

```
  shows u < v ⟹ f u ⊆ f v
proof (induct v)
  show u < 0 ⟹ f u ⊆ f 0
    by auto
next
  fix v
  assume a:(u < v ⟹ f u ⊆ f v)
  assume b:u < Suc v
  from b have c: f u ⊆ f v
    apply (case-tac u < v)
    apply (simp add: a)
    apply (case-tac u = v)
    apply simp
    by auto
  show f u ⊆ f (Suc v)
    apply (rule subset-trans[OF c])
    by (rule f)
qed

lemma subset-fSuc:
  assumes f: ⋀ u. f u ⊆ f (Suc u)
  shows u ≤ v ⟹ f u ⊆ f v
  apply (case-tac u < v)
  apply (rule subset-fSuc-strict[where f=f, OF f])
  by auto

lemma subset-𝒫k: u ≤ v ⟹ 𝒫 k u ⊆ 𝒫 k v
  by (rule subset-fSuc, rule subset-𝒫Suc)

lemma subset-𝒫𝒬k: 𝒫 k u ⊆ 𝒬 k by (auto simp add: natUnion-def)

lemma subset-𝒬𝒫Suc: 𝒬 k ⊆ 𝒫 (Suc k) u
proof −
  have a: 𝒬 k ⊆ 𝒫 (Suc k) 0 by simp
  show ?thesis
    apply (case-tac u = 0)
    apply (simp add: a)
    apply (rule subset-trans[OF a subset-𝒫k])
    by auto
qed

lemma subset-𝒬Suc: 𝒬 k ⊆ 𝒬 (Suc k)
  by (rule subset-trans[OF subset-𝒬𝒫Suc subset-𝒫𝒬k])

lemma subset-𝒬: i ≤ j ⟹ 𝒬 i ⊆ 𝒬 j
  by (rule subset-fSuc[where u=i and v=j and f = 𝒬, OF subset-𝒬Suc])

lemma empty-𝒳[simp]: k > length Doc ⟹ 𝒳 k = {}
  apply (simp add: 𝒳-def)
```

```
  apply (insert Lex-is-lexer)
  by (simp add: is-lexer-def)

lemma Sel-empty[simp]: Sel {} {} = {}
  apply (insert Sel-is-selector)
  by (auto simp add: is-selector-def)

lemma empty-𝒵[simp]: k > length Doc ⟹ 𝒵 k u = {}
  apply (induct u)
  by (simp-all add: 𝒴-def 𝒲-def)

lemma[simp]: Append {} k = id by (auto simp add: Append-def)

lemma[simp]: k > length Doc ⟹ 𝒫 k v = 𝒫 k 0
  by (induct v, simp-all add: 𝒴-def 𝒲-def)

lemma 𝒬SucEq: k ≥ length Doc ⟹ 𝒬 (Suc k) = 𝒬 k
  by (simp add: natUnion-def)

lemma 𝒬-converges:
  assumes k: k ≥ length Doc
  shows 𝒬 k = 𝔓
proof −
  {
    fix n
    have 𝒬 (length Doc + n) = 𝔓
    proof (induct n)
      show 𝒬 (length Doc + 0) = 𝔓 by (simp add: 𝔓-def)
    next
      fix n
      assume hyp: 𝒬 (length Doc + n) = 𝔓
      have 𝒬 (Suc (length Doc + n)) = 𝔓
        by (rule trans[OF 𝒬SucEq hyp], auto)
      then show 𝒬 (length Doc + Suc n) = 𝔓
        by auto
    qed
  }
  note helper = this
  from k have ∃ n. k = length Doc + n by presburger
  then obtain n where n: k = length Doc + n by blast
  then show ?thesis
    apply (simp only: n)
    by (rule helper)
qed

lemma 𝔓-covers-𝒬: 𝒬 k ⊆ 𝔓
proof (case-tac k ≥ length Doc)
  assume k ≥ length Doc
  then have 𝒬: 𝒬 k = 𝔓 by (rule 𝒬-converges)
```

```
  then show 𝒬 k ⊆ 𝔓 by (simp only: 𝒬)
next
  assume ¬ length Doc ≤ k
  then have k < length Doc by auto
  then show ?thesis
    apply (simp only: 𝔓-def)
    apply (rule subset-𝒬)
    by auto
qed

lemma Sel-upper-bound: A ⊆ B ⟹ Sel A B ⊆ B
  by (metis Sel-is-selector is-selector-def)

lemma Sel-lower-bound: A ⊆ B ⟹ A ⊆ Sel A B
  by (metis Sel-is-selector is-selector-def)

lemma 𝔓-covers-𝒫: 𝒫 k u ⊆ 𝔓
  by (rule subset-trans[OF subset-𝒫𝒬k 𝔓-covers-𝒬])

lemma 𝒲-montone: P ⊆ Q ⟹ 𝒲 P k ⊆ 𝒲 Q k
  by (auto simp add: 𝒲-def)

lemma Sel-precondition:
  𝒵 k u ⊆ 𝒲 (𝒫 k u) k
proof (induct u)
  case 0 thus ?case by simp
next
  case (Suc u)
  have 1: 𝒴 (𝒵 k u) (𝒫 k u) k ⊆ 𝒲 (𝒫 k u) k
    apply (simp add: 𝒴-def)
    apply (rule-tac Sel-upper-bound)
    using Suc by simp
  have 2: 𝒲 (𝒫 k u) k ⊆ 𝒲 (𝒫 k (Suc u)) k
    by (metis 𝒲-montone subset-𝒫Suc)
  show ?case
    apply (rule-tac subset-trans[where B=𝒲 (𝒫 k u) k])
    apply (simp add: 1)
    apply (simp only: 2)
    done
qed

lemma 𝒲-bounded-by-𝒳: 𝒲 P k ⊆ 𝒳 k
  by (metis (no-types, lifting) 𝒲-def mem-Collect-eq subsetI)

lemma 𝒵-subset-𝒳: 𝒵 k n ⊆ 𝒳 k
  by (metis Sel-precondition 𝒲-bounded-by-𝒳 rev-subsetD subsetI)

lemma 𝒵-subset-Suc: 𝒵 k n ⊆ 𝒵 k (Suc n)
apply (induct n)
```

```
apply simp
by (metis Sel-lower-bound Sel-precondition 𝒴-def 𝒵.simps(2))

lemma 𝒴-upper-bound: 𝒴 (𝒵 k u) (𝒫 k u) k ⊆ 𝒲 (𝒫 k u) k
  apply (simp add: 𝒴-def)
  by (metis Sel-precondition Sel-upper-bound)

lemma 𝔓-induct[consumes 1, case-names Base Induct]:
  assumes p: p ∈ 𝔓
  assumes base: P []
  assumes induct: ⋀ p k u. (⋀ q. q ∈ 𝒫 k u ⟹ P q) ⟹ p ∈ 𝒫 k (Suc u) ⟹
P p
  shows P p
proof −
  {
    fix p :: tokens
    fix k :: nat
    fix u :: nat
    have p ∈ 𝒫 k u ⟹ P p
    proof (induct k arbitrary: p u)
      case 0
        have p ∈ 𝒫 0 u ⟹ P p
        proof (induct u arbitrary: p)
          case 0 thus ?case using base by simp
        next
          case (Suc u) show ?case
            apply (rule induct[OF - Suc(2)])
            apply (rule Suc(1))
            by simp
        qed
        with 0 show ?case by simp
    next
      case (Suc k)
        have p ∈ 𝒫 (Suc k) u ⟹ P p
        proof (induct u arbitrary: p)
          case 0 thus ?case
            apply simp
            apply (induct rule: natUnion-decompose)
            using Suc by simp
        next
          case (Suc u) show ?case
            apply (rule induct[OF - Suc(2)])
            apply (rule Suc(1))
            by simp
        qed
        with Suc show ?case by simp
    qed
  }
  note all = this
```

from *p* **show** *?thesis*
apply (*simp add*: $\mathfrak{P}$-*def*)
apply (*induct rule*: *natUnion-decompose*)
using *all* **by** *simp*
qed

lemma *Append-mono*: $U \subseteq V \Longrightarrow P \subseteq Q \Longrightarrow$ *Append U k P* $\subseteq$ *Append V k Q*
by (*auto simp add*: *Append-def*)

lemma *pointwise-Append*: *pointwise* (*Append T k*)
by (*auto simp add*: *pointwise-def Append-def*)

lemma *regular-Append*: *regular* (*Append T k*)
proof −
have *pointwise* (*Append T k*) **using** *pointwise-Append* **by** *blast*
then have *pointbased* (*Append T k*) **using** *pointwise-implies-pointbased* **by** *blast*
then have *continuous* (*Append T k*) **using** *pointbased-implies-continuous* **by** *blast*
moreover have *setmonotone* (*Append T k*) **by** (*simp add*: *setmonotone-def Append-def*)
ultimately show *?thesis* **using** *regular-def* **by** *blast*
qed

end

end
theory *InductRules*
imports *Main*
begin

lemma *disjCases2*[*consumes 1*, *case-names 1 2*]:
assumes *AB*: $A \vee B$
and *AP*: $A \Longrightarrow P$
and *BP*: $B \Longrightarrow P$
shows *P*
proof −
from *AB AP BP* **show** *?thesis* **by** *blast*
qed

lemma *disjCases3*[*consumes 1*, *case-names 1 2 3*]:
assumes *AB*: $A \vee B \vee C$
and *AP*: $A \Longrightarrow P$
and *BP*: $B \Longrightarrow P$
and *CP*: $C \Longrightarrow P$
shows *P*
proof −
from *AB AP BP CP* **show** *?thesis* **by** *blast*
qed

```
lemma disjCases4[consumes 1, case-names 1 2 3 4]:
  assumes AB: A ∨ B ∨ C ∨ D
  and AP: A ⟹ P
  and BP: B ⟹ P
  and CP: C ⟹ P
  and DP: D ⟹ P
  shows P
proof −
  from AB AP BP CP DP show ?thesis by blast
qed

lemma disjCases5[consumes 1, case-names 1 2 3 4 5]:
  assumes AB: A ∨ B ∨ C ∨ D ∨ E
  and AP: A ⟹ P
  and BP: B ⟹ P
  and CP: C ⟹ P
  and DP: D ⟹ P
  and EP: E ⟹ P
  shows P
proof −
  from AB AP BP CP DP EP show ?thesis by blast
qed

lemma minimal-witness-ex:
  assumes k: P (k::nat)
  shows ∃ k0. k0 ≤ k ∧ P k0 ∧ (∀ k. k < k0 ⟶ ¬ (P k))
proof −
  let ?K = { h. h ≤ k ∧ P h }
  have finite-K: finite ?K by auto
  have k ∈ ?K by (simp add: k)
  then have nonempty-K: ?K ≠ {} by auto
  let ?k = Min ?K
  have witness: ?k ≤ k ∧ P ?k
    by (metis (mono-tags, lifting) Min-in finite-K mem-Collect-eq nonempty-K)
  have minimal: ∀ h. h < ?k ⟶ ¬ (P h)
    by (metis Min-le witness dual-order.strict-implies-order
        dual-order.trans finite-K leD mem-Collect-eq)
  from witness minimal show ?thesis by metis
qed

lemma minimal-witness[consumes 1, case-names Minimal]:
  assumes P (k::nat)
  and ⋀ K. K ≤ k ⟹ P K ⟹ (⋀ k. k < K ⟹ ¬ (P k)) ⟹ Q
  shows Q
proof −
  from assms minimal-witness-ex show ?thesis by metis
qed

lemma ex-minimal-witness[consumes 1, case-names Minimal]:
```

assumes $\exists\ k.\ P\ (k{::}nat)$
and $\bigwedge\ K.\ P\ K \Longrightarrow (\bigwedge\ k.\ k < K \Longrightarrow \neg\ (P\ k)) \Longrightarrow Q$
shows Q
proof –
from *assms minimal-witness-ex* **show** *?thesis* **by** *metis*
qed

end
theory *ListTools*
imports *Main*
begin

definition *is-first* :: $'a \Rightarrow 'a\ list \Rightarrow bool$
where
is-first $x\ u = (\exists\ v.\ u = [x]@v)$

definition *is-last* :: $'a \Rightarrow 'a\ list \Rightarrow bool$
where
is-last $x\ u = (\exists\ v.\ u = v@[x])$

definition *is-prefix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where
is-prefix $u\ v = (\exists\ w.\ u@w = v)$

definition *is-proper-prefix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where
is-proper-prefix $u\ v = (\exists\ w.\ w \neq [] \wedge u@w = v)$

lemma *is-prefix-eq-proper-prefix*: *is-prefix* $a\ b = (a = b \vee$ *is-proper-prefix* $a\ b)$
by (*metis append-Nil2 is-prefix-def is-proper-prefix-def*)

lemma *is-proper-prefix-eq-prefix*: *is-proper-prefix* $a\ b = (a \neq b \wedge$ *is-prefix* $a\ b)$
by (*metis append-self-conv is-prefix-eq-proper-prefix is-proper-prefix-def*)

definition *is-suffix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where
is-suffix $u\ v = (\exists\ w.\ w@u = v)$

definition *is-proper-suffix* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$
where
is-proper-suffix $u\ v = (\exists\ w.\ w \neq [] \wedge w@u = v)$

lemma *is-suffix-eq-proper-suffix*: *is-suffix* $a\ b = (a = b \vee$ *is-proper-suffix* $a\ b)$
by (*metis append-Nil is-proper-suffix-def is-suffix-def*)

lemma *is-proper-suffix-eq-suffix*: *is-proper-suffix* $a\ b = (a \neq b \wedge$ *is-suffix* $a\ b)$
by (*metis is-proper-suffix-def is-suffix-eq-proper-suffix self-append-conv2*)

lemma *is-prefix-unsplit*: *is-prefix* $u\ a \Longrightarrow u\ @\ (drop\ (length\ u)\ a) = a$

by (*metis append-eq-conv-conj is-prefix-def*)

lemma *le-take-same*: $i \leq j \Longrightarrow$ *take j a* = *take j b* $\Longrightarrow$ *take i a* = *take i b*
by (*metis min.absorb1 take-take*)

lemma *is-first-drop-length*:
assumes $k \leq$ *length a*
and $k >$ *length u*
and $v = X\#w$
and *take k a* = *take k* (*u@v*)
shows *is-first X* (*drop* (*length u*) *a*)
proof −
let *?d* = *k* − *length u*
from *assms* **have** *pos-d'*: *?d* > *0* **by** *auto*
from *assms* **have** *take-d'-v*: *take ?d* (*drop* (*length u*) *a*) = *take ?d v*
by (*metis append-eq-conv-conj drop-take*)
then have *take 1* (*drop* (*length u*) *a*) = *take 1 v*
by (*metis One-nat-def Suc-leI le-take-same pos-d'*)
then have *take 1* (*drop* (*length u*) *a*) = [*X*]
by (*simp add*: *assms*)
then show *?thesis*
by (*metis append-take-drop-id is-first-def*)
qed

lemma *is-first-cons*: *is-first x* (*y#ys*) = (*x* = *y*)
by (*auto simp add*: *is-first-def*)

lemma *list-all-pos-neg-ex*: *list-all P D* $\Longrightarrow \neg$ (*list-all Q D*) $\Longrightarrow$
$\exists$ *k*. *k* < *length D* $\wedge$ *P*(*D* ! *k*) $\wedge$ $\neg$(*Q*(*D* ! *k*))
using *list-all-length* **by** *blast*

lemma *split-list-at*: *k* < *length D* $\Longrightarrow$ *D* = (*take k D*)@[*D* ! *k*]@(*drop* (*Suc k*) *D*)
by (*metis append-Cons append-Nil id-take-nth-drop*)

lemma *take-eq-take-append*: $i \leq j \Longrightarrow j \leq$ *length a* $\Longrightarrow \exists$ *u*. *take j a* = *take i a* @ *u*
by (*metis le-Suc-ex take-add*)

lemma *is-proper-suffix-length-cmp*: *is-proper-suffix a b* $\Longrightarrow$ *length a* < *length b*
by (*metis add-diff-cancel-right' append-Nil append-eq-append-conv diff-is-0-eq is-proper-suffix-def leI length-append list.size*(*3*))

end
theory *Derivations*
imports *CFG ListTools InductRules*
begin

context *CFG* **begin**

lemma [*simp*]: *is-terminal t* $\Longrightarrow$ *is-symbol t*
by (*auto simp add*: *is-terminal-def is-symbol-def*)

lemma [*simp*]: *is-sentence* [] **by** (*auto simp add*: *is-sentence-def*)

lemma [*simp*]: *is-word* [] **by** (*auto simp add*: *is-word-def*)

lemma [*simp*]: *is-word u* $\Longrightarrow$ *is-sentence u*
by (*induct u*, *auto simp add*: *is-word-def is-sentence-def*)

definition *leftderives1* :: *sentence* ⇒ *sentence* ⇒ *bool*
where
leftderives1 u v =
(∃ *x y N α*.
u = *x* @ [*N*] @ *y*
∧ *v* = *x* @ *α* @ *y*
∧ *is-word x*
∧ *is-sentence y*
∧ (*N*, *α*) ∈ $\mathfrak{R}$)

lemma *leftderives1-implies-derives1*[*simp*]: *leftderives1 u v* $\Longrightarrow$ *derives1 u v*
apply (*auto simp add*: *leftderives1-def derives1-def*)
apply (*rule-tac x=x* **in** *exI*)
apply (*rule-tac x=y* **in** *exI*)
apply (*rule-tac x=N* **in** *exI*)
by *auto*

definition *leftderivations1* :: (*sentence* × *sentence*) *set*
where
leftderivations1 = { (*u*,*v*) | *u v*. *leftderives1 u v* }

lemma [*simp*]: *leftderivations1* ⊆ *derivations1*
by (*auto simp add*: *leftderivations1-def derivations1-def*)

definition *leftderivations* :: (*sentence* × *sentence*) *set*
where
leftderivations = *leftderivations1*^∗

lemma *rtrancl-subset-implies*: *a* ⊆ *b* $\Longrightarrow$ *a* ⊆ *b*^∗ **by** *blast*

lemma *leftderivations-subset-derivations*[*simp*]: *leftderivations* ⊆ *derivations*
apply (*simp add*: *leftderivations-def derivations-def*)
apply (*rule rtrancl-subset-rtrancl*)
apply (*rule rtrancl-subset-implies*)
by *simp*

definition *leftderives* :: *sentence* ⇒ *sentence* ⇒ *bool*

where
leftderives u v = ((*u*, *v*) ∈ *leftderivations*)

lemma *leftderives-implies-derives*[*simp*]: *leftderives u v* ⟹ *derives u v*
apply (*auto simp add*: *leftderives-def derives-def*)
by (*rule subsetD*[*OF leftderivations-subset-derivations*])

definition *is-leftderivation* :: *sentence* ⇒ *bool*
where
is-leftderivation u = *leftderives* [𝔖] *u*

lemma *leftderivation-implies-derivation*[*simp*]:
is-leftderivation u ⟹ *is-derivation u*
by (*simp add*: *is-leftderivation-def is-derivation-def*)

lemma *leftderives-refl*[*simp*]: *leftderives u u*
by (*auto simp add*: *leftderives-def leftderivations-def*)

lemma *leftderives1-implies-leftderives*[*simp*]:*leftderives1 a b* ⟹ *leftderives a b*
by (*auto simp add*: *leftderives-def leftderivations-def leftderivations1-def*)

lemma *leftderives-trans*: *leftderives a b* ⟹ *leftderives b c* ⟹ *leftderives a c*
by (*auto simp add*: *leftderives-def leftderivations-def*)

lemma *leftderives1-eq-leftderivations1*: *leftderives1 x y* = ((*x*, *y*) ∈ *leftderivations1*)
by (*simp add*: *leftderivations1-def*)

lemma *leftderives-induct*[*consumes 1*, *case-names Base Step*]:
assumes *derives*: *leftderives a b*
assumes *Pa*: *P a*
assumes *induct*: ⋀*y z*. *leftderives a y* ⟹ *leftderives1 y z* ⟹ *P y* ⟹ *P z*
shows *P b*
proof −
note *rtrancl-lemma* = *rtrancl-induct*[**where** *a* = *a* **and** *b* = *b* **and** *r* = *leftderivations1* **and** *P*=*P*]
from *derives Pa induct rtrancl-lemma* **show** *P b*
by (*metis leftderives-def leftderivations-def leftderives1-eq-leftderivations1*)
qed

end

context *CFG* **begin**

lemma *derives1-implies-derives*[*simp*]:*derives1 a b* ⟹ *derives a b*
by (*auto simp add*: *derives-def derivations-def derivations1-def*)

lemma *derives-trans*: *derives a b* ⟹ *derives b c* ⟹ *derives a c*

by (*auto simp add*: *derives-def derivations-def*)

lemma *derives1-eq-derivations1*: *derives1 x y* = ((*x*, *y*) ∈ *derivations1*)
by (*simp add*: *derivations1-def*)

lemma *derives-induct*[*consumes 1*, *case-names Base Step*]:
assumes *derives*: *derives a b*
assumes *Pa*: *P a*
assumes *induct*: ⋀*y z*. *derives a y* ⟹ *derives1 y z* ⟹ *P y* ⟹ *P z*
shows *P b*
proof −
note *rtrancl-lemma* = *rtrancl-induct*[**where** *a* = *a* **and** *b* = *b* **and** *r* = *derivations1* **and** *P*=*P*]
from *derives Pa induct rtrancl-lemma* **show** *P b*
by (*metis derives-def derivations-def derives1-eq-derivations1*)
qed

end

context *CFG* **begin**

definition *Derives1* :: *sentence* ⇒ *nat* ⇒ *rule* ⇒ *sentence* ⇒ *bool*
where
Derives1 u i r v =
(∃ *x y N α*.
u = *x* @ [*N*] @ *y*
∧ *v* = *x* @ *α* @ *y*
∧ *is-sentence x*
∧ *is-sentence y*
∧ (*N*, *α*) ∈ 𝔑
∧ *r* = (*N*, *α*) ∧ *i* = *length x*)

lemma *Derives1-split*:
Derives1 u i r v ⟹ ∃ *x y*. *u* = *x* @ [*fst r*] @ *y* ∧ *v* = *x* @ (*snd r*) @ *y* ∧ *length x* = *i*
by (*metis Derives1-def fst-conv snd-conv*)

lemma *Derives1-implies-derives1*: *Derives1 u i r v* ⟹ *derives1 u v*
by (*auto simp add*: *Derives1-def derives1-def*)

lemma *derives1-implies-Derives1*: *derives1 u v* ⟹ ∃ *i r*. *Derives1 u i r v*
by (*auto simp add*: *Derives1-def derives1-def*)

lemma *Derives1-unique-dest*: *Derives1 u i r v* ⟹ *Derives1 u i r w* ⟹ *v* = *w*
by (*auto simp add*: *Derives1-def derives1-def*)

lemma *Derives1-unique-src*: *Derives1 u i r w* ⟹ *Derives1 v i r w* ⟹ *u* = *v*
by (*auto simp add*: *Derives1-def derives1-def*)

type-synonym *derivation* = (*nat* × *rule*) *list*

fun *Derivation* :: *sentence* ⇒ *derivation* ⇒ *sentence* ⇒ *bool*
where
Derivation a [] *b* = (*a* = *b*)
| *Derivation a* (*d*#*D*) *b* = (∃ *x*. *Derives1 a* (*fst d*) (*snd d*) *x* ∧ *Derivation x D b*)

lemma *Derivation-implies-derives*: *Derivation a D b* ⟹ *derives a b*
proof (*induct D arbitrary*: *a b*)
case *Nil* **thus** *?case*
by (*simp add*: *derives-def derivations-def*)
next
case (*Cons d D*)
note *ihyps* = *this*
from *ihyps* **have** ∃ *x*. *Derives1 a* (*fst d*) (*snd d*) *x* ∧ *Derivation x D b* **by** *auto*
then obtain *x* **where** *Derives1 a* (*fst d*) (*snd d*) *x* **and** *xb*: *Derivation x D b*
by *blast*
with *Derives1-implies-derives1* **have** *d1*: *derives a x* **by** *auto*
from *ihyps xb* **have** *d2*:*derives x b* **by** *simp*
show *derives a b* **by** (*rule derives-trans*[*OF d1 d2*])
qed

lemma *Derivation-Derives1*: *Derivation a S y* ⟹ *Derives1 y i r z* ⟹ *Derivation a* (*S*@[(*i*,*r*)]) *z*
proof (*induct S arbitrary*: *a y z i r*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons s S*) **thus** *?case*
by (*metis Derivation.simps*(*2*) *append-Cons*)
qed

lemma *derives-implies-Derivation*: *derives a b* ⟹ ∃ *D*. *Derivation a D b*
proof (*induct rule*: *derives-induct*)
case *Base*
show *?case* **by** (*rule exI*[**where** *x*=[]], *simp*)
next
case (*Step y z*)
note *ihyps* = *this*
from *ihyps* **obtain** *D* **where** *ay*: *Derivation a D y* **by** *blast*
from *ihyps derives1-implies-Derives1* **obtain** *i r* **where** *yz*: *Derives1 y i r z* **by** *blast*
from *Derivation-Derives1*[*OF ay yz*] **show** *?case* **by** *auto*
qed

lemma *Derives1-take*: *Derives1 a i r b* ⟹ *take i a* = *take i b*
by (*auto simp add*: *Derives1-def*)

lemma *Derives1-drop*: *Derives1 a i r b* ⟹ *drop* (*Suc i*) *a* = *drop* (*i* + *length* (*snd*

r)) *b*
by (*auto simp add*: *Derives1-def*)

lemma *Derives1-bound*: *Derives1 a i r b* $\Longrightarrow$ *i* < *length a*
by (*auto simp add*: *Derives1-def*)

lemma *Derives1-length*: *Derives1 a i r b* $\Longrightarrow$ *length b* = *length a* + *length* (*snd r*) − *1*
by (*auto simp add*: *Derives1-def*)

definition *leftmost* :: *nat* ⇒ *sentence* ⇒ *bool*
where
leftmost i s = (*i* < *length s* ∧ *is-word* (*take i s*) ∧ *is-nonterminal* (*s* ! *i*))

lemma *set-take*: *set* (*take n s*) = { *s* ! *i* | *i*. *i* < *n* ∧ *i* < *length s*}
proof (*cases n* ≤ *length s*)
case *True* **thus** *?thesis*
by (*subst List.nth-image*[*symmetric*], *auto*)
next
case *False* **thus** *?thesis*
apply (*subst set-conv-nth*)
by (*metis less-trans linear not-le take-all*)
qed

lemma *list-all-take*: *list-all P* (*take n s*) = (∀ *i*. *i* < *n* ∧ *i* < *length s* ⟶ *P* (*s* ! *i*))
by (*auto simp add*: *set-take list-all-iff*)

lemma *is-sentence-concat*: *is-sentence* (*x*@*y*) = (*is-sentence x* ∧ *is-sentence y*)
by (*auto simp add*: *is-sentence-def*)

lemma *is-sentence-cons*: *is-sentence* (*x*#*xs*) = (*is-symbol x* ∧ *is-sentence xs*)
by (*auto simp add*: *is-sentence-def*)

lemma *rule-nonterminal-type*[*simp*]: (*N*, α) ∈ $\mathfrak{R}$ $\Longrightarrow$ *is-nonterminal N*
apply (*insert validRules*)
by (*auto simp add*: *is-nonterminal-def*)

lemma *rule-*α*-type*[*simp*]: (*N*, α) ∈ $\mathfrak{R}$ $\Longrightarrow$ *is-sentence* α
apply (*insert validRules*)
by (*auto simp add*: *is-sentence-def is-symbol-def list-all-iff is-terminal-def is-nonterminal-def*)

lemma [*simp*]: *is-nonterminal N* $\Longrightarrow$ *is-symbol N*
by (*simp add*: *is-symbol-def*)

lemma *Derives1-sentence1*[*elim*]: *Derives1 a i r b* $\Longrightarrow$ *is-sentence a*
by (*auto simp add*: *Derives1-def is-sentence-cons is-sentence-concat*)

lemma *Derives1-sentence2*[*elim*]: *Derives1 a i r b* $\Longrightarrow$ *is-sentence b*

```
  by (auto simp add: Derives1-def is-sentence-cons is-sentence-concat)

lemma [elim]: Derives1 a i r b ⟹ r ∈ ℜ
  using Derives1-def by auto

lemma is-sentence-symbol: is-sentence a ⟹ i < length a ⟹ is-symbol (a ! i)
  by (simp add: is-sentence-def list-all-iff)

lemma is-symbol-distinct: is-symbol x ⟹ is-terminal x ≠ is-nonterminal x
  apply (insert disjunct-symbols)
  apply (auto simp add: is-symbol-def is-terminal-def is-nonterminal-def)
  done

lemma is-terminal-nonterminal: is-terminal x ⟹ is-nonterminal x ⟹ False
  by (metis is-symbol-def is-symbol-distinct)

lemma Derives1-leftmost:
  assumes Derives1 a i r b
  shows ∃ j. leftmost j a ∧ j ≤ i
proof −
  let ?J = {j . j < length a ∧ is-nonterminal (a ! j)}
  let ?M = Min ?J
  from assms have J1:i ∈ ?J
    apply (auto simp add: Derives1-def is-nonterminal-def)
    by (metis (mono-tags, lifting) prod.simps(2) validRules)
  have J2:finite ?J by auto
  note J = J1 J2
  from J have M1: ?M ∈ ?J by (rule-tac Min-in, auto)
  {
    fix j
    assume j ∈ ?J
    with J have ?M ≤ j by auto
  }
  note M3 = this[OF J1]
  from assms have a-sentence: is-sentence a by (simp add: Derives1-sentence1)
  have is-word: is-word (take ?M a)
    apply (auto simp add: is-word-def list-all-take)
    proof −
      fix i
      assume i-less-M: i < ?M
      assume i-inbounds: i < length a
      show is-terminal (a ! i)
      proof(cases is-terminal (a ! i))
        case True thus ?thesis by auto
      next
        case False
        then have is-nonterminal (a ! i)
        using i-inbounds a-sentence is-sentence-symbol is-symbol-distinct by blast
        then have i ∈ ?J by (metis i-inbounds mem-Collect-eq)
```

then have *?M < i* **by** (*metis J2 Min-le i-less-M leD*)
then have *False* **by** (*metis i-less-M less-asym′*)
then show *?thesis* **by** *auto*
qed
qed
show *?thesis*
apply (*rule-tac exI*[**where** *x=?M*])
apply (*simp add: leftmost-def*)
by (*metis* (*mono-tags, lifting*) *M1 M3 is-word mem-Collect-eq*)
qed

lemma *Derivation-leftmost*: *D ≠ [] ⟹ Derivation a D b ⟹ ∃ i. leftmost i a*
apply (*case-tac D*)
apply (*auto*)
apply (*metis Derives1-leftmost*)
done

lemma *nonword-has-nonterminal*:
is-sentence a ⟹ ¬ (is-word a) ⟹ ∃ k. k < length a ∧ is-nonterminal (a ! k)
apply (*auto simp add: is-sentence-def list-all-iff is-word-def*)
by (*metis in-set-conv-nth is-symbol-distinct*)

lemma *leftmost-cons-nonterminal*:
is-nonterminal x ⟹ leftmost 0 (x#xs)
by (*metis CFG.is-word-def CFG-axioms leftmost-def length-greater-0-conv list.distinct(1)*

list-all-simps(2) nth-Cons-0 take-Cons′)

lemma *leftmost-cons-terminal*:
is-terminal x ⟹ leftmost i (x#xs) = (i > 0 ∧ leftmost (i − 1) xs)
by (*metis Suc-diff-1 Suc-less-eq is-terminal-nonterminal is-word-def leftmost-def length-Cons*
list-all-simps(1) not-gr0 nth-Cons′ take-Cons′)

lemma *is-nonterminal-cons-terminal*:
is-terminal x ⟹ k < length (x # a) ⟹ is-nonterminal ((x # a) ! k) ⟹
k > 0 ∧ k − 1 < length a ∧ is-nonterminal (a ! (k − 1))
by (*metis One-nat-def Suc-leI is-terminal-nonterminal less-diff-conv2*
list.size(4) nth-non-equal-first-eq)

lemma *leftmost-exists*:
is-sentence a ⟹ k < length a ⟹ is-nonterminal (a ! k) ⟹
∃ i. leftmost i a ∧ i ≤ k
proof (*induct a arbitrary: k*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons x a*)
show *?case*
proof(*cases is-nonterminal x*)

case *True* **thus** *?thesis*
apply(*rule-tac exI*[**where** *x=0*])
apply (*simp add*: *leftmost-cons-nonterminal*)
done
next
case *False*
then have *x*: *is-terminal x*
by (*metis Cons.prems*(*1*) *is-sentence-cons is-symbol-distinct*)
note *k* = *is-nonterminal-cons-terminal*[*OF x Cons*(*3*) *Cons*(*4*)]
with *Cons* **have** ∃ *i*. *leftmost i a* ∧ *i* ≤ *k* − *1* **by** (*metis is-sentence-cons*)
then show *?thesis*
apply (*auto simp add*: *leftmost-cons-terminal*[*OF x*])
by (*metis le-diff-conv2 Suc-leI add-Suc-right add-diff-cancel-right′ k le-0-eq le-imp-less-Suc nat-le-linear*)
qed
qed

lemma *nonword-leftmost-exists*:
is-sentence a ⟹ ¬ (*is-word a*) ⟹ ∃ *i*. *leftmost i a*
by (*metis leftmost-exists nonword-has-nonterminal*)

lemma *leftmost-unaffected-Derives1*: *leftmost j a* ⟹ *j* < *i* ⟹ *Derives1 a i r b* ⟹ *leftmost j b*
apply (*simp add*: *leftmost-def*)
proof −
assume *a1*: *j* < *length a* ∧ *is-word* (*take j a*) ∧ *is-nonterminal* (*a* ! *j*)
assume *a2*: *j* < *i*
assume *Derives1 a i r b*
then have *f3*: *take i a* = *take i b*
by (*metis Derives1-take*)
have *f4*: ⋀*n ss ssa*. *take* (*length* (*take n* (*ss*::*symbol list*))) (*ssa*::*symbol list*) = *take* (*length ss*) (*take n ssa*)
by (*metis length-take take-take*)
have *f5*: ⋀*ss*. *take j* (*ss*::*symbol list*) = *take i* (*take j ss*)
using *a2* **by** (*metis dual-order.strict-implies-order min.absorb2 take-take*)
have *f6*: *length* (*take j a*) = *j*
using *a1* **by** (*metis dual-order.strict-implies-order length-take min.absorb2*)
then have *f7*: ⋀*n*. *min j n* = *length* (*take n* (*take j a*))
by (*metis length-take*)
have *f8*: ⋀*n ss*. *n* = *length* (*take n* (*ss*::*symbol list*)) ∨ *length ss* < *n*
by (*metis leI length-take min.absorb2*)
have *f9*: ⋀*ss*. *take j* (*ss*::*symbol list*) = *take j* (*take i ss*)
using *f7 f6 f5* **by** (*metis take-take*)
have *f10*: ⋀*ss n*. *length* (*ss*::*symbol list*) ≤ *n* ∨ *length* (*take n ss*) = *n*
using *f8* **by** (*metis dual-order.strict-implies-order*)
then have *f11*: ⋀*ss ssa*. *length* (*ss*::*symbol list*) = *length* (*take* (*length ss*) (*ssa*::*symbol list*)) ∨ *length* (*take* (*length ssa*) *ss*) = *length ssa*
by (*metis length-take min.absorb2*)
have *f12*: ⋀*ss ssa n*. *take* (*length* (*ss*::*symbol list*)) (*ssa*::*symbol list*) = *take n*

```
(take (length ss) ssa) ∨ length (take n ss) = n
    using f10 by (metis min.absorb2 take-take)
  { assume ¬ j < j
    { assume ¬ j < j ∧ i ≠ j
      moreover
      { assume length a ≠ j ∧ length (take i a) ≠ i
          then have ∃ss. length (take (length (take i (take (length a) (ss::symbol
list)))) (take j ss)) ≠ length (take i (take (length a) ss))
          using f12 f11 f6 f5 f4 by metis
        then have ∃ss ssa. take (length (ss::symbol list)) (take j (ssa::symbol list))
≠ take (length ss) (take i (take (length a) ssa))
          using f11 by metis
        then have length b ≠ j
          using f9 f4 f3 by metis }
      ultimately have length b ≠ j
        using f7 f6 f5 f3 a1 by (metis length-take) }
    then have length b = j ⟶ j < j
      using a2 by metis }
  then have j < length b
    using f9 f8 f7 f6 f4 f3 by metis
  then show j < length b ∧ is-word (take j b) ∧ is-nonterminal (b ! j)
    using f9 f3 a2 a1 by (metis nth-take)
qed

definition derivation-ge :: derivation ⇒ nat ⇒ bool
where
  derivation-ge D i = (∀ d ∈ set D. fst d ≥ i)

lemma derivation-ge-cons: derivation-ge (d#D) i = (fst d ≥ i ∧ derivation-ge D
i)
  by (auto simp add: derivation-ge-def)

lemma derivation-ge-append:
  derivation-ge (D@E) i = (derivation-ge D i ∧ derivation-ge E i)
  by (auto simp add: derivation-ge-def)

lemma leftmost-unaffected-Derivation:
  derivation-ge D (Suc i) ⟹ leftmost i a ⟹ Derivation a D b ⟹ leftmost i b
proof (induct D arbitrary: a)
  case Nil thus ?case by auto
next
  case (Cons d D)
  then have ∃ x. Derives1 a (fst d) (snd d) x ∧ Derivation x D b by simp
  then obtain x where x1: Derives1 a (fst d) (snd d) x and x2: Derivation x D
b by blast
  with Cons have leftmost-x: leftmost i x
    apply (rule-tac leftmost-unaffected-Derives1[
          where a=a and j=i and b=x and i=fst d and r=snd d])
    by (auto simp add: derivation-ge-def)
```

with *Cons x2* **show** *?case* **by** (*auto simp add*: *derivation-ge-def*)
qed

lemma *le-Derives1-take*:
assumes *le*: $i \leq j$
and *D*: *Derives1 a j r b*
shows *take i a* = *take i b*
proof −
note *Derives1-take*[**where** *a*=*a* **and** *i*=*j* **and** *r*=*r* **and** *b*=*b*]
with *le D* **show** *?thesis* **by** (*rule-tac le-take-same*[**where** *i*=*i* **and** *j*=*j*], *auto*)
qed

lemma *Derivation-take*: *derivation-ge D i* ⟹ *Derivation a D b* ⟹ *take i a* = *take i b*
proof(*induct D arbitrary*: *a b*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons d D*)
then have ∃ *x*. *Derives1 a* (*fst d*) (*snd d*) *x* ∧ *Derivation x D b*
by *simp*
then obtain *x* **where** *ax*: *Derives1 a* (*fst d*) (*snd d*) *x* **and** *xb*: *Derivation x D b* **by** *blast*
from *derivation-ge-cons Cons*(*2*) **have** *d*: $i \leq$ *fst d* **and** *D*: *derivation-ge D i* **by** *auto*
note *take-i-xb* = *Cons*(*1*)[*OF D xb*]
note *take-i-ax* = *le-Derives1-take*[*OF d ax*]
from *take-i-xb take-i-ax* **show** *?case* **by** *auto*
qed

lemma *leftmost-cons-less*: *i* < *length u* ⟹ *leftmost i* (*u*@*v*) = *leftmost i u*
by (*auto simp add*: *leftmost-def nth-append*)

lemma *leftmost-is-nonterminal*: *leftmost i u* ⟹ *is-nonterminal* (*u* ! *i*)
by (*metis leftmost-def*)

lemma *is-word-is-terminal*: *i* < *length u* ⟹ *is-word u* ⟹ *is-terminal* (*u* ! *i*)
by (*metis is-word-def list-all-length*)

lemma *leftmost-append*:
assumes *leftmost*: *leftmost i* (*u*@*v*)
and *is-word*: *is-word u*
shows *length u* ≤ *i*
proof (*cases i* < *length u*)
case *False* **thus** *?thesis* **by** *auto*
next
case *True*
with *leftmost* **have** *leftmost i u* **using** *leftmost-cons-less*[*OF True*] **by** *simp*
then have *is-nonterminal*: *is-nonterminal* (*u* ! *i*) **by** (*rule leftmost-is-nonterminal*)
note *is-terminal* = *is-word-is-terminal*[*OF True is-word*]

```
  note is-terminal-nonterminal[OF is-terminal is-nonterminal]
  then show ?thesis by auto
qed

lemma derivation-ge-empty[simp]: derivation-ge [] i
by (simp add: derivation-ge-def)

lemma leftmost-notword: leftmost i a ⟹ j > i ⟹ ¬ (is-word (take j a))
by (metis is-terminal-nonterminal is-word-def leftmost-def list-all-take)

lemma leftmost-unique: leftmost i a ⟹ leftmost j a ⟹ i = j
by (metis leftmost-def leftmost-notword linorder-neqE-nat)

lemma leftmost-Derives1: leftmost i a ⟹ Derives1 a j r b ⟹ i ≤ j
by (metis Derives1-leftmost leftmost-unique)

lemma leftmost-Derives1-propagate:
  assumes leftmost: leftmost i a
      and Derives1: Derives1 a j r b
    shows (is-word b ∧ i = j) ∨ (∃ k. leftmost k b ∧ i ≤ k)
proof −
  from leftmost-Derives1[OF leftmost Derives1] have ij: i ≤ j by auto
  show ?thesis
  proof (cases is-word b)
    case True show ?thesis
      by (metis Derives1 True ij le-neq-implies-less leftmost
          leftmost-unaffected-Derives1 order-refl)
  next
    case False show ?thesis
     by (metis (no-types, opaque-lifting) Derives1 Derives1-bound Derives1-sentence2

          Derives1-take append-take-drop-id ij le-neq-implies-less leftmost
          leftmost-append leftmost-cons-less leftmost-def length-take
          min.absorb2 nat-le-linear nonword-leftmost-exists not-le)
  qed
qed

lemma is-word-Derives1[elim]: is-word a ⟹ Derives1 a i r b ⟹ False
by (metis Derives1-leftmost is-terminal-nonterminal is-word-is-terminal leftmost-def)

lemma is-word-Derivation[elim]: is-word a ⟹ Derivation a D b ⟹ D = []
by (metis Derivation-leftmost is-terminal-nonterminal is-word-def
    leftmost-def list-all-length)

lemma leftmost-Derivation:
  leftmost i a ⟹ Derivation a D b ⟹ j ≤ i ⟹ derivation-ge D j
proof (induct D arbitrary: a b i j)
  case Nil thus ?case by auto
next
```

```
  case (Cons d D)
  then have ∃ x. Derives1 a (fst d) (snd d) x ∧ Derivation x D b by auto
  then obtain x where ax:Derives1 a (fst d) (snd d) x and xb:Derivation x D b
by blast
  note ji = Cons(4)
  note i-fstd = leftmost-Derives1[OF Cons(2) ax]
  note disj = leftmost-Derives1-propagate[OF Cons(2) ax]
  thus ?case
  proof(induct rule: disjCases2)
    case 1
    with xb have D = [] by auto
    with 1 ji show ?case by (simp add: derivation-ge-def)
  next
    case 2
    then obtain k where k: leftmost k x and ik: i ≤ k by blast
    note ge = Cons(1)[OF k xb, where j=j]
    from ji ik i-fstd ge show ?case
      by (simp add: derivation-ge-cons)
  qed
qed

lemma derivation-ge-list-all: derivation-ge D i = list-all (λ d. fst d ≥ i) D
by (simp add: Ball-set derivation-ge-def)

lemma split-derivation-leftmost:
  assumes derivation-ge D i
  and ¬ (derivation-ge D (Suc i))
  shows ∃ E F r. D = E@[(i, r)]@F ∧ derivation-ge E (Suc i)
proof −
  from assms have ∃ k. k < length D ∧ fst(D ! k) ≥ i ∧ ¬(fst(D ! k) ≥ Suc i)
    by (metis derivation-ge-def in-set-conv-nth)
  then have ∃ k. k < length D ∧ fst(D ! k) = i by auto
  then show ?thesis
  proof(induct rule: ex-minimal-witness)
    case (Minimal k)
      then have k-len: k < length D and k-i: fst (D ! k) = i by auto
      let ?E = take k D
      let ?r = snd (D ! k)
      let ?F = drop (Suc k) D
      note split = split-list-at[OF k-len]
      from k-i have D-k: D ! k = (i, ?r) by auto
      show ?case
        apply (rule exI[where x=?E])
        apply (rule exI[where x=?F])
        apply (rule exI[where x=?r])
        apply (subst D-k[symmetric])
        apply (rule conjI)
        apply (rule split)
        by (metis (mono-tags, lifting) Minimal.hyps(2) Suc-leI assms(1)
```

```
        derivation-ge-list-all le-neq-implies-less list-all-length list-all-take)
  qed
qed

lemma Derives1-Derives1-swap:
  assumes i < j
  and Derives1 a j p b
  and Derives1 b i q c
  shows ∃ b′. Derives1 a i q b′ ∧ Derives1 b′ (j − 1 + length (snd q)) p c
proof −
  from Derives1-split[OF assms(2)] obtain a1 a2 where
    a-src: a = a1 @ [fst p] @ a2 and a-dest: b = a1 @ snd p @ a2
    and a1-len: length a1 = j by blast
  note a = this
  from a have is-sentence-a1: is-sentence a1
    using Derives1-sentence2 assms(2) is-sentence-concat by blast
  from a have is-sentence-a2: is-sentence a2
    using Derives1-sentence2 assms(2) is-sentence-concat by blast
  from a have is-symbol-fst-p:  is-symbol (fst p)
    by (metis Derives1-sentence1 assms(2) is-sentence-concat is-sentence-cons)
  from Derives1-split[OF assms(3)] obtain b1 b2 where
    b-src: b = b1 @ [fst q] @ b2 and b-dest: c = b1 @ snd q @ b2
    and b1-len: length b1 = i by blast
  have a-take-j: a1 = take j a by (metis a1-len a-src append-eq-conv-conj)
  have b-take-i: b1 @ [fst q] = take (Suc i) b
   by (metis append-assoc append-eq-conv-conj b1-len b-src length-append-singleton)

  from a-take-j b-take-i  take-eq-take-append[where j=j and i=Suc i and a=a]
  have ∃ u. a1 = (b1 @ [fst q]) @ u
   by (metis le-iff-add Suc-leI a1-len a-dest append-eq-conv-conj assms(1) take-add)

  then obtain u where u1: a1 = (b1 @ [fst q]) @ u by blast
  then have j-i-u: j = i + 1 + length u
   using Suc-eq-plus1 a1-len b1-len length-append length-append-singleton by auto
  from u1 is-sentence-a1 have is-sentence-b1-u: is-sentence b1 ∧ is-sentence u
    using is-sentence-concat by blast
 have u2: b2 = u @ snd p @ a2 by (metis a-dest append-assoc b-src same-append-eq
u1)
  let ?b = b1 @ (snd q) @ u @ [fst p] @ a2
  from assms have q-dom: q ∈ 𝔑 by auto
  have a-b′: Derives1 a i q ?b
    apply (subst Derives1-def)
    apply (rule exI[where x=b1])
    apply (rule exI[where x=u@[fst p]@a2])
    apply (rule exI[where x=fst q])
    apply (rule exI[where x=snd q])
    apply (auto simp add: b1-len is-sentence-cons is-sentence-concat
          is-sentence-a2 is-symbol-fst-p is-sentence-b1-u a-src u1 q-dom)
    done
```

```
  from assms have p-dom: p ∈ ℜ by auto
  have is-sentence-snd-q: is-sentence (snd q)
    using Derives1-sentence2 a-b′ is-sentence-concat by blast
  have b′-c: Derives1 ?b (j − 1 + length (snd q)) p c
    apply (subst Derives1-def)
    apply (rule exI[where x=b1 @ (snd q) @ u])
    apply (rule exI[where x=a2])
    apply (rule exI[where x=fst p])
    apply (rule exI[where x=snd p])
   apply (auto simp add: is-sentence-concat is-sentence-b1-u is-sentence-a2 p-dom
          is-sentence-snd-q b-dest u2 b1-len j-i-u)
    done
  show ?thesis
    apply (rule exI[where x=?b])
    apply (rule conjI)
    apply (rule a-b′)
    apply (rule b′-c)
    done
qed

definition derivation-shift :: derivation ⇒ nat ⇒ nat ⇒ derivation
where
  derivation-shift D left right = map (λ d. (fst d − left + right, snd d)) D

lemma derivation-shift-empty[simp]: derivation-shift [] left right = []
  by (auto simp add: derivation-shift-def)

lemma derivation-shift-cons[simp]:
  derivation-shift (d#D) left right = ((fst d − left + right, snd d)#(derivation-shift
D left right))
by (simp add: derivation-shift-def)

lemma Derivation-append: Derivation a (D@E) c = (∃ b. Derivation a D b ∧
Derivation b E c)
proof(induct D arbitrary: a c E)
  case Nil thus ?case by auto
next
  case (Cons d D) thus ?case by auto
qed

lemma Derivation-implies-append:
  Derivation a D b ⟹ Derivation b E c ⟹ Derivation a (D@E) c
using Derivation-append by blast

lemma Derivation-swap-single-end-to-front:
  i < j ⟹ derivation-ge D j ⟹ Derivation a (D@[(i,r)]) b ⟹
   Derivation a ((i,r)#(derivation-shift D 1 (length (snd r)))) b
proof(induct D arbitrary: a)
  case Nil thus ?case by auto
```

next
  case (*Cons d D*)
  from *Cons* **have** ∃ *c*. *Derives1 a* (*fst d*) (*snd d*) *c* ∧ *Derivation c* (*D* @ [(*i*, *r*)]) *b*
    by *simp*
  then obtain *c* **where** *ac*: *Derives1 a* (*fst d*) (*snd d*) *c*
    and *cb*: *Derivation c* (*D* @ [(*i*, *r*)]) *b* **by** *blast*
  from *Cons*(*3*) **have** *D-j*: *derivation-ge D j* **by** (*simp add*: *derivation-ge-cons*)
  from *Cons*(*1*)[*OF Cons*(*2*) *D-j cb*, *simplified*]
  obtain *x* **where** *cx*: *Derives1 c i r x* **and**
    xb: *Derivation x* (*derivation-shift D 1* (*length* (*snd r*))) *b* **by** *auto*
  have *i-fst-d*: *i* < *fst d* **using** *Cons derivation-ge-cons* **by** *auto*
  from *Derives1-Derives1-swap*[*OF i-fst-d ac cx*]
  obtain *b′* **where** *ab′*: *Derives1 a i r b′* **and**
    b′x: *Derives1 b′* (*fst d* − *1* + *length* (*snd r*)) (*snd d*) *x* **by** *blast*
  show *?case* **using** *ab′ b′x xb* **by** *auto*
qed

lemma *Derivation-swap-single-mid-to-front*:
  assumes *i* < *j*
  and *derivation-ge D j*
  and *Derivation a* (*D*@[(*i*,*r*)]@*E*) *b*
  shows *Derivation a* ((*i*,*r*)#((*derivation-shift D 1* (*length* (*snd r*)))@*E*)) *b*
proof −
  from *assms* **have** ∃ *x*. *Derivation a* (*D*@[(*i*, *r*)]) *x* ∧ *Derivation x E b*
    using *Derivation-append* **by** *auto*
  then obtain *x* **where** *ax*: *Derivation a* (*D*@[(*i*, *r*)]) *x* **and** *xb*: *Derivation x E b*
    by *blast*
  with *assms* **have** *Derivation a* ((*i*, *r*)#(*derivation-shift D 1* (*length* (*snd r*)))) *x*
    using *Derivation-swap-single-end-to-front* **by** *blast*
  then show *?thesis* **using** *Derivation-append xb* **by** *auto*
qed

lemma *length-derivation-shift*[*simp*]:
  length(*derivation-shift D left right*) = *length D*
  by (*simp add*: *derivation-shift-def*)

definition *LeftDerives1* :: *sentence* ⇒ *nat* ⇒ *rule* ⇒ *sentence* ⇒ *bool*
where
  LeftDerives1 u i r v = (*leftmost i u* ∧ *Derives1 u i r v*)

lemma *LeftDerives1-implies-leftderives1*: *LeftDerives1 u i r v* ⟹ *leftderives1 u v*
by (*metis Derives1-def LeftDerives1-def append-eq-conv-conj leftderives1-def*
    leftmost-def)

lemma *leftmost-Derives1-leftderives*:
  leftmost i a ⟹ *Derives1 a i r b* ⟹ *leftderives b c* ⟹ *leftderives a c*
using *LeftDerives1-def LeftDerives1-implies-leftderives1*
  leftderives1-implies-leftderives leftderives-trans **by** *blast*

```
theorem Derivation-implies-leftderives-gen:
  Derivation a D (u@v) ⟹ is-word u ⟹ (∃ w.
          leftderives a (u@w) ∧
          (v = [] ⟶ w = []) ∧
          (∀ X. is-first X v ⟶ is-first X w))
proof (induct length D arbitrary: D a u v)
  case 0
    then have a = u@v by auto
    thus ?case by (rule-tac x = v in exI, auto)
next
  case (Suc n)
    from Suc have D ≠ [] by auto
    with Suc Derivation-leftmost have ∃ i. leftmost i a by auto
    then obtain i where i: leftmost i a by blast
    show ?case
    proof (cases derivation-ge D (Suc i))
      case True
        with Suc have leftmost: leftmost i (u@v)
          by (rule-tac leftmost-unaffected-Derivation[OF True i], auto)
        have length-u: length u ≤ i
          using leftmost-append[OF leftmost Suc(4)] .
        have take-Suc: take (Suc i) a = take (Suc i) (u@v)
          using  Derivation-take[OF True Suc(3)] .
        with length-u have is-prefix-u: is-prefix u a
         by (metis append-assoc append-take-drop-id dual-order.strict-implies-order

             is-prefix-def le-imp-less-Suc take-all take-append)
        have a: u @ drop (length u) a = a
          using is-prefix-unsplit[OF is-prefix-u] .
        from take-Suc have length-take-Suc: length (take (Suc i) a) = Suc i
          by (metis Suc-leI i leftmost-def length-take min.absorb2)
        have v: v ≠ []
        proof(cases v = [])
          case False thus ?thesis by auto
        next
          case True
          with length-u have right: length(take (Suc i) (u@v)) = length u by simp
          note left = length-take-Suc
          from left right take-Suc have Suc i = length u by auto
          with length-u show ?thesis by auto
        qed
        then have ∃ X w. v = X#w by (cases v, auto)
        then obtain X w where v: v = X#w by blast
        have is-first-X: is-first X (drop (length u) a)
          apply (rule-tac is-first-drop-length[where v=v and w=w and k=Suc i])
          apply (simp-all add: take-Suc v)
          apply (metis Suc-leI i leftmost-def)
          apply (insert length-u)
```

```
      apply arith
      done
    show ?thesis
      apply (rule exI[where x=drop (length u) a])
      by (simp add: a v is-first-cons is-first-X)
  next
    case False
    have Di: derivation-ge D i
    using leftmost-Derivation[OF i Suc(3), where j=i, simplified] .
    from split-derivation-leftmost[OF Di False]
    obtain E F r where D-split: D = E @ [(i, r)] @ F
      and E-Suc: derivation-ge E (Suc i) by auto
    let ?D = (derivation-shift E 1 (length (snd r)))@F
    from D-split
    have Derivation a ((i,r) # ?D) (u @ v)
     using Derivation-swap-single-mid-to-front E-Suc Suc.prems(1) lessI by blast
    then have ∃ y. Derives1 a i r y ∧ Derivation y ?D (u @ v) by simp
    then obtain y where ay:Derives1 a i r y
      and yuv: Derivation y ?D (u @ v) by blast
    have length-D′: length ?D = n using D-split Suc.hyps(2) by auto
    from Suc(1)[OF length-D′[symmetric] yuv Suc(4)]
    obtain w where leftderives y (u @ w) and (v = [] ⟶ w = [])
      and ∀X. is-first X v ⟶ is-first X w by blast
    then show ?thesis using ay i leftmost-Derives1-leftderives by blast
  qed
qed

lemma derives-implies-leftderives-gen: derives a (u@v) ⟹ is-word u ⟹ (∃ w.
      leftderives a (u@w) ∧
      (v = [] ⟶ w = []) ∧
      (∀ X. is-first X v ⟶ is-first X w))
using Derivation-implies-leftderives-gen derives-implies-Derivation by blast

lemma derives-implies-leftderives: derives a b ⟹ is-word b ⟹ leftderives a b
using derives-implies-leftderives-gen by force

fun LeftDerivation :: sentence ⇒ derivation ⇒ sentence ⇒ bool
where
  LeftDerivation a [] b = (a = b)
| LeftDerivation a (d#D) b = (∃ x. LeftDerives1 a (fst d) (snd d) x ∧ LeftDeriva-
tion x D b)

lemma LeftDerives1-implies-Derives1: LeftDerives1 a i r b ⟹ Derives1 a i r b
by (metis LeftDerives1-def)

lemma LeftDerivation-implies-Derivation:
  LeftDerivation a D b ⟹ Derivation a D b
proof (induct D arbitrary: a)
  case (Nil) thus ?case by simp
```

next
  case (*Cons d D*)
  thus *?case*
  using *CFG.LeftDerivation.simps*(*2*) *CFG-axioms Derivation.simps*(*2*)
    LeftDerives1-implies-Derives1 **by** *blast*
qed

lemma *LeftDerivation-implies-leftderives*: *LeftDerivation a D b* $\Longrightarrow$ *leftderives a b*
proof (*induct D arbitrary*: *a b*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons d D*)
    note *ihyps* = *this*
    from *ihyps* **have** ∃ *x*. *LeftDerives1 a* (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x D b* **by** *auto*
    then obtain *x* **where** *LeftDerives1 a* (*fst d*) (*snd d*) *x* **and** *xb*: *LeftDerivation x D b* **by** *blast*
    with *LeftDerives1-implies-leftderives1* **have** *d1*: *leftderives a x* **by** *auto*
    from *ihyps xb* **have** *d2*:*leftderives x b* **by** *simp*
    show *leftderives a b* **by** (*rule leftderives-trans*[*OF d1 d2*])
qed

lemma *leftmost-witness*[*simp*]: *leftmost* (*length x*) (*x*@(*N*#*y*)) = (*is-word x* ∧ *is-nonterminal N*)
  by (*auto simp add*: *leftmost-def*)

lemma *leftderives1-implies-LeftDerives1*:
  assumes *leftderives1*: *leftderives1 u v*
  shows ∃ *i r*. *LeftDerives1 u i r v*
proof −
  from *leftderives1* **have**
    ∃ *x y N* α. *u* = *x* @ [*N*] @ *y* ∧ *v* = *x* @ α @ *y* ∧ *is-word x* ∧ *is-sentence y* ∧ (*N*, α) ∈ $\mathfrak{R}$
    by (*simp add*: *leftderives1-def*)
  then obtain *x y N* α **where**
    u:*u* = *x* @ [*N*] @ *y* **and**
    v:*v* = *x* @ α @ *y* **and**
    x:*is-word x* **and**
    y:*is-sentence y* **and**
    r:(*N*, α) ∈ $\mathfrak{R}$
    by *blast*
  show *?thesis*
    apply (*rule-tac x*=*length x* **in** *exI*)
    apply (*rule-tac x*=(*N*, α) **in** *exI*)
    apply (*auto simp add*: *LeftDerives1-def*)
    apply (*simp add*: *leftmost-def x u rule-nonterminal-type*[*OF r*])
    apply (*simp add*: *Derives1-def u v*)
    apply (*rule-tac x*=*x* **in** *exI*)
    apply (*rule-tac x*=*y* **in** *exI*)

apply (*auto simp add*: *x y r*)
done
qed

lemma *LeftDerivation-LeftDerives1*:
LeftDerivation a S y ⟹ *LeftDerives1 y i r z* ⟹ *LeftDerivation a* (*S*@[(*i*,*r*)]) *z*
proof (*induct S arbitrary*: *a y z i r*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons s S*) **thus** *?case*
by (*metis LeftDerivation.simps*(*2*) *append-Cons*)
qed

lemma *leftderives-implies-LeftDerivation*: *leftderives a b* ⟹ ∃ *D*. *LeftDerivation a D b*
proof (*induct rule*: *leftderives-induct*)
case *Base*
show *?case* **by** (*rule exI*[**where** *x*=[]], *simp*)
next
case (*Step y z*)
note *ihyps* = *this*
from *ihyps* **obtain** *D* **where** *ay*: *LeftDerivation a D y* **by** *blast*
from *ihyps leftderives1-implies-LeftDerives1* **obtain** *i r* **where** *yz*: *LeftDerives1 y i r z* **by** *blast*
from *LeftDerivation-LeftDerives1*[*OF ay yz*] **show** *?case* **by** *auto*
qed

lemma *LeftDerivation-append*:
LeftDerivation a (*D*@*E*) *c* = (∃ *b*. *LeftDerivation a D b* ∧ *LeftDerivation b E c*)
proof(*induct D arbitrary*: *a c E*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons d D*) **thus** *?case* **by** *auto*
qed

lemma *LeftDerivation-implies-append*:
LeftDerivation a D b ⟹ *LeftDerivation b E c* ⟹ *LeftDerivation a* (*D*@*E*) *c*
using *LeftDerivation-append* **by** *blast*

lemma *Derivation-unique-dest*: *Derivation a D b* ⟹ *Derivation a D c* ⟹ *b* = *c*
apply (*induct D arbitrary*: *a b c*)
apply *auto*
using *Derives1-unique-dest* **by** *blast*

lemma *Derivation-unique-src*: *Derivation a D c* ⟹ *Derivation b D c* ⟹ *a* = *b*
apply (*induct D arbitrary*: *a b c*)
apply *auto*
using *Derives1-unique-src* **by** *blast*

lemma *LeftDerives1-unique*: *LeftDerives1 a i r b* $\Longrightarrow$ *LeftDerives1 a j s b* $\Longrightarrow$ $i = j \wedge r = s$
using *Derives1-def LeftDerives1-def leftmost-unique* **by** *auto*

lemma *leftlang*: $\mathcal{L} = \{ v \mid v.$ *is-word v* $\wedge$ *is-leftderivation v* $\}$
by (*metis* (*no-types*, *lifting*) *CFG.is-derivation-def CFG.is-leftderivation-def CFG.leftderivation-implies-derivation CFG-axioms Collect-cong* $\mathcal{L}$*-def derives-implies-leftderives*)

lemma *leftprefixlang*: $\mathcal{L}_P = \{ u \mid u\ v.$ *is-word u* $\wedge$ *is-leftderivation* $(u@v)$ $\}$
apply (*auto simp add*: $\mathcal{L}_P$*-def*)
using *derives-implies-leftderives-gen is-derivation-def is-leftderivation-def* **apply** *blast*
using *leftderivation-implies-derivation* **by** *blast*

lemma *derives-implies-leftderives-cons*:
is-word a $\Longrightarrow$ *derives u* $(a@X\#b)$ $\Longrightarrow \exists$ *c. leftderives u* $(a@X\#c)$
by (*metis append-Cons append-Nil derives-implies-leftderives-gen is-first-def*)

lemma *is-word-append*[*simp*]: *is-word* $(a@b)$ = (*is-word a* $\wedge$ *is-word b*)
by (*auto simp add*: *is-word-def*)

lemma $\mathcal{L}_P$*-split*: $a@b \in \mathcal{L}_P \Longrightarrow a \in \mathcal{L}_P$
by (*auto simp add*: $\mathcal{L}_P$*-def*)

lemma $\mathcal{L}_P$*-is-word*: $a \in \mathcal{L}_P \Longrightarrow$ *is-word a*
by (*metis* (*no-types*, *lifting*) *leftprefixlang mem-Collect-eq*)

definition *Derive* :: *sentence* $\Rightarrow$ *derivation* $\Rightarrow$ *sentence*
where
Derive a D = (*THE b. Derivation a D b*)

lemma *Derivation-dest-ex-unique*: *Derivation a D b* $\Longrightarrow \exists!$ *x. Derivation a D x*
using *CFG.Derivation-unique-dest CFG-axioms* **by** *blast*

lemma *Derive*:
assumes *ab*: *Derivation a D b*
shows *Derive a D* = *b*
proof −
note *the1-equality*[*OF Derivation-dest-ex-unique*[*OF ab*] *ab*]
thus *?thesis* **by** (*simp add*: *Derive-def*)
qed

end

end
theory *Validity*
imports *LLEarleyParsing Derivations*
begin

context *LocalLexing* **begin**

definition *wellformed-token* :: *token* ⇒ *bool*
where
wellformed-token t = *is-terminal* (*terminal-of-token t*)

definition *wellformed-tokens* :: *tokens* ⇒ *bool*
where
wellformed-tokens ts = *list-all wellformed-token ts*

definition *doc-tokens* :: *tokens* ⇒ *bool*
where
doc-tokens p = (*wellformed-tokens p* ∧ *is-prefix* (*chars p*) *Doc*)

definition *wellformed-item* :: *item* ⇒ *bool*
where
wellformed-item x = (
item-rule x ∈ $\mathfrak{R}$ ∧
item-origin x ≤ *item-end x* ∧
item-end x ≤ *length Doc* ∧
item-dot x ≤ *length* (*item-rhs x*))

definition *wellformed-items* :: *items* ⇒ *bool*
where
wellformed-items X = (∀ *x* ∈ *X*. *wellformed-item x*)

lemma *is-word-terminals*: *wellformed-tokens p* ⟹ *is-word* (*terminals p*)
by (*simp add*: *is-word-def list-all-length terminals-def wellformed-token-def wellformed-tokens-def*)

lemma *is-word-subset*: *is-word x* ⟹ *set y* ⊆ *set x* ⟹ *is-word y*
by (*metis* (*mono-tags*, *opaque-lifting*) *in-set-conv-nth is-word-def list-all-length subsetCE*)

lemma *is-word-terminals-take*: *wellformed-tokens p* ⟹ *is-word*(*terminals* (*take n p*))
by (*metis append-take-drop-id is-word-terminals list-all-append wellformed-tokens-def*)

lemma *is-word-terminals-drop*: *wellformed-tokens p* ⟹ *is-word*(*terminals* (*drop n p*))
by (*metis append-take-drop-id is-word-terminals list-all-append wellformed-tokens-def*)

definition *pvalid* :: *tokens* ⇒ *item* ⇒ *bool*
where
pvalid p x = (∃ *u* γ.
wellformed-tokens p ∧
wellformed-item x ∧
u ≤ *length p* ∧

charslength p = *item-end x* ∧
charslength (*take u p*) = *item-origin x* ∧
is-derivation (*terminals* (*take u p*) @ [*item-nonterminal x*] @ γ) ∧
derives (*item-α x*) (*terminals* (*drop u p*)))

definition *Gen* :: *tokens set* ⇒ *items*
where
Gen P = { *x* | *x p*. *p* ∈ *P* ∧ *pvalid p x* }

lemma *wellformed-items* (*Gen P*)
by (*auto simp add*: *Gen-def pvalid-def wellformed-items-def*)

lemma *wellformed-items* (*Init*)
by (*auto simp add*: *wellformed-items-def Init-def init-item-def wellformed-item-def*)

definition *pvalid-left* :: *tokens* ⇒ *item* ⇒ *bool*
where
pvalid-left p x = (∃ *u* γ.
wellformed-tokens p ∧
wellformed-item x ∧
u ≤ *length p* ∧
charslength p = *item-end x* ∧
charslength (*take u p*) = *item-origin x* ∧
is-leftderivation (*terminals* (*take u p*) @ [*item-nonterminal x*] @ γ) ∧
leftderives (*item-α x*) (*terminals* (*drop u p*)))

lemma *pvalid-left*: *pvalid p x* = *pvalid-left p x*
proof −
have *right-imp-left*: *pvalid-left p x* ⟹ *pvalid p x*
by (*meson CFG.leftderives-implies-derives CFG-axioms LocalLexing.pvalid-def LocalLexing.pvalid-left-def LocalLexing-axioms leftderivation-implies-derivation*)
have *left-imp-right*: *pvalid p x* ⟹ *pvalid-left p x*
proof −
assume *pvalid p x*
then obtain *u* γ **where**
wellformed-tokens p ∧
wellformed-item x ∧
u ≤ *length p* ∧
charslength p = *item-end x* ∧
charslength (*take u p*) = *item-origin x* ∧
is-derivation (*terminals* (*take u p*) @ [*item-nonterminal x*] @ γ) ∧
derives (*item-α x*) (*terminals* (*drop u p*)) **by** (*simp add*: *pvalid-def*, *blast*)
thus *?thesis*
apply (*auto simp add*: *pvalid-left-def*)
apply (*rule-tac x*=*u* **in** *exI*)
apply *auto*
apply (*simp add*: *is-leftderivation-def*)
apply (*rule-tac derives-implies-leftderives-cons*[**where** *b*=γ])
apply (*erule is-word-terminals-take*)

apply (*simp add*: *is-derivation-def*)
by (*metis derives-implies-leftderives is-word-terminals-drop*)
qed
show *?thesis* **by** (*metis left-imp-right right-imp-left*)
qed

lemma $\mathcal{L}_P$*-wellformed-tokens*: *terminals p* $\in \mathcal{L}_P \Longrightarrow$ *wellformed-tokens p*
by (*metis* (*mono-tags*, *lifting*) $\mathcal{L}_P$*-is-word is-word-def length-map list-all-length nth-map terminals-def wellformed-token-def wellformed-tokens-def*)

end

end
theory *TheoremD2*
imports *LocalLexingLemmas Validity Derivations*
begin

context *LocalLexing* **begin**

definition *splits-at* :: *sentence* $\Rightarrow$ *nat* $\Rightarrow$ *sentence* $\Rightarrow$ *symbol* $\Rightarrow$ *sentence* $\Rightarrow$ *bool*
where
splits-at δ *i* α *N* β = (*i* < *length* $\delta \wedge \alpha$ = *take i* $\delta \wedge N = \delta$! *i* $\wedge \beta$ = *drop* (*Suc i*) δ)

lemma *splits-at-combine*: *splits-at* δ *i* α *N* $\beta \Longrightarrow \delta = \alpha$ @ [*N*] @ β
by (*simp add*: *id-take-nth-drop splits-at-def*)

lemma *splits-at-combine-dest*: *Derives1 a i r b* $\Longrightarrow$ *splits-at a i* α *N* $\beta \Longrightarrow b = \alpha$ @ (*snd r*) @ β
by (*metis* (*no-types*, *lifting*) *Derives1-drop Derives1-split append-assoc append-eq-conv-conj*

length-append splits-at-def)

lemma *Derives1-nonterminal*:
assumes *Derives1 a i r b*
assumes *splits-at a i* α *N* β
shows *fst r* = *N* $\wedge$ *is-nonterminal N*
proof −
from *assms* **have** *fst*: *fst r* = *N*
by (*metis Derives1-split append-Cons nth-append-length splits-at-def*)
then have *is-nonterminal N*
by (*metis Derives1-def assms*(*1*) *prod.collapse rule-nonterminal-type*)
with *fst* **show** *?thesis* **by** *auto*
qed

lemma *splits-at-ex*: *Derives1* δ *i r s* $\Longrightarrow \exists$ α *N* β. *splits-at* δ *i* α *N* β
by (*simp add*: *Derives1-bound splits-at-def*)

lemma *splits-at-α*: *Derives1 δ i r s* ⟹ *splits-at δ i α N β* ⟹
α = take i δ ∧ α = take i s ∧ length α = i
by (*metis Derives1-split append-eq-conv-conj splits-at-def*)

lemma *LeftDerives1-splits-at-is-word*: *LeftDerives1 δ i r s* ⟹ *splits-at δ i α N β* ⟹ *is-word α*
by (*metis LeftDerives1-def leftmost-def splits-at-def*)

lemma *splits-at-β*: *Derives1 δ i r s* ⟹ *splits-at δ i α N β* ⟹
β = drop (Suc i) δ ∧ β = drop (i + length (snd r)) s ∧ length β = length δ − i − 1
by (*metis Derives1-drop Suc-eq-plus1 diff-diff-left length-drop splits-at-def*)

lemma *Derives1-prefix*:
assumes *ab*: *Derives1 δ i r (a@b)*
assumes *split*: *splits-at δ i α N β*
shows *is-prefix α a ∨ is-prefix a α*
proof −
have *take*: *α = take i (a@b)* **using** *ab split splits-at-α* **by** *blast*
show *?thesis*
proof (*cases i ≤ length a*)
case *True*
then have *α = take i a* **by** (*simp add*: *take*)
then have *is-prefix α a*
by (*metis append-take-drop-id is-prefix-def*)
with *True* **show** *?thesis* **by** *auto*
next
case *False*
then have *is-prefix a α*
by (*simp add*: *is-prefix-def nat-le-linear take*)
with *False* **show** *?thesis* **by** *auto*
qed
qed

lemma *Derives1-suffix*:
assumes *ab*: *Derives1 δ i r (a@b)*
assumes *split*: *splits-at δ i α N β*
shows *is-suffix β b ∨ is-suffix b β*
proof −
have *drop1*: *β = drop (i + length (snd r)) (a@b)* **using** *ab split splits-at-β* **by** *blast*
have *drop2*: *b = drop (length a) (a@b)* **by** *simp*
show *?thesis*
proof (*cases (i + length (snd r)) ≤ length a*)
case *True*
with *drop1 drop2* **have** *is-suffix b β* **by** (*simp add*: *is-suffix-def*)
then show *?thesis* **by** *auto*
next
case *False*

then have *length a* $\leq$ (*i* + *length* (*snd r*)) **by** *arith*
with *drop1 drop2* **have** *is-suffix* β *b*
by (*metis append-Nil append-take-drop-id drop-append drop-eq-Nil is-suffix-def*)
then show *?thesis* **by** *auto*
qed
qed

lemma *Derives1-skip-prefix*:
length a $\leq$ *i* $\Longrightarrow$ *Derives1* (*a*@*b*) *i r* (*a*@*c*) $\Longrightarrow$ *Derives1 b* (*i* − *length a*) *r c*
apply (*auto simp add*: *Derives1-def*)
by (*metis append-eq-append-conv-if is-sentence-concat is-sentence-cons is-symbol-def*

length-drop rule-nonterminal-type)

lemma *cancel-suffix*:
assumes *a* @ *c* = *b* @ *d*
assumes *length c* $\leq$ *length d*
shows *a* = *b* @ (*take* (*length d* − *length c*) *d*)
proof −
have *a* @ *c* = (*b* @ *take* (*length d* − *length c*) *d*) @ *drop* (*length d* − *length c*) *d*
by (*metis append-assoc append-take-drop-id assms*(*1*))
then show *?thesis*
by (*metis append-eq-append-conv assms*(*2*) *diff-diff-cancel length-drop*)
qed

lemma *is-sentence-take*:
is-sentence y $\Longrightarrow$ *is-sentence* (*take n y*)
by (*metis append-take-drop-id is-sentence-concat*)

lemma *Derives1-skip-suffix*:
assumes *i*: *i* < *length a*
assumes *D*: *Derives1* (*a*@*c*) *i r* (*b*@*c*)
shows *Derives1 a i r b*
proof −
note *Derives1-def*[**where** *u*=*a*@*c* **and** *v*=*b*@*c* **and** *i*=*i* **and** *r*=*r*]
then have $\exists$ *x y N* α.
a @ *c* = *x* @ [*N*] @ *y* $\wedge$
b @ *c* = *x* @ α @ *y* $\wedge$ *is-sentence x* $\wedge$ *is-sentence y* $\wedge$ (*N*, α) $\in \mathfrak{R}$ $\wedge$ *r* = (*N*,
α) $\wedge$ *i* = *length x*
using *D* **by** *blast*
then obtain *x y N* α **where** *split*:
a @ *c* = *x* @ [*N*] @ *y* $\wedge$
b @ *c* = *x* @ α @ *y* $\wedge$ *is-sentence x* $\wedge$ *is-sentence y* $\wedge$ (*N*, α) $\in \mathfrak{R}$ $\wedge$ *r* = (*N*,
α) $\wedge$ *i* = *length x*
by *blast*
from *split* **have** *length* (*a*@*c*) = *length* (*x* @ [*N*] @ *y*) **by** *auto*
then have *length a* + *length c* = *length x* + *length y* + *1* **by** *simp*
with *split* **have** *length a* + *length c* = *i* + *length y* + *1* **by** *simp*
with *i* **have** *len-c-y*: *length c* $\leq$ *length y* **by** *arith*

let *?y* = *take* (*length y* − *length c*) *y*
from *split* **have** *ac*: *a* @ *c* = (*x* @ [*N*]) @ *y* **by** *auto*
note *cancel-suffix*[**where** *a*=*a* **and** *c* = *c* **and** *b* = *x*@[*N*] **and** *d* = *y*, *OF ac len-c-y*]
then have *a*: *a* = *x* @ [*N*] @ *?y* **by** *auto*
from *split* **have** *bc*: *b* @ *c* = (*x* @ α) @ *y* **by** *auto*
note *cancel-suffix*[**where** *a*=*b* **and** *c* = *c* **and** *b* = *x*@α **and** *d* = *y*, *OF bc len-c-y*]
then have *b*: *b* = *x* @ α @ *?y* **by** *auto*
from *split len-c-y a b* **show** *?thesis*
  apply (*simp only*: *Derives1-def*)
  apply (*rule-tac x*=*x* **in** *exI*)
  apply (*rule-tac x*=*?y* **in** *exI*)
  apply (*rule-tac x*=*N* **in** *exI*)
  apply (*rule-tac x*=α **in** *exI*)
  apply *auto*
  by (*rule is-sentence-take*)
qed

lemma *drop-cancel-suffix*: *a*@*c* = *drop n* (*b*@*c*) ⟹ *a* = *drop n b*
proof −
  assume *a1*: *a* @ *c* = *drop n* (*b* @ *c*)
  have *length* (*drop n b*) = *length b* + *length c* − *n* − *length c*
    by (*metis add-diff-cancel-right′ diff-commute length-drop*)
  then show *?thesis*
    using *a1* **by** (*metis add-diff-cancel-right′ append-eq-append-conv drop-append length-append length-drop*)
qed

lemma *drop-keep-last*: *u* ≠ [] ⟹ *u* = *drop n* (*a*@[*X*]) ⟹ *u* = *drop n a* @ [*X*]
by (*metis append-take-drop-id drop-butlast last-appendR snoc-eq-iff-butlast*)

lemma *Derives1-X-is-part-of-rule*[*consumes 2*, *case-names Suffix Prefix*]:
  assumes *aXb*: *Derives1* δ *i r* (*a*@[*X*]@*b*)
  assumes *split*: *splits-at* δ *i* α *N* β
  assumes *prefix*: ⋀ β. δ = *a* @ [*X*] @ β ⟹ *length a* < *i* ⟹
        Derives1 β (*i* − *length a* − *1*) *r b* ⟹ *False*
  assumes *suffix*: ⋀ α. δ = α @ [*X*] @ *b* ⟹ *Derives1* α *i r a* ⟹ *False*
  shows ∃ *u v*. *a* = α @ *u* ∧ *b* = *v* @ β ∧ (*snd r*) = *u*@[*X*]@*v*
proof −
  have *prefix-or*: *is-prefix* α *a* ∨ *is-proper-prefix a* α
    by (*metis Derives1-prefix split aXb is-prefix-eq-proper-prefix*)
  have *is-proper-prefix a* α ⟹ *False*
  proof −
    assume *proper*:*is-proper-prefix a* α
    then have ∃ *u*. *u* ≠ [] ∧ α = *a*@*u* **by** (*metis is-proper-prefix-def*)
    then obtain *u* **where** *u*: *u* ≠ [] ∧ α = *a*@*u* **by** *blast*
    note *splits-at* = *splits-at-*α[*OF aXb split*] *splits-at-combine*[*OF split*]
    from *splits-at* **have** α*1*: α = *take i* δ **by** *blast*

from *splits-at* **have** $\alpha 2$: $\alpha = take\ i\ (a@[X]@b)$ **by** *blast*
from *splits-at* **have** *len*α: $length\ \alpha = i$ **by** *blast*
with *proper* **have** *lena*: $length\ a < i$
using *append-eq-conv-conj drop-eq-Nil leI u* **by** *auto*
from $u\ \alpha 2$ **have** $a@u = take\ i\ (a@[X]@b)$ **by** *auto*
with *lena* **have** $u = take\ (i - length\ a)\ ([X]@b)$ **by** (*simp add*: *less-or-eq-imp-le*)

with *lena* **have** *uX*: $u = [X]@(take\ (i - length\ a - 1)\ b)$ **by** (*simp add*: *not-less take-Cons'*)
let $?\beta = (take\ (i - length\ a - 1)\ b)\ @\ [N]\ @\ \beta$
from *splits-at* **have** *f1*: $\delta = \alpha\ @\ [N]\ @\ \beta$ **by** *blast*
with *u uX* **have** *f2*: $\delta = a\ @\ [X]\ @\ ?\beta$ **by** *simp*
note *skip* = *Derives1-skip-prefix*[**where** $a = a\ @\ [X]$ **and** $b = ?\beta$ **and** $r = r$ **and** $i = i$ **and** $c = b$]
then have *D*: *Derives1* $?\beta\ (i - length\ a - 1)\ r\ b$
using *One-nat-def Suc-leI aXb append-assoc diff-diff-left f2 lena length-Cons length-append length-append-singleton list.size*(*3*) **by** *fastforce*
note *prefix*[*OF f2 lena D*]
then show *False* **.**
qed
with *prefix-or* **have** *is-prefix*: *is-prefix* α *a* **by** *blast*

from *aXb* **have** *aXb'*: *Derives1* $\delta\ i\ r\ ((a@[X])@b)$ **by** *auto*
note *Derives1-suffix*[*OF aXb' split*]
then have *suffix-or*: *is-suffix* β *b* $\vee$ *is-proper-suffix b* β
by (*metis is-suffix-eq-proper-suffix*)
have *is-proper-suffix b* $\beta \Longrightarrow$ *False*
proof −
assume *proper*: *is-proper-suffix b* β
then have $\exists\ u.\ u \neq [] \wedge \beta = u@b$ **by** (*metis is-proper-suffix-def*)
then obtain *u* **where** *u*: $u \neq [] \wedge \beta = u@b$ **by** *blast*
note *splits-at* = *splits-at-*β[*OF aXb split*] *splits-at-combine*[*OF split*]
from *splits-at* **have** $\beta 1$: $\beta = drop\ (Suc\ i)\ \delta$ **by** *blast*
from *splits-at* **have** $\beta 2$: $\beta = drop\ (i + length\ (snd\ r))\ (a\ @\ [X]\ @\ b)$ **by** *blast*
from *splits-at* **have** *len*β: $length\ \beta = length\ \delta - i - 1$ **by** *blast*
with *proper* **have** *lenb*: $length\ b < length\ \beta$ **by** (*metis is-proper-suffix-length-cmp*)

from $u\ \beta 2$ **have** $u@b = drop\ (i + length\ (snd\ r))\ ((a\ @\ [X])\ @\ b)$ **by** *auto*
hence $u = drop\ (i + length\ (snd\ r))\ (a\ @\ [X])$
by (*metis drop-cancel-suffix*)
hence *uX*: $u = drop\ (i + length\ (snd\ r))\ a\ @\ [X]$ **by** (*metis drop-keep-last u*)
let $?\alpha = \alpha\ @\ [N]\ @\ (drop\ (i + length\ (snd\ r))\ a)$
from *splits-at* **have** *f1*: $\delta = \alpha\ @\ [N]\ @\ \beta$ **by** *blast*
with *u uX* **have** *f2*: $\delta = ?\alpha\ @\ [X]\ @\ b$ **by** *simp*
note *skip* = *Derives1-skip-suffix*[**where** $a = ?\alpha$ **and** $c = [X]@b$ **and** $b{=}a$ **and** $r = r$ **and** $i = i$]
have *f3*: $i < length\ (\alpha\ @\ [N]\ @\ drop\ (i + length\ (snd\ r))\ a)$
proof −
have *f1*: $1 + i + length\ b = length\ [X] + length\ b + i$

by (*metis Groups.add-ac*(*2*) *Suc-eq-plus1-left length-Cons list.size*(*3*) *list.size*(*4*) *semiring-normalization-rules*(*22*))
have *f2*: *length* δ − *i* − *1* = *length* ((α @ [*N*] @ *drop* (*i* + *length* (*snd r*)) *a*) @ [*X*] @ *b*) − *Suc i*
by (*metis f2 length-drop splits-at*(*1*))
have *length* ([]::*symbol list*) ≠ *length* δ − *i* − *1* − *length b*
by (*metis add-diff-cancel-right′ append-Nil2 append-eq-append-conv lenβ length-append u*)
then have *length* ([]::*symbol list*) ≠ *length* α + *length* ([*N*] @ *drop* (*i* + *length* (*snd r*)) *a*) − *i*
using *f2 f1* **by** (*metis Suc-eq-plus1-left add-diff-cancel-right′ diff-diff-left length-append*)
then show *?thesis*
by *auto*
qed
from *aXb f2* **have** *D*: *Derives1* (*?α* @ [*X*] @ *b*) *i r* (*a*@[*X*]@*b*) **by** *auto*
note *skip*[*OF f3 D*]
note *suffix*[*OF f2 skip*[*OF f3 D*]]
then show *False* **.**
qed
with *suffix-or* **have** *is-suffix*: *is-suffix* β *b* **by** *blast*

from *is-prefix* **have** ∃ *u*. *a* = α @ *u* **by** (*auto simp add*: *is-prefix-def*)
then obtain *u* **where** *u*: *a* = α @ *u* **by** *blast*
from *is-suffix* **have** ∃ *v*. *b* = *v* @ β **by** (*auto simp add*: *is-suffix-def*)
then obtain *v* **where** *v*: *b* = *v* @ β **by** *blast*

from *u v splits-at-combine*[*OF split*] *aXb* **have** *D*:*Derives1* (α@[*N*]@β) *i r* (α@(*u*@[*X*]@*v*)@β)
by *simp*
from *splits-at-α*[*OF aXb split*] **have** *i*: *length* α = *i* **by** *blast*
from *i* **have** *i1*: *length* α ≤ *i* **and** *i2*: *i* ≤ *length* α **by** *auto*
note *Derives1-skip-suffix*[*OF - Derives1-skip-prefix*[*OF i1 D*], *simplified*, *OF i2*]
then have *Derives1* [*N*] *0 r* (*u* @ [*X*] @ *v*) **by** *auto*
then have *r*: *snd r* = *u* @ [*X*] @ *v*
by (*metis Derives1-split append-Cons append-Nil length-0-conv list.inject self-append-conv*)

show *?thesis* **using** *u v r* **by** *auto*
qed

lemma $\mathcal{L}_P$*-derives*: *a* ∈ $\mathcal{L}_P$ ⟹ ∃ *b*. *derives* [$\mathfrak{S}$] (*a*@*b*)
by (*simp add*: $\mathcal{L}_P$*-def is-derivation-def*)

lemma $\mathcal{L}_P$*-leftderives*: *a* ∈ $\mathcal{L}_P$ ⟹ ∃ *b*. *leftderives* [$\mathfrak{S}$] (*a*@*b*)
by (*metis* $\mathcal{L}_P$*-derives* $\mathcal{L}_P$*-is-word derives-implies-leftderives-gen*)

lemma *Derives1-rule*: *Derives1 a i r b* ⟹ *r* ∈ $\mathfrak{R}$
by (*auto simp add*: *Derives1-def*)

```
lemma is-prefix-empty[simp]: is-prefix [] a
  by (simp add: is-prefix-def)

lemma is-prefix-cons: is-prefix (x # a) b = (∃ c. b = x # c ∧ is-prefix a c)
  by (metis append-Cons is-prefix-def)

lemma is-prefix-cancel[simp]: is-prefix (a@b) (a@c) = is-prefix b c
  by (metis append-assoc is-prefix-def same-append-eq)

lemma is-prefix-chars: is-prefix a b ⟹ is-prefix (chars a) (chars b)
proof (induct a arbitrary: b)
  case Nil thus ?case by simp
next
  case (Cons x a)
  from Cons(2) have ∃ c. b = x # c ∧ is-prefix a c
    by (simp add: is-prefix-cons)
  then obtain c where c: b = x # c ∧ is-prefix a c by blast
  from c Cons(1) show ?case by simp
qed

lemma is-prefix-length: is-prefix a b ⟹ length a ≤ length b
by (auto simp add: is-prefix-def)

lemma is-prefix-take[simp]: is-prefix (take n a) a
apply (auto simp add: is-prefix-def)
apply (rule-tac x=drop n a in exI)
by simp

lemma doc-tokens-length: doc-tokens p ⟹ length (chars p) ≤ length Doc
by (metis doc-tokens-def is-prefix-length)

fun count-terminals :: sentence ⇒ nat where
  count-terminals [] = 0
| count-terminals (x#xs) = (if (is-terminal x) then Suc (count-terminals xs) else
(count-terminals xs))

lemma count-terminals-upper-bound: count-terminals p ≤ length p
  by (induct p, auto)

lemma count-terminals-append[simp]: count-terminals (a@b) = count-terminals a
+ count-terminals b
  by (induct a arbitrary: b, auto)

lemma Derives1-count-terminals:
  assumes D: Derives1 a i r b
  shows count-terminals b = count-terminals a + count-terminals (snd r)
proof −
  have ∃ α N β. splits-at a i α N β using D splits-at-ex by simp
  then obtain α N β where split: splits-at a i α N β by blast
```

```
  from D split have N: is-nonterminal N by (simp add: Derives1-nonterminal)
  have a: a = α @ [N] @ β by (metis split splits-at-combine)
  from D split have b: b = α @ (snd r) @ β using splits-at-combine-dest by simp
  show ?thesis
    apply (simp add: a b)
    using N by (metis is-terminal-nonterminal)
qed

lemma Derives1-count-terminals-leq:
  assumes D: Derives1 a i r b
  shows count-terminals a ≤ count-terminals b
by (metis Derives1-count-terminals assms le-less-linear not-add-less1)

lemma Derivation-count-terminals-leq:
  Derivation a E b ⟹ count-terminals a ≤ count-terminals b
proof (induct E arbitrary: a)
  case Nil thus ?case by auto
next
  case (Cons e E)
  then have ∃ x i r. Derives1 a i r x ∧ Derivation x E b using Derivation.simps(2)
by blast
  then obtain x i r where axb: Derives1 a i r x ∧ Derivation x E b by blast
  from axb have ax: count-terminals a ≤ count-terminals x
    using Derives1-count-terminals-leq by blast
  from axb have xb: count-terminals x ≤ count-terminals b using Cons by simp
  show ?case using ax xb by arith
qed

lemma derives-count-terminals-leq: derives a b ⟹ count-terminals a ≤ count-terminals
b
using Derivation-count-terminals-leq derives-implies-Derivation by force

lemma is-word-cons[simp]: is-word (x#xs) = (is-terminal x ∧ is-word xs)
  by (simp add: is-word-def)

lemma count-terminals-of-word: is-word w ⟹ count-terminals w = length w
  by (induct w, auto)

lemma length-terminals[simp]: length (terminals p) = length p
  by (auto simp add: terminals-def)

lemma path-length-is-upper-bound:
  assumes p: wellformed-tokens p
  assumes α: is-word α
  assumes derives: derives (α@u) (terminals p)
  shows length α ≤ length p
proof −
  have counts: count-terminals α ≤ count-terminals (terminals p)
    using derives derives-count-terminals-leq by fastforce
```

have *len1*: *length* α = *count-terminals* α **by** (*simp add*: α *count-terminals-of-word*)
have *len2*: *length* (*terminals p*) = *count-terminals* (*terminals p*)
by (*simp add*: *count-terminals-of-word is-word-terminals p*)
show *?thesis* **using** *counts len1 len2* **by** *auto*
qed

lemma *is-word-Derives1-index*:
assumes *w*: *is-word w*
assumes *derives1*: *Derives1* (*w@a*) *i r b*
shows $i \geq$ *length w*
proof −
from *derives1* **have** *n*: *is-nonterminal* ((*w@a*) ! *i*)
using *Derives1-nonterminal splits-at-def splits-at-ex* **by** *auto*
from *w* **have** *t*: $i <$ *length w* $\Longrightarrow$ *is-terminal* ((*w@a*) ! *i*)
by (*simp add*: *is-word-is-terminal nth-append*)
show *?thesis*
by (*metis t n is-terminal-nonterminal less-le-not-le nat-le-linear*)
qed

lemma *is-word-Derivation-derivation-ge*:
assumes *w*: *is-word w*
assumes *D*: *Derivation* (*w@a*) *D b*
shows *derivation-ge D* (*length w*)
by (*metis D Derivation-leftmost derivation-ge-empty leftmost-Derivation leftmost-append w*)

lemma *derives-word-is-prefix*:
assumes *w*: *is-word w*
assumes *derives*: *derives* (*w@a*) *b*
shows *is-prefix w b*
by (*metis Derivation-take append-eq-conv-conj derives derives-implies-Derivation is-prefix-take is-word-Derivation-derivation-ge w*)

lemma *terminals-take*[*simp*]: *terminals* (*take n p*) = *take n* (*terminals p*)
by (*simp add*: *take-map terminals-def*)

lemma *terminals-drop*[*simp*]: *terminals* (*drop n p*) = *drop n* (*terminals p*)
by (*simp add*: *drop-map terminals-def*)

lemma *take-prefix*[*simp*]: *is-prefix a b* $\Longrightarrow$ *take* (*length a*) *b* = *a*
by (*metis append-eq-conv-conj is-prefix-unsplit*)

lemma *Derives1-drop-prefixword*:
assumes *w*: *is-word w*
assumes *wa-b*: *Derives1* (*w@a*) *i r b*
shows *Derives1 a* ($i -$ *length w*) *r* (*drop* (*length w*) *b*)
proof −
have *i*: *length w* $\leq i$ **using** *wa-b is-word-Derives1-index w* **by** *blast*
have *is-prefix w b* **by** (*metis append-eq-conv-conj i is-prefix-take le-Derives1-take*

wa-b)
  then have *b*: *b* = *w* @ (*drop* (*length* *w*) *b*) **by** (*simp add*: *is-prefix-unsplit*)
  show *?thesis*
    apply (*rule-tac Derives1-skip-prefix*[*OF i*])
    by (*simp add*: *b*[*symmetric*] *wa-b*)
qed

lemma *derives1-drop-prefixword*:
  assumes *w*: *is-word w*
  assumes *wa-b*: *derives1* (*w*@*a*) *b*
  shows *derives1 a* (*drop* (*length w*) *b*)
by (*metis Derives1-drop-prefixword Derives1-implies-derives1 derives1-implies-Derives1 w wa-b*)

lemma *derives1-is-word-is-prefix-drop*:
  assumes *w*: *is-word w*
  assumes *w-a*: *is-prefix w a*
  assumes *ab*: *derives1 a b*
  shows *derives1* (*drop* (*length w*) *a*) (*drop* (*length w*) *b*)
by (*metis ab append-take-drop-id derives1-drop-prefixword take-prefix w w-a*)

lemma *derives-drop-prefixword-helper*:
  derives a b $\Longrightarrow$ *is-word w* $\Longrightarrow$ *is-prefix w a* $\Longrightarrow$ *derives* (*drop* (*length w*) *a*) (*drop* (*length w*) *b*)
proof (*induct rule*: *derives-induct*)
  case *Base* **thus** *?case* **by** *auto*
next
  case (*Step y z*)
  have *is-prefix-w-y*: *is-prefix w y*
      by (*metis Step.hyps*(*1*) *Step.prems*(*1*) *Step.prems*(*2*) *derives-word-is-prefix is-prefix-def*)
  thus *?case*
   by (*metis Step.hyps*(*2*) *Step.hyps*(*3*) *Step.prems*(*1*) *Step.prems*(*2*) *derives1-implies-derives*

        derives1-is-word-is-prefix-drop derives-trans)
qed

lemma *derive-drop-prefixword*:
  is-word w $\Longrightarrow$ *derives* (*w*@*a*) *b* $\Longrightarrow$ *derives a* (*drop* (*length w*) *b*)
by (*metis append-eq-conv-conj derives-drop-prefixword-helper is-prefix-take*)

lemma *thmD2′*:
  assumes *X*: *is-terminal X*
  assumes *p*: *doc-tokens p*
  assumes *pX*: (*terminals p*)@[*X*] $\in \mathcal{L}_P$
  shows $\exists$ *x*. *pvalid p x* $\wedge$ *next-symbol x* = *Some X*
proof −
  from *p* **have** *wellformed-p*: *wellformed-tokens p* **by** (*simp add*: *doc-tokens-def*)
  have $\exists$ ω. *leftderives* [$\mathfrak{S}$] (((*terminals p*)@[*X*]) @ ω) **using** $\mathcal{L}_P$*-leftderives pX* **by**

blast
then obtain *ω* **where** *leftderives* [𝔖] (((*terminals p*)@[*X*]) @ *ω*) **by** *blast*
then have ∃ *D*. *LeftDerivation* [𝔖] *D* (((*terminals p*)@[*X*]) @ *ω*)
using *leftderives-implies-LeftDerivation* **by** *blast*
then obtain *D* **where** *D*: *LeftDerivation* [𝔖] *D* (((*terminals p*)@[*X*]) @ *ω*) **by** *blast*
let *?P* = λ *k*. (∃ *a b*. *LeftDerivation* [𝔖] (*take k D*) (*a*@[*X*]@*b*) ∧ *derives a* (*terminals p*))
have *?P* (*length D*)
apply (*rule-tac x*=*terminals p* **in** *exI*)
apply (*rule-tac x*=*ω* **in** *exI*)
using *D* **by** *simp*
then show *?thesis*
proof (*induct rule*: *minimal-witness*[**where** *P*=*?P*])
case (*Minimal K*)
from *Minimal*(*2*) **obtain** *a b* **where**
aXb: *LeftDerivation* [𝔖] (*take K D*) (*a* @ [*X*] @ *b*) **and**
a: *derives a* (*terminals p*) **by** *blast*
have *KD*: *K* > *0* ∧ *length D* > *0*
proof (*cases K* = *0* ∨ *length D* = *0*)
case *True*
hence *take K D* = [] **by** *auto*
with *True aXb* **have** [𝔖] = *a* @ [*X*] @ *b* **by** *simp*
hence 𝔖 = *X*
by (*metis Nil-is-append-conv append-self-conv2 butlast.simps*(*2*)
butlast-append hd-append2 list.sel(*1*) *not-Cons-self2*)
then have *False*
using *X is-nonterminal-startsymbol is-terminal-nonterminal* **by** *auto*
then show *?thesis* **by** *blast*
next
case *False* **thus** *?thesis* **by** *arith*
qed
then have *take K D* = *take* (*K* − *1*) *D* @ [*D* ! (*K* − *1*)]
by (*metis Minimal.hyps*(*1*) *One-nat-def Suc-less-eq Suc-pred hd-drop-conv-nth*

le-imp-less-Suc take-hd-drop)
then have ∃ *δ*. *LeftDerivation* [𝔖] (*take* (*K* − *1*) *D*) *δ* ∧ *LeftDerivation δ* [*D* ! (*K* − *1*)] (*a*@[*X*]@*b*)
by (*metis LeftDerivation-append aXb*)
then obtain *δ* **where**
δ1: *LeftDerivation* [𝔖] (*take* (*K* − *1*) *D*) *δ*
and *δ2*: *LeftDerivation δ* [*D* ! (*K* − *1*)] (*a*@[*X*]@*b*)
by *blast*
from *δ2* **have** ∃ *i r*. *LeftDerives1 δ i r* (*a*@[*X*]@*b*) **by** *fastforce*
then obtain *i r* **where** *LeftDerives1-δ*: *LeftDerives1 δ i r* (*a*@[*X*]@*b*) **by** *blast*
then have *Derives1-δ*: *Derives1 δ i r* (*a*@[*X*]@*b*)
by (*metis LeftDerives1-implies-Derives1*)
then have ∃ *α N β* . *splits-at δ i α N β* **by** (*simp add*: *splits-at-ex*)
then obtain *α N β* **where** *split-δ*: *splits-at δ i α N β* **by** *blast*

have *is-word-α*: *is-word α* **by** (*metis LeftDerives1-δ LeftDerives1-splits-at-is-word split-δ*)
have ¬ (*?P* (*K* − *1*)) **using** *KD Minimal*(*3*) **by** *auto*
with *δ1* **have** *min-δ*: ¬ (∃ *a b*. *δ* = *a*@[*X*]@*b* ∧ *derives a* (*terminals p*)) **by** *blast*
from *Derives1-δ split-δ* **have** ∃ *u v*. *a* = *α* @ *u* ∧ *b* = *v* @ *β* ∧ (*snd r*) = *u*@[*X*]@*v*
proof (*induction rule*: *Derives1-X-is-part-of-rule*)
case (*Suffix γ*)
from *min-δ Suffix*(*1*) *a* **show** *?case* **by** *auto*
next
case (*Prefix γ*)
have *derives γ* (*terminals p*)
by (*metis Derives1-implies-derives1 Prefix*(*2*) *a*
derives1-implies-derives derives-trans)
with *min-δ Prefix*(*1*) **show** *?case* **by** *auto*
qed
then obtain *u v* **where** *uXv*: *a* = *α* @ *u* ∧ *b* = *v* @ *β* ∧ (*snd r*) = *u*@[*X*]@*v* **by** *blast*
let *?l* = *length α*
let *?q* = *take ?l p*
let *?x* = *Item r* (*length u*) (*charslength ?q*) (*charslength p*)
have *item-rhs ?x* = *snd r* **by** (*simp add*: *item-rhs-def*)
then have *item-rhs-x*: *item-rhs ?x* = *u*@[*X*]@*v* **using** *uXv* **by** *simp*
have *wellformed-x*: *wellformed-item ?x*
apply (*auto simp add*: *wellformed-item-def*)
apply (*metis Derives1-δ Derives1-rule*)
apply (*rule is-prefix-length*)
apply (*rule is-prefix-chars*)
apply *simp*
apply (*simp add*: *doc-tokens-length*[*OF p*])
using *item-rhs-x* **by** *simp*
from *item-rhs-x* **have** *next-symbol-x*: *next-symbol ?x* = *Some X*
by (*auto simp add*: *next-symbol-def is-complete-def*)
have *len-α-p*: *length α* ≤ *length p*
apply (*rule-tac path-length-is-upper-bound*[**where** *u*=*u*])
apply (*simp add*: *wellformed-p*)
apply (*simp add*: *is-word-α*)
using *a uXv* **by** *blast*
have *item-nonterminal-x*: *item-nonterminal ?x* = *N*
apply (*simp add*: *item-nonterminal-def*)
using *Derives1-δ Derives1-nonterminal split-δ* **by** *blast*
have *take-terminals*: *take* (*length α*) (*terminals p*) = *α*
apply (*rule-tac take-prefix*)
using *a derives-word-is-prefix is-word-α uXv* **by** *blast*
have *item-α-x*: *item-α ?x* = *u* **using** *item-α-def item-rhs-x* **by** *auto*
from *wellformed-x next-symbol-x len-α-p* **show** *?thesis*
apply (*rule-tac x*=*?x* **in** *exI*)
apply (*auto simp add*: *pvalid-def wellformed-p*)

apply (*rule-tac x=length* α **in** *exI*)
apply (*auto*)
using *item-nonterminal-x* **apply** (*simp*)
apply (*simp add*: *take-terminals*)
apply (*rule-tac x*=β **in** *exI*)
using *LeftDerivation-implies-leftderives* δ*1 is-leftderivation-def split-*δ *splits-at-combine*

apply *auto*[*1*]
using *item-*α*-x* **apply** *simp*
by (*metis a derive-drop-prefixword is-word-*α *uXv*)
qed
qed

lemma *admissible-wellformed-tokens*: *admissible p* $\Longrightarrow$ *wellformed-tokens p*
by (*auto simp add*: *admissible-def* $\mathcal{L}_P$*-wellformed-tokens*)

lemma *chars-append*[*simp*]: *chars* (*a*@*b*) = (*chars a*)@(*chars b*)
by (*induct a arbitrary*: *b*, *auto*)

lemma *chars-of-token-simp*[*simp*]: *chars-of-token* (*a*, *b*) = *b*
by (*simp add*: *chars-of-token-def*)

lemma $\mathcal{X}$*-is-prefix*: *t* $\in$ $\mathcal{X}$ *k* $\Longrightarrow$ *is-prefix* (*snd t*) (*drop k Doc*)
by (*auto simp add*: $\mathcal{X}$*-def*)

lemma *is-prefix-append*: *is-prefix* (*a*@*b*) *D* = (*is-prefix a D* $\wedge$ *is-prefix b* (*drop* (*length a*) *D*))
by (*metis append-assoc is-prefix-cancel is-prefix-def is-prefix-unsplit*)

lemma $\mathfrak{P}$*-are-doc-tokens*: *p* $\in$ $\mathfrak{P}$ $\Longrightarrow$ *doc-tokens p*
proof (*induct rule*: $\mathfrak{P}$*-induct*)
case *Base* **thus** *?case*
by (*simp add*: *doc-tokens-def wellformed-tokens-def*)
next
case (*Induct p k u*)
from *Induct*(*2*)[*simplified*] **show** *?case*
proof (*induct rule*: *limit-induct*)
case (*Init p*) **from** *Induct*(*1*)[*OF Init*] **show** *?case* **.**
next
case (*Iterate p Y*)
have *Y-is-prefix*: $\bigwedge$ *p*. *p* $\in$ *Y* $\Longrightarrow$ *is-prefix* (*chars p*) *Doc*
apply (*drule Iterate*(*1*))
by (*simp add*: *doc-tokens-def*)
have $\mathcal{Y}$ ($\mathcal{Z}$ *k u*) ($\mathcal{P}$ *k u*) *k* $\subseteq$ $\mathcal{X}$ *k* **by** (*metis* $\mathcal{Z}$*.simps*(*2*) $\mathcal{Z}$*-subset-*$\mathcal{X}$)
then have *1*: *Append* ($\mathcal{Y}$ ($\mathcal{Z}$ *k u*) ($\mathcal{P}$ *k u*) *k*) *k Y* $\subseteq$ *Append* ($\mathcal{X}$ *k*) *k Y*
by (*rule Append-mono*, *simp*)
have *2*: *p* $\in$ *Append* ($\mathcal{X}$ *k*) *k Y* $\Longrightarrow$ *doc-tokens p*
apply (*auto simp add*: *Append-def*)
apply (*simp add*: *Iterate*)

```
    apply (auto simp add: doc-tokens-def admissible-wellformed-tokens
        is-prefix-append Y-is-prefix)
    by (metis 𝒳-is-prefix snd-conv)
  show ?case
    apply (rule 2)
    by (metis (mono-tags, lifting) 1 Iterate(2) subsetCE)
 qed
qed

theorem thmD2:
 assumes X: is-terminal X
 assumes p: p ∈ 𝔓
 assumes pX: (terminals p)@[X] ∈ ℒ_P
 shows ∃ x. pvalid p x ∧ next-symbol x = Some X
by (metis X 𝔓-are-doc-tokens p pX thmD2')

end

end
theory TheoremD4
imports TheoremD2
begin

context LocalLexing begin

lemma 𝒳-are-terminals: u ∈ 𝒳 k ⟹ is-terminal (terminal-of-token u)
 by (auto simp add: 𝒳-def is-terminal-def terminal-of-token-def)

lemma terminals-append[simp]: terminals (a@b) = ((terminals a) @ (terminals
b))
 by (auto simp add: terminals-def)

lemma terminals-singleton[simp]: terminals [u] = [terminal-of-token u]
 by (simp add: terminals-def)

lemma terminal-of-token-simp[simp]: terminal-of-token (a, b) = a
 by (simp add: terminal-of-token-def)

lemma pvalid-item-end: pvalid p x ⟹ item-end x = charslength p
 by (metis pvalid-def)

lemma 𝒲-elem-in-TokensAt:
 assumes P: P ⊆ 𝔓
 assumes u-in-𝒲: u ∈ 𝒲 P k
 shows u ∈ TokensAt k (Gen P)
proof −
 have u: u ∈ 𝒳 k ∧ (∃ p∈by-length k P. admissible (p @ [u])) using u-in-𝒲
  by (auto simp add: 𝒲-def)
 then obtain p where p: p ∈ by-length k P ∧ admissible (p @ [u]) by blast
```

```
  then have charslength-p: charslength p = k
   by (metis (mono-tags, lifting) by-length.simps charslength.simps mem-Collect-eq)

  from u have u: u ∈ 𝒳 k by blast
  from p have p-in-𝔓: p ∈ 𝔓
    by (metis (no-types, lifting) P by-length.simps mem-Collect-eq subsetCE)
  then have doc-tokens-p: doc-tokens p by (metis 𝔓-are-doc-tokens)
  let ?X = terminal-of-token u
  have X-is-terminal: is-terminal ?X by (metis 𝒳-are-terminals u)
  from p have terminals p @ [terminal-of-token u] ∈ ℒ_P
    by (auto simp add: admissible-def)
  from thmD2[OF X-is-terminal p-in-𝔓 this] obtain x where
    x: pvalid p x ∧ next-symbol x = Some (terminal-of-token u) by blast
  have x-is-in-Gen-P: x ∈ Gen P
    by (metis (mono-tags, lifting) Gen-def by-length.simps mem-Collect-eq p x)
 have u-split[dest!]: ⋀ t s. u = (t, s) ⟹ t = terminal-of-token u ∧ s = chars-of-token
u
    by (metis chars-of-token-simp fst-conv terminal-of-token-def)
  show ?thesis
    apply (auto simp add: TokensAt-def bin-def)
    apply (rule-tac x=x in exI)
    apply (auto simp add: x-is-in-Gen-P x X-is-terminal)
    using x charslength-p pvalid-item-end apply (simp, blast)
    using u by (auto simp add: 𝒳-def)
qed

lemma is-derivation-is-sentence: is-derivation s ⟹ is-sentence s
by (metis (no-types, lifting) Derives1-sentence2 derives1-implies-Derives1
    derives-induct is-derivation-def is-nonterminal-startsymbol is-sentence-cons
    is-sentence-def is-symbol-def list.pred-inject(1))

lemma is-sentence-cons: is-sentence (N#s) = (is-symbol N ∧ is-sentence s)
  by (auto simp add: is-sentence-def)

lemma is-derivation-step:
  assumes uNv: is-derivation (u@[N]@v)
  assumes Nα: (N, α) ∈ ℜ
  shows is-derivation (u@α@v)
proof –
  from uNv have is-sentence (u@[N]@v) by (metis is-derivation-is-sentence)
  with is-sentence-concat is-sentence-cons
  have u-is-sentence: is-sentence u and v-is-sentence: is-sentence v
    by auto
  from Nα have derives1 (u@[N]@v) (u@α@v)
    apply (auto simp add: derives1-def)
    apply (rule-tac x=u in exI)
    apply (rule-tac x=v in exI)
    apply (rule-tac x=N in exI)
    by (auto simp add: u-is-sentence v-is-sentence)
```

then show *?thesis*
by (*metis derives1-implies-derives derives-trans is-derivation-def uNv*)
qed

lemma *is-derivation-derives*:
derives α β $\Longrightarrow$ *is-derivation* ($u@\alpha@v$) $\Longrightarrow$ *is-derivation* ($u@\beta@v$)
proof (*induct rule*: *derives-induct*)
case *Base* **thus** *?case* **by** *simp*
next
case (*Step y z*)
from *Step* **have** *1*: *is-derivation* (*u* @ *y* @ *v*) **by** *auto*
from *Step* **have** *2*: *derives1 y z* **by** *auto*
from *1 2* **show** *?case* **by** (*metis append-assoc derives1-def is-derivation-step*)
qed

lemma *item-rhs-split*: *item-rhs x* = (*item-*α *x*)@(*item-*β *x*)
by (*metis append-take-drop-id item-*α*-def item-*β*-def*)

lemma *pvalid-is-derivation-terminals-item-*β:
assumes *pvalid*: *pvalid p x*
shows $\exists$ δ. *is-derivation* ((*terminals p*)@(*item-*β *x*)@δ)
proof −
from *pvalid* **have** $\exists$ *u* γ. *is-derivation* (*terminals* (*take u p*) @ [*item-nonterminal x*] @ γ) $\wedge$
derives (*item-*α *x*) (*terminals* (*drop u p*))
by (*auto simp add*: *pvalid-def*)
then obtain *u* γ **where** *1*: *is-derivation* (*terminals* (*take u p*) @ [*item-nonterminal x*] @ γ) $\wedge$
derives (*item-*α *x*) (*terminals* (*drop u p*)) **by** *blast*
have *x-rule*: (*item-nonterminal x*, *item-rhs x*) $\in \mathfrak{R}$
by (*metis* (*no-types*, *lifting*) *LocalLexing.pvalid-def LocalLexing-axioms assms case-prodE item-nonterminal-def item-rhs-def prod.sel*(*1*) *snd-conv validRules well-formed-item-def*)
from *1 x-rule is-derivation-step* **have**
is-derivation ((*take u* (*terminals p*)) @ (*item-rhs x*) @ γ)
by *auto*
then have *is-derivation* ((*take u* (*terminals p*)) @ ((*item-*α *x*)@(*item-*β *x*)) @ γ)

by (*simp add*: *item-rhs-split*)
then have *is-derivation* ((*take u* (*terminals p*)) @ (*item-*α *x*) @ ((*item-*β *x*) @ γ))
by *simp*
then have *is-derivation* ((*take u* (*terminals p*)) @ (*drop u* (*terminals p*)) @ ((*item-*β *x*) @ γ))
by (*metis 1 is-derivation-derives terminals-drop*)
then have *is-derivation* ((*terminals p*) @ ((*item-*β *x*) @ γ))
by (*metis append-assoc append-take-drop-id*)
then show *?thesis* **by** *auto*
qed

lemma *next-symbol-not-complete*: *next-symbol x* = *Some t* ⟹ ¬ (*is-complete x*)
  by (*metis next-symbol-def option.discI*)

lemma *next-symbol-starts-item-β*:
  assumes *wf*: *wellformed-item x*
  assumes *next-symbol*: *next-symbol x* = *Some t*
  shows ∃ δ. *item-β x* = *t*#δ
proof −
  from *next-symbol* **have** *nc*: ¬ (*is-complete x*) **using** *next-symbol-not-complete* **by** *auto*
  from *next-symbol* **have** *atdot*: *item-rhs x* ! *item-dot x* = *t* **by** (*simp add*: *next-symbol-def nc*)
  from *nc* **have** *inrange*: *item-dot x* < *length* (*item-rhs x*)
    by (*simp add*: *is-complete-def*)
  from *inrange atdot* **show** *?thesis*
    apply (*simp add*: *item-β-def*)
    by (*metis Cons-nth-drop-Suc*)
qed

lemma *pvalid-prefixlang*:
  assumes *pvalid*: *pvalid p x*
  assumes *is-terminal*: *is-terminal t*
  assumes *next-symbol*: *next-symbol x* = *Some t*
  shows (*terminals p*) @ [*t*] ∈ $\mathcal{L}_P$
proof −
  have ∃ δ. *item-β x* = *t*#δ
    by (*metis next-symbol next-symbol-starts-item-β pvalid pvalid-def*)
  then obtain δ **where** δ:*item-β x* = *t*#δ **by** *blast*
  have ∃ ω. *is-derivation* ((*terminals p*)@(*item-β x*)@ω)
    by (*metis pvalid pvalid-is-derivation-terminals-item-β*)
  then obtain ω **where** *is-derivation* ((*terminals p*)@(*item-β x*)@ω) **by** *blast*
  then have *is-derivation* ((*terminals p*)@(*t*#δ)@ω) **by** (*metis* δ)
  then have *is-derivation* (((*terminals p*)@[*t*])@(δ@ω)) **by** *simp*
  then show *?thesis*
    by (*metis* (*no-types*, *lifting*) *CFG.*$\mathcal{L}_P$*-def CFG-axioms*
      append-Nil2 is-terminal is-word-append is-word-cons
      is-word-terminals mem-Collect-eq pvalid pvalid-def)
qed

lemma *TokensAt-elem-in-*$\mathcal{W}$:
  assumes *P*: $P \subseteq \mathfrak{P}$
  assumes *u-in-Tokens-at*: *u* ∈ *TokensAt k* (*Gen P*)
  shows *u* ∈ $\mathcal{W}$ *P k*
proof −
  have ∃ *t s x l*.
      u = (*t*, *s*) ∧
      x ∈ *bin* (*Gen P*) *k* ∧
       next-symbol x = *Some t* ∧ *is-terminal t* ∧ *l* ∈ *Lex t Doc k* ∧ *s* = *take l*

```
(drop k Doc)
  using u-in-Tokens-at by (auto simp add: TokensAt-def)
  then obtain t s x l where
     u: u = (t, s) ∧
         x ∈ bin (Gen P) k ∧
          next-symbol x = Some t ∧ is-terminal t ∧ l ∈ Lex t Doc k ∧ s = take l
(drop k Doc)
        by blast
  from u have t: t = terminal-of-token u by (metis terminal-of-token-simp)
  from u have s: s = chars-of-token u by (metis chars-of-token-simp)
  from u have item-end-x: item-end x = k by (metis (mono-tags, lifting) bin-def
mem-Collect-eq)
  from u have ∃ p ∈ P. pvalid p x by (auto simp add: bin-def Gen-def)
  then obtain p where p: p ∈ P and pvalid: pvalid p x by blast
  have p-len: length (chars p) = k
    by (metis charslength.simps item-end-x pvalid pvalid-item-end)
  have u-in-𝒳: u ∈ 𝒳 k
    apply (simp add: 𝒳-def)
    apply (rule-tac x=t in exI)
    apply (rule-tac x=l in exI)
    using u by (simp add: is-terminal-def)
  show ?thesis
    apply (auto simp add: 𝒲-def)
    apply (simp add: u-in-𝒳)
    apply (rule-tac x=p in exI)
    apply (simp add: p p-len)
    apply (simp add: admissible-def t[symmetric])
    apply (rule pvalid-prefixlang[where x=x])
    apply (simp add: pvalid)
    apply (simp add: u)
    apply (simp add: u)
    done
qed

theorem thmD4:
  assumes P: P ⊆ 𝔓
  shows 𝒲 P k = TokensAt k (Gen P)
using 𝒲-elem-in-TokensAt TokensAt-elem-in-𝒲
by (metis Collect-cong Collect-mem-eq assms)

end

end
theory TheoremD5
imports TheoremD4
begin

context LocalLexing begin
```

lemma *Scan-empty*: *Scan* {} *k I* = *I*
  by (*simp add*: *Scan-def*)

lemma *π-no-tokens*: *π k* {} *I* = *limit* (*λ I. Complete k* (*Predict k I*)) *I*
  by (*simp add*: *π-def Scan-empty*)

lemma *bin-elem*: *x* ∈ *bin I k* ⟹ *x* ∈ *I*
  by (*auto simp add*: *bin-def*)

lemma *Gen-implies-pvalid*: *x* ∈ *Gen P* ⟹ ∃ *p* ∈ *P*. *pvalid p x*
  by (*auto simp add*: *Gen-def*)

lemma *wellformed-init-item*[*simp*]: *r* ∈ 𝔑 ⟹ *k* ≤ *length Doc* ⟹ *wellformed-item* (*init-item r k*)
  by (*simp add*: *init-item-def wellformed-item-def*)

lemma *init-item-origin*[*simp*]: *item-origin* (*init-item r k*) = *k*
  by (*auto simp add*: *item-origin-def init-item-def*)

lemma *init-item-end*[*simp*]: *item-end* (*init-item r k*) = *k*
  by (*auto simp add*: *item-end-def init-item-def*)

lemma *init-item-nonterminal*[*simp*]: *item-nonterminal* (*init-item r k*) = *fst r*
  by (*auto simp add*: *init-item-def item-nonterminal-def*)

lemma *init-item-α*[*simp*]: *item-α* (*init-item r k*) = []
  by (*auto simp add*: *init-item-def item-α-def*)

lemma *Predict-elem-in-Gen*:
  assumes *I-in-Gen-P*: *I* ⊆ *Gen P*
  assumes *k*: *k* ≤ *length Doc*
  assumes *x-in-Predict*: *x* ∈ *Predict k I*
  shows *x* ∈ *Gen P*
proof −
  have *x* ∈ *I* ∨ (∃ *r y*. *r* ∈ 𝔑 ∧ *x* = *init-item r k* ∧ *y* ∈ *bin I k* ∧ *next-symbol y* = *Some*(*fst r*))
    using *x-in-Predict* **by** (*auto simp add*: *Predict-def*)
  then show *?thesis*
  proof (*induct rule*: *disjCases2*)
    case *1* **thus** *?case* **using** *I-in-Gen-P* **by** *blast*
  next
    case *2*
    then obtain *r y* **where** *ry*: *r* ∈ 𝔑 ∧ *x* = *init-item r k* ∧ *y* ∈ *bin I k* ∧
      next-symbol y = *Some* (*fst r*) **by** *blast*
    then have ∃ *p* ∈ *P*. *pvalid p y*
      using *Gen-implies-pvalid I-in-Gen-P bin-elem subsetCE* **by** *blast*
    then obtain *p* **where** *p*: *p* ∈ *P* ∧ *pvalid p y* **by** *blast*
    have *wellformed-p*: *wellformed-tokens p* **using** *p* **by** (*auto simp add*: *pvalid-def*)
    have *wellformed-x*: *wellformed-item x*

```
      by (simp add: ry k)
    from ry have item-end y = k by (auto simp add: bin-def)
    with p have charslength-p[simplified]: charslength p = k by (auto simp add: pvalid-def)
    have item-end-x: item-end x = k by (simp add: ry)
    have pvalid-x: pvalid p x
      apply (auto simp add: pvalid-def)
      apply (simp add: wellformed-p)
      apply (simp add: wellformed-x)
      apply (rule-tac x=length p in exI)
      apply (auto simp add: charslength-p ry)
      by (metis append-Cons next-symbol-starts-item-β p pvalid-def
        pvalid-is-derivation-terminals-item-β ry)
    then show ?case using Gen-def mem-Collect-eq p by blast
 qed
qed

lemma Predict-subset-Gen:
  assumes I ⊆ Gen P
  assumes k ≤ length Doc
  shows Predict k I ⊆ Gen P
using Predict-elem-in-Gen assms by blast

lemma nth-superfluous-append[simp]: i < length a ⟹ (a@b)!i = a!i
by (simp add: nth-append)

lemma tokens-nth-in-𝒵:
  p ∈ 𝔓 ⟹ ∀ i. i < length p ⟶ (∃ u. p ! i ∈ 𝒵 (charslength (take i p)) u)
proof (induct rule: 𝔓-induct)
  case Base thus ?case by simp
next
  case (Induct p k u)
  then have p ∈ limit (Append (𝒵 k (Suc u)) k) (𝒫 k u) by simp
  then show ?case
  proof (induct rule: limit-induct)
    case (Init p) thus ?case using Induct by auto
  next
    case (Iterate p Y)
    from Iterate(2) have p ∈ Y ∨ (∃ q t. p = q@[t] ∧ q ∈ by-length k Y ∧ t ∈ 𝒵 k (Suc u) ∧
      admissible (q @ [t]))
      by (auto simp add: Append-def)
    then show ?case
    proof (induct rule: disjCases2)
      case 1 thus ?case using Iterate(1) by auto
    next
      case 2
      then obtain q t where
        qt: p = q @ [t] ∧ q ∈ by-length k Y ∧ t ∈ 𝒵 k (Suc u) ∧ admissible (q @
```

```
[t]) by blast
      then have q-in-Y: q ∈ Y by auto
      with qt have k: k = charslength q by auto
      with qt have t: t ∈ Z k (Suc u) by auto
      show ?case
      proof(auto simp add: qt)
        fix i
        assume i: i < Suc (length q)
        then have i < length q ∨ i = length q by arith
        then show ∃ u. (q @ [t]) ! i ∈ Z (length (chars (take i q))) u
        proof (induct rule: disjCases2)
          case 1
            from Iterate(1)[OF q-in-Y]
            show ?case by (simp add: 1)
        next
          case 2
            show ?case
              apply (auto simp add: 2)
              apply (rule-tac x=Suc u in exI)
              using k t by auto
        qed
      qed
    qed
  qed
qed

lemma path-append-token:
  assumes p: p ∈ P k u
  assumes t: t ∈ Z k (Suc u)
  assumes pt: admissible (p@[t])
  assumes k: charslength p = k
  shows p@[t] ∈ P k (Suc u)
apply (simp only: P.simps)
apply (rule-tac n=Suc 0 in limit-elem)
using p t pt k apply (auto simp only: Append-def funpower.simps)
by fastforce

definition indexlt-rel :: ((nat × nat) × (nat × nat)) set where
  indexlt-rel = less-than <*lex*> less-than

definition indexlt :: nat ⇒ nat ⇒ nat ⇒ nat ⇒ bool where
  indexlt k' u' k u = (((k', u'), (k, u)) ∈ indexlt-rel)

lemma indexlt-simp: indexlt k' u' k u = (k' < k ∨ (k' = k ∧ u' < u))
  by (auto simp add: indexlt-def indexlt-rel-def)

lemma wf-indexlt-rel: wf indexlt-rel
  using indexlt-rel-def pair-less-def by auto
```

```
lemma 𝒫-induct[consumes 1, case-names Induct]:
  assumes p ∈ 𝒫 k u
  assumes induct: ⋀ p k u . (⋀ p′ k′ u′. p′ ∈ 𝒫 k′ u′ ⟹ indexlt k′ u′ k u ⟹ P
p′ k′ u′)
                    ⟹ p ∈ 𝒫 k u ⟹ P p k u
  shows P p k u
proof −
  let ?R = indexlt-rel <*lex*> {}
  have wf-R: wf ?R by (auto simp add: wf-indexlt-rel)
  let ?P = λ a. snd a ∈ 𝒫 (fst (fst a)) (snd (fst a)) ⟶ P (snd a) (fst (fst a))
(snd (fst a))
  have p ∈ 𝒫 k u ⟶ P p k u
   apply (rule wf-induct[OF wf-R, where P = ?P and a = ((k, u), p), simplified])
    apply (auto simp add: indexlt-def[symmetric])
    apply (rule-tac p=ba and k=a and u=b in induct)
    by auto
  thus ?thesis using assms by auto
qed

lemma nonempty-path-indices:
  assumes p: p ∈ 𝒫 k u
  assumes nonempty: p ≠ []
  shows k > 0 ∨ u > 0
proof (cases u = 0)
  case True
  note u = True
  have k > 0
  proof (cases k = 0)
    case True
    with p u have p = [] by simp
    with nonempty have False by auto
    then show ?thesis by auto
  next
    case False
    then show ?thesis by arith
  qed
  then show ?thesis by blast
next
  case False
  then show ?thesis by arith
qed

lemma base-paths:
  assumes p: p ∈ 𝒫 k 0
  assumes k: k > 0
  shows ∃ u. p ∈ 𝒫 (k − 1) u
proof −
  from k have ∃ i. k = Suc i by arith
  then obtain i where i: k = Suc i by blast
```

from *p* **show** *?thesis*
by (*auto simp add*: *i natUnion-def*)
qed

lemma *indexlt-trans*: *indexlt* k'' u'' k' u' $\Longrightarrow$ *indexlt* k' u' *k u* $\Longrightarrow$ *indexlt* k'' u'' *k u*
using *dual-order.strict-trans indexlt-simp* **by** *auto*

definition *is-continuation* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *tokens* $\Rightarrow$ *tokens* $\Rightarrow$ *bool* **where**
is-continuation k u q ts = ($q \in \mathcal{P}$ *k u* $\wedge$ *charslength q* = *k* $\wedge$ *admissible* (*q*@*ts*) $\wedge$
($\forall$ *t* $\in$ *set ts*. *t* $\in$ $\mathcal{Z}$ *k* (*Suc u*)) $\wedge$ ($\forall$ *t* $\in$ *set* (*butlast ts*). *chars-of-token t* = []))

lemma *limit-Append-path-nonelem-split*: *p* $\in$ *limit* (*Append T k*) ($\mathcal{P}$ *k u*) $\Longrightarrow$ *p* $\notin$ $\mathcal{P}$ *k u* $\Longrightarrow$
$\exists$ *q ts*. *p* = *q*@*ts* $\wedge$ *q* $\in$ $\mathcal{P}$ *k u* $\wedge$ *charslength q* = *k* $\wedge$ *admissible* (*q*@*ts*) $\wedge$ ($\forall$ *t* $\in$ *set ts*. *t* $\in$ *T*) $\wedge$
($\forall$ *t* $\in$ *set* (*butlast ts*). *chars-of-token t* = [])
proof (*induct rule*: *limit-induct*)
case (*Init p*) **thus** *?case* **by** *auto*
next
case (*Iterate p Y*)
show *?case*
proof (*cases p* $\in$ *Y*)
case *True*
from *Iterate*(*1*)[*OF True Iterate*(*3*)] **show** *?thesis* **by** *blast*
next
case *False*
with *Append-def Iterate*(*2*)
have $\exists$ *q t*. *p* = *q*@[*t*] $\wedge$ *q* $\in$ *by-length k Y* $\wedge$ *t* $\in$ *T* $\wedge$ *admissible* (*q* @ [*t*]) **by** *auto*
then obtain *q t* **where** *qt*: *p* = *q*@[*t*] $\wedge$ *q* $\in$ *by-length k Y* $\wedge$ *t* $\in$ *T* $\wedge$ *admissible* (*q* @ [*t*])
by *blast*
from *qt* **have** *qlen*: *charslength q* = *k* **by** *auto*
have *q* $\in$ $\mathcal{P}$ *k u* $\vee$ *q* $\notin$ $\mathcal{P}$ *k u* **by** *blast*
then show *?thesis*
proof(*induct rule*: *disjCases2*)
case *1*
show *?case*
apply (*rule-tac x*=*q* **in** *exI*)
apply (*rule-tac x*=[*t*] **in** *exI*)
using *qlen* **by** (*simp add*: *qt 1*)
next
case *2*
have *q-in-Y*: *q* $\in$ *Y* **using** *qt* **by** *auto*
from *Iterate*(*1*)[*OF q-in-Y 2*]
obtain q' *ts* **where**
q'*ts*: *q* = q' @ *ts* $\wedge$ q' $\in$ $\mathcal{P}$ *k u* $\wedge$ *charslength* q' = *k* $\wedge$ ($\forall$ *t*$\in$*set ts*. *t* $\in$ *T*) $\wedge$

```
          (∀ t∈set(butlast ts). chars-of-token t = [])
      by blast
    with qlen have charslength ts = 0 by auto
    hence empty: ∀ t ∈ set(ts). chars-of-token t = []
      apply (induct ts)
      by auto
    show ?case
      apply (rule-tac x=q′ in exI)
      apply (rule-tac x=ts@[t] in exI)
      using qt q′ts empty by auto
   qed
  qed
qed

lemma limit-Append-path-nonelem-split′:
  p ∈ limit (Append (𝒵 k (Suc u)) k) (𝒫 k u) ⟹ p ∉ 𝒫 k u ⟹
   ∃ q ts. p = q@ts ∧ is-continuation k u q ts
apply (simp only: is-continuation-def)
apply (rule-tac limit-Append-path-nonelem-split)
by auto

lemma final-step-of-path: p ∈ 𝒫 k u ⟹ p ≠ [] ⟹ (∃ q ts k′ u′. p = q@ts ∧
indexlt k′ u′ k u
  ∧ is-continuation k′ u′ q ts)
proof (induct rule: 𝒫-induct)
  case (Induct p k u)
  from Induct(2) Induct(3) have ku-0: k > 0 ∨ u > 0
    using nonempty-path-indices by blast
  show ?case
  proof (cases u = 0)
    case True
    with ku-0 have k-0: k > 0 by arith
    with True Induct(2) base-paths have ∃ u′. p ∈ 𝒫 (k − 1) u′ by auto
    then obtain u′ where u′: p ∈ 𝒫 (k − 1) u′ by blast
    have indexlt: indexlt (k − 1) u′ k u by (simp add: indexlt-simp k-0)
    from Induct(1)[OF u′ indexlt Induct(3)] show ?thesis
      using indexlt indexlt-trans by blast
  next
    case False
    then have ∃ u′. u = Suc u′ by arith
    then obtain u′ where u′: u = Suc u′ by blast
    with Induct(2) have p-limit: p ∈ limit (Append (𝒵 k (Suc u′)) k) (𝒫 k u′)
      using 𝒫.simps(2) by blast
    from u′ have indexlt: indexlt k u′ k u by (simp add: indexlt-simp)
    have p ∈ 𝒫 k u′ ∨ p ∉ 𝒫 k u′ by blast
    then show ?thesis
    proof (induct rule: disjCases2)
      case 1
      from Induct(1)[OF 1 indexlt Induct(3)] show ?case
```

```
      using indexlt indexlt-trans by blast
    next
      case 2
      from limit-Append-path-nonelem-split'[OF p-limit 2]
      show ?case using indexlt u' by auto
    qed
  qed
qed

lemma terminals-empty[simp]: terminals [] = []
  by (auto simp add: terminals-def)

lemma empty-in-ℒ_P[simp]: [] ∈ ℒ_P
  apply (simp add: ℒ_P-def is-derivation-def)
  apply (rule-tac x=[𝔖] in exI)
  by simp

lemma admissible-empty[simp]: admissible []
  by (auto simp add: admissible-def)

lemma 𝔓-are-admissible: p ∈ 𝔓 ⟹ admissible p
proof (induct rule: 𝔓-induct)
  case Base thus ?case by simp
next
  case (Induct p k u)
  from Induct(2)[simplified] show ?case
  proof (induct rule: limit-induct)
    case (Init p) from Induct(1)[OF Init] show ?case .
  next
    case (Iterate p Y)
    have 𝒴 (𝒵 k u) (𝒫 k u) k ⊆ 𝒳 k by (metis 𝒵.simps(2) 𝒵-subset-𝒳)
    then have 1: Append (𝒴 (𝒵 k u) (𝒫 k u) k) k Y ⊆ Append (𝒳 k) k Y
      by (rule Append-mono, simp)
    have 2: p ∈ Append (𝒳 k) k Y ⟹ admissible p
      apply (auto simp add: Append-def)
      by (simp add: Iterate)
    show ?case
      apply (rule 2)
      using 1 Iterate(2) by blast
  qed
qed

lemma prefix-of-empty-is-empty: is-prefix q [] ⟹ q = []
by (metis is-prefix-cons neq-Nil-conv)

lemma subset-𝒫 :
  assumes leq: k' < k ∨ (k' = k ∧ u' ≤ u)
  shows 𝒫 k' u' ⊆ 𝒫 k u
proof −
```

from *leq* **show** *?thesis*
proof (*induct rule*: *disjCases2*)
case *1*
have *s1*: $\mathcal{P}$ *k′ u′* ⊆ $\mathcal{Q}$ *k′* **by** (*rule-tac subset-*$\mathcal{PQ}$*k*)
have *s2*: $\mathcal{Q}$ *k′* ⊆ $\mathcal{Q}$ (*k* − *1*)
apply (*rule-tac subset-*$\mathcal{Q}$)
using *1* **by** *arith*
from *subset-*$\mathcal{QP}$*Suc*[**where** *k*=*k* − *1*] *1* **have** *s3*: $\mathcal{Q}$ (*k* − *1*) ⊆ $\mathcal{P}$ *k 0*
by *simp*
have *s4*: $\mathcal{P}$ *k 0* ⊆ $\mathcal{P}$ *k u* **by** (*rule-tac subset-*$\mathcal{P}$*k*, *simp*)
from *s1 s2 s3 s4 subset-trans* **show** *?case* **by** *blast*
next
case *2* **thus** *?case* **by** (*simp add* : *subset-*$\mathcal{P}$*k*)
qed
qed

lemma *empty-path-is-elem*[*simp*]: [] ∈ $\mathcal{P}$ *k u*
proof −
have [] ∈ $\mathcal{P}$ *0 0* **by** *simp*
then show [] ∈ $\mathcal{P}$ *k u* **by** (*metis le0 not-gr0 subsetCE subset-*$\mathcal{P}$)
qed

lemma *is-prefix-of-append*:
assumes *is-prefix p* (*a*@*b*)
shows *is-prefix p a* ∨ (∃ *b′*. *b′* ≠ [] ∧ *is-prefix b′ b* ∧ *p* = *a*@*b′*)
apply (*auto simp add*: *is-prefix-def*)
by (*metis append-Nil2 append-eq-append-conv2 assms is-prefix-cancel is-prefix-def*)

lemma *prefix-is-continuation*: *is-continuation k u p ts* ⟹ *is-prefix ts′ ts* ⟹
is-continuation k u p ts′
apply (*auto simp add*: *is-continuation-def is-prefix-def*)
apply (*metis* $\mathcal{L}_P$*-split admissible-def append-assoc terminals-append*)
using *in-set-butlast-appendI* **by** *fastforce*

lemma *charslength-0*: (∀ *t* ∈ *set ts*. *chars-of-token t* = []) = (*charslength ts* = *0*)
by (*induct ts*, *auto*)

lemma *is-continuation-in-*$\mathcal{P}$: *is-continuation k u p ts* ⟹ *p*@*ts* ∈ $\mathcal{P}$ *k* (*Suc u*)
proof(*induct ts rule*: *rev-induct*)
case *Nil* **thus** *?case*
apply (*auto simp add*: *is-continuation-def*)
using *subset-*$\mathcal{P}$*Suc* **by** *fastforce*
next
case (*snoc t ts*)
from *snoc*(*2*) **have** *is-continuation k u p ts*
by (*metis append-Nil2 is-prefix-cancel is-prefix-empty prefix-is-continuation*)
note *induct* = *snoc*(*1*)[*OF this*]
then have *pts*: *p*@*ts* ∈ *limit* (*Append* ($\mathcal{Z}$ *k* (*Suc u*)) *k*) ($\mathcal{P}$ *k u*) **by** *simp*
note *is-cont* = *snoc*(*2*)

```
  then have admissible: admissible (p@ts@[t]) by (simp add: is-continuation-def)
  from is-cont have t: t ∈ Z k (Suc u) by (simp add: is-continuation-def)
  from is-cont have ∀ t ∈ set ts. chars-of-token t = [] by (simp add: is-continuation-def)
  then have charslength-ts: charslength ts = 0 by (simp only: charslength-0)
  from is-cont have plen: charslength p = k by (simp add: is-continuation-def)
  show ?case
    apply (simp only: P.simps)
    apply (rule-tac limit-step-pointwise[OF pts])
    apply (simp add: pointwise-Append)
    apply (auto simp add: Append-def)
    apply (rule-tac x=fst t in exI)
    apply (rule-tac x=snd t in exI)
    apply (auto simp add: admissible)
    using charslength-ts apply simp
    using plen apply simp
    using t by simp
qed

lemma indexlt-subset-P: indexlt k' u' k u ⟹ P k' (Suc u') ⊆ P k u
apply (rule-tac subset-P)
apply (simp add: indexlt-simp)
apply arith
done

lemma prefixes-are-paths: p ∈ P k u ⟹ is-prefix x p ⟹ x ∈ P k u
proof (induct arbitrary: x rule: P-induct)
  case (Induct p k u)
  show ?case
  proof (cases p = [])
    case True
    then have x = []
      using Induct.prems prefix-of-empty-is-empty by blast
    then show x ∈ P k u by simp
  next
    case False
    from final-step-of-path[OF Induct(2) False]
    obtain q ts k' u' where step: p = q@ts ∧ indexlt k' u' k u ∧ is-continuation
k' u' q ts
      by blast
    have subset: P k' u' ⊆ P k u
      by (metis indexlt-simp less-or-eq-imp-le step subset-P)
    have is-prefix x q ∨ (∃ ts'. ts' ≠ [] ∧ is-prefix ts' ts ∧ x = q@ts')
      apply (rule-tac is-prefix-of-append)
      using Induct(3) step by auto
    then show ?thesis
    proof (induct rule: disjCases2)
      case 1
        have x: x ∈ P k' u'
        using 1 Induct step by (auto simp add: is-continuation-def)
```

```
      then show x ∈ 𝒫 k u using subset subsetCE by blast
    next
      case 2
      then obtain ts' where ts': is-prefix ts' ts ∧ x = q@ts' by blast
      have is-continuation k' u' q ts' using step prefix-is-continuation ts' by blast
      with ts' have x ∈ 𝒫 k' (Suc u')
        apply (simp only: ts')
        apply (rule-tac is-continuation-in-𝒫)
        by simp
      with subset show x ∈ 𝒫 k u using indexlt-subset-𝒫 step by blast
    qed
  qed
qed

lemma empty-or-last-of-suffix:
  assumes q = q' @ [t]
  assumes q = p @ ts
  shows ts = [] ∨ (∃ ts'. q' = p @ ts' ∧ ts'@[t] = ts)
by (metis assms(1) assms(2) butlast-append last-appendR snoc-eq-iff-butlast)

lemma is-prefix-butlast: is-prefix q (butlast p) ⟹ is-prefix q p
by (metis butlast-conv-take is-prefix-append is-prefix-def is-prefix-take)

lemma last-step-of-path:
  q ∈ 𝒫 k u ⟹ q = q'@[t] ⟹
   ∃ k' u'. indexlt k' u' k u ∧ q ∈ 𝒫 k' (Suc u') ∧ charslength q' = k' ∧ t ∈ 𝒵 k'
(Suc u')
proof (induct arbitrary: q' t rule: 𝒫-induct)
  case (Induct q k u)
    have ∃ p ts k' u'. q = p@ts ∧ indexlt k' u' k u ∧ is-continuation k' u' p ts
      apply (rule-tac final-step-of-path)
      apply (simp add: Induct(2))
      apply (simp add: Induct(3))
      done
   then obtain p ts k' u' where pts: q = p@ts ∧ indexlt k' u' k u ∧ is-continuation
k' u' p ts
      by blast
    then have indexlt: indexlt k' u' k u by auto
    from pts have ts = [] ∨ (∃ ts'. q' = p @ ts' ∧ ts'@[t] = ts)
      by (metis empty-or-last-of-suffix Induct(3))
    then show ?case
    proof (induct rule: disjCases2)
      case 1
        with pts have q: q ∈ 𝒫 k' u' by (auto simp add: is-continuation-def)
        from Induct(1)[OF this indexlt Induct(3)] show ?case
          using indexlt indexlt-trans by blast
    next
      case 2
        then obtain ts' where ts': q' = p @ ts' ∧ ts'@[t] = ts by blast
```

then have *is-prefix ts′ ts* **using** *is-prefix-def* **by** *blast*
then have *is-continuation k′ u′ p ts′* **by** (*metis prefix-is-continuation pts*)
have *charslength ts′* = *0* **using** *charslength-0 is-continuation-def pts ts′* **by** *auto*
then have *q′len*: *charslength q′* = *k′* **using** *is-continuation-def pts ts′* **by** *auto*
have $t \in$ *set ts* **using** *ts′* **by** *auto*
with *pts* **have** *t-in-𝒵*: $t \in \mathcal{Z}\ k'\ (Suc\ u')$ **using** *is-continuation-def* **by** *blast*
have *q-dom*: $q \in \mathcal{P}\ k'\ (Suc\ u')$ **using** *pts is-continuation-in-𝒫* **by** *blast*
show *?case*
apply (*rule-tac x=k′* **in** *exI*)
apply (*rule-tac x=u′* **in** *exI*)
by (*simp only*: *indexlt q′len t-in-𝒵 q-dom*)
qed
qed

lemma *charslength-of-butlast-0*: $p \in \mathcal{P}\ k\ 0 \Longrightarrow p = q@[t] \Longrightarrow charslength\ q < k$
using *last-step-of-path LocalLexing-axioms indexlt-simp* **by** *blast*

lemma *charslength-of-butlast*: $p \in \mathcal{P}\ k\ u \Longrightarrow p = q@[t] \Longrightarrow charslength\ q \leq k$
by (*metis indexlt-simp last-step-of-path eq-imp-le less-imp-le-nat*)

lemma *last-token-of-path*:
assumes $q \in \mathcal{P}\ k\ u$
assumes $q = q'@[t]$
assumes *charslength q′* = *k*
shows $t \in \mathcal{Z}\ k\ u$
proof −
from *assms* **have** $\exists\ k'\ u'.\ indexlt\ k'\ u'\ k\ u \wedge q \in \mathcal{P}\ k'\ (Suc\ u') \wedge charslength\ q' = k' \wedge$
$t \in \mathcal{Z}\ k'\ (Suc\ u')$ **using** *last-step-of-path* **by** *blast*
then obtain *k′ u′* **where** *th*: $indexlt\ k'\ u'\ k\ u \wedge q \in \mathcal{P}\ k'\ (Suc\ u') \wedge charslength\ q' = k' \wedge$
$t \in \mathcal{Z}\ k'\ (Suc\ u')$ **by** *blast*
with *assms*(*3*) **have** *k′*: *k′* = *k* **by** *blast*
with *th* **have** $t \in \mathcal{Z}\ k'\ (Suc\ u') \wedge u' < u$ **using** *indexlt-simp* **by** *auto*
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *𝒵-subset-Suc k′ linorder-neqE-nat not-less-eq*

subsetCE subset-fSuc-strict)
qed

lemma *final-step-of-path′*: $p \in \mathcal{P}\ k\ u \Longrightarrow p \notin \mathcal{P}\ k\ (u - 1) \Longrightarrow$
$\exists\ q\ ts.\ u > 0 \wedge p = q@ts \wedge is\text{-}continuation\ k\ (u - 1)\ q\ ts$
by (*metis Suc-diff-1 𝒫.simps*(*2*) *diff-0-eq-0 limit-Append-path-nonelem-split′ not-gr0*)

lemma *is-continuation-continue*:
assumes *is-continuation k u q ts*
assumes *charslength ts* = *0*

```
  assumes t ∈ Z k (Suc u)
  assumes admissible (q @ ts @ [t])
  shows is-continuation k u q (ts@[t])
proof −
  from assms show ?thesis
    by (simp add: is-continuation-def charslength-0)
qed
```

```
theorem compatibility-def:
  assumes p-in-dom: p ∈ P k u
  assumes q-in-dom: q ∈ P k u
  assumes p-charslength: charslength p = k
  assumes q-split: q = q'@[t]
  assumes q'len: charslength q' = k
  assumes admissible: admissible (p @ [t])
  shows p @ [t] ∈ P k u
proof −
  have u: u > 0
  proof (cases u = 0)
    case True
      then have charslength q' < k
        using charslength-of-butlast-0 q-in-dom q-split by blast
      with q'len have False by arith
      then show ?thesis by blast
  next
    case False
      then show ?thesis by arith
  qed
  have t-dom: t ∈ Z k u using last-token-of-path q'len q-in-dom q-split by blast
  have p ∈ P k (u − 1) ∨ p ∉ P k (u − 1) by blast
  then show ?thesis
  proof (induct rule: disjCases2)
    case 1
      with t-dom p-charslength admissible u have is-continuation k (u − 1) p [t]
        by (auto simp add: is-continuation-def)
      with u show p@[t] ∈ P k u
        by (metis One-nat-def Suc-pred is-continuation-in-P)
  next
    case 2
      from final-step-of-path'[OF p-in-dom 2]
      obtain p' ts where p': p = p' @ ts ∧ is-continuation k (u − 1) p' ts
        by blast
      from p' p-charslength is-continuation-def have charslength-ts: charslength ts
= 0
        by auto
      from u have u': Suc (u − 1) = u by arith
      have is-continuation k (u − 1) p' (ts@[t])
```

apply (*rule-tac is-continuation-continue*)
using p' **apply** *blast*
using *charslength-ts* **apply** *blast*
apply (*simp only*: u' *t-dom*)
using *admissible* p' **apply** *auto*
done
from *is-continuation-in-*$\mathcal{P}$[*OF this*] **show** *?case* **by** (*simp only*: p' u', *simp*)
qed
qed

lemma *is-prefix-admissible*:
assumes *is-prefix a b*
assumes *admissible b*
shows *admissible a*
proof −
from *assms* **show** *?thesis*
by (*auto simp add*: *is-prefix-def admissible-def* $\mathcal{L}_P$*-def*)
qed

lemma *butlast-split*: $n < length\ q \Longrightarrow butlast\ q = (take\ n\ q)@(drop\ n\ (butlast\ q))$
by (*metis append-take-drop-id take-butlast*)

lemma *in-*$\mathcal{P}$*-charslength*:
assumes *p-dom*: $p \in \mathcal{P}\ k\ u$
shows $\exists\ v.\ p \in \mathcal{P}\ (charslength\ p)\ v$
proof (*cases charslength* $p \geq k$)
case *True*
show *?thesis*
apply (*rule-tac* $x{=}u$ **in** *exI*)
by (*metis True le-neq-implies-less p-dom subsetCE subset-*$\mathcal{P}$)
next
case *False*
then have *charslength*: $charslength\ p < k$ **by** *arith*
have $p = [] \vee p \neq []$ **by** *blast*
thus *?thesis*
proof (*induct rule*: *disjCases2*)
case *1* **thus** *?case* **by** *simp*
next
case *2*
from *final-step-of-path*[*OF p-dom 2*] **obtain** $q\ ts\ k'\ u'$ **where**
step: $p = q\ @\ ts \wedge indexlt\ k'\ u'\ k\ u \wedge is\text{-}continuation\ k'\ u'\ q\ ts$ **by** *blast*
from *step* **have** k': $charslength\ q = k'$ **using** *is-continuation-def* **by** *blast*
from *step* **have** $charslength\ q \leq charslength\ p$ **by** *simp*
with k' **have** k': $k' \leq charslength\ p$ **by** *simp*
from *step* **have** $p \in \mathcal{P}\ k'\ (Suc\ u')$ **using** *is-continuation-in-*$\mathcal{P}$ **by** *blast*
with k' **have** $p \in \mathcal{P}\ (charslength\ p)\ (Suc\ u')$
by (*metis le-neq-implies-less subsetCE subset-*$\mathcal{P}$)
then show *?case* **by** *blast*
qed

```
qed

theorem general-compatibility:
  p ∈ 𝒫 k u ⟹ q ∈ 𝒫 k u ⟹ charslength p = charslength (take n q)
    ⟹ charslength p ≤ k ⟹ admissible (p @ (drop n q)) ⟹ p @ (drop n q) ∈
𝒫 k u
proof (induct length q − n arbitrary: p q n k u)
  case 0
    from 0 have 0 = length q − n by auto
    then have n: n ≥ length q by arith
    then have drop n q = [] by auto
    then show ?case by (simp add: 0.prems(1))
next
  case (Suc l)
    have n ≥ length q ∨ n < length q by arith
    then show ?case
    proof (induct rule: disjCases2)
      case 1
        then have drop n q = [] by auto
        then show ?case by (simp add: Suc.prems(1))
    next
      case 2
        then have length q > 0 by auto
        then have q-nonempty: q ≠ [] by auto
        let ?q′ = butlast q
        from q-nonempty Suc(2) have h1: l = length ?q′ − n by auto
        have h2: ?q′ ∈ 𝒫 k u
          by (metis Suc.prems(2) butlast-conv-take is-prefix-take prefixes-are-paths)
        have h3: charslength p = charslength (take n ?q′)
          using 2.hyps Suc.prems(3) take-butlast by force
        have is-prefix (p @ drop n ?q′) (p @ drop n q)
          by (simp add: butlast-conv-take drop-take)
        note h4 = is-prefix-admissible[OF this Suc.prems(5)]
        note induct = Suc(1)[OF h1 Suc(3) h2 h3 Suc.prems(4) h4]
        let ?p′ = p @ (drop n (butlast q))
        from induct have ?p′ ∈ 𝒫 k u .
        let ?i = charslength ?p′
        have charslength-i[symmetric]: charslength ?q′ = ?i
          using Suc.prems(3) apply simp
          apply (subst butlast-split[OF 2])
          by simp
        have q-split: q = ?q′@[last q] by (simp add: q-nonempty)
         with Suc.prems(2) charslength-of-butlast have charslength-q′: charslength
?q′ ≤ k
          by blast
        from q-nonempty have p′last: ?p′@[last q] = p@(drop n q)
          by (metis 2.hyps append-assoc drop-eq-Nil drop-keep-last not-le q-split)
        have ?i ≤ k by (simp only: charslength-i charslength-q′)
        then have ?i = k ∨ ?i < k by auto
```

then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have *charslength-q′*: *charslength ?q′* = *k* **using** *charslength-i*[*symmetric*] *1* **by** *blast*
from *compatibility-def*[*OF induct Suc.prems*(*2*) *1 q-split charslength-q′*]
show *?case* **by** (*simp only*: *p′last Suc.prems*(*5*))
next
case *2*
from *in-𝒫-charslength*[*OF induct*]
obtain *v1* **where** *v1*: *?p′* ∈ 𝒫 *?i v1* **by** *blast*
from *last-step-of-path*[*OF Suc.prems*(*2*) *q-split*]
have ∃ *u*. *q* ∈ 𝒫 *?i u* **by** (*metis charslength-i*)
then obtain *v2* **where** *v2*: *q* ∈ 𝒫 *?i v2* **by** *blast*
let *?v* = *max v1 v2*
have *v1* ≤ *?v* **by** *auto*
with *v1* **have** *dom1*: *?p′* ∈ 𝒫 *?i ?v* **by** (*metis* (*no-types*, *opaque-lifting*) *subsetCE subset-𝒫k*)
have *v2* ≤ *?v* **by** *auto*
with *v2* **have** *dom2*: *q* ∈ 𝒫 *?i ?v* **by** (*metis* (*no-types*, *opaque-lifting*) *subsetCE subset-𝒫k*)
from *compatibility-def*[*OF dom1 dom2 - q-split*]
have *p* @ *drop n q* ∈ 𝒫 *?i ?v*
by (*simp only*: *p′last charslength-i*[*symmetric*] *Suc.prems*(*5*))
then show *p* @ *drop n q* ∈ 𝒫 *k u* **by** (*meson 2.hyps subsetCE subset-𝒫*)
qed
qed
qed

lemma *wellformed-item-derives*:
assumes *wellformed*: *wellformed-item x*
shows *derives* [*item-nonterminal x*] (*item-rhs x*)
proof −
from *wellformed* **have** (*item-nonterminal x*, *item-rhs x*) ∈ 𝔑
by (*simp add*: *item-nonterminal-def item-rhs-def wellformed-item-def*)
then show *?thesis*
by (*metis append-Nil2 derives1-def derives1-implies-derives is-sentence-concat rule-α-type self-append-conv2*)
qed

lemma *wellformed-complete-item-β*:
assumes *wellformed*: *wellformed-item x*
assumes *complete*: *is-complete x*
shows *item-β x* = []
using *complete is-complete-def item-β-def* **by** *auto*

lemma *wellformed-complete-item-derives*:
assumes *wellformed*: *wellformed-item x*
assumes *complete*: *is-complete x*

shows *derives* [*item-nonterminal x*] (*item-α x*)
using *complete is-complete-def item-α-def wellformed wellformed-item-derives* **by** *auto*

lemma *is-derivation-implies-admissible*:
is-derivation (*terminals p* @ *δ*) $\Longrightarrow$ *is-word* (*terminals p*) $\Longrightarrow$ *admissible p*
using $\mathcal{L}_P$*-def admissible-def* **by** *blast*

lemma *item-rhs-of-inc-item*[*simp*]: *item-rhs* (*inc-item x k*) = *item-rhs x*
by (*auto simp add*: *inc-item-def item-rhs-def*)

lemma *item-rule-of-inc-item*[*simp*]: *item-rule* (*inc-item x k*) = *item-rule x*
by (*simp add*: *inc-item-def*)

lemma *item-origin-of-inc-item*[*simp*]: *item-origin* (*inc-item x k*) = *item-origin x*
by (*simp add*: *inc-item-def*)

lemma *item-end-of-inc-item*[*simp*]: *item-end* (*inc-item x k*) = *k*
by (*simp add*: *inc-item-def*)

lemma *item-dot-of-inc-item*[*simp*]: *item-dot* (*inc-item x k*) = (*item-dot x*) + *1*
by (*simp add*: *inc-item-def*)

lemma *item-nonterminal-of-inc-item*[*simp*]: *item-nonterminal* (*inc-item x k*) = *item-nonterminal x*
by (*simp add*: *inc-item-def item-nonterminal-def*)

lemma *wellformed-inc-item*:
assumes *wellformed*: *wellformed-item x*
assumes *next-symbol*: *next-symbol x* = *Some s*
assumes *k-upper-bound*: $k \leq$ *length Doc*
assumes *k-lower-bound*: $k \geq$ *item-end x*
shows *wellformed-item* (*inc-item x k*)
proof −
have *k-lower-bound′*: $k \geq$ *item-origin x*
using *k-lower-bound wellformed wellformed-item-def* **by** *auto*
show *?thesis*
apply (*auto simp add*: *wellformed-item-def k-upper-bound k-lower-bound′*)
using *wellformed wellformed-item-def* **apply** *blast*
using *is-complete-def next-symbol next-symbol-not-complete not-less-eq-eq* **by** *blast*
qed

lemma *item-α-of-inc-item*:
assumes *wellformed*: *wellformed-item x*
assumes *next-symbol*: *next-symbol x* = *Some s*
shows *item-α* (*inc-item x k*) = *item-α x* @ [*s*]
by (*metis* (*mono-tags*, *lifting*) *item-dot-of-inc-item item-rhs-of-inc-item*
One-nat-def add.right-neutral add-Suc-right is-complete-def item-α-def item-β-def

le-neq-implies-less list.sel(*1*) *next-symbol next-symbol-not-complete next-symbol-starts-item-β*

take-hd-drop wellformed wellformed-item-def)

lemma *derives1-pad*:
assumes *derives1*: *derives1* α β
assumes *u*: *is-sentence u*
assumes *v*: *is-sentence v*
shows *derives1* (*u*@α@*v*) (*u*@β@*v*)
proof −
from *derives1* **have**
∃ *x y N* δ. α = *x* @ [*N*] @ *y* ∧ β = *x* @ δ @ *y* ∧ *is-sentence x* ∧ *is-sentence y* ∧ (*N*, δ) ∈ 𝔑
by (*auto simp add*: *derives1-def*)
then obtain *x y N* δ **where**
1: α = *x* @ [*N*] @ *y* ∧ β = *x* @ δ @ *y* ∧ *is-sentence x* ∧ *is-sentence y* ∧ (*N*, δ) ∈ 𝔑 **by** *blast*
show *?thesis*
apply (*simp only*: *derives1-def*)
apply (*rule-tac x*=*u*@*x* **in** *exI*)
apply (*rule-tac x*=*y*@*v* **in** *exI*)
apply (*rule-tac x*=*N* **in** *exI*)
apply (*rule-tac x*=δ **in** *exI*)
using *1 u v is-sentence-concat* **by** *auto*
qed

lemma *derives-pad*:
derives α β ⟹ *is-sentence u* ⟹ *is-sentence v* ⟹ *derives* (*u*@α@*v*) (*u*@β@*v*)
proof (*induct rule*: *derives-induct*)
case *Base* **thus** *?case* **by** *simp*
next
case (*Step y z*)
from *Step* **have** *1*: *derives* (*u*@α@*v*) (*u*@*y*@*v*) **by** *auto*
from *Step* **have** *2*: *derives1 y z* **by** *auto*
then have *derives1* (*u*@*y*@*v*) (*u*@*z*@*v*) **by** (*simp add*: *Step.prems derives1-pad*)

then show *?case*
using *1 derives1-implies-derives derives-trans* **by** *blast*
qed

lemma *derives1-is-sentence*: *derives1* α β ⟹ *is-sentence* α ∧ *is-sentence* β
using *Derives1-sentence1 Derives1-sentence2 derives1-implies-Derives1* **by** *blast*

lemma *derives-is-sentence*: *derives* α β ⟹ (α = β) ∨ (*is-sentence* α ∧ *is-sentence* β)
proof (*induct rule*: *derives-induct*)
case *Base* **thus** *?case* **by** *simp*
next

case (*Step y z*)
  show *?case* **using** *Step.hyps*(*2*) *Step.hyps*(*3*) *derives1-is-sentence* **by** *blast*
qed

lemma *derives-append*:
  assumes *au*: *derives a u*
  assumes *bv*: *derives b v*
  assumes *is-sentence-a*: *is-sentence a*
  assumes *is-sentence-b*: *is-sentence b*
  shows *derives* (*a*@*b*) (*u*@*v*)
proof −
  from *au* **have** $a = u \vee$ (*is-sentence a* $\wedge$ *is-sentence u*)
    using *derives-is-sentence* **by** *blast*
  then have *au-sentences*: *is-sentence a* $\wedge$ *is-sentence u* **using** *is-sentence-a* **by** *blast*
  from *bv* **have** $b = v \vee$ (*is-sentence b* $\wedge$ *is-sentence v*)
    using *derives-is-sentence* **by** *blast*
  then have *bv-sentences*: *is-sentence b* $\wedge$ *is-sentence v* **using** *is-sentence-b* **by** *blast*
  have *1*: *derives* (*a*@*b*) (*u*@*b*)
    apply (*rule-tac derives-pad*[*OF au*, **where** *u*=[], *simplified*])
    using *is-sentence-b* **by** *auto*
  have *2*: *derives* (*u*@*b*) (*u*@*v*)
    apply (*rule-tac derives-pad*[*OF bv*, **where** *v*=[], *simplified*])
    apply (*simp add*: *au-sentences*)
    done
  from *1 2 derives-trans* **show** *?thesis* **by** *blast*
qed

lemma *is-sentence-item-*α: *wellformed-item x* $\Longrightarrow$ *is-sentence* (*item-*α *x*)
  by (*metis is-sentence-take item-*α*-def item-rhs-def prod.collapse rule-*α*-type wellformed-item-def*)

lemma *is-nonterminal-item-nonterminal*: *wellformed-item x* $\Longrightarrow$ *is-nonterminal* (*item-nonterminal x*)
  by (*metis item-nonterminal-def prod.collapse rule-nonterminal-type wellformed-item-def*)

lemma *Complete-elem-in-Gen*:
  assumes *I-in-Gen*: $I \subseteq$ *Gen* ($\mathcal{P}$ *k u*)
  assumes *k*: $k \leq$ *length Doc*
  assumes *x-in-Complete*: $x \in$ *Complete k I*
  shows $x \in$ *Gen* ($\mathcal{P}$ *k u*)
proof −
  let *?P* = $\mathcal{P}$ *k u*
  from *x-in-Complete* **have** $x \in I \vee$ ($\exists$ *x1 x2*. *x* = *inc-item x1 k* $\wedge$
    x1 $\in$ *bin I* (*item-origin x2*) $\wedge$ *x2* $\in$ *bin I k* $\wedge$ *is-complete x2* $\wedge$
    next-symbol x1 = *Some* (*item-nonterminal x2*))
    by (*auto simp add*: *Complete-def*)
  then show *?thesis*

```
  proof (induct rule: disjCases2)
    case 1 thus ?case using I-in-Gen subsetCE by blast
  next
    case 2
    then obtain x1 x2 where x12: x = inc-item x1 k ∧
      x1 ∈ bin I (item-origin x2) ∧ x2 ∈ bin I k ∧ is-complete x2 ∧
      next-symbol x1 = Some (item-nonterminal x2) by blast
    from x12 have ∃ p1 p2. p1 ∈ ?P ∧ pvalid p1 x1 ∧ p2 ∈ ?P ∧ pvalid p2 x2
      by (meson Gen-implies-pvalid I-in-Gen bin-elem subsetCE)
     then obtain p1 p2 where p1: p1 ∈ ?P ∧ pvalid p1 x1 and p2: p2 ∈ ?P ∧
pvalid p2 x2
      by blast
    from p1 obtain w δ where p1valid:
      wellformed-tokens p1 ∧
       wellformed-item x1 ∧
       w ≤ length p1 ∧
       charslength p1 = item-end x1 ∧
       charslength (take w p1) = item-origin x1 ∧
       is-derivation (terminals (take w p1) @ [item-nonterminal x1] @ δ) ∧
       derives (item-α x1) (terminals (drop w p1))
       using pvalid-def by blast
    from p2 obtain y γ where p2valid:
      wellformed-tokens p2 ∧
       wellformed-item x2 ∧
       y ≤ length p2 ∧
       charslength p2 = item-end x2 ∧
       charslength (take y p2) = item-origin x2 ∧
       is-derivation (terminals (take y p2) @ [item-nonterminal x2] @ γ) ∧
       derives (item-α x2) (terminals (drop y p2))
       using pvalid-def by blast
    let ?r = p1 @ (drop y p2)
    have charslength-p1-eq: charslength p1 = item-end x1 by (simp only: p1valid)
    from x12 have item-end-x1: item-end x1 = item-origin x2
      using bin-def mem-Collect-eq by blast
    have item-end-x2: item-end x2 = k using bin-def x12 by blast
    then have charslength-p1-leq: charslength p1 ≤ k
      using charslength-p1-eq item-end-x1 p2valid wellformed-item-def by auto
    have ∃δ'. item-β x1 = [item-nonterminal x2] @ δ'
      by (simp add: next-symbol-starts-item-β p1valid x12)
    then obtain δ' where δ': item-β x1 = [item-nonterminal x2] @ δ' by blast
    have is-derivation ((terminals (take w p1))@(item-rhs x1)@δ)
      using is-derivation-derives p1valid wellformed-item-derives by blast
   then have is-derivation ((terminals (take w p1))@(item-α x1 @ item-β x1)@δ)
      by (simp add: item-rhs-split)
    then have is-derivation ((terminals (take w p1))@((terminals (drop w p1)) @
item-β x1)@δ)
      using is-derivation-derives p1valid by auto
    then have is-derivation ((terminals p1)@(item-β x1)@δ)
      by (metis append-assoc append-take-drop-id terminals-append)
```

then have *is-derivation* ((*terminals p1*)@([*item-nonterminal x2*] @ δ')@δ)
using *is-derivation-derives* δ' **by** *auto*
then have *is-derivation* ((*terminals p1*)@(*terminals* (*drop y p2*)) @ δ' @δ)
using *is-complete-def is-derivation-derives is-derivation-step item-α-def item-nonterminal-def item-rhs-def p2valid wellformed-item-def x12* **by** *auto*
then have *is-derivation* (*terminals* (*p1* @ (*drop y p2*)) @ (δ' @ δ)) **by** *simp*
then have *admissible-r*: *admissible* (*p1* @ (*drop y p2*))
apply (*rule-tac is-derivation-implies-admissible*)
apply *auto*
apply (*rule is-word-terminals*)
apply (*simp add*: *p1valid*)
using *p2valid* **using** *is-word-terminals-drop terminals-drop* **by** *auto*
have *r-in-dom*: *?r* ∈ $\mathcal{P}$ *k u*
apply (*rule-tac general-compatibility*)
apply (*simp add*: *p1*)
apply (*simp add*: *p2*)
apply (*simp only*: *p2valid charslength-p1-eq item-end-x1*)
apply (*simp only*: *charslength-p1-leq*)
by (*simp add*: *admissible-r*)
have *wellformed-r*: *wellformed-tokens ?r*
using *admissible-r admissible-wellformed-tokens* **by** *blast*
have *wellformed-x*: *wellformed-item x*
apply (*simp add*: *x12*)
apply (*rule-tac wellformed-inc-item*)
apply (*simp add*: *p1valid*)
apply (*simp add*: *x12*)
apply (*simp add*: *k*)
using *charslength-p1-eq charslength-p1-leq* **by** *auto*
have *charslength-p1-as-p2*: *charslength p1* = *charslength* (*take y p2*)
using *charslength-p1-eq item-end-x1 p2valid* **by** *linarith*
then have *charslength-r*: *charslength ?r* = *item-end x*
apply (*simp add*: *x12*)
apply (*subst length-append*[*symmetric*])
apply (*subst chars-append*[*symmetric*])
apply (*subst append-take-drop-id*)
using *item-end-x2 p2valid* **by** *auto*
have *item-α-x*: *item-α x* = *item-α x1* @ [*item-nonterminal x2*]
using *x12 p1valid* **by** (*simp add*: *item-α-of-inc-item*)
from *p2valid* **have** *derives-item-nonterminal-x2*:
derives [*item-nonterminal x2*] (*terminals* (*drop y p2*))
using *derives-trans wellformed-complete-item-derives x12* **by** *blast*
have *pvalid ?r x*
apply (*auto simp only*: *pvalid-def*)
apply (*rule-tac x=w* **in** *exI*)
apply (*rule-tac x=δ* **in** *exI*)
apply (*auto simp only*:)
apply (*simp add*: *wellformed-r*)
apply (*simp add*: *wellformed-x*)
using *p1valid* **apply** *simp*

apply (*simp only*: *charslength-r*)
using *x12 p1valid* **apply** *simp*
using *x12 p1valid* **apply** *simp*
apply (*simp add*: *item-α-x*)
apply (*rule-tac derives-append*)
using *p1valid* **apply** *simp*
using *derives-item-nonterminal-x2 p1valid* **apply** *auto*[*1*]
using *is-sentence-item-α p1valid* **apply** *blast*
using *is-derivation-is-sentence is-sentence-concat p2valid* **by** *blast*
with *r-in-dom* **show** *?case* **using** *Gen-def mem-Collect-eq* **by** *blast*
qed
qed

lemma *Complete-subset-Gen*:
assumes *I-in-Gen-P*: *I* ⊆ *Gen* (𝒫 *k u*)
assumes *k*: *k* ≤ *length Doc*
shows *Complete k I* ⊆ *Gen* (𝒫 *k u*)
using *Complete-elem-in-Gen I-in-Gen-P k* **by** *blast*

lemma 𝒫*-are-admissible*: *p* ∈ 𝒫 *k u* ⟹ *admissible p*
apply (*rule-tac* 𝔓*-are-admissible*)
using 𝔓*-covers-*𝒫 *subsetCE* **by** *blast*

lemma *is-continuation-base*:
assumes *p-dom*: *p* ∈ 𝒫 *k u*
assumes *charslength-p*: *charslength p* = *k*
shows *is-continuation k u p* []
apply (*auto simp add*: *is-continuation-def*)
apply (*simp add*: *p-dom*)
using *charslength-p* **apply** *simp*
using 𝒫*-are-admissible p-dom* **by** *blast*

lemma *is-continuation-empty-chars*:
is-continuation k u q ts ⟹ *charslength* (*q*@*ts*) = *k* ⟹ *chars ts* = []
by (*simp add*: *is-continuation-def*)

lemma 𝒵*-subset*: *u* ≤ *v* ⟹ 𝒵 *k u* ⊆ 𝒵 *k v*
using 𝒵*-subset-Suc subset-fSuc* **by** *blast*

lemma *is-continuation-increase-u*:
assumes *cont*: *is-continuation k u q ts*
assumes *uv*: *u* ≤ *v*
shows *is-continuation k v q ts*
proof −
have *q* ∈ 𝒫 *k u* **using** *cont is-continuation-def* **by** *blast*
with *uv* **have** *q-dom*: *q* ∈ 𝒫 *k v* **by** (*meson subsetCE subset-*𝒫*k*)
from *uv* **have** 𝒵: ⋀ *t*. *t* ∈ 𝒵 *k* (*Suc u*) ⟹ *t* ∈ 𝒵 *k* (*Suc v*)
using 𝒵*-subset le-neq-implies-less less-imp-le-nat not-less-eq subsetCE* **by** *blast*

```
  show ?thesis
    apply (auto simp only: is-continuation-def)
    apply (simp add: q-dom)
    using cont is-continuation-def apply simp
    using cont is-continuation-def apply simp
    using cont is-continuation-def 2 apply simp
    using cont is-continuation-def apply (simp only:)
    done
qed

lemma pvalid-next-symbol-derivable:
  assumes pvalid: pvalid p x
  assumes next-symbol: next-symbol x = Some s
  shows ∃ δ. is-derivation((terminals p)@[s]@δ)
proof −
  from pvalid pvalid-def have wellformed-x: wellformed-item x by auto
  from next-symbol-starts-item-β[OF wellformed-x next-symbol]
  obtain ω where ω: item-β x = [s] @ ω by auto
  from pvalid have ∃ γ. is-derivation((terminals p)@(item-β x)@γ)
    using pvalid-is-derivation-terminals-item-β by blast
  then obtain γ where is-derivation((terminals p)@(item-β x)@γ) by blast
  with ω have is-derivation((terminals p)@[s]@ω@γ) by auto
  then show ?thesis by blast
qed

lemma pvalid-admissible:
  assumes pvalid: pvalid p x
  shows admissible p
proof −
  have ∃ δ. is-derivation((terminals p)@(item-β x)@δ)
    by (simp add: pvalid pvalid-is-derivation-terminals-item-β)
  then obtain δ where δ: is-derivation((terminals p)@(item-β x)@δ) by blast
  have is-word: is-word (terminals p)
    using pvalid-def is-word-terminals pvalid by blast
  show ?thesis using δ is-derivation-implies-admissible is-word by blast
qed

lemma pvalid-next-terminal-admissible:
  assumes pvalid: pvalid p x
  assumes next-symbol: next-symbol x = Some t
  assumes terminal: is-terminal t
  shows admissible (p@[(t, c)])
proof −
  have is-word (terminals p)
    using is-word-terminals pvalid pvalid-def by blast
  then show ?thesis
   using is-derivation-implies-admissible next-symbol pvalid pvalid-next-symbol-derivable

     terminal by fastforce
```

qed

lemma *𝒳-wellformed*: $t \in \mathcal{X}\ k \Longrightarrow$ *wellformed-token t*
by (*simp add*: *𝒳-are-terminals wellformed-token-def*)

lemma *𝒵-wellformed*: $t \in \mathcal{Z}\ k\ u \Longrightarrow$ *wellformed-token t*
using *𝒳-wellformed 𝒵-subset-𝒳* **by** *blast*

lemma *Scan-elem-in-Gen*:
assumes *I-in-Gen*: $I \subseteq Gen\ (\mathcal{P}\ k\ u)$
assumes *k*: $k \leq length\ Doc$
assumes *T*: $T \subseteq \mathcal{Z}\ k\ u$
assumes *x-in-Scan*: $x \in Scan\ T\ k\ I$
shows $x \in Gen\ (\mathcal{P}\ k\ u)$
proof −
have $u = 0 \Longrightarrow x \in I$
proof −
assume $u = 0$
then have $\mathcal{Z}\ k\ u = \{\}$ **by** *simp*
then have $T = \{\}$ **using** *T* **by** *blast*
then have $Scan\ T\ k\ I = I$ **by** (*simp add*: *Scan-empty*)
then show $x \in I$ **using** *x-in-Scan* **by** *simp*
qed
then have $x \in I \vee (u > 0 \wedge (\exists\ y\ t\ c.\ x = inc\text{-}item\ y\ (k + length\ c) \wedge y \in bin\ I\ k \wedge$
$(t, c) \in T \wedge next\text{-}symbol\ y = Some\ t))$ **using** *x-in-Scan Scan-def* **by** *auto*
then show *?thesis*
proof (*induct rule*: *disjCases2*)
case *1* **thus** *?case* **using** *I-in-Gen* **by** *blast*
next
case *2*
then obtain *y t c* **where** *x-is-scan*: $x = inc\text{-}item\ y\ (k + length\ c) \wedge y \in bin\ I\ k \wedge$
$(t, c) \in T \wedge next\text{-}symbol\ y = Some\ t$ **by** *blast*
have *u-gt-0*: $0 < u$ **using** *2* **by** *blast*
have $\exists\ p \in \mathcal{P}\ k\ u.\ pvalid\ p\ y$ **using** *Gen-implies-pvalid I-in-Gen bin-elem x-is-scan* **by** *blast*
then obtain *p* **where** *p*: $p \in \mathcal{P}\ k\ u \wedge pvalid\ p\ y$ **by** *blast*
have *p-dom*: $p \in \mathcal{P}\ k\ u$ **using** *p* **by** *blast*
from *p pvalid-def x-is-scan* **have** *charslength-p*: $charslength\ p = k$
using *bin-def mem-Collect-eq* **by** *auto*
obtain *tok* **where** *tok*: $tok = (t, c)$ **using** *x-is-scan* **by** *blast*
have *tok-dom*: $tok \in \mathcal{Z}\ k\ u$ **using** *tok x-is-scan T* **by** *blast*
have $p = [] \vee p \neq []$ **by** *blast*
then have $\exists\ q\ ts\ u'.\ p = q@ts \wedge u' < u \wedge charslength\ ts = 0 \wedge is\text{-}continuation\ k\ u'\ q\ ts$
proof (*induct rule*: *disjCases2*)
case *1* **thus** *?case*
apply (*rule-tac x=p* **in** *exI*)

apply (*rule-tac x*=[] **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*simp add*: *2 is-continuation-def*)
using *charslength-p* **by** *simp*
next
case *2*
from *final-step-of-path*[*OF p-dom 2*] **obtain** q ts k' u'
where *final-step*: $p = q$ @ $ts \wedge$ *indexlt* k' u' k $u \wedge$ *is-continuation* k' u' q ts
by *blast*
then have $k' \leq k$ **using** *indexlt-simp* **by** *auto*
then have $k' < k \vee k' = k$ **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have $p \in \mathcal{P}$ k' (*Suc* u') **using** *final-step is-continuation-in-*$\mathcal{P}$ **by** *blast*
then have *p-dom*: $p \in \mathcal{P}$ k *0* **by** (*meson 1 subsetCE subset-*$\mathcal{P}$)
with *charslength-p* **have** *is-continuation k 0 p* [] **using** *is-continuation-base*
by *blast*
then show *?case*
apply (*rule-tac x*=*p* **in** *exI*)
apply (*rule-tac x*=[] **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*simp add*: *u-gt-0*)
done
next
case *2*
with *final-step indexlt-simp* **have** $u' < u$ **by** *auto*
then show *?case*
apply (*rule-tac x*=*q* **in** *exI*)
apply (*rule-tac x*=*ts* **in** *exI*)
apply (*rule-tac x*=u' **in** *exI*)
using *final-step 2* **apply** *auto*
using *charslength-p is-continuation-empty-chars* **by** *blast*
qed
qed
then obtain q ts u' **where**
p-split: $p = q$@$ts \wedge u' < u \wedge$ *charslength* $ts = 0 \wedge$ *is-continuation* k u' q ts
by *blast*
then have $\exists\ u''.\ u' \leq u'' \wedge$ *Suc* $u'' = u$ **by** (*auto*, *arith*)
then obtain u'' **where** u'': $u' \leq u'' \wedge$ *Suc* $u'' = u$ **by** *blast*
with *p-split* **have** *cont-*u'': *is-continuation* k u'' q ts
using *is-continuation-increase-u* **by** *blast*
have *admissible*: *admissible* (*p*@[*tok*])
apply (*simp add*: *tok*)
apply (*rule-tac pvalid-next-terminal-admissible*[**where** *x*=*y*])
apply (*simp add*: *p*)
apply (*simp add*: *x-is-scan*)
using $\mathcal{Z}$*-wellformed tok tok-dom wellformed-token-def* **by** *auto*
have *is-continuation* k u'' q (*ts*@[*tok*])

apply (*rule is-continuation-continue*)
apply (*simp add*: *cont-u′′*)
using *p-split* **apply** *simp*
using *u′′ tok-dom* **apply** *simp*
using *admissible p-split* **by** *auto*
with *p-split u′′* **have** *ptok-dom*: *p*@[*tok*] ∈ 𝒫 *k u*
using *append-assoc is-continuation-in-𝒫* **by** *auto*
from *p* **obtain** *i* γ **where** *valid*:
wellformed-tokens p ∧
wellformed-item y ∧
i ≤ *length p* ∧
charslength p = *item-end y* ∧
charslength (*take i p*) = *item-origin y* ∧
is-derivation (*terminals* (*take i p*) @ [*item-nonterminal y*] @ γ) ∧
derives (*item-α y*) (*terminals* (*drop i p*)) **using** *pvalid-def* **by** *blast*
have *clen-ptok*: *k* + *length c* = *charslength* (*p*@[*tok*])
using *charslength-p tok* **by** *simp*
from *ptok-dom* **have** *ptok-doc-tokens*: *doc-tokens* (*p*@[*tok*])
using 𝔓*-are-doc-tokens* 𝔓*-covers-𝒫 rev-subsetD* **by** *blast*
have *wellformed-x*: *wellformed-item x*
apply (*simp add*: *x-is-scan*)
apply (*rule-tac wellformed-inc-item*)
apply (*simp add*: *valid*)
apply (*simp add*: *x-is-scan*)
apply (*simp only*: *clen-ptok*)
using *ptok-doc-tokens charslength.simps doc-tokens-length* **apply** *presburger*
apply (*simp only*: *clen-ptok*)
using *valid* **by** *auto*
have *pvalid* (*p*@[*tok*]) *x*
apply (*auto simp only*: *pvalid-def*)
apply (*rule-tac x*=*i* **in** *exI*)
apply (*rule-tac x*=γ **in** *exI*)
apply (*auto simp only*:)
using *ptok-dom admissible admissible-wellformed-tokens* **apply** *blast*
apply (*simp add*: *wellformed-x*)
using *valid* **apply** *simp*
apply (*simp add*: *x-is-scan clen-ptok*)
using *valid* **apply** (*simp add*: *x-is-scan*)
using *valid* **apply** (*simp add*: *x-is-scan*)
using *valid* **apply** (*simp add*: *x-is-scan*)
apply (*subst item-α-of-inc-item*)
using *valid* **apply** *simp*
using *x-is-scan* **apply** *simp*
apply (*rule-tac derives-append*)
apply *simp*
apply (*simp add*: *tok*)
using *is-sentence-item-α* **apply** *blast*
by (*meson pvalid-next-symbol-derivable LocalLexing-axioms is-derivation-is-sentence*

```
      is-sentence-concat p x-is-scan)
    with ptok-dom show ?thesis
      using Gen-def mem-Collect-eq by blast
  qed
qed

lemma Scan-subset-Gen:
  assumes I-in-Gen: I ⊆ Gen (𝒫 k u)
  assumes k: k ≤ length Doc
  assumes T: T ⊆ 𝒵 k u
  shows Scan T k I ⊆ Gen (𝒫 k u)
using I-in-Gen Scan-elem-in-Gen T k by blast

theorem thmD5:
  assumes I: I ⊆ Gen (𝒫 k u)
  assumes k: k ≤ length Doc
  assumes T: T ⊆ 𝒵 k u
  shows π k T I ⊆ Gen (𝒫 k u)
apply (simp add: π-def)
apply (rule-tac limit-upperbound)
using I k T Predict-subset-Gen Complete-subset-Gen Scan-subset-Gen apply metis
by (simp add: I)

end

end
theory TheoremD6
imports TheoremD5
begin

context LocalLexing begin

definition inc-dot :: nat ⇒ item ⇒ item
where
  inc-dot d x = Item (item-rule x) (item-dot x + d) (item-origin x) (item-end x)

lemma inc-dot-0[simp]: inc-dot 0 x = x
  by (simp add: inc-dot-def)

lemma Predict-mk-regular1:
  ∃ (P :: rule ⇒ item ⇒ bool) F. Predict k = mk-regular1 P F
proof −
  let ?P = λ r x::item. r ∈ ℜ ∧ item-end x = k ∧ next-symbol x = Some(fst r)
  let ?F = λ r (x::item). init-item r k
  show ?thesis
    apply (rule-tac x=?P in exI)
    apply (rule-tac x=?F in exI)
    apply (rule-tac ext)
    by (auto simp add: mk-regular1-def bin-def Predict-def)
```

qed

lemma *Complete-mk-regular2*:
  ∃ (*P* :: *dummy* ⇒ *item* ⇒ *item* ⇒ *bool*) *F*. *Complete k* = *mk-regular2 P F*
proof −
  let *?P* = λ (*r*::*dummy*) *x y*. *item-end x* = *item-origin y* ∧ *item-end y* = *k* ∧ *is-complete y* ∧
      next-symbol x = *Some* (*item-nonterminal y*)
  let *?F* = λ (*r*::*dummy*) *x y*. *inc-item x k*
  show *?thesis*
    apply (*rule-tac x*=*?P* **in** *exI*)
    apply (*rule-tac x*=*?F* **in** *exI*)
    apply (*rule-tac ext*)
    by (*auto simp add*: *mk-regular2-def bin-def Complete-def*)
qed

lemma *Scan-mk-regular1*:
  ∃ (*P* :: *token* ⇒ *item* ⇒ *bool*) *F*. *Scan T k* = *mk-regular1 P F*
proof −
  let *?P* = λ (*tok*::*token*) (*x*::*item*). *item-end x* = *k* ∧ *tok* ∈ *T* ∧ *next-symbol x* = *Some* (*fst tok*)
  let *?F* = λ (*tok*::*token*) (*x*::*item*). *inc-item x* (*k* + *length* (*snd tok*))
  show *?thesis*
    apply (*rule-tac x*=*?P* **in** *exI*)
    apply (*rule-tac x*=*?F* **in** *exI*)
    apply (*rule-tac ext*)
    by (*auto simp add*: *mk-regular1-def bin-def Scan-def*)
qed

lemma *Predict-regular*: *regular* (*Predict k*)
  by (*metis Predict-mk-regular1 regular1*)

lemma *Complete-regular*: *regular* (*Complete k*)
  by (*metis Complete-mk-regular2 regular2*)

lemma *Scan-regular*: *regular* (*Scan T k*)
  by (*metis Scan-mk-regular1 regular1*)

lemma *π-functional*: π *k T* = *limit* ((*Scan T k*) *o* (*Complete k*) *o* (*Predict k*))
proof −
  have π *k T* = *limit* (λ *I*. *Scan T k* (*Complete k* (*Predict k I*)))
    using *π-def* **by** *blast*
  moreover have (λ *I*. *Scan T k* (*Complete k* (*Predict k I*))) =
      (*Scan T k*) *o* (*Complete k*) *o* (*Predict k*)
      apply (*rule ext*)
      by *simp*
  ultimately show *?thesis* **by** *simp*
qed

lemma *π-step-regular*: *regular* ((*Scan T k*) *o* (*Complete k*) *o* (*Predict k*))
  by (*simp add*: *Complete-regular Predict-regular Scan-regular regular-comp*)

lemma *π-regular*: *regular* (*π k T*)
  by (*simp add*: *π-functional π-step-regular regular-limit*)

lemma *π-fix*: *Scan T k* (*Complete k* (*Predict k* (*π k T I*))) = *π k T I*
  using *π-functional π-step-regular regular-fixpoint* **by** *fastforce*

lemma *π-fix′*: ((*Scan T k*) *o* (*Complete k*) *o* (*Predict k*)) (*π k T I*) = *π k T I*
  using *π-functional π-step-regular regular-fixpoint* **by** *fastforce*

lemma *setmonotone-cases*:
  assumes *setmonotone f*
  shows *f X* = *X* ∨ *X* ⊂ *f X*
using *assms elem-setmonotone* **by** *fastforce*

lemma *distribute-fixpoint-over-setmonotone-comp*:
  assumes *f*: *setmonotone f*
  assumes *g*: *setmonotone g*
  assumes *fixpoint*: (*f o g*) *I* = *I*
  shows *f I* = *I* ∧ *g I* = *I*
proof −
  from *setmonotone-cases*[*OF g*, **where** *X*=*I*] **show** *?thesis*
  proof(*induct rule*: *disjCases2*)
    case *1*
      thus *?case* **using** *fixpoint* **by** *simp*
  next
    case *2*
      with *f* **have** *I* ⊂ (*f o g*) *I*
        by (*metis comp-apply fixpoint less-asym′ setmonotone-cases*)
      with *fixpoint* **have** *False* **by** *simp*
      then show *?case* **by** *blast*
  qed
qed

lemma *distribute-fixpoint-over-setmonotone-comp-3*:
  assumes *f*: *setmonotone f*
  assumes *g*: *setmonotone g*
  assumes *h*: *setmonotone h*
  assumes *fixpoint*: (*f o g o h*) *I* = *I*
  shows *f I* = *I* ∧ *g I* = *I* ∧ *h I* = *I*
by (*meson distribute-fixpoint-over-setmonotone-comp f fixpoint g h setmonotone-comp*)

lemma *Predict-π-fix*: *Predict k* (*π k T I*) = *π k T I*
by (*meson Complete-regular Predict-regular Scan-regular π-fix′*
  distribute-fixpoint-over-setmonotone-comp-3 regular-implies-setmonotone)

lemma *Scan-π-fix*: *Scan T k* (*π k T I*) = *π k T I*

by (*meson Complete-regular Predict-regular Scan-regular* π*-fix′*
distribute-fixpoint-over-setmonotone-comp-3 regular-implies-setmonotone)

lemma *Complete-*π*-fix*: *Complete k* (π *k T I*) = π *k T I*
by (*meson Complete-regular Predict-regular Scan-regular* π*-fix′*
distribute-fixpoint-over-setmonotone-comp-3 regular-implies-setmonotone)

lemma π*-idempotent*: π *k T* (π *k T I*) = π *k T I*
by (*simp add*: π*-functional* π*-step-regular limit-is-idempotent*)

lemma *derivation-shift-identity*[*simp*]: *derivation-shift D 0 0* = *D*
by (*simp add*: *derivation-shift-def*)

lemma *Derivation-skip-prefix*: *Derivation* (*u*@*v*) *D w* ⟹ *derivation-ge D* (*length u*) ⟹
Derivation v (*derivation-shift D* (*length u*) *0*) (*drop* (*length u*) *w*)
proof (*induct D arbitrary*: *u v w*)
case *Nil*
thus *?case* **by** (*simp add*: *append-eq-conv-conj*)
next
case (*Cons d D*)
from *Cons* **have** ∃*x*. *Derives1* (*u*@*v*) (*fst d*) (*snd d*) *x* ∧ *Derivation x D w* **by** *auto*
then obtain *x* **where** *x*: *Derives1* (*u*@*v*) (*fst d*) (*snd d*) *x* ∧ *Derivation x D w* **by** *blast*
from *Cons* **have** *d*: *fst d* ≥ *length u* **and** *D*: *derivation-ge D* (*length u*)
using *derivation-ge-cons* **apply** *blast*
using *Cons.prems*(*2*) *derivation-ge-cons* **by** *blast*
have ∃ *x′*. *x* = *u*@*x′* **by** (*metis append-eq-conv-conj d le-Derives1-take x*)
then obtain *x′* **where** *x′*: *x* = *u*@*x′* **by** *blast*
show *?case*
apply *simp*
apply (*rule-tac x*=*x′* **in** *exI*)
using *Cons.hyps D Derives1-skip-prefix d x x′* **by** *blast*
qed

lemma *leftmost-skip-prefix*: *leftmost i* (*u*@*v*) ⟹ *i* ≥ *length u* ⟹ *leftmost* (*i* − *length u*) *v*
by (*simp add*: *leftmost-def less-diff-conv2 nth-append*)

lemma *LeftDerivation-skip-prefix*: *LeftDerivation* (*u*@*v*) *D w* ⟹ *derivation-ge D* (*length u*) ⟹
LeftDerivation v (*derivation-shift D* (*length u*) *0*) (*drop* (*length u*) *w*)
proof (*induct D arbitrary*: *u v w*)
case *Nil*
thus *?case* **by** (*simp add*: *append-eq-conv-conj*)
next
case (*Cons d D*)
from *Cons* **have** ∃*x*. *LeftDerives1* (*u*@*v*) (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x*

D w **by** *auto*
then obtain *x* **where** *x*: *LeftDerives1* (*u*@*v*) (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x D w* **by** *blast*
from *Cons* **have** *d*: *fst d* ≥ *length u* **and** *D*: *derivation-ge D* (*length u*)
using *derivation-ge-cons* **apply** *blast*
using *Cons.prems*(*2*) *derivation-ge-cons* **by** *blast*
have ∃ *x'*. *x* = *u*@*x'*
by (*metis LeftDerives1-implies-Derives1 append-eq-conv-conj d le-Derives1-take x*)
then obtain *x'* **where** *x'*: *x* = *u*@*x'* **by** *blast*
have *leftmost*: *leftmost* (*fst d*) (*u*@*v*) **using** *LeftDerives1-def x* **by** *blast*
have *1*: *LeftDerives1 v* (*fst d* − *length u*) (*snd d*) *x'*
apply (*auto simp add*: *LeftDerives1-def*)
apply (*simp add*: *leftmost d leftmost-skip-prefix*)
using *Derives1-skip-prefix LeftDerives1-implies-Derives1 d x x'* **by** *blast*
have *2*: *LeftDerivation x'* (*derivation-shift D* (*length u*) *0*) (*drop* (*length u*) *w*)
using *Cons.hyps D x x'* **by** *blast*
show *?case*
apply *simp*
apply (*rule-tac x*=*x'* **in** *exI*)
using *1 2* **by** *blast*
qed

lemma *splits-at-append*: *splits-at u i u1 N u2* ⟹ *splits-at* (*u*@*v*) *i u1 N* (*u2*@*v*)
by (*auto simp add*: *splits-at-def*)

lemma *LeftDerives1-append-leftmost-unique*: *LeftDerives1* (*a*@*b*) *i r c* ⟹ *leftmost j a* ⟹ *i* = *j*
by (*meson LeftDerives1-def leftmost-cons-less leftmost-def leftmost-unique*)

lemma *drop-derivation-shift*:
drop n (*derivation-shift D left right*) = *derivation-shift* (*drop n D*) *left right*
by (*auto simp add*: *derivation-shift-def drop-map*)

lemma *take-derivation-shift*:
take n (*derivation-shift D left right*) = *derivation-shift* (*take n D*) *left right*
by (*auto simp add*: *derivation-shift-def take-map*)

lemma *derivation-shift-0-shift*: *derivation-shift* (*derivation-shift D left1 0*) *left2 right2* =
derivation-shift D (*left1* + *left2*) *right2*
by (*auto simp add*: *derivation-shift-def*)

lemma *splits-at-append-prefix*:
splits-at v i α *N* β ⟹ *splits-at* (*u*@*v*) (*i* + *length u*) (*u*@α) *N* β
apply (*auto simp add*: *splits-at-def*)
by (*simp add*: *nth-append*)

lemma *splits-at-implies-Derives1*: *splits-at* δ *i* α *N* β ⟹ *is-sentence* δ ⟹ *r*∈ $\mathfrak{R}$

```
⟹ fst r = N
 ⟹ Derives1 δ i r (α@(snd r)@β)
by (metis (no-types, lifting) Derives1-def is-sentence-concat length-take
  less-or-eq-imp-le min.absorb2 prod.collapse splits-at-combine splits-at-def)

lemma Derives1-append-prefix:
  assumes Derives1: Derives1 v i r w
  assumes u: is-sentence u
  shows Derives1 (u@v) (i + length u) r (u@w)
proof −
  have ∃ α N β. splits-at v i α N β using assms splits-at-ex by auto
  then obtain α N β where split-v: splits-at v i α N β by blast
   have split-w: w = α@(snd r)@β using assms split-v splits-at-combine-dest by
blast
  have split-uv: splits-at (u@v) (i + length u) (u@α) N β
    by (simp add: split-v splits-at-append-prefix)
  have is-sentence-uv: is-sentence (u@v)
    using Derives1 Derives1-sentence1 is-sentence-concat u by blast
  show ?thesis
   by (metis Derives1 Derives1-nonterminal Derives1-rule append-assoc is-sentence-uv

       split-uv split-v split-w splits-at-implies-Derives1)
qed

lemma leftmost-prepend-word: leftmost i v ⟹ is-word u ⟹ leftmost (i + length
u) (u@v)
by (simp add: leftmost-def nth-append)

lemma LeftDerives1-append-prefix:
  assumes Derives1: LeftDerives1 v i r w
  assumes u: is-word u
  shows LeftDerives1 (u@v) (i + length u) r (u@w)
proof −
  have 1: Derives1 v i r w
    by (simp add: Derives1 LeftDerives1-implies-Derives1)
  have 2: leftmost i v
    using Derives1 LeftDerives1-def by blast
  have 3: is-sentence u using u by fastforce
  have 4: Derives1 (u@v) (i + length u) r (u@w)
    by (simp add: 1 3 Derives1-append-prefix)
  have 5: leftmost (i + length u) (u@v)
    by (simp add: 2 leftmost-prepend-word u)
  show ?thesis
    by (simp add: 4 5 LeftDerives1-def)
qed

lemma Derivation-append-prefix: Derivation v D w ⟹ is-sentence u ⟹
  Derivation (u@v) (derivation-shift D 0 (length u)) (u@w)
proof (induct D arbitrary: u v w)
```

case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons d D*)
then have ∃ *x*. *Derives1 v* (*fst d*) (*snd d*) *x* ∧ *Derivation x D w* **by** *auto*
then obtain *x* **where** *x*: *Derives1 v* (*fst d*) (*snd d*) *x* ∧ *Derivation x D w* **by** *blast*
with *Cons* **have** *induct*: *Derivation* (*u*@*x*) (*derivation-shift D 0* (*length u*)) (*u*@*w*) **by** *auto*
have *Derives1*: *Derives1* (*u*@*v*) ((*fst d*) + *length u*) (*snd d*) (*u*@*x*)
by (*simp add*: *Cons.prems*(*2*) *Derives1-append-prefix x*)
show *?case*
apply *simp*
apply (*rule-tac x*=*u*@*x* **in** *exI*)
by (*simp add*: *Cons.hyps Cons.prems*(*2*) *Derives1 x*)
qed

lemma *LeftDerivation-append-prefix*: *LeftDerivation v D w* ⟹ *is-word u* ⟹
LeftDerivation (*u*@*v*) (*derivation-shift D 0* (*length u*)) (*u*@*w*)
proof (*induct D arbitrary*: *u v w*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons d D*)
then have ∃ *x*. *LeftDerives1 v* (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x D w* **by** *auto*
then obtain *x* **where** *x*: *LeftDerives1 v* (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x D w* **by** *blast*
with *Cons* **have** *induct*: *LeftDerivation* (*u*@*x*) (*derivation-shift D 0* (*length u*)) (*u*@*w*) **by** *auto*
have *Derives1*: *LeftDerives1* (*u*@*v*) ((*fst d*) + *length u*) (*snd d*) (*u*@*x*)
by (*simp add*: *Cons.prems*(*2*) *LeftDerives1-append-prefix x*)
show *?case*
apply *simp*
apply (*rule-tac x*=*u*@*x* **in** *exI*)
by (*simp add*: *Cons.hyps Cons.prems*(*2*) *Derives1 x*)
qed

lemma *derivation-ge-shift-simp*: *derivation-ge D i* ⟹ *i* ≥ *l* ⟹ *r* ≥ *l* ⟹
derivation-shift D l r = *derivation-shift D 0* (*r* − *l*)
proof (*induct D*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Cons d D*)
have *fst-d*: *fst d* ≥ *l*
using *Cons.prems*(*1*) *Cons.prems*(*2*) *derivation-ge-cons le-trans* **by** *blast*
show *?case*
apply *auto*
using *Cons fst-d* **apply** *arith*
using *Cons derivation-ge-cons* **apply** *auto*
done

```
qed

lemma append-dropped-prefix: is-prefix u v ⟹ drop (length u) v = w ⟹ u@w
= v
using is-prefix-unsplit by blast

lemma derivation-ge-shift-plus:
  assumes derivation-ge D u
  assumes derivation-ge (derivation-shift D u 0) v
  shows derivation-ge D (u + v)
proof −
  from assms show ?thesis
    apply (auto simp add: derivation-ge-def derivation-shift-def)
    by fastforce
qed

lemma LeftDerivation-breakdown:
  LeftDerivation (u@v) D w ⟹ ∃ n w1 w2. w = w1 @ w2 ∧
     LeftDerivation u (take n D) w1 ∧
     derivation-ge (drop n D) (length w1) ∧
     LeftDerivation v (derivation-shift (drop n D) (length w1) 0) w2
proof (induct length D arbitrary: u v D w)
  case 0
    then have D: D = [] by auto
    with 0 have u@v = w by auto
    with D show ?case
      apply (rule-tac x=0 in exI)
      apply (rule-tac x=u in exI)
      apply (rule-tac x=v in exI)
      by auto
next
  case (Suc l)
    then have ∃ d D'. D = d#D'
      by (metis LeftDerivation.elims(2) length-0-conv nat.simps(3))
    then obtain d D' where D-split: D = d#D' by blast
    from Suc have is-sentence-uv: is-sentence (u@v)
    by (metis D-split Derives1-sentence1 LeftDerivation.simps(2) LeftDerives1-implies-Derives1)
    then have is-sentence-u: is-sentence u and is-sentence-v: is-sentence v
      by (simp add: is-sentence-concat)+
    have is-word u ∨ (¬ is-word u) by blast
    then show ?case
      proof(induct rule: disjCases2)
        case 1
          then have derivation-ge-u: derivation-ge D (length u)
         using LeftDerivation-implies-Derivation Suc.prems is-word-Derivation-derivation-ge
by blast
          have is-prefix: is-prefix u w
            using 1.hyps LeftDerivation-implies-leftderives Suc.prems
              derives-word-is-prefix leftderives-implies-derives by blast
```

have *u-w*: *w* = *u* @ (*drop* (*length u*) *w*)
by (*metis 1.hyps LeftDerivation-implies-leftderives Suc.prems derives-word-is-prefix is-prefix-unsplit leftderives-implies-derives*)
show *?case*
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=u* **in** *exI*)
apply (*rule-tac x=drop* (*length u*) *w* **in** *exI*)
apply (*auto*)
apply (*rule u-w*)
apply (*rule derivation-ge-u*)
by (*simp add*: *LeftDerivation-skip-prefix Suc.prems derivation-ge-u*)
next
case *2*
with *is-sentence-u* **have** ∃ *i u1 N u2*. *splits-at u i u1 N u2* ∧ *leftmost i u*
using *leftmost-def nonword-leftmost-exists splits-at-def* **by** *auto*
then obtain *i u1 N u2* **where** *split-u*: *splits-at u i u1 N u2* ∧ *leftmost i u* **by** *blast*
have *is-word-u1*: *is-word u1* **by** (*metis leftmost-def split-u splits-at-def*)
have *LeftDerivation* (*u*@*v*) (*d*#*D′*) *w* **using** *D-split Suc.prems* **by** *blast*
then have ∃ *x*. *LeftDerives1* (*u*@*v*) (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x D′ w*
by *simp*
then obtain *x* **where** *x*: *LeftDerives1* (*u*@*v*) (*fst d*) (*snd d*) *x* ∧ *LeftDerivation x D′ w*
by *blast*
then have *fst-d-eq-i*: *fst d* = *i* **using** *splits-at-combine LeftDerives1-append-leftmost-unique split-u*
by *metis*
have *split-uv*: *splits-at* (*u*@*v*) *i u1 N* (*u2*@*v*) **by** (*simp add*: *split-u splits-at-append*)
have *split-x*: *x* = *u1* @ ((*snd* (*snd d*)) @ *u2* @ *v*)
using *LeftDerives1-implies-Derives1 fst-d-eq-i split-uv splits-at-combine-dest x* **by** *blast*
have *derivation-ge-D′*: *derivation-ge D′* (*length u1*)
using *LeftDerivation-implies-Derivation is-word-Derivation-derivation-ge*

leftmost-def split-u split-x splits-at-def x **by** *fastforce*
have *D1*: *LeftDerivation* ((*snd* (*snd d*)) @ *u2* @ *v*) (*derivation-shift D′* (*length u1*) *0*)
(*drop* (*length u1*) *w*)
using *LeftDerivation-skip-prefix derivation-ge-D′ split-x x* **by** *blast*
then have *D2*: *LeftDerivation* (((*snd* (*snd d*)) @ *u2*) @ *v*) (*derivation-shift D′* (*length u1*) *0*)
(*drop* (*length u1*) *w*) **by** *auto*
have *l* = *length* (*derivation-shift D′* (*length u1*) *0*)
using *D-split Suc.hyps*(*2*) **by** *auto*
from *Suc*(*1*)[*OF this D2*] **obtain** *n w1 w2* **where** *induct*:
drop (*length u1*) *w* = *w1* @ *w2* ∧
LeftDerivation (*snd* (*snd d*) @ *u2*)

```
            (take n (derivation-shift D′ (length u1) 0)) w1 ∧
          derivation-ge (drop n (derivation-shift D′ (length u1) 0)) (length w1) ∧
            LeftDerivation v (derivation-shift (drop n (derivation-shift D′ (length
u1) 0))
            (length w1) 0) w2 by blast
      have derivation-ge-D′-u1-w1: derivation-ge (drop n D′) (length u1 + length
w1)
        proof −
          from induct have 1: derivation-ge (derivation-shift (drop n D′) (length
u1) 0) (length w1)
            apply (subst drop-derivation-shift[symmetric])
            by blast
          have 2: derivation-ge (drop n D′) (length u1)
            by (metis append-take-drop-id derivation-ge-D′ derivation-ge-append)
          show ?thesis using 1 2 derivation-ge-shift-plus by blast
        qed
        have LeftDerivation (u1@(snd (snd d) @ u2)) (derivation-shift
              (take n (derivation-shift D′ (length u1) 0)) 0 (length u1)) (u1@w1)
        using induct LeftDerivation-append-prefix is-word-u1 by blast
        then have der1: LeftDerivation (u1@(snd (snd d) @ u2))
            (derivation-shift (take n D′) (length u1) (length u1)) (u1@w1)
          using take-derivation-shift derivation-shift-0-shift by auto
        have eq1: derivation-shift (take n D′) (length u1) (length u1) = take n D′
          apply (subst derivation-ge-shift-simp[where i = length u1])
          apply auto
          by (metis append-take-drop-id derivation-ge-D′ derivation-ge-append)
        from der1 eq1 have der2: LeftDerivation (u1@(snd (snd d) @ u2)) (take
n D′) (u1@w1)
          by auto
        have eq2: take (Suc n) D = d#(take n D′)
          by (simp add: D-split)
        have der3: LeftDerivation u (take (Suc n) D) (u1@w1)
          apply (simp add: eq2)
          apply (rule-tac x=u1@(snd (snd d) @ u2) in exI)
        by (metis Derives1-skip-suffix LeftDerives1-def append-assoc der2 fst-d-eq-i

            split-u split-x splits-at-def x)
        have is-prefix u1 w
          using LeftDerivation-implies-leftderives derives-word-is-prefix is-word-u1

            leftderives-implies-derives split-x x by blast
        then have eq3: u1 @ (w1@w2) = w
          apply (rule-tac append-dropped-prefix)
          apply (auto simp add: induct)
          done
        show ?case
          apply (rule-tac x=Suc n in exI)
          apply (rule-tac x=u1@w1 in exI)
          apply (rule-tac x=w2 in exI)
```

```
        apply auto
        apply (simp add: eq3)
        apply (simp add: der3)
        apply (simp add: D-split)
        apply (rule derivation-ge-D'-u1-w1)
        apply (simp add: D-split)
        using induct derivation-shift-0-shift drop-derivation-shift apply auto
        done
    qed
qed

lemma Derives1-terminals-stay:
  assumes Derives1: Derives1 u i r v
  assumes t-dom: t ∈ set u
  assumes terminal: is-terminal t
  shows t ∈ set v
proof −
  have ∃ α β N. splits-at u i α N β using Derives1 splits-at-ex by blast
  then obtain α β N where split-u: splits-at u i α N β by blast
  then have t ∈ set (α @ [N] @ β) using splits-at-combine t-dom by auto
  then have t-possible-locations: t ∈ set α ∨ t = N ∨ t ∈ set β by auto
   have is-nonterminal: is-nonterminal N using Derives1 Derives1-nonterminal
split-u by auto
  with t-possible-locations terminal have t-locations: t ∈ set α ∨ t ∈ set β
    using is-terminal-nonterminal by blast
 from Derives1 split-u have v = α @ (snd r) @ β by (simp add: splits-at-combine-dest)

  with t-locations show ?thesis by auto
qed

lemma Derivation-terminals-stay: Derivation u D v ⟹ t ∈ set u ⟹ is-terminal
t ⟹ t ∈ set v
proof (induct D arbitrary: u v)
  case Nil thus ?case by auto
next
  case (Cons d D)
  then have ∃ x. Derives1 u (fst d) (snd d) x ∧ Derivation x D v by auto
   then obtain x where x: Derives1 u (fst d) (snd d) x ∧ Derivation x D v by
auto
  show ?case using Cons Derives1-terminals-stay x by blast
qed

lemma Derivation-empty-no-terminals: Derivation u D [] ⟹ t ∈ set u ⟹ is-nonterminal
t
  by (metis Ball-set Derivation-implies-derives Derivation-terminals-stay
    derives-is-sentence is-sentence-def is-symbol-distinct list.pred-inject(1))

lemma mono-subset-elem: mono f ⟹ A ⊆ B ⟹ x ∈ f A ⟹ x ∈ f B using
mono-def by blast
```

lemma *wellformed-inc-dot*: *wellformed-item x* $\Longrightarrow$ *item-dot x* + *d* $\leq$ *length* (*item-rhs x*) $\Longrightarrow$
wellformed-item(*inc-dot d x*)
by (*simp add*: *inc-dot-def item-rhs-def wellformed-item-def*)

lemma *init-item-dot*[*simp*]: *item-dot* (*init-item r k*) = *0*
by (*simp add*: *init-item-def*)

lemma *init-item-rhs*[*simp*]: *item-rhs* (*init-item r k*) = *snd r*
by (*simp add*: *init-item-def item-rhs-def*)

lemma *init-item-*β[*simp*]: *item-*β (*init-item r k*) = *snd r*
by (*simp add*: *item-*β*-def*)

lemma *mono-*π: *mono* (π *k T*)
by (*simp add*: π*-regular regular-implies-mono*)

lemma π*-subset-elem-trans*:
assumes *Y*: $Y \subseteq \pi\ k\ T\ X$
assumes *z*: $z \in \pi\ k\ T\ Y$
shows $z \in\ \pi\ k\ T\ X$
proof −
from *Y* **have** $\pi\ k\ T\ Y \subseteq \pi\ k\ T\ (\pi\ k\ T\ X)$ **by** (*simp add*: *monoD mono-*π)
then have $\pi\ k\ T\ Y \subseteq \pi\ k\ T\ X$ **using** π*-idempotent* **by** *blast*
with *z* **show** *?thesis* **using** *contra-subsetD* **by** *blast*
qed

lemma *inc-dot-origin*[*simp*]: *item-origin* (*inc-dot d x*) = *item-origin x*
by (*simp add*: *inc-dot-def*)

lemma *inc-dot-end*[*simp*]: *item-end* (*inc-dot d x*) = *item-end x*
by (*simp add*: *inc-dot-def*)

lemma *inc-dot-rhs*[*simp*]: *item-rhs* (*inc-dot d x*) = *item-rhs x*
by (*simp add*: *inc-dot-def item-rhs-def*)

lemma *inc-dot-dot*[*simp*]: *item-dot* (*inc-dot d x*) = *item-dot x* + *d*
by (*simp add*: *inc-dot-def*)

lemma *inc-dot-nonterminal*[*simp*]: *item-nonterminal* (*inc-dot d x*) = *item-nonterminal x*
by (*simp add*: *inc-dot-def item-nonterminal-def*)

lemma *Predict-subset-*π: *Predict k X* $\subseteq \pi\ k\ T\ X$
proof −
have *setmonotone* (π *k T*)
by (*simp add*: π*-regular regular-implies-setmonotone*)
then have *s*: $X \subseteq \pi\ k\ T\ X$ **by** (*simp add*: *subset-setmonotone*)

have *mono* (*Predict k*) **by** (*simp add*: *Predict-regular regular-implies-mono*)
with *s* **have** *Predict k X* ⊆ *Predict k* (*π k T X*) **by** (*simp add*: *monoD*)
then show *Predict k X* ⊆ *π k T X* **by** (*simp add*: *Predict-π-fix*)
qed

lemma *Complete-subset-π*: *Complete k X* ⊆ *π k T X*
proof −
have *setmonotone* (*π k T*)
by (*simp add*: *π-regular regular-implies-setmonotone*)
then have *s*: *X* ⊆ *π k T X* **by** (*simp add*: *subset-setmonotone*)
have *mono* (*Complete k*) **by** (*simp add*: *Complete-regular regular-implies-mono*)

with *s* **have** *Complete k X* ⊆ *Complete k* (*π k T X*) **by** (*simp add*: *monoD*)
then show *Complete k X* ⊆ *π k T X* **by** (*simp add*: *Complete-π-fix*)
qed

lemma *inc-inc-dot*[*simp*]: *inc-dot a* (*inc-dot b x*) = *inc-dot* (*a* + *b*) *x*
by (*simp add*: *inc-dot-def*)

lemma *thmD6-Left*: *wellformed-item x* ⟹ *item-β x* = *δ* @ *ω* ⟹ *item-end x* =
k ⟹
LeftDerivation δ D [] ⟹ *inc-dot* (*length δ*) *x* ∈ *π k* {} {*x*}
proof (*induct length D arbitrary*: *x δ ω D rule*: *less-induct*)
case *less*
have *length δ* = *0* ∨ *length δ* = *1* ∨ *length δ* ≥ *2* **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases3*)
case *1*
then have *δ* = [] **by** *auto*
then show *?case* **by** (*simp add*: *π-regular elem-setmonotone regular-implies-setmonotone*)
next
case *2*
then have ∃ *N*. *δ* = [*N*]
by (*metis One-nat-def append-self-conv2 drop-all id-take-nth-drop*
le-numeral-extra(*4*) *lessI take-0*)
then obtain *N* **where** *N*: *δ* = [*N*] **by** *blast*
then have *N* ∈ *set δ* **by** *auto*
then have *is-nonterminal-N*: *is-nonterminal N* **using** *Derivation-empty-no-terminals*

LeftDerivation-implies-Derivation less.prems(*4*) **by** *blast*
have *D* ≠ [] **using** *LeftDerivation.elims*(*2*) *N less.prems*(*4*) **by** *blast*
then have ∃ *e E*. *D* = *e*#*E* **using** *LeftDerivation.elims*(*2*) *less.prems*(*4*)
by *blast*
then obtain *e E* **where** *eE*: *D* = *e*#*E* **by** *blast*
then have ∃ *γ*. *LeftDerives1 δ* (*fst e*) (*snd e*) *γ* ∧
LeftDerivation γ E [] **using** *LeftDerivation.simps*(*2*) *less.prems*(*4*) **by** *blast*

then obtain *γ* **where** *γ*: *LeftDerives1 δ* (*fst e*) (*snd e*) *γ* ∧ *LeftDerivation*
γ E [] **by** *blast*

```
with N have γ-def: γ = snd (snd e)
  by (metis 2.hyps Derives1-split LeftDerives1-def One-nat-def append-Cons

      append-Nil append-Nil2 leftmost-def length-0-conv less-nat-zero-code
linorder-neqE-nat
    list.inject not-less-eq)
have next-symbol-x: next-symbol x = Some N
using N less.prems(1) less.prems(2) next-symbol-def next-symbol-starts-item-β

    wellformed-complete-item-β by fastforce
have x-subset: {x} ⊆ π k {} {x}
  using π-regular regular-implies-setmonotone subset-setmonotone by blast
let ?y = init-item (snd e) k
have ?y ∈ Predict k {x}
  apply (simp add: Predict-def)
  apply (rule disjI2)
  apply (rule-tac x=fst (snd e) in exI)
  apply (rule-tac x=snd (snd e) in exI)
  apply auto
  using Derives1-rule LeftDerives1-implies-Derives1 γ apply blast
  apply (rule-tac x=x in exI)
  by (metis (mono-tags, lifting) Derives1-split LeftDerives1-def N γ
        append.simps(1) append.simps(2) bin-def is-nonterminal-N left-
most-cons-nonterminal
      leftmost-unique length-greater-0-conv less.prems(3) less-nat-zero-code
      list.inject mem-Collect-eq next-symbol-x singletonI)
then have y-dom: ?y ∈ π k {} {x} using Predict-subset-π by blast
let ?z = inc-dot (length γ) ?y
have item-dot ?y = 0 and item-rhs ?y = γ by (auto simp add: γ-def)
note y-props = this
then have wellformed-y: wellformed-item ?y
using Derives1-rule LeftDerives1-implies-Derives1 γ less.prems(1) less.prems(3)

  wellformed-init-item wellformed-item-def by blast
with y-props have wellformed-z: wellformed-item ?z by (simp add: well-
formed-inc-dot)
have item-β-y: item-β ?y = γ @ [] using item-rhs-split y-props(2) by auto
have is-complete-z: is-complete ?z by (simp add: is-complete-def γ-def)
have ?z ∈ π k {} {?y}
  apply (rule less(1)[where D=E])
  apply (auto simp add: eE wellformed-y γ)
  apply (simp add: γ-def)
  done
with y-dom have z-dom: ?z ∈ π k {} {x}
  using π-subset-elem-trans empty-subsetI insert-subset by blast
let ?w = inc-dot (length δ) x
have ?w ∈ Complete k {x, ?z}
 apply (simp add: Complete-def)
 apply (rule-tac disjI2)+
```

apply (*rule-tac x=x* **in** *exI*)
apply (*auto simp add*: *2*)
apply (*simp add*: *inc-dot-def inc-item-def less*)
apply (*rule-tac x=?z* **in** *exI*)
apply (*auto simp add*: *bin-def less is-complete-z next-symbol-x*)
by (*metis Derives1-split LeftDerives1-def N* γ *append-Cons append-self-conv2*

is-nonterminal-N leftmost-cons-nonterminal leftmost-unique length-0-conv list.inject)
then have *?w* $\in$ π *k* {} {*x*, *?z*} **using** *Complete-subset-*π **by** *blast*
then show *?case* **by** (*meson* π*-subset-elem-trans insert-subset x-subset z-dom*)

next
case *3*
then have $\exists$ *N* α. δ = [*N*] @ α
by (*metis append-Cons append-Nil count-terminals.cases le0 le-0-eq list.size*(*3*)

numeral-le-iff semiring-norm(*69*))
then obtain *N* α **where** δ*-split*: δ = [*N*] @ α **by** *blast*
with *3* **have** α*-nonempty*: $\alpha \neq$ []
by (*metis* (*full-types*) *One-nat-def Suc-eq-plus1 append-Nil2 impossible-Cons length-Cons*
list.size(*3*) *nat-1-add-1*)
have *LeftDerivation* ([*N*] @ α) *D* [] **using** δ*-split less.prems*(*4*) **by** *blast*
from *LeftDerivation-breakdown*[*OF this*, *simplified*]
obtain *n* **where** *n*: *LeftDerivation* [*N*] (*take n D*) [] $\wedge$ *LeftDerivation* α (*drop n D*) [] **by** *blast*
let *?E* = *take n D*
from *n* **have** *E*: *LeftDerivation* [*N*] *?E* [] **by** *auto*
let *?F* = *drop n D*
from *n* **have** *F*: *LeftDerivation* α *?F* [] **by** *auto*
have *length-add*: *length ?E* + *length ?F* = *length D* **by** *simp*
have *?E* $\neq$ [] **using** *E* **by** *force*
then have *length-E-0*: *length ?E* > *0* **by** *auto*
have *?F* $\neq$ [] **using** *F* α*-nonempty* **by** *force*
then have *length-F-0*: *length ?F* > *0* **by** *auto*
from *length-add length-E-0 length-F-0*
have *length ?E* < *length D* $\wedge$ *length ?F* < *length D*
using *add.commute nat-add-left-cancel-less nat-neq-iff not-add-less2* **by** *linarith*
then have *length-E*: *length ?E* < *length D* **and** *length-F*: *length ?F* < *length D* **by** *auto*
let *?y* = *inc-dot* (*length* [*N*]) *x*
have *y-dom*: *?y* $\in$ π *k* {} {*x*}
apply (*rule-tac less*(*1*)[**where** *D*=*?E* **and** ω=α@ω])
apply (*rule length-E*)
by (*auto simp add*: *less* δ*-split E*)
let *?z* = *inc-dot* (*length* α) *?y*
have *wellformed-y*: *wellformed-item ?y*

using *δ-split is-complete-def less.prems(1) less.prems(2) wellformed-complete-item-β*

wellformed-inc-dot **by** *fastforce*
have *?z* ∈ *π k* {} {*?y*}
apply (*rule-tac less(1)*[**where** *D*=*?F* **and** *ω*=*ω*])
apply (*rule length-F*)
apply (*rule wellformed-y*)
apply (*auto simp add*: *F less*)
by (*metis δ-split add.commute append-assoc append-eq-conv-conj drop-drop inc-dot-dot*
inc-dot-rhs item-β-def length-Cons less.prems(2) list.size(3))
then have *z-dom*: *?z* ∈ *π k* {} {*x*} **using** *π-subset-elem-trans y-dom* **by** *blast*

have *?z* = *inc-dot* (*length δ*) *x* **by** (*simp add*: *δ-split*)
with *z-dom* **show** *?case* **by** *auto*
qed
qed

lemma *derives-empty-implies-LeftDerivation*: *derives δ* [] ⟹ ∃ *D*. *LeftDerivation δ D* []
using *derives-implies-leftderives is-word-def leftderives-implies-LeftDerivation list.pred-inject(1)* **by** *blast*

lemma *thmD6*: *wellformed-item x* ⟹ *item-β x* = *δ* @ *ω* ⟹ *item-end x* = *k* ⟹

derives δ [] ⟹ *inc-dot* (*length δ*) *x* ∈ *π k* {} {*x*}
using *derives-empty-implies-LeftDerivation thmD6-Left* **by** *blast*

end

end
theory *TheoremD7*
imports *TheoremD6*
begin

context *LocalLexing* **begin**

lemma *Derives1-keep-first-terminal*: *Derives1* (*x*#*u*) *i r* (*y*#*v*) ⟹ *is-terminal x* ⟹ *x* = *y*
by (*metis Derives1-leftmost Derives1-take leftmost-cons-terminal list.sel(1) not-le take-Cons′*)

lemma *Derives1-nonterminal-head*:
assumes *Derives1 u i r* (*N*#*v*)
assumes *is-nonterminal N*
shows ∃ *u′ M*. *u* = *M*#*u′* ∧ *is-nonterminal M*
proof −
from *assms* **have** *nonempty-u*: *u* ≠ []
by (*metis Derives1-bound less-nat-zero-code list.size(3)*)

```
  have ∃ u' M. u = M#u'
    using count-terminals.cases nonempty-u by blast
  then obtain u' M where split-u: u = M#u' by blast
  have is-sentence-u: is-sentence u using assms
    using Derives1-sentence1 by blast
  then have is-terminal M ∨ is-nonterminal M
    using is-sentence-cons is-symbol-distinct split-u by blast
  then show ?thesis
  proof (induct rule: disjCases2)
    case 1
      have is-terminal N
        using 1.hyps Derives1-keep-first-terminal
        assms(1) split-u by blast
      with assms have False using is-terminal-nonterminal by blast
      then show ?case by blast
  next
    case 2 with split-u show ?case by blast
  qed
qed

lemma sentence-starts-with-nonterminal:
  assumes is-nonterminal N
  assumes derives u []
  shows ∃ X r. u@[N] = X#r ∧ is-nonterminal X
proof (cases u = [])
  case True thus ?thesis using assms(1) by blast
next
  case False
  then have ∃ X r. u = X#r using count-terminals.cases by blast
  then obtain X r where Xr: u = X#r by blast
  then have is-nonterminal X
    by (metis False assms(2) count-terminals.simps derives-count-terminals-leq
      derives-is-sentence is-sentence-cons is-symbol-distinct not-le zero-less-Suc)
  with Xr show ?thesis by auto
qed

lemma Derives1-nonterminal-head':
  assumes Derives1 u i r (v1@[N]@v2)
  assumes is-nonterminal N
  assumes derives v1 []
  shows ∃ u' M. u = M#u' ∧ is-nonterminal M
proof -
  from sentence-starts-with-nonterminal[OF assms(2,3)]
  obtain X r where v1 @ [N] = X # r ∧ is-nonterminal X by blast
  then show ?thesis
    by (metis Derives1-nonterminal-head append-Cons append-assoc assms(1))
qed

lemma thmD7-helper:
```

```
  assumes LeftDerivation [𝔖] D (N#v)
  assumes is-nonterminal N
  assumes 𝔖 ≠ N
  shows ∃ n M a a1 a2 w. n < length D ∧ (M, a) ∈ ℜ ∧ LeftDerivation [𝔖] (take
n D) (M#w) ∧
    a = a1 @ [N] @ a2 ∧ derives a1 []
proof −
  have ∃ n u v. LeftDerivation [𝔖] (take n D) (u@[N]@v) ∧ derives u []
    apply (rule-tac x=length D in exI)
    apply (rule-tac x=[] in exI)
    apply (rule-tac x=v in exI)
    using assms by simp
  then show ?thesis
  proof (induct rule: ex-minimal-witness)
    case (Minimal K)
      have nonzero-K: K ≠ 0
      proof (cases K = 0)
        case True
          with Minimal have ∃ u v. [𝔖] = u@[N]@v
            using LeftDerivation.simps(1) take-0 by auto
          with assms have False by (simp add: Cons-eq-append-conv)
          then show ?thesis by simp
      next
        case False
          then show ?thesis by arith
      qed
      from Minimal(1)
     obtain u v where uv: LeftDerivation [𝔖] (take K D) (u @ [N] @ v) ∧ derives
u [] by blast
      from nonzero-K have take K D ≠ []
       using Minimal.hyps(2) less-nat-zero-code nat-neq-iff take-eq-Nil uv by force

      then have ∃ E e. (take K D) = E@[e] using rev-exhaust by blast
      then obtain E e where Ee: take K D = E@[e] by blast
       with uv have ∃ x. LeftDerivation [𝔖] E x ∧ LeftDerives1 x (fst e) (snd e)
(u @ [N] @ v)
        by (simp add: LeftDerivation-append)
     then obtain x where x: LeftDerivation [𝔖] E x ∧ LeftDerives1 x (fst e) (snd
e) (u @ [N] @ v)
        by blast
      then have ∃ w M. x = M#w ∧ is-nonterminal M
         using Derives1-nonterminal-head' LeftDerives1-implies-Derives1 assms(2)
uv by blast
    then obtain w M where split-x: x = M#w and is-nonterminal-M: is-nonterminal
M by blast
      from Ee nonzero-K have E: E = take (K − 1) D
      by (metis Minimal.hyps(2) butlast-snoc butlast-take dual-order.strict-implies-order

          le-less-linear take-all uv)
```

have *leftmost* (*fst e*) (*M#w*) **using** *x LeftDerives1-def split-x* **by** *blast*
with *is-nonterminal-M* **have** *fst-e*: *fst e* = *0*
by (*simp add*: *leftmost-cons-nonterminal leftmost-unique*)
have *Derives1*: *Derives1 x 0* (*snd e*) (*u* @ [*N*] @ *v*)
using *LeftDerives1-implies-Derives1 fst-e x* **by** *auto*
have *x-splits-at*: *splits-at x 0* [] *M w*
by (*simp add*: *split-x splits-at-def*)
from *Derives1 x-splits-at*
have *pq*: ∃ *p q*. *u* = [] @ *p* ∧ *v* = *q* @ *w* ∧ *snd* (*snd e*) = *p* @ [*N*] @ *q*
proof (*induct rule*: *Derives1-X-is-part-of-rule*)
case (*Suffix* α) **thus** *?case* **by** *blast*
next
case (*Prefix* β)
then have *derives-β*: *derives* β []
using *Derives1-implies-derives1 derives1-implies-derives derives-trans uv* **by** *blast*
with *Prefix*(*1*) *x Minimal E nonzero-K* **show** *False*
by (*meson diff-less less-nat-zero-code less-one nat-neq-iff*)
qed
from *this*[*simplified*] **obtain** *q* **where** *q*: *v* = *q* @ *w* ∧ *snd* (*snd e*) = *u* @ *N* # *q* **by** *blast*
have *M-def*: *fst* (*snd e*) = *M*
using *Derives1 Derives1-nonterminal x-splits-at* **by** *blast*
show *?case*
apply (*rule-tac x*=*K*−*1* **in** *exI*)
apply (*rule-tac x*=*M* **in** *exI*)
apply (*rule-tac x*=*snd* (*snd e*) **in** *exI*)
apply (*rule-tac x*=*u* **in** *exI*)
apply (*rule-tac x*=*q* **in** *exI*)
apply (*rule-tac x*=*w* **in** *exI*)
by (*metis Derives1 Derives1-rule E Ee M-def One-nat-def Suc-pred pq append-Nil*
append-same-eq dual-order.strict-implies-order le-less-linear nonzero-K not-Cons-self2
not-gr0 not-less-eq prod.collapse q self-append-conv split-x take-all uv x)
qed
qed

lemma *head-of-item-β-is-next-symbol*:
wellformed-item x ⟹ *item-β x* = *t#δ* ⟹ *next-symbol x* = *Some t*
using *next-symbol-def next-symbol-starts-item-β wellformed-complete-item-β* **by** *fastforce*

lemma *next-symbol-predicts*: *next-symbol x* = *Some N* ⟹ (*N*, *a*) ∈ 𝔑 ⟹ *k* = *item-end x* ⟹
init-item (*N*, *a*) *k* ∈ *Predict k* {*x*}
using *Predict-def bin-def* **by** *auto*

lemma *thmD7-LeftDerivation*: *LeftDerivation* [𝔖] *D* (*N#γ*) ⟹ *is-nonterminal N*

```
⟹ (N, α) ∈ ℜ ⟹
  init-item (N, α) 0 ∈ π 0 {} Init
proof (induct length D arbitrary: D N γ α rule: less-induct)
  case less
    let ?trivial = 𝔖 = N
    have ?trivial ∨ ¬ ?trivial by blast
    then show ?case
    proof (induct rule: disjCases2)
      case 1
        then have init-item (N, α) 0 ∈ Init
          apply (subst Init-def)
          by (auto simp add: less)
        then show ?case
            by (meson π-regular contra-subsetD regular-implies-setmonotone sub-
set-setmonotone)
    next
      case 2
        from thmD7-helper[OF less(2) less(3) 2]
        obtain n M a a1 a2 w where n < length D and (M, a) ∈ ℜ and
            LeftDerivation [𝔖] (take n D) (M # w) and a = a1 @ [N] @ a2 and
derives a1 []
          by blast
        note M = this
        let ?x = init-item (M, a) 0
        have x-dom: ?x ∈ π 0 {} Init
          apply (rule less(1)[OF - M(3) - M(2)])
          using M(1) apply simp
          using M(2) by simp
        have wellformed-x: wellformed-item ?x by (simp add: M(2))
        let ?y = inc-dot (length a1) ?x
        have ?y ∈ π 0 {} {?x}
          apply (rule thmD6[where ω=[N] @ a2])
          using wellformed-x by (auto simp add: M)
        with x-dom have y-dom: ?y ∈ π 0 {} Init
          using π-subset-elem-trans empty-subsetI insert-subset by blast
        have wellformed-y: wellformed-item ?y
          using M(4) wellformed-inc-dot wellformed-x by auto
        have item-β ?y = N#a2 by (simp add: M(4) item-β-def)
        then have next-symbol-y: next-symbol ?y = Some N
          by (simp add: head-of-item-β-is-next-symbol wellformed-y)
        let ?z = init-item (N, α) 0
        have ?z ∈ Predict 0 {?y}
          by (simp add: less.prems(3) next-symbol-predicts next-symbol-y)
        then have ?z ∈ π 0 {} {?y} using Predict-subset-π by auto
        with y-dom show ?z ∈ π 0 {} Init
          using π-subset-elem-trans empty-subsetI insert-subset by blast
    qed
qed
```

theorem *thmD7*: *is-derivation* $(N\#\gamma) \Longrightarrow$ *is-nonterminal* $N \Longrightarrow (N, \alpha) \in \mathfrak{R} \Longrightarrow$

init-item (N, α) *0* $\in \pi$ *0* $\{\}$ *Init*
by (*metis* $\mathcal{L}_P$*-is-word derives-implies-leftderives-cons empty-in-*$\mathcal{L}_P$ *is-derivation-def*

leftderives-implies-LeftDerivation self-append-conv2 thmD7-LeftDerivation)

end

end
theory *TheoremD8*
imports *TheoremD7*
begin

context *LocalLexing* **begin**

lemma *wellformed-tokens-empty-path*[*simp*]: *wellformed-tokens* []
  by (*simp add*: *wellformed-tokens-def*)

lemma $\mathcal{P}$*-0-0-Gen*: *Gen* ($\mathcal{P}$ *0 0*) = { *x* . *wellformed-item x* $\wedge$ *item-origin x* = *0* $\wedge$ *item-end x* = *0* $\wedge$
  derives (*item-*α *x*) [] $\wedge$ ($\exists$ γ. *is-derivation* ([*item-nonterminal x*] @ γ)) }
by (*auto simp add*: *Gen-def pvalid-def*)

lemma *Init-subset-Gen*: *Init* $\subseteq$ *Gen* ($\mathcal{P}$ *0 0*)
  apply (*subst* $\mathcal{P}$*-0-0-Gen*)
  apply (*auto simp add*: *Init-def*)
  apply (*rule-tac x*=[] **in** *exI*)
  apply (*simp add*: *is-derivation-def*)
  done

lemma $\mathcal{J}$*-0-0-subset-Gen*: $\mathcal{J}$ *0 0* $\subseteq$ *Gen* ($\mathcal{P}$ *0 0*)
  apply (*simp only*: $\mathcal{J}$*.simps*)
  apply (*rule-tac thmD5*)
  apply (*rule Init-subset-Gen*)
  by *auto*

lemma *inc-dot-rule*[*simp*]: *item-rule* (*inc-dot d x*) = *item-rule x*
  by (*simp add*: *inc-dot-def*)

lemma *init-item-rule*[*simp*]: *item-rule* (*init-item r k*) = *r*
  by (*simp add*: *init-item-def*)

lemma *item-dot-is-*α*-length*: *wellformed-item x* $\Longrightarrow$ *item-dot x* = *length* (*item-*α *x*)
  apply (*simp add*: *item-*α*-def*)
  by (*simp add*: *min-absorb2 wellformed-item-def*)

lemma *Gen-subset-*$\mathcal{J}$*-0-0-helper*:

```
  assumes wellformed-item x
  assumes item-origin x = 0
  assumes item-end x = 0
  assumes derives (item-α x) []
  assumes is-derivation (item-nonterminal x # γ)
  shows x ∈ π 0 {} Init
proof −
  let ?y = init-item (item-nonterminal x, item-rhs x) 0
  have y-dom: ?y ∈ π 0 {} Init
    apply (rule-tac thmD7)
    using assms apply auto
    using is-nonterminal-item-nonterminal apply blast
    by (simp add: item-nonterminal-def item-rhs-def wellformed-item-def)
  let ?x = inc-dot (length (item-α x)) ?y
  have x1: item-rule x = item-rule ?x
    apply (simp)
    by (simp add: item-nonterminal-def item-rhs-def)
  have x2: item-dot x = item-dot ?x
    apply simp
    by (simp add: assms(1) item-dot-is-α-length)
  have x3: item-origin x = item-origin ?x
    using assms by auto
  have x4: item-end x = item-end ?x
    using assms by auto
  from x1 x2 x3 x4 have x-is-inc: x = ?x using item.expand by blast
  have wellformed-item-y: wellformed-item ?y
   using assms(1) item-nonterminal-def item-rhs-def wellformed-item-def by auto
  have x ∈ π 0 {} {?y}
    apply (subst x-is-inc)
    apply (rule-tac thmD6)
    apply (simp add: wellformed-item-y)
    apply (simp add: item-rhs-split)
    apply simp
    using assms apply simp
    done
  with y-dom show ?thesis
    using π-subset-elem-trans empty-subsetI insert-subset by blast
qed

lemma Gen-subset-𝒥-0-0: Gen (𝒫 0 0) ⊆ 𝒥 0 0
  apply (subst 𝒫-0-0-Gen)
  apply auto
  using Gen-subset-𝒥-0-0-helper by blast

theorem thmD8: 𝒥 0 0 = Gen (𝒫 0 0)
  using Gen-subset-𝒥-0-0 𝒥-0-0-subset-Gen by blast

end
```

end
theory *TheoremD9*
imports *TheoremD8*
begin

context *LocalLexing* **begin**

definition *items-le* :: *nat* ⇒ *items* ⇒ *items*
where
items-le k I = { *x* . *x* ∈ *I* ∧ *item-end x* ≤ *k* }

definition *items-eq* :: *nat* ⇒ *items* ⇒ *items*
where
items-eq k I = { *x* . *x* ∈ *I* ∧ *item-end x* = *k* }

definition *paths-le* :: *nat* ⇒ *tokens set* ⇒ *tokens set*
where
paths-le k P = { *p* . *p* ∈ *P* ∧ *charslength p* ≤ *k* }

definition *paths-eq* :: *nat* ⇒ *tokens set* ⇒ *tokens set*
where
paths-eq k P = { *p* . *p* ∈ *P* ∧ *charslength p* = *k* }

lemma *items-le-pointwise*: *pointwise* (*items-le k*)
by (*auto simp add*: *pointwise-def items-le-def*)

lemma *items-le-is-filter*: *items-le k I* ⊆ *I*
by (*auto simp add*: *items-le-def*)

lemma *items-eq-pointwise*: *pointwise* (*items-eq k*)
by (*auto simp add*: *pointwise-def items-eq-def*)

lemma *items-eq-is-filter*: *items-eq k I* ⊆ *I*
by (*auto simp add*: *items-eq-def*)

lemma *paths-le-pointwise*: *pointwise* (*paths-le k*)
by (*auto simp add*: *pointwise-def paths-le-def*)

lemma *paths-le-continuous*: *continuous* (*paths-le k*)
by (*simp add*: *paths-le-pointwise pointbased-implies-continuous pointwise-implies-pointbased*)

lemma *paths-le-mono*: *mono* (*paths-le k*)
by (*simp add*: *continuous-imp-mono paths-le-continuous*)

lemma *paths-le-is-filter*: *paths-le k P* ⊆ *P*
by (*auto simp add*: *paths-le-def*)

lemma *paths-eq-pointwise*: *pointwise* (*paths-eq k*)
by (*auto simp add*: *pointwise-def paths-eq-def*)

lemma *paths-eq-is-filter*: *paths-eq k P* $\subseteq$ *P*
by (*auto simp add*: *paths-eq-def*)

lemma *Predict-item-end*: *x* $\in$ *Predict k Y* $\Longrightarrow$ *item-end x* = *k* $\vee$ *x* $\in$ *Y*
using *Predict-def* **by** *auto*

lemma *Complete-item-end*: *x* $\in$ *Complete k Y* $\Longrightarrow$ *item-end x* = *k* $\vee$ *x* $\in$ *Y*
using *Complete-def* **by** *auto*

lemma *$\mathcal{J}$-0-0-item-end*: *x* $\in$ $\mathcal{J}$ *0 0* $\Longrightarrow$ *item-end x* = *0*
apply (*simp add*: *π-def*)
proof (*induct rule*: *limit-induct*)
case (*Init x*) **thus** *?case* **by** (*auto simp add*: *Init-def*)
next
case (*Iterate x Y*)
then have *x* $\in$ *Complete 0* (*Predict 0 Y*) **by** (*simp add*: *Scan-empty*)
then have *item-end x* = *0* $\vee$ *x* $\in$ *Predict 0 Y* **using** *Complete-item-end* **by** *blast*
then have *item-end x* = *0* $\vee$ *x* $\in$ *Y* **using** *Predict-item-end* **by** *blast*
then show *?case* **using** *Iterate* **by** *blast*
qed

lemma *items-le-$\mathcal{J}$-0-0*: *items-le 0* ($\mathcal{J}$ *0 0*) = $\mathcal{J}$ *0 0*
using *LocalLexing.$\mathcal{J}$-0-0-item-end LocalLexing.items-le-def LocalLexing-axioms* **by** *blast*

lemma *paths-le-$\mathcal{P}$-0-0*: *paths-le 0* ($\mathcal{P}$ *0 0*) = $\mathcal{P}$ *0 0*
by (*auto simp add*: *paths-le-def*)

definition *empty-tokens* :: *token set* $\Rightarrow$ *token set*
where
empty-tokens T = { *t* . *t* $\in$ *T* $\wedge$ *chars-of-token t* = [] }

lemma *items-le-Predict*: *items-le k* (*Predict k I*) = *Predict k* (*items-le k I*)
by (*auto simp add*: *items-le-def Predict-def bin-def*)

lemma *items-le-Complete*:
wellformed-items I $\Longrightarrow$ *items-le k* (*Complete k I*) = *Complete k* (*items-le k I*)
by (*auto simp add*: *items-le-def Complete-def bin-def is-complete-def wellformed-items-def*
wellformed-item-def)

lemma *items-le-Scan*:
items-le k (*Scan T k I*) = *Scan* (*empty-tokens T*) *k* (*items-le k I*)
by (*auto simp add*: *items-le-def Scan-def bin-def empty-tokens-def*)

lemma *wellformed-items-Gen*: *wellformed-items* (*Gen P*)
using *Gen-implies-pvalid pvalid-def wellformed-items-def* **by** *blast*

lemma *wellformed-*$\mathcal{J}$*-0-0*: *wellformed-items* ($\mathcal{J}$ *0 0*)
using *thmD8 wellformed-items-Gen* **by** *auto*

lemma *wellformed-items-Predict*:
wellformed-items I $\Longrightarrow$ *wellformed-items* (*Predict k I*)
by (*auto simp add*: *wellformed-items-def wellformed-item-def Predict-def bin-def*)

lemma *wellformed-items-Complete*:
wellformed-items I $\Longrightarrow$ *wellformed-items* (*Complete k I*)
apply (*auto simp add*: *wellformed-items-def wellformed-item-def Complete-def bin-def*)
apply (*metis dual-order.trans*)
using *is-complete-def next-symbol-not-complete not-less-eq-eq* **by** *blast*

lemma $\mathcal{X}$*-length-bound*: $(t, c) \in \mathcal{X}\ k \Longrightarrow k + length\ c \leq length\ Doc$
using $\mathcal{X}$*-is-prefix is-prefix-length not-le* **by** *fastforce*

lemma *wellformed-items-Scan*:
wellformed-items I $\Longrightarrow T \subseteq \mathcal{X}\ k \Longrightarrow$ *wellformed-items* (*Scan T k I*)
apply (*auto simp add*: *wellformed-items-def wellformed-item-def Scan-def bin-def* $\mathcal{X}$*-length-bound*)
using *is-complete-def next-symbol-not-complete not-less-eq-eq* **by** *blast*

lemma *wellformed-items-*π:
assumes *wellformed-items I*
assumes $T \subseteq \mathcal{X}\ k$
shows *wellformed-items* (π *k T I*)
proof −
{
fix *x* :: *item*
have $x \in \pi\ k\ T\ I \Longrightarrow$ *wellformed-item x*
proof (*simp add*: π*-def*, *induct rule*: *limit-induct*)
case (*Init x*) **thus** *?case* **using** *assms*(*1*) **by** (*simp add*: *wellformed-items-def*)

next
case (*Iterate x Y*)
have *wellformed-items Y* **by** (*simp add*: *Iterate.hyps*(*1*) *wellformed-items-def*)
then have *wellformed-items* (*Scan T k* (*Complete k* (*Predict k Y*)))
by (*simp add*: *assms*(*2*) *wellformed-items-Complete wellformed-items-Predict*

wellformed-items-Scan)
then show *?case* **by** (*simp add*: *Iterate.hyps*(*2*) *wellformed-items-def*)
qed
}
then show *?thesis* **using** *wellformed-items-def* **by** *auto*
qed

lemma $\mathcal{J}$*-subset-Suc-u*: $\mathcal{J}\ k\ u \subseteq \mathcal{J}\ k\ (Suc\ u)$

```
  by (metis Complete-π-fix Complete-subset-π 𝒥.simps(1) 𝒥.simps(2) 𝒥.simps(3)
not0-implies-Suc)

lemma mono-TokensAt: mono (TokensAt k)
  by (auto simp add: mono-def TokensAt-def bin-def)

lemma 𝒯-subset-TokensAt: 𝒯 k u ⊆ TokensAt k (𝒥 k u)
proof (induct u)
  case 0 thus ?case by simp
next
  case (Suc u)
    have 1: Tokens k (𝒯 k u) (𝒥 k u) = Sel (𝒯 k u) (TokensAt k (𝒥 k u))
      by (simp add: Tokens-def)
    have 2: Sel (𝒯 k u) (TokensAt k (𝒥 k u)) ⊆ TokensAt k (𝒥 k u)
      by (simp add: Sel-upper-bound Suc.hyps)
    have 𝒯 k (Suc u) ⊆ TokensAt k (𝒥 k u)
      by (simp add: 1 2)
    then show ?case
      by (meson 𝒥-subset-Suc-u mono-TokensAt mono-subset-elem subset-iff)
qed

lemma TokensAt-subset-𝒳: TokensAt k I ⊆ 𝒳 k
  using TokensAt-def 𝒳-def is-terminal-def by auto

lemma wellformed-items-𝒥-induct-u:
  assumes wellformed-items (𝒥 k u)
  shows wellformed-items (𝒥 k (Suc u))
proof -
  {
    fix x :: item
    have x ∈ 𝒥 k (Suc u) ⟹ wellformed-item x
    proof (simp add: π-def, induct rule: limit-induct)
      case (Init x)
        with assms show ?case by (auto simp add: wellformed-items-def)
    next
      case (Iterate p Y)
        from Iterate(1) have wellformed-Y: wellformed-items Y
          by (auto simp add: wellformed-items-def)
        then have wellformed-items (Complete k (Predict k Y))
          by (simp add: wellformed-items-Complete wellformed-items-Predict)
        then have wellformed-items (Scan (Tokens k (𝒯 k u) (𝒥 k u)) k (Complete
k (Predict k Y)))
          apply (rule-tac wellformed-items-Scan)
          apply simp
          apply (simp add: Tokens-def)
            by (meson Sel-upper-bound TokensAt-subset-𝒳 𝒯-subset-TokensAt sub-
set-trans)
        then show ?case
          using Iterate.hyps(2) wellformed-items-def by blast
```

qed
}
then show *?thesis*
using *wellformed-items-def* **by** *blast*
qed

lemma *wellformed-items-$\mathcal{J}$-k-u-if-0*: *wellformed-items* ($\mathcal{J}$ *k 0*) $\Longrightarrow$ *wellformed-items* ($\mathcal{J}$ *k u*)
apply (*induct u*)
apply (*simp*)
using *wellformed-items-$\mathcal{J}$-induct-u* **by** *blast*

lemma *wellformed-items-natUnion*: ($\bigwedge$ *k*. *wellformed-items* (*I k*)) $\Longrightarrow$ *wellformed-items* (*natUnion I*)
by (*auto simp add*: *natUnion-def wellformed-items-def*)

lemma *wellformed-items-$\mathcal{I}$-k-if-0*: *wellformed-items* ($\mathcal{J}$ *k 0*) $\Longrightarrow$ *wellformed-items* ($\mathcal{I}$ *k*)
apply (*simp*)
apply (*rule wellformed-items-natUnion*)
using *wellformed-items-$\mathcal{J}$-k-u-if-0* **by** *blast*

lemma *wellformed-items-$\mathcal{J}$-$\mathcal{I}$*: *wellformed-items* ($\mathcal{J}$ *k u*) $\wedge$ *wellformed-items* ($\mathcal{I}$ *k*)
proof (*induct k arbitrary*: *u*)
case *0*
show *?case*
using *wellformed-$\mathcal{J}$-0-0 wellformed-items-$\mathcal{I}$-k-if-0 wellformed-items-$\mathcal{J}$-k-u-if-0* **by** *blast*
next
case (*Suc k*)
have *0*: *wellformed-items* ($\mathcal{J}$ (*Suc k*) *0*)
apply *simp*
using *Suc.hyps wellformed-items-π* **by** *auto*
then show *?case*
using *wellformed-items-$\mathcal{I}$-k-if-0 wellformed-items-$\mathcal{J}$-k-u-if-0* **by** *blast*
qed

lemma *wellformed-items-$\mathcal{J}$*: *wellformed-items* ($\mathcal{J}$ *k u*)
by (*simp add*: *wellformed-items-$\mathcal{J}$-$\mathcal{I}$*)

lemma *wellformed-items-$\mathcal{I}$*: *wellformed-items* ($\mathcal{I}$ *k*)
using *wellformed-items-$\mathcal{J}$-$\mathcal{I}$* **by** *blast*

lemma *funpower-consume-function*:
assumes *law*: $\bigwedge X.\ P\ X \Longrightarrow f\ (g\ X) = h\ (f\ X) \wedge P\ (g\ X)$
shows $P\ I \Longrightarrow P$ (*funpower g n I*) $\wedge$ *f* (*funpower g n I*) = *funpower h n* (*f I*)
proof (*induct n*)
case *0*
then show *?case* **by** *simp*

next
  case (*Suc n*)
  then have *S1*: *P* (*funpower g n I*) **and** *S2*: *f* (*funpower g n I*) = *funpower h n* (*f I*)
    by *auto*
  have *law1*: $\bigwedge$ *X*. *P X* $\Longrightarrow$ *f* (*g X*) = *h* (*f X*) **using** *law* **by** *auto*
  have *law2*: $\bigwedge$ *X*. *P X* $\Longrightarrow$ *P* (*g X*) **using** *law* **by** *auto*
  show *?case*
    apply *simp*
    apply (*subst law1*[**where** *X=funpower g n I*])
    apply (*simp add*: *S1*)
    apply (*subst S2*)
    apply *auto*
    apply (*rule law2*)
    apply (*simp add*: *S1*)
    done
qed

lemma *limit-consume-function*:
  assumes *continuous*: *continuous f*
  assumes *law*: $\bigwedge$ *X*. *P X* $\Longrightarrow$ *f* (*g X*) = *h* (*f X*) $\wedge$ *P* (*g X*)
  assumes *setmonotone*: *setmonotone g*
  shows *P I* $\Longrightarrow$ *f* (*limit g I*) = *limit h* (*f I*)
proof −
  have *1*: *f* (*limit g I*) = *f* (*natUnion* (λ*n*. *funpower g n I*))
    by (*simp add*: *limit-def*)
  have *chain* (λ*n*. *funpower g n I*) **by** (*simp add*: *setmonotone setmonotone-implies-chain-funpower*)
  from *continuous-apply*[*OF continuous this*]
  have *swap*: *f* (*natUnion* (λ*n*. *funpower g n I*)) = *natUnion* (*f* $\circ$ (λ*n*. *funpower g n I*)) **by** *blast*
  have *f o* (λ*n*. *funpower g n I*) = (λ *n*. *f* (*funpower g n I*)) **by** *auto*
  also have *P I* $\Longrightarrow$ (λ *n*. *f* (*funpower g n I*)) = (λ *n*. *funpower h n* (*f I*))
    by (*metis funpower-consume-function*[**where** *P=P* **and** *f=f* **and** *g=g* **and** *h=h*, *OF law*, *simplified*])
  ultimately have *P I* $\Longrightarrow$ *f o* (λ*n*. *funpower g n I*) = (λ *n*. *funpower h n* (*f I*)) **by** *auto*
  with *swap* **have** *2*: *P I* $\Longrightarrow$ *f* (*natUnion* (λ*n*. *funpower g n I*)) = *natUnion* (λ *n*. *funpower h n* (*f I*))
    by *auto*
  have *3*: *natUnion* (λ *n*. *funpower h n* (*f I*)) = *limit h* (*f I*)
    by (*simp add*: *limit-def*)
  assume *P I*
  with *1 2 3* **show** *?thesis* **by** *auto*
qed

lemma *items-le-*π*-swap*:
  assumes *wellformed-I*: *wellformed-items I*
  assumes *T*: *T* $\subseteq$ $\mathcal{X}$ *k*
  shows *items-le k* (π *k T I*) = π *k* (*empty-tokens T*) (*items-le k I*)

proof −
let *?g* = (*Scan T k*) *o* (*Complete k*) *o* (*Predict k*)
let *?h* = (*Scan* (*empty-tokens T*) *k*) *o* (*Complete k*) *o* (*Predict k*)
have *law1*: $\bigwedge$ *I*. *wellformed-items I* $\Longrightarrow$ *items-le k* (*?g I*) = *?h* (*items-le k I*)
using *LocalLexing.wellformed-items-Predict LocalLexing-axioms items-le-Complete*

items-le-Predict items-le-Scan **by** *auto*
have *law2*: $\bigwedge$ *I*. *wellformed-items I* $\Longrightarrow$ *wellformed-items* (*?g I*)
by (*simp add*: *T wellformed-items-Complete wellformed-items-Predict well-formed-items-Scan*)
show *?thesis*
apply (*subst* π*-functional*)
apply (*subst limit-consume-function*[**where** *P*=*wellformed-items* **and** *h*=*?h*])
apply (*simp add*: *items-le-pointwise pointbased-implies-continuous pointwise-implies-pointbased*)
using *law1 law2* **apply** *blast*
apply (*simp add*: π*-step-regular regular-implies-setmonotone*)
apply (*rule wellformed-I*)
apply (*subst* π*-functional*)
apply *blast*
done
qed

lemma *items-le-idempotent*: *items-le k* (*items-le k I*) = *items-le k I*
using *items-le-def* **by** *auto*

lemma *paths-le-idempotent*: *paths-le k* (*paths-le k P*) = *paths-le k P*
using *paths-le-def* **by** *auto*

lemma *items-le-fix-D*:
assumes *items-le-fix*: *items-le k I* = *I*
assumes *x-dom*: $x \in I$
shows *item-end* $x \leq k$
using *items-le-def items-le-fix x-dom* **by** *blast*

lemma *remove-paths-le-in-subset-Gen*:
assumes *items-le k I* = *I*
assumes $I \subseteq$ *Gen P*
shows $I \subseteq$ *Gen* (*paths-le k P*)
proof −
{
fix *x* :: *item*
assume *x-dom*: $x \in I$
then have *x-item-end*: *item-end* $x \leq k$ **using** *assms items-le-fix-D* **by** *auto*
have $x \in$ *Gen P* **using** *assms x-dom* **by** *auto*
then obtain *p* **where** *p*: $p \in P \wedge$ *pvalid p x* **using** *Gen-implies-pvalid* **by** *blast*

have *charslength-p*: *charslength* $p \leq k$ **using** *p pvalid-item-end x-item-end* **by** *auto*
then have $p \in$ *paths-le k P* **by** (*simp add*: *p paths-le-def*)

then have *x* ∈ *Gen* (*paths-le k P*) **using** *Gen-def p* **by** *blast*
}
then show *?thesis* **by** *blast*
qed

lemma *mono-Gen*: *mono Gen*
by (*auto simp add*: *mono-def Gen-def*)

lemma *empty-tokens-idempotent*: *empty-tokens* (*empty-tokens T*) = *empty-tokens T*
by (*auto simp add*: *empty-tokens-def*)

lemma *empty-tokens-is-filter*: *empty-tokens T* ⊆ *T*
by (*auto simp add*: *empty-tokens-def*)

lemma *items-le-paths-le*: *items-le k* (*Gen P*) = *Gen* (*paths-le k P*)
using *LocalLexing.Gen-def LocalLexing.items-le-def LocalLexing-axioms paths-le-def*

pvalid-item-end **by** *auto*

lemma *bin-items-le*[*symmetric*]: *bin I k* = *bin* (*items-le k I*) *k*
by (*auto simp add*: *bin-def items-le-def*)

lemma *TokensAt-items-le*[*symmetric*]: *TokensAt k I* = *TokensAt k* (*items-le k I*)
using *TokensAt-def bin-items-le* **by** *blast*

lemma *by-length-paths-le*[*symmetric*]: *by-length k P* = *by-length k* (*paths-le k P*)
using *by-length.simps paths-le-def* **by** *auto*

lemma $\mathcal{W}$*-paths-le*[*symmetric*]: $\mathcal{W}$ *P k* = $\mathcal{W}$ (*paths-le k P*) *k*
using $\mathcal{W}$*-def by-length-paths-le* **by** *blast*

theorem $\mathcal{T}$*-equals-*$\mathcal{Z}$*-induct-step*:
assumes *induct*: *items-le k* ($\mathcal{J}$ *k u*) = *Gen* (*paths-le k* ($\mathcal{P}$ *k u*))
assumes *induct-tokens*: $\mathcal{T}$ *k u* = $\mathcal{Z}$ *k u*
shows $\mathcal{T}$ *k* (*Suc u*) = $\mathcal{Z}$ *k* (*Suc u*)
proof −
have *TokensAt k* ($\mathcal{J}$ *k u*) = *TokensAt k* (*items-le k* ($\mathcal{J}$ *k u*))
using *TokensAt-items-le* **by** *blast*
also have *TokensAt k* (*items-le k* ($\mathcal{J}$ *k u*)) = *TokensAt k* (*Gen* (*paths-le k* ($\mathcal{P}$ *k u*)))
using *induct* **by** *auto*
ultimately have *TokensAt-le*: *TokensAt k* ($\mathcal{J}$ *k u*) = *TokensAt k* (*Gen* (*paths-le k* ($\mathcal{P}$ *k u*)))
by *auto*
have *TokensAt k* ($\mathcal{J}$ *k u*) = $\mathcal{W}$ ($\mathcal{P}$ *k u*) *k*
apply (*subst TokensAt-le*)
apply (*subst* $\mathcal{W}$*-paths-le*[*symmetric*])
apply (*rule-tac thmD4*[*symmetric*])

```
    using 𝔓-covers-𝒫 paths-le-is-filter by blast
  then show ?thesis
    by (simp add: induct-tokens Tokens-def 𝒴-def)
qed

theorem thmD9:
  assumes induct: items-le k (𝒥 k u) = Gen (paths-le k (𝒫 k u))
  assumes induct-tokens: 𝒯 k u = 𝒵 k u
  assumes k: k ≤ length Doc
  shows items-le k (𝒥 k (Suc u)) ⊆ Gen (paths-le k (𝒫 k (Suc u)))
proof −
  have t1: items-le k (𝒥 k (Suc u)) = items-le k (π k (𝒯 k (Suc u)) (𝒥 k u))
    by auto
  have t2: items-le k (π k (𝒯 k (Suc u)) (𝒥 k u)) =
    π k (empty-tokens (𝒯 k (Suc u))) (items-le k (𝒥 k u))
    apply (subst items-le-π-swap)
    apply (simp add: wellformed-items-𝒥)
    using TokensAt-subset-𝒳 𝒯-subset-TokensAt apply blast
    by blast
  have t3: π k (empty-tokens (𝒯 k (Suc u))) (items-le k (𝒥 k u)) =
    π k (empty-tokens (𝒯 k (Suc u))) (Gen (paths-le k (𝒫 k u)))
    using induct by auto
  have 𝒫-subset: 𝒫 k u ⊆ 𝒫 k (Suc u) using subset-𝒫Suc by blast
  then have paths-le k (𝒫 k u) ⊆ paths-le k (𝒫 k (Suc u))
    by (simp add: mono-subset-elem paths-le-mono subsetI)
  with mono-Gen have Gen (paths-le k (𝒫 k u)) ⊆ Gen (paths-le k (𝒫 k (Suc u)))
    by (simp add: mono-subset-elem subsetI)
  then have t4: π k (empty-tokens (𝒯 k (Suc u))) (Gen (paths-le k (𝒫 k u))) ⊆
    π k (empty-tokens (𝒯 k (Suc u))) (Gen (paths-le k (𝒫 k (Suc u))))
    by (rule monoD[OF mono-π])
  have 𝒯-eq-𝒵: 𝒯 k (Suc u) = 𝒵 k (Suc u)
    using 𝒯-equals-𝒵-induct-step assms(1) induct-tokens by blast
  have t5: π k (empty-tokens (𝒯 k (Suc u))) (Gen (paths-le k (𝒫 k (Suc u)))) ⊆
    Gen (paths-le k (𝒫 k (Suc u)))
    apply (rule-tac remove-paths-le-in-subset-Gen)
    apply (subst items-le-π-swap)
    using wellformed-items-Gen apply blast
    using 𝒯-eq-𝒵 𝒵-subset-𝒳 empty-tokens-is-filter apply blast
    apply (simp only: empty-tokens-idempotent paths-le-idempotent items-le-paths-le)
    apply (rule-tac thmD5)
    using items-le-is-filter items-le-paths-le apply blast
    apply (rule k)
    using 𝒯-eq-𝒵 empty-tokens-is-filter by blast
  from t1 t2 t3 t4 t5 show ?thesis using subset-trans by blast
qed

end

end
```

theory *Ladder*
imports *TheoremD9*
begin

context *LocalLexing* **begin**

definition *LeftDerivationFix* :: *sentence* ⇒ *nat* ⇒ *derivation* ⇒ *nat* ⇒ *sentence* ⇒ *bool*
where
LeftDerivationFix α *i D j* β = (*is-sentence* α ∧ *is-sentence* β
∧ *LeftDerivation* α *D* β ∧ *i* < *length* α ∧ *j* < *length* β
∧ α ! *i* = β ! *j* ∧ (∃ *E F*. *D* = *E*@(*derivation-shift F 0* (*Suc j*)) ∧
LeftDerivation (*take i* α) *E* (*take j* β) ∧
LeftDerivation (*drop* (*Suc i*) α) *F* (*drop* (*Suc j*) β)))

definition *LeftDerivationIntro* ::
sentence ⇒ *nat* ⇒ *rule* ⇒ *nat* ⇒ *derivation* ⇒ *nat* ⇒ *sentence* ⇒ *bool*
where
LeftDerivationIntro α *i r ix D j* γ = (∃ β. *LeftDerives1* α *i r* β ∧
ix < *length* (*snd r*) ∧ (*snd r*) ! *ix* = γ ! *j* ∧
LeftDerivationFix β (*i* + *ix*) *D j* γ)

lemma *LeftDerivationFix-empty*[*simp*]: *is-sentence* α ⟹ *i* < *length* α ⟹ *LeftDerivationFix* α *i* [] *i* α
apply (*auto simp add*: *LeftDerivationFix-def*)
apply (*rule-tac x*=[] **in** *exI*)
apply *auto*
done

lemma *Derive-empty*[*simp*]: *Derive a* [] = *a*
by (*auto simp add*: *Derive-def*)

lemma *LeftDerivation-append1*: *LeftDerivation a* (*D*@[(*i*, *r*)]) *c* ⟹ ∃ *b*. *LeftDerivation a D b*
∧ *LeftDerives1 b i r c*
by (*simp add*: *LeftDerivation-append*)

lemma *Derivation-append1*: *Derivation a* (*D*@[(*i*, *r*)]) *c* ⟹ ∃ *b*. *Derivation a D b*
∧ *Derives1 b i r c*
by (*simp add*: *Derivation-append*)

lemma *Derivation-take-derive*:
assumes *Derivation a D b*
shows *Derivation a* (*take n D*) (*Derive a* (*take n D*))
by (*metis Derivation-append Derive append-take-drop-id assms*)

lemma *LeftDerivation-take-derive*:
assumes *LeftDerivation a D b*

shows *LeftDerivation a* (*take n D*) (*Derive a* (*take n D*))
by (*metis Derive LeftDerivation-append LeftDerivation-implies-Derivation append-take-drop-id assms*)

lemma *Derivation-Derive-take-Derives1*:
assumes $N \neq 0$
assumes $N \leq$ *length D*
assumes *Derivation a D b*
assumes α: α = *Derive a* (*take* (*N* − *1*) *D*)
assumes β = *Derive a* (*take N D*)
shows *Derives1* α (*fst* (*D* ! (*N* − *1*))) (*snd* (*D* ! (*N* − *1*))) β
proof −
let *?D1* = *take* (*N* − *1*) *D*
let *?D2* = *take N D*
from *assms* **have** *app*: *?D2* = *?D1* @ [*D* ! (*N* − *1*)]
apply *auto*
by (*metis Suc-less-eq Suc-pred le-imp-less-Suc take-Suc-conv-app-nth*)
from *assms* **have** *Derivation a ?D2* β
using *Derivation-take-derive* **by** *blast*
with *app* **show** *?thesis*
using *Derivation.simps Derivation-append Derive* α **by** *auto*
qed

lemma *LeftDerivation-Derive-take-LeftDerives1*:
assumes $N \neq 0$
assumes $N \leq$ *length D*
assumes *LeftDerivation a D b*
assumes α: α = *Derive a* (*take* (*N* − *1*) *D*)
assumes β = *Derive a* (*take N D*)
shows *LeftDerives1* α (*fst* (*D* ! (*N* − *1*))) (*snd* (*D* ! (*N* − *1*))) β
proof −
let *?D1* = *take* (*N* − *1*) *D*
let *?D2* = *take N D*
from *assms* **have** *app*: *?D2* = *?D1* @ [*D* ! (*N* − *1*)]
apply *auto*
by (*metis Suc-less-eq Suc-pred le-imp-less-Suc take-Suc-conv-app-nth*)
from *assms* **have** *LeftDerivation a ?D2* β
using *LeftDerivation-take-derive* **by** *blast*
with *app* **show** *?thesis*
by (*metis Derive LeftDerivation-append1 LeftDerivation-implies-Derivation* α *prod.collapse*)
qed

lemma *LeftDerives1-skip-prefix*:
length a $\leq$ *i* $\Longrightarrow$ *LeftDerives1* (*a*@*b*) *i r* (*a*@*c*) $\Longrightarrow$ *LeftDerives1 b* (*i* − *length a*) *r c*
apply (*auto simp add*: *LeftDerives1-def*)
using *leftmost-skip-prefix* **apply** *blast*
by (*simp add*: *Derives1-skip-prefix*)

lemma *LeftDerives1-skip-suffix*:
assumes *i*: *i* < *length a*
assumes *D*: *LeftDerives1* (*a*@*c*) *i r* (*b*@*c*)
shows *LeftDerives1 a i r b*
proof −
note *Derives1-def*[**where** *u*=*a*@*c* **and** *v*=*b*@*c* **and** *i*=*i* **and** *r*=*r*]
then have ∃ *x y N α*.
a @ *c* = *x* @ [*N*] @ *y* ∧
b @ *c* = *x* @ *α* @ *y* ∧ *is-sentence x* ∧ *is-sentence y* ∧ (*N*, *α*) ∈ 𝔑 ∧ *r* = (*N*, *α*) ∧ *i* = *length x*
using *D LeftDerives1-implies-Derives1* **by** *auto*
then obtain *x y N α* **where** *split*:
a @ *c* = *x* @ [*N*] @ *y* ∧
b @ *c* = *x* @ *α* @ *y* ∧ *is-sentence x* ∧ *is-sentence y* ∧ (*N*, *α*) ∈ 𝔑 ∧ *r* = (*N*, *α*) ∧ *i* = *length x*
by *blast*
from *split* **have** *length* (*a*@*c*) = *length* (*x* @ [*N*] @ *y*) **by** *auto*
then have *length a* + *length c* = *length x* + *length y* + *1* **by** *simp*
with *split* **have** *length a* + *length c* = *i* + *length y* + *1* **by** *simp*
with *i* **have** *len-c-y*: *length c* ≤ *length y* **by** *arith*
let *?y* = *take* (*length y* − *length c*) *y*
from *split* **have** *ac*: *a* @ *c* = (*x* @ [*N*]) @ *y* **by** *auto*
note *cancel-suffix*[**where** *a*=*a* **and** *c* = *c* **and** *b* = *x*@[*N*] **and** *d* = *y*, *OF ac len-c-y*]
then have *a*: *a* = *x* @ [*N*] @ *?y* **by** *auto*
from *split* **have** *bc*: *b* @ *c* = (*x* @ *α*) @ *y* **by** *auto*
note *cancel-suffix*[**where** *a*=*b* **and** *c* = *c* **and** *b* = *x*@*α* **and** *d* = *y*, *OF bc len-c-y*]
then have *b*: *b* = *x* @ *α* @ *?y* **by** *auto*
from *split len-c-y a b* **show** *?thesis*
apply (*simp only*: *LeftDerives1-def Derives1-def*)
apply (*rule-tac conjI*)
using *D LeftDerives1-def i leftmost-cons-less* **apply** *blast*
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*=*?y* **in** *exI*)
apply (*rule-tac x*=*N* **in** *exI*)
apply (*rule-tac x*=*α* **in** *exI*)
apply *auto*
by (*rule is-sentence-take*)
qed

lemma *LeftDerives1-X-is-part-of-rule*[*consumes 2*, *case-names Suffix Prefix*]:
assumes *aXb*: *LeftDerives1 δ i r* (*a*@[*X*]@*b*)
assumes *split*: *splits-at δ i α N β*
assumes *prefix*: ⋀ *β*. *δ* = *a* @ [*X*] @ *β* ⟹ *length a* < *i* ⟹ *is-word* (*a* @ [*X*]) ⟹
LeftDerives1 β (*i* − *length a* − *1*) *r b* ⟹ *False*
assumes *suffix*: ⋀ *α*. *δ* = *α* @ [*X*] @ *b* ⟹ *LeftDerives1 α i r a* ⟹ *False*

shows $\exists\ u\ v.\ a = \alpha\ @\ u \wedge b = v\ @\ \beta \wedge (snd\ r) = u@[X]@v$
proof −
have *aXb-old*: *Derives1* δ *i r* $(a@[X]@b)$
using *LeftDerives1-implies-Derives1 aXb* **by** *blast*
have *prefix-or*: *is-prefix* α *a* $\vee$ *is-proper-prefix a* α
by (*metis Derives1-prefix split aXb-old is-prefix-eq-proper-prefix*)
have *is-word-*α: *is-word* α
using *LeftDerives1-splits-at-is-word aXb assms*(*2*) **by** *blast*
have *is-proper-prefix a* $\alpha \Longrightarrow$ *False*
proof −
assume *proper*:*is-proper-prefix a* α
then have $\exists\ u.\ u \neq [] \wedge \alpha = a@u$ **by** (*metis is-proper-prefix-def*)
then obtain *u* **where** *u*: $u \neq [] \wedge \alpha = a@u$ **by** *blast*
note *splits-at* = *splits-at-*α[*OF aXb-old split*] *splits-at-combine*[*OF split*]
from *splits-at* **have** $\alpha 1$: α = *take i* δ **by** *blast*
from *splits-at* **have** $\alpha 2$: α = *take i* $(a@[X]@b)$ **by** *blast*
from *splits-at* **have** *len*α: *length* α = *i* **by** *blast*
with *proper* **have** *lena*: *length a* < *i*
using *append-eq-conv-conj drop-eq-Nil leI u* **by** *auto*
with *is-word-*α $\alpha 2$ **have** *is-word-aX*: *is-word* $(a@[X])$
by (*simp add*: *is-word-terminals not-less take-Cons′ u*)
from *u* $\alpha 2$ **have** $a@u$ = *take i* $(a@[X]@b)$ **by** *auto*
with *lena* **have** *u* = *take* (*i* − *length a*) $([X]@b)$ **by** (*simp add*: *less-or-eq-imp-le*)

with *lena* **have** *uX*: *u* = $[X]@($*take* (*i* − *length a* − *1*) *b*) **by** (*simp add*: *not-less take-Cons′*)
let *?β* = (*take* (*i* − *length a* − *1*) *b*) @ [*N*] @ β
from *splits-at* **have** *f1*: $\delta = \alpha$ @ [*N*] @ β **by** *blast*
with *u uX* **have** *f2*: δ = *a* @ [*X*] @ *?β* **by** *simp*
note *skip* = *LeftDerives1-skip-prefix*[**where** *a* = *a* @ [*X*] **and** *b* = *?β* **and** *r* = *r* **and** *i* = *i* **and** *c* = *b*]
then have *D*: *LeftDerives1 ?β* (*i* − *length a* − *1*) *r b*
using *One-nat-def Suc-leI aXb append-assoc diff-diff-left f2 lena length-Cons length-append length-append-singleton list.size*(*3*) **by** *fastforce*
note *prefix*[*OF f2 lena is-word-aX D*]
then show *False* **.**
qed
with *prefix-or* **have** *is-prefix*: *is-prefix* α *a* **by** *blast*

from *aXb* **have** *aXb′*: *LeftDerives1* δ *i r* $((a@[X])@b)$ **by** *auto*
then have *aXb′-old*: *Derives1* δ *i r* $((a@[X])@b)$ **by** (*simp add*: *LeftDerives1-implies-Derives1*)

note *Derives1-suffix*[*OF aXb′-old split*]
then have *suffix-or*: *is-suffix* β *b* $\vee$ *is-proper-suffix b* β
by (*metis is-suffix-eq-proper-suffix*)
have *is-proper-suffix b* $\beta \Longrightarrow$ *False*
proof −
assume *proper*: *is-proper-suffix b* β
then have $\exists\ u.\ u \neq [] \wedge \beta = u@b$ **by** (*metis is-proper-suffix-def*)

then obtain *u* **where** *u*: $u \neq [] \wedge \beta = u@b$ **by** *blast*
note *splits-at* = *splits-at-β*[*OF aXb-old split*] *splits-at-combine*[*OF split*]
from *splits-at* **have** *β1*: $\beta = drop\ (Suc\ i)\ \delta$ **by** *blast*
from *splits-at* **have** *β2*: $\beta = drop\ (i + length\ (snd\ r))\ (a\ @\ [X]\ @\ b)$ **by** *blast*
from *splits-at* **have** *lenβ*: $length\ \beta = length\ \delta - i - 1$ **by** *blast*
with *proper* **have** *lenb*: $length\ b < length\ \beta$ **by** (*metis is-proper-suffix-length-cmp*)

from *u β2* **have** $u@b = drop\ (i + length\ (snd\ r))\ ((a\ @\ [X])\ @\ b)$ **by** *auto*
hence $u = drop\ (i + length\ (snd\ r))\ (a\ @\ [X])$
by (*metis drop-cancel-suffix*)
hence *uX*: $u = drop\ (i + length\ (snd\ r))\ a\ @\ [X]$ **by** (*metis drop-keep-last u*)
let $?\alpha = \alpha\ @\ [N]\ @\ (drop\ (i + length\ (snd\ r))\ a)$
from *splits-at* **have** *f1*: $\delta = \alpha\ @\ [N]\ @\ \beta$ **by** *blast*
with *u uX* **have** *f2*: $\delta = ?\alpha\ @\ [X]\ @\ b$ **by** *simp*
note *skip* = *LeftDerives1-skip-suffix*[**where** $a = ?\alpha$ **and** $c = [X]@b$ **and** $b=a$
and
$r = r$ **and** $i = i$]
have *f3*: $i < length\ (\alpha\ @\ [N]\ @\ drop\ (i + length\ (snd\ r))\ a)$
proof −
have *f1*: $1 + i + length\ b = length\ [X] + length\ b + i$
by (*metis Groups.add-ac*(*2*) *Suc-eq-plus1-left length-Cons list.size*(*3*) *list.size*(*4*)
semiring-normalization-rules(*22*))
have *f2*: $length\ \delta - i - 1 = length\ ((\alpha\ @\ [N]\ @\ drop\ (i + length\ (snd\ r))\ a)$
$@\ [X]\ @\ b) - Suc\ i$
by (*metis f2 length-drop splits-at*(*1*))
have $length\ ([]::symbol\ list) \neq length\ \delta - i - 1 - length\ b$
by (*metis add-diff-cancel-right′ append-Nil2 append-eq-append-conv lenβ*
length-append u)
then have $length\ ([]::symbol\ list) \neq length\ \alpha + length\ ([N]\ @\ drop\ (i + length$
$(snd\ r))\ a) - i$
using *f2 f1* **by** (*metis Suc-eq-plus1-left add-diff-cancel-right′ diff-diff-left*
length-append)
then show *?thesis*
by *auto*
qed
from *aXb f2* **have** *D*: *LeftDerives1* $(?\alpha\ @\ [X]\ @\ b)\ i\ r\ (a@[X]@b)$ **by** *auto*
note *skip*[*OF f3 D*]
note *suffix*[*OF f2 skip*[*OF f3 D*]]
then show *False* **.**
qed
with *suffix-or* **have** *is-suffix*: *is-suffix* β *b* **by** *blast*

from *is-prefix* **have** $\exists\ u.\ a = \alpha\ @\ u$ **by** (*auto simp add*: *is-prefix-def*)
then obtain *u* **where** *u*: $a = \alpha\ @\ u$ **by** *blast*
from *is-suffix* **have** $\exists\ v.\ b = v\ @\ \beta$ **by** (*auto simp add*: *is-suffix-def*)
then obtain *v* **where** *v*: $b = v\ @\ \beta$ **by** *blast*

from *u v splits-at-combine*[*OF split*] *aXb* **have** *D*:*LeftDerives1* $(\alpha@[N]@\beta)\ i\ r$
$(\alpha@(u@[X]@v)@\beta)$

by *simp*
from *splits-at-α*[*OF aXb-old split*] **have** *i*: *length α = i* **by** *blast*
from *i* **have** *i1*: *length α ≤ i* **and** *i2*: *i ≤ length α* **by** *auto*
note *LeftDerives1-skip-suffix*[*OF - LeftDerives1-skip-prefix*[*OF i1 D*], *simplified*, *OF i2*]
then have *LeftDerives1* [*N*] *0 r* (*u* @ [*X*] @ *v*) **by** *auto*
then have *Derives1* [*N*] *0 r* (*u* @ [*X*] @ *v*)
using *LeftDerives1-implies-Derives1* **by** *auto*
then have *r*: *snd r = u* @ [*X*] @ *v*
by (*metis Derives1-split append-Cons append-Nil length-0-conv list.inject self-append-conv*)

show *?thesis* **using** *u v r* **by** *auto*
qed

lemma *LeftDerivationFix-grow-suffix*:
assumes *LDF*: *LeftDerivationFix* (*b1*@[*X*]@*b2*) (*length b1*) *D j c*
assumes *suffix-b2*: *LeftDerives1 suffix e r b2*
assumes *is-word-b1X*: *is-word* (*b1*@[*X*])
shows *LeftDerivationFix* (*b1*@[*X*]@*suffix*) (*length b1*) ((*e* + *length* (*b1*@[*X*]), *r*)#*D*) *j c*
proof −
from *LDF* **have** *LDF'*: *is-sentence* (*b1*@[*X*]@*b2*) ∧ *is-sentence c* ∧
LeftDerivation (*b1* @ [*X*] @ *b2*) *D c* ∧ *length b1* < *length* (*b1* @ [*X*] @ *b2*) ∧
j < *length c* ∧
(*b1* @ [*X*] @ *b2*) ! *length b1* = *c* ! *j* ∧
(∃ *E F*. *D = E* @ *derivation-shift F 0* (*Suc j*) ∧
LeftDerivation (*take* (*length b1*) (*b1* @ [*X*] @ *b2*)) *E* (*take j c*) ∧
LeftDerivation (*drop* (*Suc* (*length b1*)) (*b1* @ [*X*] @ *b2*)) *F* (*drop* (*Suc j*) *c*))
using *LeftDerivationFix-def* **by** *blast*
then obtain *E F* **where** *EF*: *D = E* @ *derivation-shift F 0* (*Suc j*) ∧
LeftDerivation (*take* (*length b1*) (*b1* @ [*X*] @ *b2*)) *E* (*take j c*) ∧
LeftDerivation (*drop* (*Suc* (*length b1*)) (*b1* @ [*X*] @ *b2*)) *F* (*drop* (*Suc j*) *c*)
by *blast*
then have *LD-b1-c*: *LeftDerivation b1 E* (*take j c*) **by** *simp*
with *is-word-b1X* **have** *E*: *E* = []
using *LeftDerivation-implies-Derivation is-word-Derivation is-word-append* **by** *blast*
then have *b1-def*: *b1 = take j c* **using** *LD-b1-c* **by** *auto*
then have *b1-len*: *j = length b1*
by (*simp add*: *LDF' dual-order.strict-implies-order min.absorb2*)
have *D*: *D = derivation-shift F 0* (*Suc j*) **using** *EF E* **by** *simp*
have *step*: *LeftDerives1* (*b1* @ [*X*] @ *suffix*) (*Suc* (*e* + *length b1*)) *r* (*b1* @ [*X*] @ *b2*) ∧
LeftDerivation (*b1* @ [*X*] @ *b2*) *D c*
by (*metis LDF' LeftDerives1-append-prefix add-Suc-right append-assoc assms*(*2*) *is-word-b1X*
length-append-singleton)
then have *is-sentence-b1Xsuffix*: *is-sentence* (*b1* @ [*X*] @ *suffix*)
using *Derives1-sentence1 LeftDerives1-implies-Derives1* **by** *blast*

```
  have X-eq-cj: X = c ! j using LDF' by auto
  show ?thesis
    apply (simp add: LeftDerivationFix-def)
    apply (rule conjI)
    using is-sentence-b1Xsuffix apply simp
    apply (rule conjI)
    using LDF' apply simp
    apply (rule conjI)
    using step apply force
    apply (rule conjI)
    using LDF' apply simp
    apply (rule conjI)
    apply (rule X-eq-cj)
    apply (rule-tac x=[] in exI)
    apply (rule-tac x=(e, r)#F in exI)
    apply auto
    apply (rule b1-len[symmetric])
    apply (rule D)
    apply (rule b1-def)
    apply (rule-tac x=b2 in exI)
    apply (simp add: suffix-b2)
    using EF by auto
qed

lemma Derives1-append-suffix:
  assumes Derives1: Derives1 v i r w
  assumes u: is-sentence u
  shows Derives1 (v@u) i r (w@u)
proof -
  have ∃ α N β. splits-at v i α N β using assms splits-at-ex by auto
  then obtain α N β where split-v: splits-at v i α N β by blast
   have split-w: w = α@(snd r)@β using assms split-v splits-at-combine-dest by
blast
  have split-uv: splits-at (v@u) i α N (β@u)
    by (simp add: split-v splits-at-append)
  have is-sentence-uv: is-sentence (v@u)
    using Derives1 Derives1-sentence1 is-sentence-concat u by blast
  show ?thesis
   by (metis Derives1 Derives1-nonterminal Derives1-rule append-assoc is-sentence-uv

        split-uv split-v split-w splits-at-implies-Derives1)
qed

lemma leftmost-append-suffix: leftmost i v ⟹ leftmost i (v@u)
by (simp add: leftmost-def nth-append)

lemma LeftDerives1-append-suffix:
  assumes Derives1: LeftDerives1 v i r w
  assumes u: is-sentence u
```

shows *LeftDerives1* (*v*@*u*) *i r* (*w*@*u*)
proof −
have *1*: *Derives1 v i r w*
by (*simp add*: *Derives1 LeftDerives1-implies-Derives1*)
have *2*: *leftmost i v*
using *Derives1 LeftDerives1-def* **by** *blast*
have *3*: *is-sentence u* **using** *u* **by** *fastforce*
have *4*: *Derives1* (*v*@*u*) *i r* (*w*@*u*)
by (*simp add*: *1 3 Derives1-append-suffix*)
have *5*: *leftmost i* (*v*@*u*)
by (*simp add*: *2 leftmost-append-suffix u*)
show *?thesis*
by (*simp add*: *4 5 LeftDerives1-def*)
qed

lemma *LeftDerivationFix-is-sentence*:
LeftDerivationFix a i D j b $\Longrightarrow$ *is-sentence a* $\wedge$ *is-sentence b*
using *LeftDerivationFix-def* **by** *blast*

lemma *LeftDerivationIntro-is-sentence*:
LeftDerivationIntro α *i r ix D j* $\gamma \Longrightarrow$ *is-sentence* $\alpha \wedge$ *is-sentence* γ
by (*meson Derives1-sentence1 LeftDerivationFix-is-sentence LeftDerivationIntro-def*
LeftDerives1-implies-Derives1)

lemma *LeftDerivationFix-grow-prefix*:
assumes *LDF*: *LeftDerivationFix* (*b1*@[*X*]@*b2*) (*length b1*) *D j c*
assumes *prefix-b1*: *LeftDerives1 prefix e r b1*
shows *LeftDerivationFix* (*prefix*@[*X*]@*b2*) (*length prefix*) ((*e*, *r*)#*D*) *j c*
proof −
from *LDF* **have** *LDF′*: *LeftDerivation* (*b1* @ [*X*] @ *b2*) *D c* $\wedge$
length b1 < *length* (*b1* @ [*X*] @ *b2*) $\wedge$
j < *length c* $\wedge$
(*b1* @ [*X*] @ *b2*) ! *length b1* = *c* ! *j* $\wedge$
($\exists$ *E F*. *D* = *E* @ *derivation-shift F 0* (*Suc j*) $\wedge$
LeftDerivation (*take* (*length b1*) (*b1* @ [*X*] @ *b2*)) *E* (*take j c*) $\wedge$
LeftDerivation (*drop* (*Suc* (*length b1*)) (*b1* @ [*X*] @ *b2*)) *F* (*drop* (*Suc j*) *c*))
using *LeftDerivationFix-def* **by** *blast*
then obtain *E F* **where** *EF*: *D* = *E* @ *derivation-shift F 0* (*Suc j*) $\wedge$
LeftDerivation (*take* (*length b1*) (*b1* @ [*X*] @ *b2*)) *E* (*take j c*) $\wedge$
LeftDerivation (*drop* (*Suc* (*length b1*)) (*b1* @ [*X*] @ *b2*)) *F* (*drop* (*Suc j*) *c*)
by *blast*
then have *E-b1-c*: *LeftDerivation b1 E* (*take j c*) **by** *simp*
with *EF* **have** *F-b2-c*: *LeftDerivation b2 F* (*drop* (*Suc j*) *c*) **by** *simp*
have *step*: *LeftDerives1* (*prefix* @ [*X*] @ *b2*) *e r* (*b1* @ [*X*] @ *b2*)
using *LDF LeftDerivationFix-is-sentence LeftDerives1-append-suffix*
is-sentence-concat prefix-b1 **by** *blast*
show *?thesis*
apply (*simp add*: *LeftDerivationFix-def*)

apply (*rule conjI*)
apply (*metis Derives1-sentence1 LDF LeftDerivationFix-def LeftDerives1-implies-Derives1*

is-sentence-concat is-sentence-cons prefix-b1)
apply (*rule conjI*)
using *LDF LeftDerivationFix-is-sentence* **apply** *blast*
apply (*rule conjI*)
apply (*rule-tac x=b1@[X]@b2* **in** *exI*)
using *step* **apply** *simp*
using *LDF′* **apply** *auto*[*1*]
apply (*rule conjI*)
using *LDF′* **apply** *simp*
apply (*rule conjI*)
using *LDF′* **apply** *auto*[*1*]
apply (*rule-tac x=(e,r)#E* **in** *exI*)
apply (*rule-tac x=F* **in** *exI*)
apply (*auto simp add*: *EF F-b2-c*)
apply (*rule-tac x=b1* **in** *exI*)
apply (*simp add*: *prefix-b1 E-b1-c*)
done
qed

lemma *LeftDerivationFixOrIntro*:
LeftDerivation a D $\gamma \Longrightarrow$ *is-sentence* $\gamma \Longrightarrow j <$ *length* $\gamma \Longrightarrow$
($\exists$ *i*. *LeftDerivationFix a i D j* γ) $\vee$
($\exists$ *d* α *ix*. *d* < *length D* $\wedge$ *LeftDerivation a* (*take d D*) α $\wedge$
LeftDerivationIntro α (*fst* (*D* ! *d*)) (*snd* (*D* ! *d*)) *ix* (*drop* (*Suc d*) *D*) *j* γ)
proof (*induct length D arbitrary*: *a D* γ *j rule*: *less-induct*)

case *less*
have *length D* = *0* $\vee$ *length D* $\neq$ *0* **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
then have *D*: *D* = [] **by** *auto*
with *less* **have** $\exists$ *i*. *LeftDerivationFix a i D j* γ
apply (*rule-tac x=j* **in** *exI*)
by *auto*
then show *?case* **by** *blast*
next
case *2*
note *less2* = *2*
have $\exists$ *n* β *i*. *n* $\leq$ *length D* $\wedge$ β = *Derive a* (*take n D*) $\wedge$ *LeftDerivationFix* β
i (*drop n D*) *j* γ
apply (*rule-tac x=length D* **in** *exI*)
apply *auto*
using *Derive LeftDerivationFix-empty LeftDerivation-implies-Derivation less*
by *blast*
then show *?case*

proof (*induct rule*: *ex-minimal-witness*)
case (*Minimal N*)
then obtain *β i* **where** *Minimal-N*:
N ≤ length D ∧ β = Derive a (*take N D*) *∧ LeftDerivationFix β i* (*drop N D*) *j γ* **by** *blast*
have *N = 0 ∨ N ≠ 0* **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
with *Minimal-N* **have** *β = a* **by** *auto*
with *1 Minimal-N* **show** *?case*
apply (*rule-tac disjI1*)
by *auto*
next
case *2*
let *?δ = Derive a* (*take* (*N − 1*) *D*)
have *LeftDerives1-δ*: *LeftDerives1 ?δ* (*fst* (*D* ! (*N − 1*))) (*snd* (*D* ! (*N − 1*))) *β*
using *2.hyps LeftDerivation-Derive-take-LeftDerives1 Minimal-N less.prems*(*1*) **by** *blast*
then have *Derives1-δ*: *Derives1 ?δ* (*fst* (*D* ! (*N − 1*))) (*snd* (*D* ! (*N − 1*))) *β*
using *LeftDerives1-implies-Derives1* **by** *blast*
have *i-len*: *i < length β* **using** *Minimal-N*
by (*auto simp add*: *LeftDerivationFix-def*)
then have ∃ *X β-1 β-2*. *splits-at β i β-1 X β-2*
using *splits-at-def* **by** *blast*
then obtain *X β-1 β-2* **where** *β-split*: *splits-at β i β-1 X β-2* **by** *blast*
then have *β-combine*: *β = β-1* @ [*X*] @ *β-2* **using** *splits-at-combine* **by** *blast*
then have *LeftDerives1-δ-hyp*:
LeftDerives1 ?δ (*fst* (*D* ! (*N − 1*))) (*snd* (*D* ! (*N − 1*))) (*β-1* @ [*X*] @ *β-2*)
using *LeftDerives1-δ* **by** *blast*
from *β-split* **have** *i-def*: *i = length β-1*
by (*simp add*: *dual-order.strict-implies-order min.absorb2 splits-at-def*)
have ∃ *Y δ-1 δ-2*. *splits-at ?δ* (*fst* (*D* ! (*N − 1*))) *δ-1 Y δ-2*
using *Derives1-δ splits-at-ex* **by** *blast*
then obtain *Y δ-1 δ-2* **where** *δ-split*: *splits-at ?δ* (*fst* (*D* ! (*N − 1*))) *δ-1 Y δ-2* **by** *blast*
have *NFix*: *LeftDerivationFix* (*β-1* @ [*X*] @ *β-2*) (*length β-1*) (*drop N D*) *j γ*
using *Minimal-N β-combine i-def* **by** *auto*
from *LeftDerives1-δ-hyp δ-split*
have ∃ *u v*. *β-1 = δ-1* @ *u ∧ β-2 = v* @ *δ-2 ∧ snd* (*snd* (*D* ! (*N − 1*))) = *u* @ [*X*] @ *v*
proof (*induct rule*: *LeftDerives1-X-is-part-of-rule*)
case (*Suffix suffix*)
let *?k = N − 1*

let *?β = Derive a (take ?k D)*
let *?i = length β-1*
have *k-less*: *?k < length D* **using** *2.hyps Minimal-N* **by** *linarith*
then have *k-leq*: *?k ≤ length D* **by** *auto*
have *drop-k-d*: *drop ?k D = (D ! (N − 1))#(drop N D)*
using *2.hyps Cons-nth-drop-Suc k-less* **by** *fastforce*
from *LeftDerivationFix-grow-suffix[OF NFix Suffix(4) Suffix(3)] Suffix(1) Suffix(2) 2*
have *LeftDerivationFix ?β ?i (drop ?k D) j γ*
apply *auto*
by (*metis One-nat-def drop-k-d*)
with *Minimal(2)*[**where** *k=?k*] **show** *False*
using *2.hyps k-leq* **by** *auto*
next
case (*Prefix prefix*)
have *collapse*: (*fst (D ! (N − 1)), snd (D ! (N − 1))) # drop N D = drop (N − 1) D*
by (*metis 2.hyps Cons-nth-drop-Suc Minimal-N Suc-diff-1 neq0-conv not-less*
not-less-eq prod.collapse)
from *LeftDerivationFix-grow-prefix[OF NFix Prefix(2)] Prefix(1) collapse*
have *LeftDerivationFix ?δ (length prefix) (drop (N − 1) D) j γ* **by** *auto*
with *Minimal(2)*[**where** *k = N − 1*] **show** *False*
by (*metis Minimal-N collapse diff-le-self le-neq-implies-less less-imp-diff-less*
less-or-eq-imp-le not-Cons-self2)
qed
then obtain *u v* **where** *uv*:
β-1 = δ-1 @ u ∧ β-2 = v @ δ-2 ∧ snd (snd (D ! (N − 1))) = u @ [X] @ v **by** *blast*
have *X-1*: *snd (snd (D ! (N − Suc 0))) ! length u = X* **using** *uv* **by** *auto*
have *X-2*: *γ ! j = X* **using** *LeftDerivationFix-def NFix* **by** *auto*
show *?case*
apply (*rule disjI2*)
apply (*rule-tac x=N − 1* **in** *exI*)
apply (*rule-tac x=?δ* **in** *exI*)
apply (*rule-tac x=length u* **in** *exI*)
apply (*rule conjI*)
using *Minimal-N less2* **apply** *linarith*
apply (*rule conjI*)
using *LeftDerivation-take-derive less.prems(1)* **apply** *blast*
apply (*subst LeftDerivationIntro-def*)
apply (*rule-tac x=β* **in** *exI*)
apply *auto*
using *LeftDerives1-δ One-nat-def* **apply** *presburger*
using *uv* **apply** *auto[1]*
using *X-1 X-2* **apply** *auto[1]*
by (*metis (no-types, lifting) 2.hyps Derives1-δ Derives1-split Minimal-N One-nat-def*

Suc-diff-1 δ-split append-eq-conv-conj i-def length-append neq0-conv splits-at-def uv)
qed
qed
qed
qed

type-synonym *deriv* = *nat* × *nat* × *nat*
type-synonym *ladder* = *deriv list*

definition *deriv-n* :: *deriv* ⇒ *nat* **where**
deriv-n d = *fst d*

definition *deriv-j* :: *deriv* ⇒ *nat* **where**
deriv-j d = *fst* (*snd d*)

definition *deriv-ix* :: *deriv* ⇒ *nat* **where**
deriv-ix d = *snd* (*snd d*)

definition *deriv-i* :: *deriv* ⇒ *nat* **where**
deriv-i d = *snd* (*snd d*)

definition *ladder-j* :: *ladder* ⇒ *nat* ⇒ *nat* **where**
ladder-j L index = *deriv-j* (*L* ! *index*)

definition *ladder-i* :: *ladder* ⇒ *nat* ⇒ *nat* **where**
ladder-i L index = (*if index* = *0 then deriv-i* (*hd L*) *else ladder-j L* (*index* − *1*))

definition *ladder-n* :: *ladder* ⇒ *nat* ⇒ *nat* **where**
ladder-n L index = *deriv-n* (*L* ! *index*)

definition *ladder-prev-n* :: *ladder* ⇒ *nat* ⇒ *nat* **where**
ladder-prev-n L index = (*if index* = *0 then 0 else* (*ladder-n L* (*index* − *1*)))

definition *ladder-ix* :: *ladder* ⇒ *nat* ⇒ *nat* **where**
ladder-ix L index = (*if index* = *0 then undefined else deriv-ix* (*L* ! *index*))

definition *ladder-last-j* :: *ladder* ⇒ *nat* **where**
ladder-last-j L = *ladder-j L* (*length L* − *1*)

definition *ladder-last-n* :: *ladder* ⇒ *nat* **where**
ladder-last-n L = *ladder-n L* (*length L* − *1*)

definition *is-ladder* :: *derivation* ⇒ *ladder* ⇒ *bool* **where**
is-ladder D L = (*L* ≠ [] ∧
(∀ *u*. *u* < *length L* ⟶ *ladder-n L u* ≤ *length D*) ∧
(∀ *u v*. *u* < *v* ∧ *v* < *length L* ⟶ *ladder-n L u* < *ladder-n L v*) ∧
ladder-last-n L = *length D*)

definition *ladder-γ* :: *sentence ⇒ derivation ⇒ ladder ⇒ nat ⇒ sentence* **where**
ladder-γ a D L index = Derive a (take (ladder-n L index) D)

definition *ladder-α* :: *sentence ⇒ derivation ⇒ ladder ⇒ nat ⇒ sentence* **where**
ladder-α a D L index = (if index = 0 then a else ladder-γ a D L (index − 1))

definition *LeftDerivationIntrosAt* :: *sentence ⇒ derivation ⇒ ladder ⇒ nat ⇒ bool* **where**
LeftDerivationIntrosAt a D L index = (
let α = ladder-α a D L index in
let i = ladder-i L index in
let j = ladder-j L index in
let ix = ladder-ix L index in
let γ = ladder-γ a D L index in
let n = ladder-n L (index − 1) in
let m = ladder-n L index in
let e = D ! n in
let E = drop (Suc n) (take m D) in
i = fst e ∧
LeftDerivationIntro α i (snd e) ix E j γ)

definition *LeftDerivationIntros* :: *sentence ⇒ derivation ⇒ ladder ⇒ bool* **where**
LeftDerivationIntros a D L = (
∀ index. 1 ≤ index ∧ index < length L ⟶ LeftDerivationIntrosAt a D L index)

definition *LeftDerivationLadder* :: *sentence ⇒ derivation ⇒ ladder ⇒ sentence ⇒ bool* **where**
LeftDerivationLadder a D L b = (
LeftDerivation a D b ∧
is-ladder D L ∧
LeftDerivationFix a (ladder-i L 0) (take (ladder-n L 0) D) (ladder-j L 0) (ladder-γ a D L 0) ∧
LeftDerivationIntros a D L)

definition *mk-deriv-fix* :: *nat ⇒ nat ⇒ nat ⇒ deriv* **where**
mk-deriv-fix i n j = (n, j, i)

definition *mk-deriv-intro* :: *nat ⇒ nat ⇒ nat ⇒ deriv* **where**
mk-deriv-intro ix n j = (n, j, ix)

lemma *mk-deriv-fix-i*[*simp*]: *deriv-i (mk-deriv-fix i n j) = i*
by (*simp add*: *deriv-i-def mk-deriv-fix-def*)

lemma *mk-deriv-fix-j*[*simp*]: *deriv-j (mk-deriv-fix i n j) = j*
by (*simp add*: *deriv-j-def mk-deriv-fix-def*)

lemma *mk-deriv-fix-n*[*simp*]: *deriv-n (mk-deriv-fix i n j) = n*
by (*simp add*: *deriv-n-def mk-deriv-fix-def*)

lemma *mk-deriv-intro-i*[*simp*]: *deriv-i* (*mk-deriv-intro i n j*) = *i*
by (*simp add*: *deriv-i-def mk-deriv-intro-def*)

lemma *mk-deriv-intro-ix*[*simp*]: *deriv-ix* (*mk-deriv-intro ix n j*) = *ix*
by (*simp add*: *deriv-ix-def mk-deriv-intro-def*)

lemma *mk-deriv-intro-j*[*simp*]: *deriv-j* (*mk-deriv-intro i n j*) = *j*
by (*simp add*: *deriv-j-def mk-deriv-intro-def*)

lemma *mk-deriv-intro-n*[*simp*]: *deriv-n* (*mk-deriv-intro i n j*) = *n*
by (*simp add*: *deriv-n-def mk-deriv-intro-def*)

lemma *LeftDerivationFix-implies-ex-ladder*:
LeftDerivationFix a i D j $\gamma \Longrightarrow \exists$ *L*. *LeftDerivationLadder a D L* γ ∧
ladder-last-j L = *j* ∧ *ladder-last-n L* = *length D*
apply (*rule-tac x*=[*mk-deriv-fix i* (*length D*) *j*] **in** *exI*)
apply (*auto simp add*: *LeftDerivationLadder-def*)
apply (*simp add*: *LeftDerivationFix-def*)
apply (*simp add*: *is-ladder-def*)
apply (*auto simp add*: *ladder-i-def ladder-j-def ladder-n-def ladder-*γ*-def*)
apply (*simp add*: *ladder-last-n-def ladder-n-def*)
using *Derive LeftDerivationFix-def LeftDerivation-implies-Derivation* **apply** *blast*
apply (*simp add*: *LeftDerivationIntros-def*)
apply (*simp add*: *ladder-last-j-def ladder-j-def*)
apply (*simp add*: *ladder-last-n-def ladder-n-def*)
done

lemma *trivP*[*case-names prems*]: *P* ⟹ *P* **by** *blast*

lemma *LeftDerivationLadder-ladder-n-bound*:
assumes *LeftDerivationLadder a D L b*
assumes *index* < *length L*
shows *ladder-n L index* ≤ *length D*
using *LeftDerivationLadder-def assms*(*1*) *assms*(*2*) *is-ladder-def* **by** *blast*

lemma *LeftDerivationLadder-deriv-n-bound*:
assumes *LeftDerivationLadder a D L b*
assumes *index* < *length L*
shows *deriv-n* (*L* ! *index*) ≤ *length D*
using *LeftDerivationLadder-def assms*(*1*) *assms*(*2*) *is-ladder-def ladder-n-def* **by** *auto*

lemma *ladder-n-simp1*[*simp*]: *u* < *length L* ⟹ *ladder-n* (*L*@*L'*) *u* = *ladder-n L u*
by (*simp add*: *ladder-n-def*)

lemma *ladder-n-simp2*[*simp*]: *ladder-n* (*L*@[*d*]) (*length L*) = *deriv-n d*
by (*simp add*: *ladder-n-def*)

lemma *ladder-j-simp1*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{ladder\text{-}j}\ (L@L')\ u = \mathit{ladder\text{-}j}\ L\ u$
by (*simp add*: *ladder-j-def*)

lemma *ladder-j-simp2*[*simp*]: $\mathit{ladder\text{-}j}\ (L@[d])\ (\mathit{length}\ L) = \mathit{deriv\text{-}j}\ d$
by (*simp add*: *ladder-j-def*)

lemma *ladder-i-simp1*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{ladder\text{-}i}\ (L@L')\ u = \mathit{ladder\text{-}i}\ L\ u$
by (*auto simp add*: *ladder-i-def*)

lemma *ladder-ix-simp1*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{ladder\text{-}ix}\ (L@L')\ u = \mathit{ladder\text{-}ix}\ L$
u
by (*auto simp add*: *ladder-ix-def*)

lemma *ladder-ix-simp2*[*simp*]: $L \neq [] \Longrightarrow \mathit{ladder\text{-}ix}\ (L@[d])\ (\mathit{length}\ L) = \mathit{deriv\text{-}ix}$
d
by (*auto simp add*: *ladder-ix-def*)

lemma *ladder-γ-simp1*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{ladder\text{-}\gamma}\ a\ D\ (L@L')\ u = \mathit{ladder\text{-}\gamma}$
$a\ D\ L\ u$
by (*simp add*: *ladder-γ-def*)

lemma *ladder-γ-simp2*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{is\text{-}ladder}\ D\ L \Longrightarrow$
$\mathit{ladder\text{-}\gamma}\ a\ (D@D')\ L\ u = \mathit{ladder\text{-}\gamma}\ a\ D\ L\ u$
by (*simp add*: *is-ladder-def ladder-γ-def*)

lemma *ladder-α-simp1*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{ladder\text{-}\alpha}\ a\ D\ (L@L')\ u = \mathit{ladder\text{-}\alpha}$
$a\ D\ L\ u$
by (*simp add*: *ladder-α-def*)

lemma *ladder-α-simp2*[*simp*]: $u < \mathit{length}\ L \Longrightarrow \mathit{is\text{-}ladder}\ D\ L \Longrightarrow$
$\mathit{ladder\text{-}\alpha}\ a\ (D@D')\ L\ u = \mathit{ladder\text{-}\alpha}\ a\ D\ L\ u$
by (*simp add*: *is-ladder-def ladder-α-def*)

lemma *ladder-n-minus-1-bound*: $\mathit{is\text{-}ladder}\ D\ L \Longrightarrow \mathit{index} \geq 1 \Longrightarrow \mathit{index} < \mathit{length}$
$L \Longrightarrow$
$\mathit{ladder\text{-}n}\ L\ (\mathit{index} - \mathit{Suc}\ 0) < \mathit{length}\ D$
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-diff-1 Suc-le-lessD dual-order.strict-implies-order*

is-ladder-def le-neq-implies-less not-less)

lemma *LeftDerivationIntrosAt-ignore-appendix*:
assumes *is-ladder*: *is-ladder D L*
assumes *hyp*: *LeftDerivationIntrosAt a D L index*
assumes *index-ge*: $\mathit{index} \geq 1$
assumes *index-less*: $\mathit{index} < \mathit{length}\ L$
shows *LeftDerivationIntrosAt* $a\ (D\ @\ D')\ (L\ @\ L')$ *index*
proof −
have *index-minus-1*: $\mathit{index} - \mathit{Suc}\ 0 < \mathit{length}\ L$

using *index-less* **by** *arith*
have *is-0*: *ladder-n L index* $-$ *length D* = *0*
using *index-less is-ladder is-ladder-def* **by** *auto*
from *index-ge index-less* **show** *?thesis*
apply (*simp add*: *LeftDerivationIntrosAt-def Let-def*)
apply (*simp add*: *index-minus-1 is-ladder ladder-n-minus-1-bound is-0*)
using *hyp* **apply** (*auto simp add*: *LeftDerivationIntrosAt-def Let-def*)
done
qed

lemma *ladder-i-eq-last-j*: $L \neq [] \Longrightarrow$ *ladder-i* (*L* @ *L'*) (*length L*) = *ladder-last-j L*
by (*simp add*: *ladder-i-def ladder-last-j-def*)

lemma *ladder-last-n-intro*: $L \neq [] \Longrightarrow$ *ladder-n L* (*length L* $-$ *Suc 0*) = *ladder-last-n L*
by (*simp add*: *ladder-last-n-def*)

lemma *is-ladder-not-empty*: *is-ladder D L* $\Longrightarrow L \neq []$
using *is-ladder-def* **by** *blast*

lemma *last-ladder-*γ:
assumes *is-ladder*: *is-ladder D L*
assumes *ladder-last-n*: *ladder-last-n L* = *length D*
shows *ladder-*γ *a D L* (*length L* $-$ *Suc 0*) = *Derive a D*
proof $-$
from *is-ladder is-ladder-not-empty* **have** $L \neq []$ **by** *blast*
then show *?thesis*
by (*simp add*: *ladder-*γ*-def ladder-last-n-intro ladder-last-n*)
qed

lemma *ladder-*α*-full*:
assumes *is-ladder*: *is-ladder D L*
assumes *ladder-last-n*: *ladder-last-n L* = *length D*
shows *ladder-*α *a* (*D* @ *D'*) (*L* @ *L'*) (*length L*) = *Derive a D*
proof $-$
from *is-ladder* **have** *L-not-empty*: $L \neq []$ **by** (*simp add*: *is-ladder-def*)
with *is-ladder ladder-last-n* **show** *?thesis*
apply (*simp add*: *ladder-*α*-def*)
apply (*simp add*: *last-ladder-*γ)
done
qed

lemma *LeftDerivationIntro-implies-LeftDerivation*:
LeftDerivationIntro α *i r ix D j* $\gamma \Longrightarrow$ *LeftDerivation* α ((*i*,*r*)#*D*) γ
using *LeftDerivationFix-def LeftDerivationIntro-def* **by** *auto*

lemma *LeftDerivationLadder-grow*:
LeftDerivationLadder a D L $\alpha \Longrightarrow$ *ladder-last-j L* = *i* $\Longrightarrow$

```
  LeftDerivationIntro α i r ix E j γ ⟹
  LeftDerivationLadder a (D@[(i, r)]@E) (L@[mk-deriv-intro ix (Suc(length D +
length E)) j]) γ
proof (induct arbitrary: a D L α i r ix E j γ rule: trivP)
  case prems
  {
    fix u :: nat
    assume u < Suc (length L)
    then have u < length L ∨ u = length L by arith
    then have ladder-n (L @ [mk-deriv-intro ix (Suc (length D + length E)) j]) u
≤
      Suc (length D + length E)
    proof (induct rule: disjCases2)
      case 1
      then show ?case
        apply simp
         by (meson LeftDerivationLadder-ladder-n-bound le-Suc-eq le-add1 le-trans
prems(1))
    next
      case 2
      then show ?case
        by (simp add: ladder-n-def)
    qed
  }
  note ladder-n-ineqs = this
  {
    fix u :: nat
    fix v :: nat
    assume u-less-v: u < v
    assume v < Suc (length L)
    then have v < length L ∨ v = length L by arith
    then have ladder-n (L @ [mk-deriv-intro ix (Suc (length D + length E)) j]) u
          < ladder-n (L @ [mk-deriv-intro ix (Suc (length D + length E)) j]) v
    proof (induct rule: disjCases2)
      case 1
      with u-less-v have u-bound: u < length L by arith
      show ?case using 1 u-bound apply simp
      using prems u-less-v LeftDerivationLadder-def is-ladder-def by auto
    next
      case 2
      with u-less-v have u-bound: u < length L by arith
      have deriv-n (L ! u) ≤ length D
        using LeftDerivationLadder-deriv-n-bound prems(1) u-bound by blast
      then show ?case
        apply (simp add: u-bound)
        apply (simp add: ladder-n-def 2)
        done
    qed
  }
```

```
note ladder-n-ineqs = ladder-n-ineqs this
have is-ladder:
  is-ladder (D @ (i, r) # E) (L @ [mk-deriv-intro ix (Suc (length D + length E))
j])
  apply (auto simp add: is-ladder-def)
  using ladder-n-ineqs apply auto
  apply (simp add: ladder-last-n-def)
  done
have is-ladder-L: is-ladder D L
  using LeftDerivationLadder-def prems.prems(1) by blast
have ladder-last-n-eq-length: ladder-last-n L = length D
  using is-ladder-L is-ladder-def by blast
have L-not-empty: L ≠ []
  using LeftDerivationLadder-def is-ladder-def prems(1) by blast
{
  fix index :: nat
  assume index-ge: Suc 0 ≤ index
  assume index < Suc (length L)
  then have index < length L ∨ index = length L by arith
  then have LeftDerivationIntrosAt a (D @ (i, r) # E)
    (L @ [mk-deriv-intro ix (Suc (length D + length E)) j]) index
  proof (induct rule: disjCases2)
    case 1
    then show ?case
      using LeftDerivationIntrosAt-ignore-appendix
        LeftDerivationIntros-def LeftDerivationLadder-def One-nat-def
        index-ge prems.prems(1) by presburger
  next
    case 2
    have min-simp: ⋀ n E. min n (Suc (n + length E)) = n
      by auto
    with 2 prems is-ladder-L ladder-last-n-eq-length show ?case
      apply (simp add: LeftDerivationIntrosAt-def)
      apply (simp add: L-not-empty ladder-i-eq-last-j ladder-last-n-intro)
      apply (simp add: ladder-α-full min-simp)
      apply (simp add: ladder-γ-def)
      by (metis Derive LeftDerivationIntro-implies-LeftDerivation LeftDerivation-
Ladder-def
        LeftDerivation-implies-Derivation LeftDerivation-implies-append)
  qed
}
then show ?case
  apply (auto simp add: LeftDerivationLadder-def)
  using prems apply (auto simp add: LeftDerivationLadder-def)[1]
  using LeftDerivationFix-def LeftDerivationIntro-def LeftDerivation-append ap-
ply auto[1]
  using is-ladder apply simp
  using L-not-empty apply simp
  using LeftDerivationLadder-def LeftDerivationLadder-ladder-n-bound ladder-γ-def
```

prems.prems(*1*) **apply** *auto*[*1*]
apply (*subst LeftDerivationIntros-def*)
apply *auto*
done
qed

lemma *LeftDerivationIntro-bounds-ij*:
LeftDerivationIntro α *i r ix D j* $\beta \Longrightarrow i <$ *length* $\alpha \wedge j <$ *length* β
by (*meson Derives1-bound LeftDerivationFix-def LeftDerivationIntro-def LeftDerives1-implies-Derives1*)

theorem *LeftDerivationLadder-exists*: *LeftDerivation a D* $\gamma \Longrightarrow$ *is-sentence* $\gamma \Longrightarrow$ $j <$ *length* $\gamma \Longrightarrow$
$\exists$ *L*. *LeftDerivationLadder a D L* $\gamma \wedge$ *ladder-last-j L* = *j*
proof (*induct length D arbitrary*: *a D* γ *j rule*: *less-induct*)
case *less*
from *LeftDerivationFixOrIntro*[*OF less*(*2*,*3*,*4*)] **show** *?case*
proof (*induct rule*: *disjCases2*)
case *1*
then obtain *i* **where** *LeftDerivationFix a i D j* γ **by** *blast*
show *?case*
using *1.hyps LeftDerivationFix-implies-ex-ladder* **by** *blast*
next
case *2*
then obtain *d* α *ix* **where** *inductrule*: $d <$ *length D* $\wedge$
LeftDerivation a (*take d D*) α $\wedge$
LeftDerivationIntro α (*fst* (*D* ! *d*)) (*snd* (*D* ! *d*)) *ix* (*drop* (*Suc d*) *D*) *j* γ **by** *blast*
then have *less-length-D*: *length* (*take d D*) < *length D*
and *LeftDerivation-*α: *LeftDerivation a* (*take d D*) α **by** *auto*
have *is-sentence-*α: *is-sentence* α **using** *LeftDerivationIntro-is-sentence inductrule* **by** *blast*
have *fst* (*D* ! *d*) < *length* α **using** *LeftDerivationIntro-bounds-ij inductrule* **by** *blast*
from *less*(*1*)[*OF less-length-D LeftDerivation-*α *is-sentence-*α, **where** *j*= *fst* (*D* ! *d*), *OF this*]
obtain *L* **where** *induct-Ladder*:
LeftDerivationLadder a (*take d D*) *L* α **and** *induct-last*: *ladder-last-j L* = *fst* (*D* ! *d*)
by *blast*
have *induct-intro*: *LeftDerivationIntro* α (*fst* (*D* ! *d*)) (*snd* (*D* ! *d*)) *ix* (*drop* (*Suc d*) *D*) *j* γ
using *inductrule* **by** *blast*
have *d* < *length D* **using** *inductrule* **by** *blast*
then have *simp-to-D*: *take d D* @ *D* ! *d* # *drop* (*Suc d*) *D* = *D*
using *id-take-nth-drop* **by** *force*
from *LeftDerivationLadder-grow*[*OF induct-Ladder induct-last induct-intro*] *simp-to-D*

```
    show ?case
      apply auto
      apply (rule-tac x=
        L @ [mk-deriv-intro ix (Suc (min (length D) d + (length D − Suc d))) j] in
exI)
      apply (simp add: ladder-last-j-def)
      done
  qed
qed

lemma LeftDerivationLadder-L-0:
  assumes LeftDerivationLadder α D L β
  assumes length L = 1
  shows ∃ i. LeftDerivationFix α i D (ladder-last-j L) β
proof −
  have is-ladder D L using assms by (auto simp add: LeftDerivationLadder-def)
  then have ladder-n: ladder-n L 0 = length D
    by (simp add: assms(2) is-ladder-def ladder-last-n-def)
  show ?thesis
    apply (rule-tac x = ladder-i L 0 in exI)
    using assms(1) apply (auto simp add: LeftDerivationLadder-def)
   by (metis Derive LeftDerivationFix-def LeftDerivation-implies-Derivation One-nat-def
assms(2)
      diff-Suc-1 ladder-last-j-def ladder-n order-refl take-all)
qed

lemma LeftDerivationFix-splits-at-derives:
  assumes LeftDerivationFix a i D j b
  shows ∃ U a1 a2 b1 b2. splits-at a i a1 U a2 ∧ splits-at b j b1 U b2 ∧
    derives a1 b1 ∧ derives a2 b2
proof −
   note hyp = LeftDerivationFix-def[where α=a and β=b and D=D and i=i
and j=j]
  from hyp obtain E F where EF:
    D = E @ derivation-shift F 0 (Suc j) ∧
      LeftDerivation (take i a) E (take j b) ∧ LeftDerivation (drop (Suc i) a) F
(drop (Suc j) b)
    using assms by blast
  show ?thesis
    apply (rule-tac x=a ! i in exI)
    apply (rule-tac x=take i a in exI)
    apply (rule-tac x=drop (Suc i) a in exI)
    apply (rule-tac x=take j b in exI)
    apply (rule-tac x=drop (Suc j) b in exI)
    using Derivation-implies-derives LeftDerivation-implies-Derivation assms hyp
      splits-at-def by blast
qed

lemma  LeftDerivation-append-suffix:
```

LeftDerivation a D b ⟹ *is-sentence c* ⟹ *LeftDerivation* (*a*@*c*) *D* (*b*@*c*)
proof (*induct D arbitrary*: *a b c*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons d D*)
then show *?case*
apply *auto*
apply (*rule-tac x*=*x*@*c* **in** *exI*)
apply *auto*
using *LeftDerives1-append-suffix* **by** *simp*
qed

lemma *LeftDerivation-impossible*: *LeftDerivation a D b* ⟹ *i* < *length a* ⟹
is-nonterminal (*a* ! *i*) ⟹ *derivation-ge D* (*Suc i*) ⟹ *D* = []
proof (*induct D*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons d D*)
then have *lm*: ⋀ *j*. *leftmost j a* ⟹ *j* ≤ *i*
by (*metis Derives1-sentence1 LeftDerivation.simps*(*2*) *LeftDerives1-implies-Derives1*

leftmost-exists leftmost-unique)
from *Cons* **show** *?case*
apply *auto*
apply (*auto simp add*: *derivation-ge-def LeftDerives1-def*)
using *lm*[**where** *j*=*fst d*] **by** *arith*
qed

lemma *derivation-ge-shift*: *derivation-ge* (*derivation-shift F 0 j*) *j*
apply (*induct F*)
apply (*auto simp add*: *derivation-ge-def*)
done

lemma *LeftDerivationFix-splits-at-nonterminal*:
assumes *LeftDerivationFix a i D j b*
assumes *is-nonterminal* (*a* ! *i*)
shows ∃ *U a1 a2 b1*. *splits-at a i a1 U a2* ∧ *splits-at b j b1 U a2* ∧ *LeftDerivation a1 D b1*
proof −
note *hyp* = *LeftDerivationFix-def*[**where** *α*=*a* **and** *β*=*b* **and** *D*=*D* **and** *i*=*i* **and** *j*=*j*]
from *hyp* **obtain** *E F* **where** *EF*:
D = *E* @ *derivation-shift F 0* (*Suc j*) ∧ *LeftDerivation* (*take i a*) *E* (*take j b*) ∧
LeftDerivation (*drop* (*Suc i*) *a*) *F* (*drop* (*Suc j*) *b*)
using *assms* **by** *blast*
have ∃ *β*. *LeftDerivation a E β* ∧ *LeftDerivation β* (*derivation-shift F 0* (*Suc j*)) *b*

using *EF LeftDerivation-append assms*(*1*) *hyp* **by** *blast*
then obtain β **where** β*-intro*:
LeftDerivation a E $\beta \wedge$ *LeftDerivation* β (*derivation-shift F 0* (*Suc j*)) *b* **by** *blast*
have *LeftDerivation* ((*take i a*)@(*drop i a*)) *E* ((*take j b*)@(*drop i a*))
by (*metis EF LeftDerivation-append-suffix append-take-drop-id assms*(*1*) *hyp is-sentence-concat*)
then have *LeftDerivation a E* ((*take j b*)@(*drop i a*)) **by** *simp*
then have β*-decomposed*: β = (*take j b*)@(*drop i a*)
using *Derivation-unique-dest LeftDerivation-implies-Derivation* β*-intro* **by** *blast*

then have β ! *j* = *a* ! *i*
by (*metis Cons-nth-drop-Suc assms*(*1*) *hyp length-take min.absorb2 nth-append-length*

order.strict-implies-order)
then have *is-nt*: *is-nonterminal* (β ! *j*) **by** (*simp add*: *assms*(*2*))
have *index-j*: *j* < *length* β **using** β*-decomposed assms*(*1*) *hyp* **by** *auto*
have *derivation*: *LeftDerivation* β (*derivation-shift F 0* (*Suc j*)) *b*
by (*simp add*: β*-intro*)
from *LeftDerivation-impossible*[*OF derivation index-j is-nt derivation-ge-shift*]
have *F*: *F* = [] **by** (*metis length-0-conv length-derivation-shift*)
then have β*-is-b*: β = *b* **using** β*-intro* **by** *auto*
show *?thesis*
apply (*rule-tac x=a* ! *i* **in** *exI*)
apply (*rule-tac x=take i a* **in** *exI*)
apply (*rule-tac x=drop* (*Suc i*) *a* **in** *exI*)
apply (*rule-tac x=take j b* **in** *exI*)
using *EF F assms*(*1*) *hyp splits-at-def* **by** *auto*
qed

lemma *LeftDerivationIntro-implies-nonterminal*:
LeftDerivationIntro α *i* (*snd e*) *ix E j* $\gamma \Longrightarrow$ *is-nonterminal* (α ! *i*)
by (*simp add*: *LeftDerivationIntro-def LeftDerives1-def leftmost-is-nonterminal*)

lemma *LeftDerivationIntrosAt-implies-nonterminal*:
LeftDerivationIntrosAt a D L index $\Longrightarrow$ *is-nonterminal*((*ladder-*α *a D L index*) ! (*ladder-i L index*))
by (*meson LeftDerivationIntro-implies-nonterminal LeftDerivationIntrosAt-def*)

lemma *LeftDerivationIntro-examine-rule*:
LeftDerivationIntro α *i r ix D j* $\gamma \Longrightarrow$ *splits-at* α *i* $\alpha 1$ *M* $\alpha 2 \Longrightarrow$
$\exists\ \eta.\ M = fst\ r \wedge \eta = snd\ r \wedge (M, \eta) \in \mathfrak{R}$
by (*metis Derives1-nonterminal Derives1-rule LeftDerivationIntro-def LeftDerives1-implies-Derives1*

prod.collapse)

lemma *LeftDerivation-skip-prefixword-ex*:
assumes *LeftDerivation* (*u*@*v*) *D w*
assumes *is-word u*

```
  shows ∃ w'. w = u@w' ∧ LeftDerivation v (derivation-shift D (length u) 0) w'
by (metis LeftDerivation.simps(1) LeftDerivation-breakdown LeftDerivation-implies-Derivation

  LeftDerivation-skip-prefix append-eq-conv-conj assms(1) assms(2) is-word-Derivation

  is-word-Derivation-derivation-ge)

definition ladder-cut :: ladder ⇒ nat ⇒ ladder
where ladder-cut L n = (let i = length L − 1 in L[i := (n, snd (L ! i))])

fun deriv-shift :: nat ⇒ nat ⇒ deriv ⇒ deriv
where deriv-shift dn dj (n, j, i) = (n − dn, j − dj, i)

definition ladder-shift :: ladder ⇒ nat ⇒ nat ⇒ ladder
where ladder-shift L dn dj = map (deriv-shift dn dj) L

lemma splits-at-append-suffix-prevails:
  assumes splits-at (a@b) i u N v
  assumes i < length a
  shows ∃ v'. v = v'@b ∧ a=u@[N]@v'
proof −
  have min (length a) (Suc i) = Suc i
    using Suc-leI assms(2) min.absorb2 by blast
  then show ?thesis
      by (metis (no-types) append-assoc append-eq-conv-conj append-take-drop-id
assms(1)
      hd-drop-conv-nth length-take splits-at-def take-hd-drop)
qed

lemma derivation-shift-right-left-cancel:
  derivation-shift (derivation-shift D 0 r) r 0 = D
by (induct D, auto)

lemma derivation-shift-left-right-cancel:
  assumes derivation-ge D r
  shows derivation-shift (derivation-shift D r 0) 0 r = D
using assms derivation-ge-shift-simp derivation-shift-0-shift by auto

lemma LeftDerivation-ge-take:
  assumes derivation-ge D k
  assumes LeftDerivation a D b
  assumes D ≠ []
  shows take k a = take k b ∧ is-word (take k a)
proof −
  obtain d D' where d: d#D' = D using assms(3) list.exhaust by blast
  then have ∃ x. LeftDerives1 a (fst d) (snd d) x ∧ LeftDerivation x D' b
    using LeftDerivation.simps(2) assms(2) by blast
  then obtain x where x: LeftDerives1 a (fst d) (snd d) x ∧ LeftDerivation x D'
b by blast
```

have *fst-d-k*: *fst d* $\geq$ *k* **using** *d assms*(*1*) *derivation-ge-cons* **by** *blast*
from *x fst-d-k* **have** *is-word*: *is-word* (*take k a*)
by (*metis LeftDerives1-def append-take-drop-id is-word-append leftmost-def min.absorb2 take-append take-take*)
have *is-eq*: *take k a* = *take k b*
using *Derivation-take LeftDerivation-implies-Derivation assms*(*1*) *assms*(*2*) **by** *blast*
show *?thesis* **using** *is-word is-eq* **by** *blast*
qed

lemma *LeftDerivationFix-splits-at-symbol*:
assumes *LeftDerivationFix a i D j b*
shows $\exists$ *U a1 a2 b1 b2 n. splits-at a i a1 U a2* $\wedge$ *splits-at b j b1 U b2* $\wedge$
n $\leq$ *length D* $\wedge$ *LeftDerivation a1* (*take n D*) *b1* $\wedge$ *derivation-ge* (*drop n D*) (*Suc*(*length b1*)) $\wedge$
LeftDerivation a2 (*derivation-shift* (*drop n D*) (*Suc*(*length b1*)) *0*) *b2* $\wedge$
(*n* = *length D* $\vee$ (*n* < *length D* $\wedge$ *is-word* (*b1*@[*U*])))
proof −
note *hyp* = *LeftDerivationFix-def*[**where** α=*a* **and** β=*b* **and** *D*=*D* **and** *i*=*i* **and** *j*=*j*]
from *hyp* **obtain** *E F* **where** *EF*:
D = *E* @ *derivation-shift F 0* (*Suc j*) $\wedge$ *LeftDerivation* (*take i a*) *E* (*take j b*) $\wedge$
LeftDerivation (*drop* (*Suc i*) *a*) *F* (*drop* (*Suc j*) *b*)
using *assms* **by** *blast*
have $\exists$ β. *LeftDerivation a E* β $\wedge$ *LeftDerivation* β (*derivation-shift F 0* (*Suc j*)) *b*
using *EF LeftDerivation-append assms*(*1*) *hyp* **by** *blast*
then obtain β **where** β-*intro*:
LeftDerivation a E β $\wedge$ *LeftDerivation* β (*derivation-shift F 0* (*Suc j*)) *b* **by** *blast*
have *LeftDerivation* ((*take i a*)@(*drop i a*)) *E* ((*take j b*)@(*drop i a*))
by (*metis EF LeftDerivation-append-suffix append-take-drop-id assms*(*1*) *hyp is-sentence-concat*)
then have *LeftDerivation a E* ((*take j b*)@(*drop i a*)) **by** *simp*
then have β-*decomposed*: β = (*take j b*)@(*drop i a*)
using *Derivation-unique-dest LeftDerivation-implies-Derivation* β-*intro* **by** *blast*

have *derivation*: *LeftDerivation* β (*derivation-shift F 0* (*Suc j*)) *b*
by (*simp add*: β-*intro*)
have $\exists$ *n. n* $\leq$ *length D* $\wedge$ *E* = *take n D*
by (*metis EF append-eq-conv-conj is-prefix-length is-prefix-take*)
then obtain *n* **where** *n*: *n* $\leq$ *length D* $\wedge$ *E* = *take n D* **by** *blast*
have *F-def*: *drop n D* = *derivation-shift F 0* (*Suc j*)
by (*metis EF append-eq-conv-conj length-take min.absorb2 n*)
have *min-j*: *min* (*length b*) *j* = *j* **using** *assms hyp* **by** *linarith*
have *derivation-ge-Suc-j*: *derivation-ge* (*drop n D*) (*Suc j*)
using *F-def derivation-ge-shift* **by** *simp*
have *LeftDerivation-*β*-b*: *LeftDerivation* β (*drop n D*) *b* **by** (*simp add*: *F-def*

β-intro)
have *is-word-Suc-j-b*: *n* ≠ *length D* ⟹ *is-word* (*take* (*Suc j*) *b*)
by (*metis EF F-def LeftDerivation-ge-take β-intro append-Nil2 derivation-ge-Suc-j*

length-take min.absorb2 n)
have *take-Suc-j-b-decompose*: *take* (*Suc j*) *b* = *take j b* @ [*a* ! *i*]
using *assms hyp take-Suc-conv-app-nth* **by** *auto*
show *?thesis*
apply (*rule-tac x=a* ! *i* **in** *exI*)
apply (*rule-tac x=take i a* **in** *exI*)
apply (*rule-tac x=drop* (*Suc i*) *a* **in** *exI*)
apply (*rule-tac x=take j b* **in** *exI*)
apply (*rule-tac x=drop* (*Suc j*) *b* **in** *exI*)
apply (*rule-tac x=n* **in** *exI*)
apply (*auto simp add*: *min-j*)
using *assms hyp splits-at-def* **apply** *blast*
using *assms hyp splits-at-def* **apply** *blast*
using *n* **apply** *blast*
using *EF n* **apply** *simp*
using *F-def* **apply** *simp*
using *derivation-ge-shift* **apply** *blast*
using *F-def derivation-shift-right-left-cancel* **apply** *simp*
using *EF* **apply** *blast*
using *n* **apply** *arith*
using *is-word-Suc-j-b take-Suc-j-b-decompose is-word-append* **apply** *simp+*
done
qed

lemma *LeftDerivation-breakdown′*: *LeftDerivation* (*u* @ *v*) *D w* ⟹
∃ *n w1 w2*.
n ≤ *length D* ∧
w = *w1* @ *w2* ∧
LeftDerivation u (*take n D*) *w1* ∧
derivation-ge (*drop n D*) (*length w1*) ∧
LeftDerivation v (*derivation-shift* (*drop n D*) (*length w1*) *0*) *w2*
proof −
assume *hyp*: *LeftDerivation* (*u* @ *v*) *D w*
from *LeftDerivation-breakdown*[*OF hyp*] **obtain** *n w1 w2* **where** *breakdown*:
w = *w1* @ *w2* ∧
LeftDerivation u (*take n D*) *w1* ∧
derivation-ge (*drop n D*) (*length w1*) ∧
LeftDerivation v (*derivation-shift* (*drop n D*) (*length w1*) *0*) *w2* **by** *blast*
obtain *m* **where** *m*: *m* = *min* (*length D*) *n* **by** *blast*
have *take-m*: *take m D* = *take n D* **using** *m is-prefix-take take-prefix* **by** *fastforce*
have *drop-m*: *drop m D* = *drop n D*
by (*metis append-eq-conv-conj append-take-drop-id length-take m*)
have *m-bound*: *m* ≤ *length D* **by** (*simp add*: *m*)
show *?thesis*
apply (*rule-tac x=m* **in** *exI*)

apply (*rule-tac x=w1* **in** *exI*)
apply (*rule-tac x=w2* **in** *exI*)
using *breakdown m-bound take-m drop-m* **by** *auto*
qed

lemma *LeftDerives1-append-replace-in-left*:
assumes *ld1*: *LeftDerives1* (α@δ) *i r* β
assumes *i-bound*: $i <$ *length* α
shows $\exists\ \alpha'$. $\beta = \alpha'$@$\delta \wedge$ *LeftDerives1* α *i r* $\alpha' \wedge i +$ *length* (*snd r*) $\leq$ *length* α'
proof −
obtain α' **where** α': $\alpha' =$ (*take i* α)@(*snd r*)@(*drop* (*i+1*) α) **by** *blast*
have *fst-r*: *fst r* = α ! *i*
proof −
have $\forall$ *ss n p ssa*. $\neg$ *LeftDerives1 ss n p ssa* $\vee$ *Derives1 ss n p ssa*
using *LeftDerives1-implies-Derives1* **by** *blast*
then have *Derives1* (α @ δ) *i r* β
using *ld1* **by** *blast*
then show *?thesis*
using *Derives1-nonterminal i-bound splits-at-def* **by** *auto*
qed
have *Derives1* α *i r* α'
using *i-bound ld1*
apply (*auto simp add*: α' *Derives1-def*)
apply (*rule-tac x=take i* α **in** *exI*)
apply (*rule-tac x=drop* (*i+1*) α **in** *exI*)
apply (*rule-tac x=fst r* **in** *exI*)
apply *auto*
apply (*simp add*: *fst-r*)
using *id-take-nth-drop* **apply** *blast*
using *Derives1-sentence1 LeftDerives1-implies-Derives1 is-sentence-concat*
is-sentence-take **apply** *blast*
apply (*metis Derives1-sentence1 LeftDerives1-implies-Derives1 append-take-drop-id*

is-sentence-concat)
using *Derives1-rule LeftDerives1-implies-Derives1* **by** *blast*
then have *leftderives1-*α*-*α': *LeftDerives1* α *i r* α'
using *LeftDerives1-def i-bound ld1 leftmost-cons-less* **by** *auto*
have *i-bound-*α': $i +$ *length* (*snd r*) $\leq$ *length* α'
using α' *i-bound*
by (*simp add*: *add-mono-thms-linordered-semiring*(*2*) *le-add1 less-or-eq-imp-le*
min.absorb2)
have *is-sentence-*δ: *is-sentence* δ
using *Derives1-sentence1 LeftDerives1-implies-Derives1 is-sentence-concat ld1*
by *blast*
then have β: $\beta = \alpha'$@δ
using *ld1 leftderives1-*α*-*α' *Derives1-append-suffix Derives1-unique-dest*
LeftDerives1-implies-Derives1 **by** *blast*
show *?thesis*
apply (*rule-tac x=*α' **in** *exI*)

using β *i-bound-*α' *leftderives1-*α*-*α' **by** *blast*
qed

lemma *LeftDerivationIntro-propagate*:
assumes *intro*: *LeftDerivationIntro* $(\alpha@\delta)$ *i r ix D j* γ
assumes *i-*α: $i <$ *length* α
assumes *non*: *is-nonterminal* $(\gamma ! j)$
shows $\exists$ ω. *LeftDerivation* α $((i,r)\#D)$ ω $\wedge$ $\gamma = \omega@\delta$ $\wedge$ $j <$ *length* ω
proof −
from *intro LeftDerivationIntro-def*[**where** $\alpha=\alpha@\delta$ **and** $i=i$ **and** $r=r$ **and** $ix=ix$ **and** $D=D$ **and**
$j=j$ **and** $\gamma=\gamma$]
obtain β **where** *ld-*β: *LeftDerives1* $(\alpha$ @ $\delta)$ *i r* β **and**
ix: $ix <$ *length* (*snd r*) $\wedge$ *snd r* ! $ix = \gamma$! j **and**
β*-fix*: *LeftDerivationFix* β $(i + ix)$ *D j* γ
by *blast*
from *LeftDerives1-append-replace-in-left*[*OF ld-*β *i-*α]
obtain α' **where** α': $\beta = \alpha'$ @ δ $\wedge$ *LeftDerives1* α *i r* α' $\wedge$ $i +$ *length* (*snd r*) $\leq$ *length* α'
by *blast*
have *i-plus-ix-bound*: $i + ix <$ *length* α' **using** α' *ix* **by** *linarith*
have *ld-*γ: *LeftDerivationFix* $(\alpha'$ @ $\delta)$ $(i + ix)$ *D j* γ
using β*-fix* α' **by** *simp*
then have *non-i-ix*: *is-nonterminal* $((\alpha'$ @ $\delta)$! $(i + ix))$
by (*simp add*: *LeftDerivationFix-def non*)
from *LeftDerivationFix-splits-at-nonterminal*[*OF ld-*γ *non-i-ix*]
obtain *U a1 a2 b1* **where** *U*:
splits-at $(\alpha'$ @ $\delta)$ $(i + ix)$ *a1 U a2* $\wedge$ *splits-at* γ *j b1 U a2* $\wedge$ *LeftDerivation a1 D b1*
by *blast*
have $\exists$ *q*. $a2 = q@\delta$ $\wedge$ $\alpha' = a1$ @ $[U]$ @ q
using *splits-at-append-suffix-prevails*[*OF - i-plus-ix-bound*, **where** $b=\delta$] *U* **by** *blast*
then obtain *q* **where** *q*: $a2 = q@\delta$ $\wedge$ $\alpha' = a1$ @ $[U]$ @ q **by** *blast*
show *?thesis*
apply (*rule-tac* $x=b1@[U]@q$ **in** *exI*)
apply *auto*
apply (*rule-tac* $x=\alpha'$ **in** *exI*)
apply (*metis LeftDerivationFix-def LeftDerivation-append-suffix U* α'
q append-Cons append-Nil is-sentence-concat ld-γ)
using *U q splits-at-combine* **apply** *auto*[*1*]
using *U splits-at-def* **by** *auto*
qed

lemma *LeftDerivationIntro-finish*:
assumes *intro*: *LeftDerivationIntro* $(\alpha@\delta)$ *i r ix D j* γ
assumes *i-*α: $i <$ *length* α
shows $\exists$ *k* ω δ'.
$k \leq$ *length D* $\wedge$

LeftDerivation α ((i, r)#(take k D)) ω ∧
LeftDerivation (α @ δ) ((i, r)#(take k D)) (ω @ δ) ∧
derivation-ge (drop k D) (length ω) ∧
LeftDerivation δ (derivation-shift (drop k D) (length ω) 0) δ′ ∧
γ = ω @ δ′ ∧ j < length ω
proof −
from *intro LeftDerivationIntro-def*[**where** *α=α@δ* **and** *i=i* **and** *r=r* **and** *ix=ix* **and** *D=D* **and**
j=j **and** *γ=γ*]
obtain *β* **where** *ld-β*: *LeftDerives1 (α @ δ) i r β* **and**
ix: *ix < length (snd r) ∧ snd r ! ix = γ ! j* **and**
β-fix: *LeftDerivationFix β (i + ix) D j γ*
by *blast*
from *LeftDerives1-append-replace-in-left*[*OF ld-β i-α*]
obtain *α′* **where** *α′*: *β = α′ @ δ ∧ LeftDerives1 α i r α′ ∧ i + length (snd r) ≤ length α′*
by *blast*
have *i-plus-ix-bound*: *i + ix < length α′* **using** *α′ ix* **by** *linarith*
have *ld-γ*: *LeftDerivationFix (α′ @ δ) (i + ix) D j γ*
using *β-fix α′* **by** *simp*
from *LeftDerivationFix-splits-at-symbol*[*OF ld-γ*]
obtain *U a1 a2 b1 b2 n* **where** *U*:
splits-at (α′ @ δ) (i + ix) a1 U a2 ∧
splits-at γ j b1 U b2 ∧
n ≤ length D ∧
LeftDerivation a1 (take n D) b1 ∧
derivation-ge (drop n D) (Suc (length b1)) ∧
LeftDerivation a2 (derivation-shift (drop n D) (Suc (length b1)) 0) b2 ∧
(n = length D ∨ n < length D ∧ is-word (b1 @ [U]))
by *blast*
have *n-bound*: *n ≤ length D* **using** *U* **by** *blast*
have *∃ q. a2 = q@δ ∧ α′ = a1 @ [U] @ q*
using *splits-at-append-suffix-prevails*[*OF - i-plus-ix-bound*, **where** *b=δ*] *U* **by** *blast*
then obtain *q* **where** *q*: *a2 = q@δ ∧ α′ = a1 @ [U] @ q* **by** *blast*
have *j*: *j = length b1*
using *U* **by** (*simp add*: *dual-order.strict-implies-order min.absorb2 splits-at-def*)
have *n = length D ∨ n < length D ∧ is-word (b1 @ [U])* **using** *U* **by** *blast*
then show *?thesis*
proof (*induct rule*: *disjCases2*)
case *1*
from *1* **have** *drop-n-D*: *drop n D = []* **by** (*simp add*: *U*)
then have *LeftDerivation a2 [] b2* **using** *U* **by** *simp*
then have *a2-eq-b2*: *a2 = b2* **by** *simp*
show *?case*
apply (*rule-tac x=n* **in** *exI*)
apply (*rule-tac x=b1@[U]@q* **in** *exI*)
apply (*rule-tac x=δ* **in** *exI*)
apply *auto*

```
        apply (simp add: 1)
        apply (rule-tac x=α′ in exI)
       apply (metis LeftDerivationFix-is-sentence LeftDerivation-append-suffix U
α′
          append-Cons append-Nil is-sentence-concat ld-γ q)
        apply (rule-tac x=α′ @ δ in exI)
        apply (metis 1.hyps LeftDerivationFix-def U α′ a2-eq-b2 id-take-nth-drop
ld-β
          ld-γ q splits-at-def take-all)
        apply (simp add: drop-n-D)+
        apply (metis U a2-eq-b2 id-take-nth-drop q splits-at-def)
        using j apply arith
        done
  next
    case 2
      obtain E where E: E = (derivation-shift (drop n D) (Suc (length b1)) 0)
by blast
     then have LeftDerivation (q@δ) E b2 using U q  by simp
     from LeftDerivation-breakdown′[OF this] obtain n′ w1 w2 where w1w2:
       n′ ≤ length E ∧
        b2 = w1 @ w2 ∧
        LeftDerivation q (take n′ E) w1 ∧
        derivation-ge (drop n′ E) (length w1) ∧
        LeftDerivation δ (derivation-shift (drop n′ E) (length w1) 0) w2 by blast
     have length-E-D: length E = length D − n using E n-bound by simp
      have n-plus-n′-bound: n + n′ ≤ length D using length-E-D w1w2 n-bound
by arith
     have take-breakdown: take (n + n′) D = (take n D) @ (take n′ (drop n D))
       using take-add by blast
     have q-w1: LeftDerivation q (take n′ E) w1 using w1w2 by blast
     have isw: is-word (b1 @ [U]) using 2 by blast
    have take-n′: take n′ (drop n D) = derivation-shift (take n′ E) 0 (Suc (length
b1))
       by (metis E U derivation-shift-left-right-cancel take-derivation-shift)
      have α′-derives-b1-U-w1: LeftDerivation α′ (take (n + n′) D) (b1 @ U #
w1)
       apply (subst take-breakdown)
       apply (rule-tac LeftDerivation-implies-append[where b=b1@[U]@q])
       apply (metis LeftDerivationFix-is-sentence LeftDerivation-append-suffix U
         is-sentence-concat ld-γ q)
       apply (simp add: take-n′)
        by (rule LeftDerivation-append-prefix[OF q-w1, where u = b1@[U], OF
isw, simplified])
     have dge: derivation-ge (drop (n + n′) D) (Suc (length b1 + length w1))
     proof −
       have derivation-ge (drop n′ (drop n D)) (length b1 + 1 + length w1)
            by (metis (no-types) E Suc-eq-plus1 U append-take-drop-id deriva-
tion-ge-append derivation-ge-shift-plus drop-derivation-shift w1w2)
       then show derivation-ge (drop (n + n′) D) (Suc (length b1 + length w1))
```

by (*metis* (*no-types*) *Suc-eq-plus1 add.commute drop-drop semiring-normalization-rules*(*23*))
qed
show *?case*
apply (*rule-tac* $x=n+n'$ **in** *exI*)
apply (*rule-tac* $x=b1$ @ [*U*] @ *w1* **in** *exI*)
apply (*rule-tac* $x=w2$ **in** *exI*)
apply *auto*
using *n-plus-n′-bound* **apply** *simp*
apply (*rule-tac* $x=\alpha'$ **in** *exI*)
using α' α'*-derives-b1-U-w1* **apply** *blast*
apply (*rule-tac* $x=\alpha'$ @ δ **in** *exI*)
apply (*metis Cons-eq-appendI LeftDerivationFix-is-sentence LeftDerivation-append-suffix*
α' α'*-derives-b1-U-w1 append-assoc is-sentence-concat ld-*β *ld-*γ)
apply (*rule dge*)
apply (*metis E Suc-eq-plus1 add.commute derivation-shift-0-shift drop-derivation-shift*

drop-drop w1w2)
using *U splits-at-combine w1w2* **apply** *auto*[*1*]
by (*simp add*: *j*)
qed
qed

lemma *LeftDerivationLadder-propagate*:
LeftDerivationLadder (α@δ) *D L* $\gamma \Longrightarrow$ *ladder-i L 0* < *length* $\alpha \Longrightarrow n =$ *ladder-n L index*
$\Longrightarrow$ *index* < *length L* $\Longrightarrow$
if (*index* + *1* < *length L*) *then*
($\exists$ β. *LeftDerivation* α (*take n D*) β $\wedge$ *ladder-*γ (α@δ) *D L index* = β@δ $\wedge$
ladder-j L index < *length* β)
else
($\exists$ n' β δ'. (*index* = *0* $\vee$ *ladder-prev-n L index* < n') $\wedge$ $n' \leq n$ $\wedge$ *LeftDerivation* α (*take* n' *D*) β $\wedge$
LeftDerivation (α@δ) (*take* n' *D*) (β@δ) $\wedge$
derivation-ge (*drop* n' *D*) (*length* β) $\wedge$
LeftDerivation δ (*derivation-shift* (*drop* n' *D*) (*length* β) *0*) δ' $\wedge$
ladder-γ (α@δ) *D L index* = β@δ' $\wedge$ *ladder-j L index* < *length* β)
proof (*induct index arbitrary*: *n*)
case *0*
have *ldfix*:
LeftDerivationFix (α@δ) (*ladder-i L 0*) (*take n D*) (*ladder-j L 0*) (*ladder-*γ (α@δ) *D L 0*)
using *0.prems*(*1*) *0.prems*(*3*) *LeftDerivationLadder-def* **by** *blast*
from *0* **have** *1* < *length L* $\vee$ *1* = *length L* **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have *LeftDerivationIntrosAt* (α@δ) *D L 1*
using *0.prems*(*1*) *1.hyps LeftDerivationIntros-def LeftDerivationLadder-def*

by *blast*
from *LeftDerivationIntrosAt-implies-nonterminal*[*OF this*]
have *is-nonterminal* (*ladder-*γ (α @ δ) *D L 0* ! *ladder-j L 0*)
by (*simp add*: *ladder-*α*-def ladder-i-def*)
with *ldfix* **have** *is-nonterminal* ((α@δ) ! (*ladder-i L 0*)) **by** (*simp add*: *LeftDerivationFix-def*)
from *LeftDerivationFix-splits-at-nonterminal*[*OF ldfix this*] **obtain** *U a1 a2 b* **where** *thesplit*:
splits-at (α @ δ) (*ladder-i L 0*) *a1 U a2* $\wedge$
splits-at (*ladder-*γ (α @ δ) *D L 0*) (*ladder-j L 0*) *b U a2* $\wedge$
LeftDerivation a1 (*take n D*) *b* **by** *blast*
have $\exists$ δ'. *a2* = δ' @ δ $\wedge$ α = *a1* @ [*U*] @ δ'
using *thesplit splits-at-append-suffix-prevails* **using** *0.prems*(*2*) **by** *blast*
then obtain δ' **where** δ': *a2* = δ' @ δ $\wedge$ α = *a1* @ ([*U*] @ δ') **by** *blast*
obtain β **where** β: β = *b* @ ([*U*] @ δ') **by** *blast*
have *is-sentence* α **using** *LeftDerivationFix-is-sentence is-sentence-concat ldfix* **by** *blast*
then have *is-sentence* ([*U*] @ δ') **using** δ' *is-sentence-concat* **by** *blast*
with δ' *thesplit* **have** *LeftDerivation* (*a1* @ ([*U*] @ δ')) (*take n D*) (*b* @ ([*U*] @ δ'))
using *LeftDerivation-append-suffix* **by** *blast*
then have α*-derives-*β: *LeftDerivation* α (*take n D*) β **using** β δ' **by** *blast*
have β*-append-*δ: *ladder-*γ (α @ δ) *D L 0* = β@δ
by (*metis* β δ' *append-assoc splits-at-combine thesplit*)
have *ladder-j-bound*: *ladder-j L 0* < *length* β
by (*metis One-nat-def* β *diff-add-inverse dual-order.strict-implies-order leD le-add1*
length-Cons length-append length-take list.size(*3*) *min.absorb2 neq0-conv splits-at-def*
thesplit zero-less-diff zero-less-one)
show *?case*
using *1* **apply** *simp*
apply (*rule-tac x*=β **in** *exI*)
by (*auto simp add*: α*-derives-*β β*-append-*δ *ladder-j-bound*)
next
case *2*
note *case-2* = *2*
have *n-def*: *n* = *length D*
by (*metis 0.prems*(*1*) *0.prems*(*3*) *2.hyps LeftDerivationLadder-def One-nat-def*
diff-Suc-1 is-ladder-def ladder-last-n-intro)
then have *take-n-D*: *take n D* = *D* **by** (*simp add*: *eq-imp-le*)
from *LeftDerivationFix-splits-at-symbol*[*OF ldfix*] **obtain** *U a1 a2 b1 b2 m* **where** *U*:
splits-at (α @ δ) (*ladder-i L 0*) *a1 U a2* $\wedge$
splits-at (*ladder-*γ (α @ δ) *D L 0*) (*ladder-j L 0*) *b1 U b2* $\wedge$
m $\leq$ *length* (*take n D*) $\wedge$
LeftDerivation a1 (*take m* (*take n D*)) *b1* $\wedge$
derivation-ge (*drop m* (*take n D*)) (*Suc* (*length b1*)) $\wedge$

LeftDerivation a2 (*derivation-shift* (*drop m* (*take n D*)) (*Suc* (*length b1*)) *0*) *b2* $\wedge$
(m = *length* (*take n D*) $\vee$ ($m <$ *length* (*take n D*) $\wedge$ *is-word* (*b1* @ [*U*]))) **by** *blast*
obtain D' **where** D': D' = *derivation-shift* (*drop m D*) (*Suc* (*length b1*)) *0* **by** *blast*
then have *a2-derives-b2*: *LeftDerivation a2* D' *b2* **using** *U take-n-D* **by** *auto*
from *U* **have** *m-leq-n*: $m \leq n$
by (*simp add*: *0.prems*(*1*) *0.prems*(*3*) *0.prems*(*4*) *LeftDerivationLadder-def is-ladder-def*
min.absorb2)
from *U* **have** *splits-at* (α @ δ) (*ladder-i L 0*) *a1 U a2* **by** *blast*
from *splits-at-append-suffix-prevails*[*OF this 0*(*2*)] **obtain** v' **where**
v': *a2* = v' @ δ $\wedge$ α = *a1* @ [*U*] @ v' **by** *blast*
have *a1-derives-b1*: *LeftDerivation a1* (*take m D*) *b1* **using** *m-leq-n U*
by (*metis 0.prems*(*1*) *0.prems*(*3*) *2.hyps LeftDerivationLadder-def One-nat-def*
cancel-comm-monoid-add-class.diff-cancel is-ladder-def ladder-last-n-intro order-refl
take-all)
have *LeftDerivation* (v' @ δ) D' *b2* **using** *a2-derives-b2* v' **by** *simp*
from *LeftDerivation-breakdown'*[*OF this*] **obtain** m' *w1 w2* **where** *w12*:
b2 = *w1* @ *w2* $\wedge$
$m' \leq$ *length* D' $\wedge$
LeftDerivation v' (*take* m' D') *w1* $\wedge$
derivation-ge (*drop* m' D') (*length w1*) $\wedge$
LeftDerivation δ (*derivation-shift* (*drop* m' D') (*length w1*) *0*) *w2* **by** *blast*
have *length* $D' \leq$ *length* $D - m$ **by** (*simp add*: D')
then have $m' \leq$ *length* $D - m$ **using** *w12 dual-order.trans* **by** *blast*
then have *m-m'-leq-n*: $m + m' \leq n$ **using** *n-def m-leq-n le-diff-conv2 add.commute*
by *linarith*
obtain β **where** β: β = *b1* @ ([*U*] @ *w1*) **by** *blast*
have *is-sentence* ([*U*] @ v')
using *LeftDerivationFix-is-sentence is-sentence-concat ldfix* v' **by** *blast*
then have *LeftDerivation* (*a1* @ ([*U*] @ v')) (*take m D*) (*b1* @ ([*U*] @ v'))
using *LeftDerivation-append-suffix a1-derives-b1* **by** *blast*
then have α*-derives-pre-*β: *LeftDerivation* α (*take m D*) (*b1* @ ([*U*] @ v'))
using v' **by** *blast*
have $m = n \vee (m < n \wedge$ *is-word* (*b1* @ [*U*]))
using *U n-def*[*symmetric*] *take-n-D* **by** *simp*
then have *pre-*β*-derives-*β: *LeftDerivation* (*b1* @ ([*U*] @ v')) (*take* m' (*drop m D*)) β
proof (*induct rule*: *disjCases2*)
case *1*
then have $m' = 0$ **using** *m-m'-leq-n* **by** *arith*
then show *?case*
apply (*simp add*: β)
using *w12* **by** *simp*

next
case *2*
then have *is-word-prefix*: *is-word* (*b1* @ [*U*]) **by** *blast*
have *take-drop-eq*: *take m′* (*drop m D*) = *derivation-shift* (*take m′ D′*)
0 (*length* (*b1* @ [*U*]))
apply (*simp add*: *D′ take-derivation-shift*)
by (*metis U derivation-shift-left-right-cancel take-derivation-shift take-n-D*)
have *v′-derives-w1*: *LeftDerivation v′* (*take m′ D′*) *w1*
by (*simp add*: *w12*)
with *is-word-prefix* **have**
LeftDerivation ((*b1* @ [*U*]) @ *v′*) (*derivation-shift* (*take m′ D′*)
0 (*length* (*b1* @ [*U*]))) ((*b1* @ [*U*]) @ *w1*)
using *LeftDerivation-append-prefix* **by** *blast*
with *take-drop-eq* **show** *?case* **by** (*simp add*: *β*)
qed
have (*take m D*) @ (*take m′* (*drop m D*)) = (*take* (*m* + *m′*) *D*)
by (*simp add*: *take-add*)
then have *α-derives-β*: *LeftDerivation α* (*take* (*m* + *m′*) *D*) *β*
using *LeftDerivation-implies-append α-derives-pre-β pre-β-derives-β* **by** *fast-force*
have *derivation-ge-drop-m-m′*: *derivation-ge* (*drop* (*m* + *m′*) *D*) (*length β*)
proof −
have *f1*: *drop m′* (*drop m D*) = *drop* (*m* + *m′*) *D*
by (*simp add*: *add.commute*)
have *derivation-ge* (*drop m′* (*drop m D*)) (*Suc* (*length b1*))
by (*metis* (*no-types*) *U append-take-drop-id derivation-ge-append take-n-D*)
then show *derivation-ge* (*drop* (*m* + *m′*) *D*) (*length β*)
using *f1* **by** (*metis* (*no-types*) *D′ β append-assoc derivation-ge-shift-plus*
drop-derivation-shift length-append length-append-singleton w12)
qed
have *δ-derives-w2*: *LeftDerivation δ* (*derivation-shift* (*drop* (*m* + *m′*) *D*) (*length β*) *0*) *w2*
proof −
have *derivation-shift* (*drop m′ D′*) (*length w1*) *0* = *derivation-shift* (*drop* (*m* + *m′*) *D*) (*length β*) *0*
by (*simp add*: *D′ β add.commute derivation-shift-0-shift drop-derivation-shift*)
then show *LeftDerivation δ* (*derivation-shift* (*drop* (*m* + *m′*) *D*) (*length β*) *0*) *w2*
using *w12* **by** *presburger*
qed
have *ladder-γ-def*: *ladder-γ* (*α* @ *δ*) *D L 0* = *β* @ *w2*
using *U β splits-at-combine w12* **by** *auto*
have *ladder-j-bound*: *ladder-j L 0* < *length β* **using** *U β splits-at-def* **by** *auto*
show *?case*
using *2* **apply** *simp*
apply (*rule-tac x*=*m* + *m′* **in** *exI*)
apply (*auto simp add*: *m-m′-leq-n*)
apply (*rule-tac x*=*β* **in** *exI*)
apply (*auto simp add*: *α-derives-β*)

```
    using LeftDerivationFix-is-sentence LeftDerivation-append-suffix α-derives-β

      is-sentence-concat ldfix apply blast
    using derivation-ge-drop-m-m' apply blast
    apply (rule-tac x=w2 in exI)
    apply auto
    using δ-derives-w2 apply blast
    using ladder-γ-def apply blast
    using ladder-j-bound apply blast
    done
  qed
next
  case (Suc index)
  have step: LeftDerivationIntrosAt (α@δ) D L (Suc index)
   by (metis LeftDerivationIntros-def LeftDerivationLadder-def Suc.prems(1) Suc.prems(4)

      Suc-eq-plus1-left le-add1)
  have index-plus-1-bound: index + 1 < length L
    using Suc.prems(4) by linarith
  then have index-bound: index < length L by arith
  obtain n' where n': n' = ladder-n L index by blast
 from Suc.hyps[OF Suc.prems(1) Suc.prems(2) n' index-bound] index-plus-1-bound

  have ∃ α'. LeftDerivation α (take n' D) α' ∧
    ladder-γ (α@δ) D L index = α'@δ ∧ ladder-j L index < length α'
    by auto
  then obtain α' where α': LeftDerivation α (take n' D) α' ∧
    ladder-γ (α@δ) D L index = α'@δ ∧ ladder-j L index < length α'
    by blast
  have Suc-index-bound: Suc index < length L using Suc.prems by auto
 have is-ladder: is-ladder D L using Suc.prems LeftDerivationLadder-def by auto

  have n-def: n = ladder-n L (Suc index)
    using Suc-index-bound n' by (simp add: Suc.prems(3))
   with n' have n'-less-n: n' < n using is-ladder Suc-index-bound is-ladder-def
lessI by blast
 have ladder-α-is-γ: ladder-α (α@δ) D L (Suc index) = ladder-γ (α@δ) D L index
    by (simp add: ladder-α-def)
  obtain i where i: i = ladder-i L (Suc index) by blast
  obtain e where e: e = (D ! n') by blast
  obtain E where E: E = drop (Suc n') (take n D) by blast
  obtain ix where ix: ix = ladder-ix L (Suc index) by blast
  obtain j where j: j = ladder-j L (Suc index) by blast
  obtain γ where γ: γ = ladder-γ (α@δ) D L (Suc index) by blast
  have intro: LeftDerivationIntro (α'@δ) i (snd e) ix E j γ
    by (metis E LeftDerivationIntrosAt-def α' γ ladder-α-is-γ
      diff-Suc-1 e i ix j local.step n' n-def)
  have is-eq-fst-e: i = fst e
    by (metis LeftDerivationIntrosAt-def diff-Suc-1 e i local.step n')
```

```
have i-less-α′: i < length α′ using i α′ ladder-i-def by simp
have (Suc index) + 1 < length L ∨ (Suc index) + 1 = length L
  using Suc-index-bound by arith
then show ?case
proof (induct rule: disjCases2)
  case 1
    from 1 have LeftDerivationIntrosAt (α@δ) D L (Suc (Suc index))
      by (metis LeftDerivationIntros-def LeftDerivationLadder-def Suc.prems(1)
        Suc-eq-plus1 Suc-eq-plus1-left le-add1)
    from LeftDerivationIntrosAt-implies-nonterminal[OF this] have
       is-nonterminal (ladder-α (α @ δ) D L (Suc (Suc index)) ! ladder-i L (Suc
(Suc index)))
      by blast
     then have non-γ-j: is-nonterminal (γ ! j) by (simp add: γ j ladder-α-def
ladder-i-def)
    from LeftDerivationIntro-propagate[OF intro i-less-α′ non-γ-j]
    obtain ω where ω: LeftDerivation α′ ((i, snd e) # E) ω ∧ γ = ω @ δ ∧ j
< length ω
      by blast
    have α-ω: LeftDerivation α ((take n′ D)@((i, snd e) # E)) ω
      using α′ ω LeftDerivation-implies-append by blast
    have i-e: (i, snd e) = e by (simp add: is-eq-fst-e)
    have take-n-D-e: ((take n′ D)@(e # E)) = take n D
    proof −
      have f2: ladder-last-n L = length D
        using is-ladder is-ladder-def by blast
      have f3: min (ladder-last-n L) n = n
        using is-ladder-def
       by (metis (no-types) Suc-eq-plus1 index-plus-1-bound is-ladder min.absorb2
n-def)
      then have take n′ (take n D) @ take n D ! n′ # E = take n D
        using f2 by (metis E id-take-nth-drop length-take n′-less-n)
      then show ?thesis
        using f3 f2 by (metis (no-types) append-assoc append-eq-conv-conj
              dual-order.strict-implies-order e length-take min.absorb2 n′-less-n
nth-append)
    qed
    from 1 show ?case
      apply auto
      apply (rule-tac x=ω in exI)
      apply auto
      using α-ω i-e take-n-D-e apply auto[1]
      using γ ω apply blast
      using ω j by blast
next
  case 2
  from LeftDerivationIntro-finish[OF intro i-less-α′] obtain k ω δ′ where kwδ′:
    k ≤ length E ∧
     LeftDerivation α′ ((i, snd e) # take k E) ω ∧
```

```
      LeftDerivation (α′ @ δ) ((i, snd e) # take k E) (ω @ δ) ∧
      derivation-ge (drop k E) (length ω) ∧
      LeftDerivation δ (derivation-shift (drop k E) (length ω) 0) δ′ ∧
      γ = ω @ δ′ ∧ j < length ω by blast
    have ladder-last-n-1: ladder-last-n L = n
      by (metis 2.hyps Suc-eq-plus1 diff-Suc-1 ladder-last-n-def n-def)
    from is-ladder have ladder-last-n-2: ladder-last-n L = length D
      using is-ladder-def by blast
    from ladder-last-n-1 ladder-last-n-2 have n-eq-length-D: n = length D by blast

    have take-split: take (Suc (n′ + k)) D = (take n′ D) @ ((i, snd e) # take k E)
      apply (simp add: E n-eq-length-D)
      by (metis (no-types, lifting) Cons-eq-appendI add-Suc append-eq-appendI e
        is-eq-fst-e n′-less-n n-eq-length-D prod.collapse
        self-append-conv2 take-Suc-conv-app-nth take-add)
    have α-ω: LeftDerivation α (take (Suc (n′ + k)) D) ω
      apply (subst take-split)
      apply (rule LeftDerivation-implies-append[where b=α′])
      apply (simp add: α′)
      using kwδ′ by blast
    have Suc-n′-k-bound: Suc (n′ + k) ≤ n using E kwδ′ n′-less-n by auto[1]
    from 2 show ?case
      apply auto
      apply (rule-tac x=Suc (n′ + k) in exI)
      apply auto
      apply (simp add: ladder-prev-n-def n′)
      using Suc-n′-k-bound apply blast
      apply (rule-tac x=ω in exI)
      apply auto
      using α-ω apply blast
    using α-ω LeftDerivationFix-def LeftDerivationLadder-def LeftDerivation-append-suffix

       Suc.prems(1) is-sentence-concat apply auto[1]
       apply (metis E add.commute add-Suc-right drop-drop kwδ′ n-eq-length-D
nat-le-linear
       take-all)
      apply (rule-tac x=δ′ in exI)
      apply auto
    apply (metis E LeftDerivationLadder-ladder-n-bound Suc.prems(1) Suc-index-bound

       add.commute add-Suc-right drop-drop kwδ′ n-def n-eq-length-D take-all)
      using γ kwδ′ apply blast
      using j kwδ′ by blast
  qed
qed

lemma ladder-i-of-cut-at-0:
  assumes L-non-empty: L ≠ []
  shows ladder-i (ladder-cut L n) 0 = ladder-i L 0
```

proof −
  have *length L* ≠ *0* **using** *L-non-empty* **by** *auto*
  then have *length L* = *1* ∨ *length L* > *1* **by** *arith*
  then show *?thesis*
  proof (*induct rule*: *disjCases2*)
    case *1*
      then show *?case*
        apply (*simp add*: *ladder-cut-def ladder-i-def deriv-i-def*)
        by (*simp add*: *assms hd-conv-nth*)
  next
    case *2*
      then show *?case*
        apply (*simp add*: *ladder-cut-def ladder-i-def deriv-i-def*)
      by (*metis diff-is-0-eq hd-conv-nth leD list-update-nonempty nth-list-update-neq*)
  qed
qed

lemma *ladder-last-j-of-cut*:
  assumes *L-non-empty*: *L* ≠ []
  shows *ladder-last-j* (*ladder-cut L n*) = *ladder-last-j L*
proof −
  have *length-L-nonzero*: *length L* ≠ *0* **using** *L-non-empty* **by** *auto*
  then have *length-ladder-cut*: *length* (*ladder-cut L n*) = *length L*
    by (*metis ladder-cut-def length-list-update*)
  show *?thesis*
    apply (*simp add*: *ladder-last-j-def length-ladder-cut*)
    apply (*simp add*: *ladder-cut-def ladder-j-def deriv-j-def*)
   by (*metis length-L-nonzero diff-less neq0-conv nth-list-update-eq snd-conv zero-less-Suc*)
qed

lemma *length-ladder-cut*:
  assumes *L-non-empty*: *L* ≠ []
  shows *length* (*ladder-cut L n*) = *length L*
by (*metis ladder-cut-def length-list-update*)

lemma *ladder-last-n-of-cut*:
  assumes *L-non-empty*: *L* ≠ []
  shows *ladder-last-n* (*ladder-cut L n*) = *n*
proof −
  show *?thesis*
    apply (*simp add*: *ladder-last-n-def length-ladder-cut*[*OF L-non-empty*])
    apply (*simp add*: *ladder-n-def ladder-cut-def deriv-n-def*)
    by (*metis assms diff-Suc-less fst-conv length-greater-0-conv nth-list-update-eq*)
qed

lemma *ladder-n-of-cut*:
  assumes *L-non-empty*: *L* ≠ []
  assumes *index* < *length L* − *1*
  shows *ladder-n* (*ladder-cut L n*) *index* = *ladder-n L index*

```
by (metis assms(2) ladder-cut-def ladder-n-def nat-neq-iff nth-list-update-neq)

lemma ladder-n-prev-bound:
  assumes ladder: is-ladder D L
  assumes u-bound: u < length L − 1
  shows ladder-n L u ≤ ladder-prev-n L (length L − 1)
proof −
  have ladder-n L u ≤ ladder-n L (length L − 2)
  proof −
    have u < Suc (length L − 2)
      using u-bound by linarith
    then show ?thesis
     by (metis (no-types) diff-Suc-less is-ladder-def ladder leI length-greater-0-conv

        not-less-eq numeral-2-eq-2 order.order-iff-strict)
  qed
  then show ?thesis
   by (metis One-nat-def Suc-diff-Suc diff-Suc-1 ladder-prev-n-def neq0-conv not-less0

      numeral-2-eq-2 u-bound zero-less-diff)
qed

lemma ladder-n-last-is-length:
  assumes is-ladder D L
  shows ladder-n L (length L − 1) = length D
using assms is-ladder-def ladder-last-n-intro by auto

lemma derivation-ge-shift-implies-derivation-ge:
  assumes dge: derivation-ge (derivation-shift F 0 j) k
  shows derivation-ge F (k − j)
proof −
  have ⋀ i r. (i, r) ∈ set (derivation-shift F 0 j) ⟹ i ≥ k
    using dge derivation-ge-def by auto
  {
    fix i :: nat
    fix r :: symbol × (symbol list)
    assume ir: (i, r) ∈ set F
    then have (i + j, r) ∈ set (derivation-shift F 0 j)
    proof −
      have (i + j, r) ∈ (λp. (fst p − 0 + j, snd p)) ‘ set F
        by (metis (lifting) ir diff-zero image-eqI prod.collapse prod.inject)
      then show ?thesis
        by (simp add: derivation-shift-def)
    qed
    then have i + j ≥ k using dge derivation-ge-def by auto
    then have i ≥ k − j by auto
  }
  then show ?thesis using derivation-ge-def by auto
qed
```

lemma *Derives1-bound′*: *Derives1 a i r b* $\Longrightarrow$ *i* $\leq$ *length b*
by (*metis Derives1-bound Derives1-take append-Nil2 append-take-drop-id drop-eq-Nil*

dual-order.strict-implies-order length-take min.absorb2 nat-le-linear)

lemma *LeftDerivation-Derives1-last*:
assumes *LeftDerivation a D b*
assumes *D* $\neq$ []
shows *Derives1* (*Derive a* (*take* (*length D* $-$ *1*) *D*)) (*fst* (*last D*)) (*snd* (*last D*))
b
by (*metis Derive LeftDerivation-Derive-take-LeftDerives1 LeftDerivation-implies-Derivation*

LeftDerives1-implies-Derives1 assms(*1*) *assms*(*2*) *last-conv-nth le-refl length-0-conv*
take-all)

lemma *last-of-prefix-in-set*:
assumes *n* $<$ *length E*
assumes *D* $=$ *E*@*F*
shows *last E* $\in$ *set* (*drop n D*)
proof $-$
have *f1*: *last* (*drop n E*) $=$ *last E*
by (*simp add*: *assms*(*1*))
have *drop n E* $\neq$ []
by (*metis* (*no-types*) *Cons-nth-drop-Suc assms*(*1*) *list.simps*(*3*))
then show *?thesis*
using *f1* **by** (*metis* (*no-types*) *append.simps*(*2*) *append-butlast-last-id append-eq-conv-conj*
assms(*2*) *drop-append in-set-dropD insertCI list.set*(*2*))
qed

lemma *LeftDerivationFix-cut-appendix*:
assumes *ldfix*: *LeftDerivationFix* (α@δ) *i D j* (β@δ')
assumes α-β: *LeftDerivation* α (*take n D*) β
assumes *n-bound*: *n* $\leq$ *length D*
assumes *dge*: *derivation-ge* (*drop n D*) (*length* β)
assumes *i-in*: *i* $<$ *length* α
assumes *j-in*: *j* $<$ *length* β
shows *LeftDerivationFix* α *i* (*take n D*) *j* β
proof $-$
from *LeftDerivationFix-def*[**where** α=α@δ **and** *i*=*i* **and** *D*=*D* **and** *j*=*j* **and**
β=β@δ']
obtain *E F* **where** *EF*:
is-sentence (α @ δ) $\wedge$
is-sentence (β @ δ') $\wedge$
LeftDerivation (α @ δ) *D* (β @ δ') $\wedge$
i $<$ *length* (α @ δ) $\wedge$
j $<$ *length* (β @ δ') $\wedge$
(α @ δ) ! *i* $=$ (β @ δ') ! *j* $\wedge$
D $=$ *E* @ *derivation-shift F 0* (*Suc j*) $\wedge$

LeftDerivation (*take i* (α @ δ)) *E* (*take j* (β @ δ')) $\wedge$
LeftDerivation (*drop* (*Suc i*) (α @ δ)) *F* (*drop* (*Suc j*) (β @ δ')) **using** *ldfix*
by *auto*
with *i-in j-in* **have** *take-i-E-take-j*: *LeftDerivation* (*take i* α) *E* (*take j* β)
by (*simp add*: *less-or-eq-imp-le*)
obtain *m* **where** *m*: *m* = *length E* **by** *blast*
{
assume *n-less-m*: *n* < *m*
then have *E-nonempty*: *E* $\neq$ [] **using** *gr-implies-not0 list.size*(*3*) *m* **by** *auto*
have *last-E-in-set*: *last E* $\in$ *set* (*drop n D*)
using *last-of-prefix-in-set n-less-m m EF* **by** *blast*
obtain *k r* **where** *kr*: (*k*, *r*) = *last E* **by** (*metis old.prod.exhaust*)
have *k-lower-bound*: *k* $\geq$ *length* β **using** *dge last-E-in-set kr*
by (*metis derivation-ge-def fst-conv*)
have $\exists$ α'. *Derives1* α' *k r* (*take j* β) **using** *LeftDerivation-Derives1-last take-i-E-take-j*
by (*metis E-nonempty kr fst-conv snd-conv*)
then have *k* $\leq$ *j* **by** (*metis Derives1-bound'* *j-in length-take less-imp-le-nat min.absorb2*)
then have *k-upper-bound*: *k* < *length* β **using** *j-in* **by** *arith*
from *k-lower-bound k-upper-bound* **have** *False* **by** *arith*
}
then have *m-le-n*: *m* $\leq$ *n* **by** *arith*
have *take-i-E-take-j*: *LeftDerivation* (*take i* α) *E* (*take j* β)
by (*simp add*: *take-i-E-take-j*)
have *take n D* = *E* @ (*take* (*n* − *m*) (*derivation-shift F 0* (*Suc j*)))
using *EF m m-le-n* **by** *auto*
then have *take-n-D*: *take n D* = *E* @ (*derivation-shift* (*take* (*n* − *m*) *F*) *0* (*Suc j*))
using *take-derivation-shift* **by** *auto*
obtain *F'* **where** *F'*: *F'* = *derivation-shift* (*take* (*n* − *m*) *F*) *0* (*Suc j*) **by** *blast*

have *LeftDerivation* ((*take i* α)@(*drop i* α)) *E* ((*take j* β)@(*drop i* α))
using *take-i-E-take-j*
by (*metis EF LeftDerivation-append-suffix append-take-drop-id is-sentence-concat*)

then have *LeftDerivation* α *E* ((*take j* β)@(*drop i* α)) **by** *simp*
with *take-n-D* **have** *take-j-drop-i*: *LeftDerivation* ((*take j* β)@(*drop i* α)) *F'* β **using** *F'*
by (*metis Derivation-unique-dest LeftDerivation-append LeftDerivation-implies-Derivation* α-β)
have *F'-ge*: *derivation-ge F'* (*Suc j*) **using** *F' derivation-ge-shift* **by** *blast*
have *drop-append*: *drop i* α = [α!*i*] @ (*drop* (*Suc i*) α) **by** (*simp add*: *Cons-nth-drop-Suc i-in*)
have *take-append*: *take j* β @ [α!*i*] = *take* (*Suc j*) β
by (*metis LeftDerivationFix-def i-in j-in ldfix nth-superfluous-append take-Suc-conv-app-nth*)
have *take-drop-Suc*: (*take j* β)@(*drop i* α) = (*take* (*Suc j*) β)@(*drop* (*Suc i*) α)
by (*simp add*: *drop-append take-append*)
with *take-drop-Suc take-j-drop-i* **have** *LeftDerivation* ((*take* (*Suc j*) β)@(*drop*

(*Suc i*) α)) *F′* β
  by *auto*
 note *helper* = *LeftDerivation-skip-prefix*[*OF this*]
 have *len-take*: *length* (*take* (*Suc j*) β) = *Suc j* **by** (*simp add*: *Suc-leI j-in min.absorb2*)
 have *F′-shift*: *derivation-shift F′* (*Suc j*) *0* = *take* (*n* − *m*) *F*
  using *F′ derivation-shift-right-left-cancel* **by** *blast*
 have *LeftDerivation-drop*: *LeftDerivation* (*drop* (*Suc i*) α) (*take* (*n* − *m*) *F*) (*drop* (*Suc j*) β)
  using *helper len-take F′-shift F′-ge* **by** *auto*
 show *?thesis*
  apply (*auto simp add*: *LeftDerivationFix-def*)
  using *LeftDerivationFix-is-sentence is-sentence-concat ldfix* **apply** *blast*
  using *LeftDerivationFix-is-sentence is-sentence-concat ldfix* **apply** *blast*
  using α-β **apply** *blast*
  using *i-in* **apply** *blast*
  using *j-in* **apply** *blast*
  using *LeftDerivationFix-def i-in j-in ldfix* **apply** *auto*[*1*]
  apply (*rule-tac x*=*E* **in** *exI*)
  apply (*rule-tac x*=*take* (*n* − *m*) *F* **in** *exI*)
  apply *auto*
  using *take-n-D* **apply** *blast*
  using *take-i-E-take-j* **apply** *blast*
  using *LeftDerivation-drop* **by** *blast*
qed

lemma *LeftDerivationFix-cut-appendix′*:
 assumes *ldfix*: *LeftDerivationFix* (α@δ) *i D j* (β@δ′)
 assumes α-β: *LeftDerivation* α *D* β
 assumes *i-in*: *i* < *length* α
 assumes *j-in*: *j* < *length* β
 shows *LeftDerivationFix* α *i D j* β
proof −
 obtain *n* **where** *n*: *n* = *length D* **by** *blast*
 have *LeftDerivationFix* α *i* (*take n D*) *j* β
  apply (*rule-tac LeftDerivationFix-cut-appendix*)
  apply (*rule ldfix*)
  using α-β *n* **apply** *auto*[*1*]
  using *n* **apply** *auto*[*1*]
  using *n* **apply** *auto*[*1*]
  using *i-in* **apply** *blast*
  using *j-in* **apply** *blast*
  done
 then show *?thesis* **using** *n* **by** *auto*
qed

lemma *LeftDerivationIntro-cut-appendix*:
 assumes *ldfix*: *LeftDerivationIntro* (α@δ) *i r ix D j* (β@δ′)
 assumes α-β: *LeftDerivation* α ((*i*,*r*)#(*take n D*)) β

assumes *n-bound*: $n \le length\ D$
assumes *dge*: *derivation-ge* (*drop n D*) (*length* β)
assumes *i-in*: $i < length\ \alpha$
assumes *j-in*: $j < length\ \beta$
shows *LeftDerivationIntro* α *i r ix* (*take n D*) *j* β
proof −
from *LeftDerivationIntro-def*[**where** $\alpha{=}\alpha@\delta$ **and** $i{=}i$ **and** $r{=}r$ **and** $ix{=}ix$ **and** $D{=}D$ **and** $j{=}j$ **and** $\gamma{=}\beta@\delta'$]
obtain ω **where** ω: *LeftDerives1* (α @ δ) *i r* ω $\wedge$
ix < *length* (*snd r*) $\wedge$ *snd r* ! *ix* = (β @ δ') ! *j* $\wedge$ *LeftDerivationFix* ω (*i* + *ix*) *D j* (β @ δ')
using *ldfix* **by** *blast*
then have $\exists\ \alpha'.\ \omega = \alpha'$ @ $\delta \wedge i + length\ (snd\ r) \le length\ \alpha'$
using *i-in* **using** *LeftDerives1-append-replace-in-left* **by** *blast*
then obtain α' **where** α': $\omega = \alpha'$ @ $\delta \wedge i + length\ (snd\ r) \le length\ \alpha'$ **by** *blast*
have α-α': *LeftDerives1* α *i r* α' **using** α' ω **using** *LeftDerives1-skip-suffix i-in* **by** *blast*
from α-β **obtain** *u* **where** *u*: *LeftDerives1* α *i r u* $\wedge$ *LeftDerivation u* (*take n D*) β **by** *auto*
with α-α' **have** $u = \alpha'$ **using** *Derives1-unique-dest LeftDerives1-implies-Derives1* **by** *blast*
with *u* **have** α'-β: *LeftDerivation* α' (*take n D*) β **by** *auto*
have *ldfix-append*: *LeftDerivationFix* (α' @ δ) (*i* + *ix*) *D j* (β @ δ') **using** α' ω **by** *blast*
have *i-plus-ix-bound*: $i + ix < length\ \alpha'$ **using** ω α'
using *add-lessD1 le-add-diff-inverse less-asym' linorder-neqE-nat nat-add-left-cancel-less*

by *linarith*
from *LeftDerivationFix-cut-appendix*[*OF ldfix-append* α'-β *n-bound dge i-plus-ix-bound j-in*]
have *ldfix*: *LeftDerivationFix* α' (*i* + *ix*) (*take n D*) *j* β **.**
show *?thesis*
apply (*simp add*: *LeftDerivationIntro-def*)
apply (*rule-tac* $x{=}\alpha'$ **in** *exI*)
apply *auto*
using α-α' **apply** *blast*
using ω **apply** *blast*
apply (*simp add*: ω *j-in*)
using *ldfix* **by** *blast*
qed

lemma *LeftDerivationIntro-cut-appendix'*:
assumes *ldfix*: *LeftDerivationIntro* ($\alpha@\delta$) *i r ix D j* ($\beta@\delta'$)
assumes α-β: *LeftDerivation* α ((*i*,*r*)#*D*) β
assumes *i-in*: $i < length\ \alpha$
assumes *j-in*: $j < length\ \beta$
shows *LeftDerivationIntro* α *i r ix D j* β
proof −
obtain *n* **where** *n*: $n = length\ D$ **by** *blast*

have *LeftDerivationIntro* α *i r ix* (*take n D*) *j* β
apply (*rule-tac LeftDerivationIntro-cut-appendix*)
apply (*rule ldfix*)
using α-β *n* **apply** *auto*[*1*]
using *n* **apply** *auto*[*1*]
using *n* **apply** *auto*[*1*]
using *i-in* **apply** *blast*
using *j-in* **apply** *blast*
done
then show *?thesis* **using** *n* **by** *auto*
qed

lemma *ladder-n-monotone*: *is-ladder D L* $\Longrightarrow$ $u \leq v$ $\Longrightarrow$ $v <$ *length L* $\Longrightarrow$ *ladder-n L u* $\leq$ *ladder-n L v*
by (*metis is-ladder-def le-neq-implies-less linear not-less*)

lemma *ladder-i-cut*:
assumes *index-bound*: *index* < *length L*
shows *ladder-i* (*ladder-cut L n*) *index* = *ladder-i L index*
proof −
have *index* = *0* ∨ *index* > *0* **by** *arith*
then show *?thesis*
proof (*induct rule*: *disjCases2*)
case *1*
with *index-bound* **have** *L* ≠ [] **by** (*simp add*: *less-numeral-extra*(*3*))
then show *?case* **using** *1* **by** (*simp add*: *ladder-i-of-cut-at-0*)
next
case *2*
then show *?case*
apply (*auto simp add*: *ladder-i-def ladder-cut-def ladder-j-def deriv-j-def Let-def*)
using *index-bound* **by** *auto*
qed
qed

lemma *ladder-j-cut*:
assumes *index-bound*: *index* < *length L*
shows *ladder-j* (*ladder-cut L n*) *index* = *ladder-j L index*
by (*metis gr-implies-not0 index-bound ladder-cut-def ladder-j-def ladder-last-j-def ladder-last-j-of-cut length-ladder-cut list.size*(*3*) *nth-list-update-neq*)

lemma *ladder-ix-cut*:
assumes *index-lower-bound*: *index* > *0*
assumes *index-upper-bound*: *index* < *length L*
shows *ladder-ix* (*ladder-cut L n*) *index* = *ladder-ix L index*
proof −
show *?thesis*
using *index-lower-bound* **apply** (*simp add*: *ladder-ix-def deriv-ix-def*)
by (*metis index-upper-bound ladder-cut-def nth-list-update-eq nth-list-update-neq*

snd-conv)
qed

lemma *LeftDerivation-from-in-between*:
assumes α-β: *LeftDerivation* α (*take u D*) β
assumes α-γ: *LeftDerivation* α (*take v D*) γ
assumes *u-le-v*: $u \leq v$
shows *LeftDerivation* β (*drop u* (*take v D*)) γ
proof −
have *take-split*: *take v D* = *take u D* @ (*drop u* (*take v D*))
by (*metis u-le-v add-diff-cancel-left′ drop-take le-Suc-ex take-add*)
then show *?thesis* **using** α-γ α-β
by (*metis* (*no-types*, *lifting*) *Derivation-unique-dest LeftDerivation-append LeftDerivation-implies-Derivation*)
qed

lemma *LeftDerivationLadder-cut-appendix-helper*:
assumes *LDLadder*: *LeftDerivationLadder* (α@δ) *D L* γ
assumes *ladder-i-in-*α: *ladder-i L 0* < *length* α
shows ∃ *E F* γ*1* γ*2 L′*. *D* = *E*@*F* ∧
γ = γ*1* @ γ*2* ∧
LeftDerivationLadder α *E L′* γ*1* ∧
derivation-ge F (*length* γ*1*) ∧
LeftDerivation δ (*derivation-shift F* (*length* γ*1*) *0*) γ*2* ∧
L′ = *ladder-cut L* (*length E*)
proof −
have *is-ladder-D-L*: *is-ladder D L* **using** *LDLadder LeftDerivationLadder-def* **by** *blast*
obtain *n* **where** *n*: *n* = *ladder-last-n L* **by** *blast*
then have *n-eq-ladder-n*: *n* = *ladder-n L* (*length L* − *1*) **using** *ladder-last-n-def* **by** *blast*
have *length-L-nonzero*: *length L* > *0*
using *LeftDerivationLadder-def assms*(*1*) *is-ladder-def* **by** *blast*
from *LeftDerivationLadder-propagate*[*OF LDLadder ladder-i-in-*α *n-eq-ladder-n*]
obtain *n′* β δ' **where** *finish*:
(*length L* − *1* = *0* ∨ *ladder-prev-n L* (*length L* − *1*) < *n′*) ∧
n′ ≤ *n* ∧
LeftDerivation α (*take n′ D*) β ∧
LeftDerivation (α @ δ) (*take n′ D*) (β @ δ) ∧
derivation-ge (*drop n′ D*) (*length* β) ∧
LeftDerivation δ (*derivation-shift* (*drop n′ D*) (*length* β) *0*) δ' ∧
ladder-γ (α @ δ) *D L* (*length L* − *1*) = β @ δ' ∧ *ladder-j L* (*length L* − *1*) < *length* β
using *length-L-nonzero* **by** *auto*
obtain *E* **where** *E*: *E* = *take n′ D* **by** *blast*
obtain *F* **where** *F*: *F* = *drop n′ D* **by** *blast*
obtain *L′* **where** *L′*: *L′* = *ladder-cut L* (*length E*) **by** *blast*
have γ*-ladder*: γ = *ladder-*γ (α @ δ) *D L* (*length L* − *1*)
by (*metis Derive LDLadder LeftDerivationLadder-def LeftDerivation-implies-Derivation*

append-Nil2 append-take-drop-id drop-eq-Nil is-ladder-def ladder-γ-def le-refl n
n-eq-ladder-n)
then have γ: $\gamma = \beta$ @ δ' **using** *finish* **by** *auto*
have *is-sentence* (α@δ)
using *LDLadder LeftDerivationFix-is-sentence LeftDerivationLadder-def* **by** *blast*
then have *is-sentence-α*: *is-sentence* α **using** *is-sentence-concat* **by** *blast*
have *is-sentence* γ
using *Derivation-implies-derives LDLadder LeftDerivationFix-is-sentence LeftDerivationLadder-def LeftDerivation-implies-Derivation derives-is-sentence* **by** *blast*
then have *is-sentence-β*: *is-sentence* β **using** γ *is-sentence-concat* **by** *blast*
have *length-L'*: *length* L' = *length* L
by (*metis* L' *ladder-cut-def length-list-update*)
have *ladder-last-n-L'*: *ladder-last-n* L' = *length* E
using L' *ladder-last-n-of-cut length-L-nonzero* **by** *blast*
have *length-D-eq-n*: *length* $D = n$
using *LDLadder LeftDerivationLadder-def is-ladder-def n* **by** *auto*
then have *length-E-eq-n'*: *length* $E = n'$ **by** (*simp add*: *E finish min.absorb2*)
{
fix u :: *nat*
assume $u <$ *length* L'
then have $u <$ *length* $L' - 1 \lor u =$ *length* $L' - 1$ **by** *arith*
then have *ladder-n* L' $u \leq$ *length* E
proof (*induct rule*: *disjCases2*)
case *1*
have *u-bound*: $u <$ *length* $L - 1$ **using** *1* **by** (*simp add*: *length-L'*)
then have *L'-eq-L*: *ladder-n* L' u = *ladder-n* L u **using** L' *ladder-n-of-cut length-L-nonzero length-greater-0-conv* **by** *blast*
from *u-bound* **have** *ladder-n* L $u \leq$ *ladder-prev-n* L (*length* $L - 1$)
using *ladder-n-prev-bound LeftDerivationLadder-def assms*(*1*) **by** *blast*
then show *?case*
using *L'-eq-L finish length-E-eq-n' u-bound* **by** *linarith*
next
case *2*
then have *ladder-n* L' u = *length* E **using** *ladder-last-n-L' ladder-last-n-def* **by** *auto*
then show *?case* **by** *auto*
qed
}
note *ladder-n-bound = this*
{
fix u :: *nat*
fix v :: *nat*
assume *u-less-v*: $u < v$
assume *v-bound*: $v <$ *length* L'
have $v <$ *length* $L' - 1 \lor v =$ *length* $L' - 1$ **using** *v-bound* **by** *arith*

then have *ladder-n L′ u* < *ladder-n L′ v*
proof (*induct rule*: *disjCases2*)
case *1*
show *?case*
using *1.hyps L′ LeftDerivationLadder-def assms*(*1*) *is-ladder-def ladder-n-of-cut*
length-L′ u-less-v **by** *auto*
next
case *2*
note *v-def* = *2*
have *v* = *0* ∨ *v* ≠ *0* **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
then show *?case* **using** *u-less-v* **by** *auto*
next
case *2*
then have *ladder-prev-n L* (*length L* − *1*) < *n′* **using** *finish v-def length-L′*

by *linarith*
then show *?case*
by (*metis* (*no-types*, *lifting*) *L′ LeftDerivationLadder-def assms*(*1*)
ladder-last-n-L′ ladder-last-n-def ladder-n-of-cut ladder-n-prev-bound
le-neq-implies-less length-E-eq-n′ length-L′ length-greater-0-conv
less-trans u-less-v v-def)
qed
qed
}
note *ladder-n-pairwise-bound* = *this*

have *is-ladder-E-L′*: *is-ladder E L′*
apply (*auto simp add*: *is-ladder-def ladder-last-n-L′*)
using *length-L-nonzero length-L′* **apply** *simp*
using *ladder-n-bound* **apply** *blast*
using *ladder-n-pairwise-bound* **by** *blast*

{
fix *index* :: *nat*
assume *index-bound*: *index* + *1* < *length L*
then have *index-le*: *index* < *length L* **by** *arith*
from *index-bound* **have** *len-L-minus-1*: *length L* − *1* ≠ *0* **by** *arith*
obtain *m* **where** *m*: *m* = *ladder-n L index* **by** *blast*
from *LeftDerivationLadder-propagate*[*OF LDLadder ladder-i-in-α m index-le*]
obtain *ω* **where**
ω: *LeftDerivation α* (*take m D*) *ω* ∧ *ladder-γ* (*α* @ *δ*) *D L index* = *ω* @ *δ* ∧
ladder-j L index < *length ω* **using** *index-bound* **by** *auto*
have *L′-Derive*: *ladder-γ α E L′ index* = *Derive α* (*take* (*ladder-n L′ index*) *E*)
by (*simp add*: *ladder-γ-def*)

```
have ladder-n-L′-eq-L: ladder-n L′ index = ladder-n L index
  using L′ index-bound ladder-n-of-cut length-L-nonzero by auto
have ladder-prev-n L (length L − 1) < n′ using finish len-L-minus-1 by blast
then have n′-is-upper-bound: ladder-n L (length L − 2) < n′ using index-bound
  by (metis diff-diff-left ladder-prev-n-def len-L-minus-1 one-add-one)
have index-upper-bound: index ≤ length L − 2 using index-bound by linarith

have ladder-n-upper-bound: ladder-n L index ≤ ladder-n L (length L − 2)
  apply (rule-tac ladder-n-monotone)
  apply (rule-tac is-ladder-D-L)
  apply (rule index-upper-bound)
  using length-L-nonzero by linarith
with n′-is-upper-bound have m-le-n′: m ≤ n′
  using dual-order.strict-implies-order le-less-trans m by linarith
then have take m E = take m D
  by (metis E le-take-same length-E-eq-n′ order-refl take-all)
then have take-helper: (take (ladder-n L′ index) E) = take m D
  by (simp add: ladder-n-L′-eq-L m)
then have Derive-eq-ω: Derive α (take (ladder-n L′ index) E) = ω
  by (simp add: Derive LeftDerivation-implies-Derivation ω)
then have t1: ladder-γ (α@δ) D L index = (ladder-γ α E L′ index) @ δ
  by (simp add: L′-Derive ω)
have ω-eq: ω = ladder-γ α E L′ index by (simp add: Derive-eq-ω L′-Derive)
then have t2: LeftDerivation α (take (ladder-n L index) D) (ladder-γ α E L′
index)
  using ω m by blast
have t3: (take (ladder-n L′ index) E) = take (ladder-n L index) D
  by (simp add: m take-helper)
have t4: ladder-j L index < length (ladder-γ α E L′ index)
  using ω ω-eq by blast
have t5: E ! (ladder-n L′ index) = D ! (ladder-n L index)
  using E ladder-n-L′-eq-L ladder-n-upper-bound n′-is-upper-bound by auto
note t = t1 t2 t3 t4 t5
}
note ladder-early-stage = this

{
  fix index :: nat
  assume index-bound: index < length L
  have i: ladder-i L′ index = ladder-i L index
    using L′ ladder-i-cut by (simp add: index-bound)
  have j: ladder-j L′ index = ladder-j L index
    using L′ ladder-j-cut by (simp add: index-bound)
  have ix: index > 0 ⟹ ladder-ix L′ index = ladder-ix L index
    using L′ ladder-ix-cut by (simp add: index-bound)
  have α: ladder-α (α@δ) D L index = (ladder-α α E L′ index) @ δ
    by (simp add: index-bound ladder-α-def ladder-early-stage(1))
  have i-bound: index > 0 ⟹ ladder-i L′ index < length (ladder-α α E L′ index)
    using i index-bound ladder-α-def ladder-early-stage(4) ladder-i-def by auto
```

note *ij* = *i j ix* α *i-bound*
}
note *ladder-every-stage* = *this*

{
fix *u* :: *nat*
fix *v* :: *nat*
assume *u-le-v*: $u \leq v$
assume *v-bound*: $v <$ *length L*
have *ladder-n L u* $\leq$ *ladder-n L v*
using *is-ladder-D-L ladder-n-monotone u-le-v v-bound* **by** *blast*
}
note *ladder-L-n-pairwise-le* = *this*

{
fix *index* :: *nat*
assume *index-lower-bound*: *index* > *0*
assume *index-upper-bound*: *index* + *1* < *length L*
note *derivation* = *ladder-early-stage*(*2*)
have *derivation1*:
LeftDerivation α (*take* (*ladder-n L* (*index* − *Suc 0*)) *D*) (*ladder-*γ α *E L*′ (*index* − *Suc 0*))
using *derivation*[*of index* − *Suc 0*] *index-lower-bound index-upper-bound*
using *One-nat-def Suc-diff-1 Suc-eq-plus1 le-less-trans lessI less-or-eq-imp-le* **by** *linarith*
have *derivation2*:
LeftDerivation α (*take* (*ladder-n L index*) *D*) (*ladder-*γ α *E L*′ *index*)
using *derivation*[*of index*] *index-upper-bound* **by** *blast*
have *ladder-*α*-is-*γ[*symmetric*]: *ladder-*γ α *E L*′ (*index* − *Suc 0*) = *ladder-*α α *E L*′ *index*
using *index-lower-bound ladder-*α*-def* **by** *auto*
have *ladder-n-le*: *ladder-n L* (*index* − *Suc 0*) $\leq$ *ladder-n L index*
apply (*rule-tac ladder-L-n-pairwise-le*)
apply *arith*
using *index-upper-bound* **by** *arith*
from *LeftDerivation-from-in-between*[*OF derivation1 derivation2 ladder-n-le*] *ladder-*α*-is-*γ
have *LeftDerivation* (*ladder-*α α *E L*′ *index*) (*drop* (*ladder-n L*′ (*index* − *Suc 0*))
(*take* (*ladder-n L*′ *index*) *E*)) (*ladder-*γ α *E L*′ *index*)
by (*metis L*′ *Suc-leI add-lessD1 index-lower-bound index-upper-bound ladder-early-stage*(*3*)
ladder-n-of-cut le-add-diff-inverse2 length-L-nonzero length-greater-0-conv less-diff-conv)
}
note *LeftDerivation-delta-early* = *this*

have *LeftDerivationFix-*α*-0*: *LeftDerivationFix* α (*ladder-i L*′ *0*) (*take* (*ladder-n*

L′ 0) *E*)
(*ladder-j L′ 0*) (*ladder-γ α E L′ 0*)
proof −
have *ldfix*: *LeftDerivationFix* (*α*@*δ*) (*ladder-i L 0*) (*take* (*ladder-n L 0*) *D*) (*ladder-j L 0*)
(*ladder-γ* (*α*@*δ*) *D L 0*)
using *LDLadder LeftDerivationLadder-def* **by** *blast*
have *ladder-i-L′*: *ladder-i L′ 0* = *ladder-i L 0*
using *L′ ladder-i-of-cut-at-0 length-L-nonzero* **by** *blast*
have *ladder-j-L′*: *ladder-j L′ 0* = *ladder-j L 0*
by (*metis L′ ladder-cut-def ladder-j-def ladder-last-j-def ladder-last-j-of-cut length-L′ length-greater-0-conv nth-list-update-neq*)
have *length L* = *1* ∨ *length L* > *1* **using** *length-L-nonzero* **by** *linarith*
then show *?thesis*
proof (*induct rule*: *disjCases2*)
case *1*
from *1* **have** *ladder-n-L′-0*: *ladder-n L′ 0* = *n′*
using *diff-is-0-eq′ ladder-last-n-L′ ladder-last-n-def length-E-eq-n′ length-L′ less-or-eq-imp-le* **by** *auto*
have *take-n′-E*: *take n′ E* = *E* **by** (*simp add*: *length-E-eq-n′*)
from *ladder-n-L′-0 take-n′-E* **have** *take-ladder-n-L′*: *take* (*ladder-n L′ 0*) *E* = *E* **by** *auto*
have *ladder-n L 0* = *length D*
by (*simp add*: *1.hyps length-D-eq-n n-eq-ladder-n*)
then have *take-ladder-n-L-0*: *take* (*ladder-n L 0*) *D* = *D* **by** *simp*
have *ladder-γ-α*: *ladder-γ α E L′ 0* = *β*
apply (*simp add*: *ladder-γ-def take-ladder-n-L′*)
by (*simp add*: *Derive E LeftDerivation-implies-Derivation finish*)
have *ladder-j-in-β*: *ladder-j L 0* < *length β*
using *finish 1.hyps* **by** *auto*
have *ldfix-1*: *LeftDerivationFix* (*α*@*δ*) (*ladder-i L 0*) *D* (*ladder-j L 0*) (*β*@*δ′*)
using *1.hyps γ γ-ladder ldfix take-ladder-n-L-0* **by** *auto*
then have *LeftDerivationFix α* (*ladder-i L 0*) *E* (*ladder-j L 0*) *β*
by (*simp add*: *E LeftDerivationFix-cut-appendix finish ladder-i-in-α ladder-j-in-β length-D-eq-n*)
then show *?case*
by (*simp add*: *ladder-i-L′ ladder-j-L′ take-ladder-n-L′ ladder-γ-α*)
next
case *2*
have *h*: *0* + *1* < *length L* **using** *2.hyps* **by** *auto*
note *stage* = *ladder-early-stage*[*of 0*, *OF h*]
have *ldfix0*: *LeftDerivationFix* (*α*@*δ*) (*ladder-i L 0*) (*take* (*ladder-n L 0*) *D*) (*ladder-j L 0*)
((*ladder-γ α E L′ 0*) @ *δ*)
using *ladder-i-L′ ladder-j-L′ ldfix stage*(*1*) *stage*(*3*) **by** *auto*
from *LeftDerivationFix-cut-appendix′*[*OF ldfix0 stage*(*2*) *ladder-i-in-α stage*(*4*)]
show *?case*

by (*simp add*: *ladder-i-L′ ladder-j-L′ stage*(*3*))
qed
qed

{
fix *index* :: *nat*
assume *index-bounds*: *1* ≤ *index* ∧ *index* + *1* < *length L*
have *introsAt-appendix*: *LeftDerivationIntrosAt* (*α*@*δ*) *D L index*
using *LDLadder LeftDerivationIntros-def LeftDerivationLadder-def add-lessD1 index-bounds*
by *blast*
have *index-plus-1-upper-bound*: *index* + *1* < *length L* **using** *index-bounds* **by** *arith*
note *early-stage* = *ladder-early-stage*[*of index*, *OF index-plus-1-upper-bound*]
have *ladder-i-L-index-eq-fst*: *ladder-i L index* = *fst* (*D* ! *ladder-n L* (*index* − *Suc 0*))
using *introsAt-appendix LeftDerivationIntrosAt-def index-bounds* **by** (*metis One-nat-def*)
have *E-at-D-at*: (*E* ! *ladder-n L′* (*index* − *Suc 0*)) = (*D* ! *ladder-n L* (*index* − *Suc 0*))
using *ladder-early-stage*[*of index* − *Suc 0*]
by (*metis One-nat-def add-lessD1 index-bounds le-add-diff-inverse2*)
then have *ladder-i-L′-index-eq-fst*: *ladder-i L′ index* = *fst* (*E* ! *ladder-n L′* (*index* − *Suc 0*))
using *index-bounds ladder-i-L-index-eq-fst ladder-every-stage*(*1*) *le-add1 le-less-trans* **by** *auto*
have *same-derivation*: (*drop* (*Suc* (*ladder-n L′* (*index* − *Suc 0*))) (*take* (*ladder-n L′ index*) *E*)) =
(*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) (*take* (*ladder-n L index*) *D*))
using *L′ early-stage*(*3*) *index-bounds ladder-n-of-cut length-L-nonzero* **by** *auto*
have *deriv-split*: (*drop* (*ladder-n L′* (*index* − *Suc 0*)) (*take* (*ladder-n L′ index*) *E*)) =
((*ladder-i L′ index*, *snd* (*E* ! *ladder-n L′* (*index* − *Suc 0*))) #
drop (*Suc* (*ladder-n L′* (*index* − *Suc 0*))) (*take* (*ladder-n L′ index*) *E*))
by (*metis Cons-nth-drop-Suc One-nat-def Suc-le-lessD add-lessD1 diff-Suc-less index-bounds*
ladder-i-L′-index-eq-fst ladder-n-bound ladder-n-pairwise-bound length-L′
length-take min.absorb2 nth-take prod.collapse)
have *LeftDerivationIntrosAt α E L′ index*
apply (*auto simp add*: *LeftDerivationIntrosAt-def Let-def*)
using *ladder-i-L′-index-eq-fst* **apply** *blast*
apply (*rule-tac LeftDerivationIntro-cut-appendix′*[**where** *δ*=*δ* **and** *δ′* = *δ*])
apply (*metis E-at-D-at LeftDerivationIntrosAt-def One-nat-def Suc-le-lessD add-lessD1*
early-stage(*1*) *index-bounds introsAt-appendix ladder-every-stage*(*2*)
ladder-every-stage(*3*) *ladder-every-stage*(*4*) *ladder-i-L′-index-eq-fst same-derivation*)
defer *1*
using *index-bounds ladder-every-stage*(*5*) **apply** *auto*[*1*]
using *early-stage*(*4*) *index-bounds ladder-every-stage*(*2*) **apply** *auto*[*1*]

using *LeftDerivation-delta-early deriv-split*
by (*metis One-nat-def Suc-le-eq index-bounds*)
}
note *LeftDerivationIntrosAt-early* = *this*

{
fix *index* :: *nat*
assume *index-bounds*: $1 \leq index \wedge index + 1 = length\ L$
have *introsAt-appendix*: *LeftDerivationIntrosAt* (α@δ) *D L index*
using *LDLadder LeftDerivationIntros-def LeftDerivationLadder-def add-lessD1 index-bounds*
by (*metis Suc-eq-plus1 not-less-eq*)
have *ladder-i-L-index-eq-fst*: *ladder-i L index* = *fst* (*D* ! *ladder-n L* (*index* − *Suc 0*))
using *introsAt-appendix LeftDerivationIntrosAt-def index-bounds* **by** (*metis One-nat-def*)
have *E-at-D-at*: (*E* ! *ladder-n L′* (*index* − *Suc 0*)) = (*D* ! *ladder-n L* (*index* − *Suc 0*))
using *ladder-early-stage*[*of index* − *Suc 0*]
by (*metis One-nat-def Suc-eq-plus1 index-bounds le-add-diff-inverse2 lessI*)
then have *ladder-i-L′-index-eq-fst*: *ladder-i L′ index* = *fst* (*E* ! *ladder-n L′* (*index* − *Suc 0*))
using *index-bounds ladder-i-L-index-eq-fst ladder-every-stage*(*1*) *le-add1 le-less-trans* **by** *auto*
obtain *D′* **where** *D′*: *D′* = (*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) (*take* (*ladder-n L index*) *D*))
by *blast*
obtain *k* **where** *k*: *k* = (*ladder-n L′ index*) − (*Suc* (*ladder-n L′* (*index* − *Suc 0*)))
by *blast*
have *ladder-n-L′-index*: *ladder-n L′ index* = *length E*
by (*metis diff-add-inverse2 index-bounds ladder-last-n-L′ ladder-last-n-def length-L′*)
have *take-is-E*: *take* (*ladder-n L′ index*) *E* = *E* **by** (*simp add*: *ladder-n-L′-index*)
have *ladder-n-L-index*: *ladder-n L index* = *length D*
by (*metis diff-add-inverse2 index-bounds length-D-eq-n n-eq-ladder-n*)
have *take-is-D*: *take* (*ladder-n L index*) *D* = *D*
by (*simp add*: *ladder-n-L-index*)
have *write-as-take-k-D′*: (*drop* (*Suc* (*ladder-n L′* (*index* − *Suc 0*))) *E*) = *take k D′*
using *take-is-D*
by (*metis D′ E L′ One-nat-def Suc-le-lessD add-diff-cancel-right′ diff-Suc-less*
drop-take index-bounds k ladder-n-L′-index ladder-n-of-cut length-E-eq-n′ length-L-nonzero length-greater-0-conv)
have *k-bound*: *k* ≤ *length D′*
by (*metis le-iff-add append-take-drop-id k ladder-n-L′-index length-append length-drop write-as-take-k-D′*)
have *D′-alt*: *D′* = *drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) *D*

by (*simp add*: D' *take-is-D*)
have *LeftDerivationIntrosAt* α E L' *index*
apply (*auto simp add*: *LeftDerivationIntrosAt-def Let-def*)
using *ladder-i-L'-index-eq-fst* **apply** *blast*
apply (*subst take-is-E*)
apply (*subst write-as-take-k-D'*)
apply (*rule-tac LeftDerivationIntro-cut-appendix*[**where** δ=δ **and** $\delta' = \delta'$])
apply (*metis D' Derive E E-at-D-at LeftDerivationIntrosAt-def LeftDerivation-implies-Derivation One-nat-def Suc-le-lessD add-diff-cancel-right'*

diff-Suc-less finish index-bounds introsAt-appendix ladder-γ*-def ladder-every-stage*(*2*)

ladder-every-stage(*3*) *ladder-every-stage*(*4*) *ladder-i-L'-index-eq-fst length-L-nonzero*

take-is-E)
apply (*metis Cons-nth-drop-Suc E L' LeftDerivation-from-in-between LeftDerivation-take-derive*
One-nat-def Suc-le-lessD add-diff-cancel-right' diff-Suc-less finish index-bounds

ladder-α*-def ladder-*γ*-def ladder-i-L'-index-eq-fst ladder-n-L'-index ladder-n-of-cut*
ladder-prev-n-def length-E-eq-n' length-L-nonzero less-imp-le-nat less-numeral-extra(*3*)

list.size(*3*) *prod.collapse take-is-E write-as-take-k-D'*)
using *k-bound* **apply** *blast*
using *D'-alt* **apply** (*metis* (*no-types*, *lifting*) *Derive E L' LeftDerivation-implies-Derivation*

One-nat-def Suc-leI Suc-le-lessD add-diff-cancel-right' diff-Suc-less drop-drop finish
index-bounds k ladder-γ*-def ladder-n-L'-index ladder-n-of-cut ladder-prev-n-def*

le-add-diff-inverse2 length-E-eq-n' length-L-nonzero length-greater-0-conv
less-not-refl2 take-is-E)
using *index-bounds ladder-every-stage*(*5*) **apply** *auto*[*1*]
by (*metis Derive E LeftDerivation-implies-Derivation One-nat-def add-diff-cancel-right'*

diff-Suc-less finish index-bounds ladder-γ*-def ladder-every-stage*(*2*) *length-L-nonzero*

take-is-E)
}
note *LeftDerivationIntrosAt-last* = *this*

have *ladder-E-L'*: *LeftDerivationLadder* α E L' β
apply (*auto simp add*: *LeftDerivationLadder-def*)
using *finish E* **apply** *blast*
using *is-ladder-E-L'* **apply** *blast*
using *LeftDerivationFix-*α*-0* **apply** *blast*
using *LeftDerivationIntros-def LeftDerivationIntrosAt-early LeftDerivationIntrosAt-last*

by (*metis Suc-eq-plus1 Suc-leI le-neq-implies-less length-L′*)

show *?thesis*
apply (*rule-tac x=E* **in** *exI*)
apply (*rule-tac x=F* **in** *exI*)
apply (*rule-tac x=β* **in** *exI*)
apply (*rule-tac x=δ′* **in** *exI*)
apply (*rule-tac x=L′* **in** *exI*)
apply *auto*
using *E F* **apply** *simp*
apply (*rule γ*)
using *ladder-E-L′* **apply** *blast*
using *F finish* **apply** *blast*
using *F finish* **apply** *blast*
by (*rule L′*)
qed

theorem *LeftDerivationLadder-cut-appendix*:
assumes *LDLadder*: *LeftDerivationLadder* (*α*@*δ*) *D L γ*
assumes *ladder-i-in-α*: *ladder-i L 0* < *length α*
shows ∃ *E F γ1 γ2 L′*. *D* = *E*@*F* ∧
γ = *γ1* @ *γ2* ∧
LeftDerivationLadder α E L′ γ1 ∧
derivation-ge F (*length γ1*) ∧
LeftDerivation δ (*derivation-shift F* (*length γ1*) *0*) *γ2* ∧
length L′ = *length L* ∧ *ladder-i L′ 0* = *ladder-i L 0* ∧
ladder-last-j L′ = *ladder-last-j L*
proof −
from *LeftDerivationLadder-cut-appendix-helper*[*OF LDLadder ladder-i-in-α*]
obtain *E F γ1 γ2 L′* **where** *helper*:
D = *E* @ *F* ∧
γ = *γ1* @ *γ2* ∧
LeftDerivationLadder α E L′ γ1 ∧
derivation-ge F (*length γ1*) ∧
LeftDerivation δ (*derivation-shift F* (*length γ1*) *0*) *γ2* ∧ *L′* = *ladder-cut L*
(*length E*)
by *blast*
show *?thesis*
apply (*rule-tac x=E* **in** *exI*)
apply (*rule-tac x=F* **in** *exI*)
apply (*rule-tac x=γ1* **in** *exI*)
apply (*rule-tac x=γ2* **in** *exI*)
apply (*rule-tac x=L′* **in** *exI*)
using *helper LDLadder LeftDerivationLadder-def is-ladder-def ladder-i-of-cut-at-0*

ladder-last-j-of-cut length-ladder-cut **by** *force*
qed

definition *ladder-stepdown-diff* :: *ladder* ⇒ *nat* **where**

ladder-stepdown-diff L = *Suc* (*ladder-n L 0*)

definition *ladder-stepdown-α-0* :: *sentence* ⇒ *derivation* ⇒ *ladder* ⇒ *sentence*
where
ladder-stepdown-α-0 a D L = *Derive a* (*take* (*ladder-stepdown-diff L*) *D*)

lemma *LeftDerivationIntro-LeftDerives1*:
assumes *LeftDerivationIntro α i r ix D j γ*
assumes *splits-at α i a1 A a2*
shows *LeftDerives1 α i r* (*a1*@(*snd r*)@*a2*)
by (*metis LeftDerivationIntro-def LeftDerivationIntro-examine-rule LeftDerivation-Intro-is-sentence*
LeftDerives1-def assms(*1*) *assms*(*2*) *prod.collapse splits-at-implies-Derives1*)

lemma *LeftDerives1-Derive*:
assumes *LeftDerives1 α i r γ*
shows *Derive α* [(*i*, *r*)] = *γ*
by (*metis Derive LeftDerivation.simps*(*1*) *LeftDerivation-LeftDerives1*
LeftDerivation-implies-Derivation append-Nil assms)

lemma *ladder-stepdown-α-0-altdef*:
assumes *ladder*: *LeftDerivationLadder α D L γ*
assumes *length-L*: *length L* > *1*
assumes *split*: *splits-at* (*ladder-α α D L 1*) (*ladder-i L 1*) *a1 A a2*
shows *ladder-stepdown-α-0 α D L* = *a1* @ (*snd* (*snd* (*D* ! (*ladder-n L 0*)))) @ *a2*
proof −
have *1*: *ladder-α α D L 1* = *Derive α* (*take* (*ladder-n L 0*) *D*)
by (*simp add*: *ladder-α-def ladder-γ-def*)
obtain *rule* **where** *rule*: *rule* = *snd* (*D* ! (*ladder-n L 0*)) **by** *blast*
have ∃ *E ω*. *LeftDerivationIntro* (*ladder-α α D L 1*) (*ladder-i L 1*) *rule* (*ladder-ix L 1*)
E (*ladder-j L 1*) *ω*
by (*metis LeftDerivationIntrosAt-def LeftDerivationIntros-def LeftDerivation-Ladder-def*
One-nat-def diff-Suc-1 ladder length-L order-refl rule)
then obtain *E ω* **where** *intro*:
LeftDerivationIntro (*ladder-α α D L 1*) (*ladder-i L 1*) *rule* (*ladder-ix L 1*) *E* (*ladder-j L 1*) *ω*
by *blast*
then have *2*: *LeftDerives1* (*ladder-α α D L 1*) (*ladder-i L 1*) *rule* (*a1*@(*snd rule*)@*a2*)
using *LeftDerivationIntro-LeftDerives1 split* **by** *blast*
have *fst-D*: *fst* (*D* ! (*ladder-n L 0*)) = *ladder-i L 1*
by (*metis LeftDerivationIntrosAt-def LeftDerivationIntros-def LeftDerivation-Ladder-def*
One-nat-def diff-Suc-1 ladder le-numeral-extra(*4*) *length-L*)
have *derive-derive*: *Derive α* (*take* (*Suc* (*ladder-n L 0*)) *D*) =
Derive (*Derive α* (*take* (*ladder-n L 0*) *D*)) [*D* ! (*ladder-n L 0*)]

proof −
have *f1*: *Derivation* α (*take* (*Suc* (*ladder-n L 0*)) *D*) (*Derive* α (*take* (*Suc* (*ladder-n L 0*)) *D*))
using *Derivation-take-derive LeftDerivationLadder-def LeftDerivation-implies-Derivation ladder* **by** *blast*
have *f2*: *length L* − *1* < *length L*
using *length-L* **by** *linarith*
have *0* < *length L* − *1*
using *length-L* **by** *linarith*
then have *f3*: *take* (*Suc* (*ladder-n L 0*)) *D* = *take* (*ladder-n L 0*) *D* @ [*D* ! *ladder-n L 0*]
using *f2* **by** (*metis* (*full-types*) *LeftDerivationLadder-def is-ladder-def ladder ladder-last-n-def take-Suc-conv-app-nth*)
obtain *sss* :: *symbol list* ⇒ (*nat* × *symbol* × *symbol list*) *list* ⇒ (*nat* × *symbol* × *symbol list*) *list* ⇒ *symbol list* ⇒ *symbol list* **where**
∀ *x0 x1 x2 x3*. (∃ *v4*. *Derivation x3 x2 v4* ∧ *Derivation v4 x1 x0*) = (*Derivation x3 x2* (*sss x0 x1 x2 x3*) ∧ *Derivation* (*sss x0 x1 x2 x3*) *x1 x0*)
by *moura*
then show *?thesis*
using *f3 f1 Derivation-append Derive* **by** *auto*
qed
then have *3*: *ladder-stepdown-α-0* α *D L* = *Derive* (*ladder-α* α *D L 1*) [*D* ! (*ladder-n L 0*)]
using *1* **by** (*simp add*: *ladder-stepdown-α-0-def ladder-stepdown-diff-def*)
have *4*: *D* ! (*ladder-n L 0*) = (*ladder-i L 1*, *rule*)
using *rule fst-D* **by** (*metis prod.collapse*)
then show *?thesis* **using** *2 3 4 LeftDerives1-Derive snd-conv* **by** *auto*
qed

lemma *ladder-i-0-bound*:
assumes *ld*: *LeftDerivationLadder* α *D L* γ
shows *ladder-i L 0* < *length* α
proof −
have *LeftDerivationFix* α (*ladder-i L 0*) (*take* (*ladder-n L 0*) *D*) (*ladder-j L 0*) (*ladder-γ* α *D L 0*)
using *ld LeftDerivationLadder-def* **by** *simp*
then show *?thesis* **using** *LeftDerivationFix-def* **by** *simp*
qed

lemma *ladder-j-bound*:
assumes *ld*: *LeftDerivationLadder* α *D L* γ
assumes *index-bound*: *index* < *length L*
shows *ladder-j L index* < *length* (*ladder-γ* α *D L index*)
proof −
have *ld′*: *LeftDerivationLadder* (α@[]) *D L* γ **using** *ld* **by** *simp*
have *ladder-i-0*: *ladder-i L 0* < *length* α **using** *ladder-i-0-bound ld* **by** *auto*
obtain *n* **where** *n*: *n* = *ladder-n L index* **by** *blast*
note *propagate* = *LeftDerivationLadder-propagate*[*OF ld′ ladder-i-0 n index-bound*]
from *index-bound* **have** *index* + *1* < *length L* ∨ *index* + *1* = *length L* **by** *arith*

```
  then show ?thesis
  proof (induct rule: disjCases2)
    case 1
      then have ∃β. LeftDerivation α (take n D) β ∧
         ladder-γ (α @ []) D L index = β @ [] ∧ ladder-j L index < length β
         using propagate by auto
      then show ?case by auto
  next
    case 2
      then have
        ∃n' β δ'.
          (index = 0 ∨ ladder-prev-n L index < n') ∧
          n' ≤ n ∧
          LeftDerivation α (take n' D) β ∧
          LeftDerivation (α @ []) (take n' D) (β @ []) ∧
          derivation-ge (drop n' D) (length β) ∧
          LeftDerivation [] (derivation-shift (drop n' D) (length β) 0) δ' ∧
          ladder-γ (α @ []) D L index = β @ δ' ∧ ladder-j L index < length β
          using propagate by auto
      then show ?case by auto
  qed
qed

lemma ladder-last-j-bound:
  assumes ld: LeftDerivationLadder α D L γ
  shows ladder-last-j L < length γ
proof −
  have length L − 1 < length L
    using LeftDerivationLadder-def assms is-ladder-def by auto
  from ladder-j-bound[OF ld this]
  show ?thesis
     by (metis Derive LeftDerivationLadder-def LeftDerivation-implies-Derivation
One-nat-def
      is-ladder-def ladder-last-j-def last-ladder-γ ld)
qed

fun ladder-shift-n :: nat ⇒ ladder ⇒ ladder where
  ladder-shift-n N [] = []
| ladder-shift-n N ((n, j, i)#L) = ((n − N, j, i)#(ladder-shift-n N L))

fun ladder-stepdown :: ladder ⇒ ladder
where
  ladder-stepdown [] = undefined
| ladder-stepdown [v] = undefined
| ladder-stepdown ((n0, j0, i0)#(n1, j1, ix1)#L) =
  (n1 − Suc n0, j1, j0 + ix1) # (ladder-shift-n (Suc n0) L)

lemma ladder-shift-n-length:
  length (ladder-shift-n N L) = length L
```

```
  by (induct L, auto)

lemma ladder-stepdown-prepare:
  assumes length L > 1
  shows L = (ladder-n L 0, ladder-j L 0, ladder-i L 0)#
    (ladder-n L 1, ladder-j L 1, ladder-ix L 1)#(drop 2 L)
proof -
  have ∃ n0 j0 i0 n1 j1 ix1 L'. L = ((n0, j0, i0)#(n1, j1, ix1)#L')
     by (metis One-nat-def Suc-eq-plus1 assms ladder-stepdown.cases less-not-refl
list.size(3)
      list.size(4) not-less0)
 then obtain n0 j0 i0 n1 j1 ix1 L' where L': L = ((n0, j0, i0)#(n1, j1, ix1)#L')
by blast
  have n0: n0 = ladder-n L 0 using L'
    by (auto simp add: ladder-n-def deriv-n-def)
  show ?thesis using L'
    by (auto simp add: ladder-n-def deriv-n-def ladder-j-def deriv-j-def
      ladder-i-def deriv-i-def ladder-ix-def deriv-ix-def)
qed

lemma ladder-stepdown-length:
  assumes length L > 1
  shows length (ladder-stepdown L) = length L − 1
apply (subst ladder-stepdown-prepare[OF assms(1)])
apply (simp add: ladder-shift-n-length)
using assms apply arith
done

lemma ladder-stepdown-i-0:
  assumes length L > 1
  shows ladder-i (ladder-stepdown L) 0 = ladder-i L 1 + ladder-ix L 1
  using ladder-stepdown-prepare[OF assms] ladder-i-def ladder-j-def deriv-j-def
  by (metis One-nat-def deriv-i-def diff-Suc-1 ladder-stepdown.simps(3) list.sel(1)

    snd-conv zero-neq-one)

lemma ladder-shift-n-cons: ladder-shift-n N (x#L) = (fst x − N, snd x)#(ladder-shift-n
N L)
  apply (induct L)
  by (cases x, simp)+

lemma ladder-shift-n-drop: ladder-shift-n N (drop n L) = drop n (ladder-shift-n N
L)
proof (induct L arbitrary: n)
  case Nil then show ?case by simp
next
  case (Cons x L)
    show ?case
    proof (cases n)
```

```
      case 0 then show ?thesis
        by simp
    next
      case (Suc n) then show ?thesis
        by (simp add: ladder-shift-n-cons Cons)
    qed
qed

lemma drop-2-shift:
  assumes index > 0
  assumes length L > 1
  shows drop 2 L ! (index − Suc 0) = L ! Suc index
proof −
  define l1 l2 and L′ where l1 = L ! 0 l2 = L ! 1
    and L′ = drop 2 L
  with ‹length L > 1› have L = l1 # l2 # L′
    by (cases L) (auto simp add: neq-Nil-conv)
  with ‹index > 0› show ?thesis
    by simp
qed

lemma ladder-shift-n-at:
  index < length L ⟹ (ladder-shift-n N L) ! index = (fst (L ! index) − N, snd
(L ! index))
proof (induct L arbitrary: index)
  case Nil then show ?case by auto
next
  case (Cons x L)
    show ?case
      apply (simp add: ladder-shift-n-cons)
      apply (cases index)
      apply (auto)
      apply (rule-tac Cons(1))
      using Cons(2) by auto
qed

lemma ladder-stepdown-j:
  assumes length-L-greater-1: length L > 1
  assumes L′: L′ = ladder-stepdown L
  assumes index-bound: index < length L′
  shows ladder-j L′ index = ladder-j L (Suc index)
proof −
  note L-prepare = ladder-stepdown-prepare[OF length-L-greater-1]
  have ladder-stepdown-L-def: ladder-stepdown L = ((ladder-n L (Suc 0) − Suc
(ladder-n L 0), ladder-j L (Suc 0), ladder-j L 0 + ladder-ix L (Suc 0)) #
    ladder-shift-n (Suc (ladder-n L 0)) (drop 2 L))
    by (subst L-prepare, simp)
  have index = 0 ∨ index > 0 by arith
  then show ladder-j L′ index = ladder-j L (Suc index)
```

proof (*induct rule*: *disjCases2*)
case *1*
show *?case*
by (*simp add*: *L′ ladder-stepdown-L-def 1 ladder-j-def deriv-j-def*)
next
case *2*
have *index-bound′*: *Suc index* < *length L*
using *index-bound L′ ladder-stepdown-length length-L-greater-1* **by** *auto*
show *?case*
apply (*simp add*: *L′ ladder-stepdown-L-def 2 ladder-j-def ladder-shift-n-drop drop-2-shift*)
apply (*subst drop-2-shift*)
apply (*simp add*: *2*)
using *length-L-greater-1* **apply** (*simp add*: *ladder-shift-n-length*)
apply (*simp add*: *deriv-j-def*)
apply (*simp add*: *ladder-shift-n-at*[*OF index-bound′*])
done
qed
qed

lemma *ladder-stepdown-last-j*:
assumes *length-L-greater-1*: *length L* > *1*
shows *ladder-last-j* (*ladder-stepdown L*) = *ladder-last-j L*
using *ladder-stepdown-j Suc-diff-Suc diff-Suc-1 ladder-last-j-def ladder-stepdown-length*

length-L-greater-1 lessI **by** *auto*

lemma *ladder-stepdown-n*:
assumes *length-L-greater-1*: *length L* > *1*
assumes *L′*: *L′* = *ladder-stepdown L*
assumes *index-bound*: *index* < *length L′*
shows *ladder-n L′ index* = *ladder-n L* (*Suc index*) − *ladder-stepdown-diff L*
proof −
note *L-prepare* = *ladder-stepdown-prepare*[*OF length-L-greater-1*]
have *ladder-stepdown-L-def*: *ladder-stepdown L* = ((*ladder-n L* (*Suc 0*) − *Suc* (*ladder-n L 0*), *ladder-j L* (*Suc 0*), *ladder-j L 0* + *ladder-ix L* (*Suc 0*)) # *ladder-shift-n* (*Suc* (*ladder-n L 0*)) (*drop 2 L*))
by (*subst L-prepare*, *simp*)
have *index* = *0* ∨ *index* > *0* **by** *arith*
then show *ladder-n L′ index* = *ladder-n L* (*Suc index*) − *ladder-stepdown-diff L*
proof (*induct rule*: *disjCases2*)
case *1*
show *?case*
by (*simp add*: *L′ ladder-stepdown-L-def 1 ladder-n-def deriv-n-def ladder-stepdown-diff-def*)
next
case *2*
have *index-bound′*: *Suc index* < *length L*

```
      using index-bound L′ ladder-stepdown-length length-L-greater-1 by auto
    show ?case
     apply (simp add: L′ ladder-stepdown-L-def 2 ladder-n-def ladder-shift-n-drop
drop-2-shift
        ladder-stepdown-diff-def)
      apply (subst drop-2-shift)
      apply (simp add: 2)
      using length-L-greater-1 apply (simp add: ladder-shift-n-length)
      apply (simp add: deriv-n-def)
      apply (simp add: ladder-shift-n-at[OF index-bound′])
      done
  qed
qed

lemma ladder-stepdown-ix:
  assumes length-L-greater-1: length L > 1
  assumes L′: L′ = ladder-stepdown L
  assumes index-lower-bound: 0 < index
  assumes index-upper-bound: index < length L′
  shows ladder-ix L′ index = ladder-ix L (Suc index)
proof −
  note L-prepare = ladder-stepdown-prepare[OF length-L-greater-1]
  have ladder-stepdown-L-def: ladder-stepdown L = ((ladder-n L (Suc 0) − Suc
(ladder-n L 0), ladder-j L (Suc 0), ladder-j L 0 + ladder-ix L (Suc 0)) #
    ladder-shift-n (Suc (ladder-n L 0)) (drop 2 L))
    by (subst L-prepare, simp)

  have index-bound′: Suc index < length L
    using index-upper-bound L′ ladder-stepdown-length length-L-greater-1 by auto
  show ?thesis
      apply (simp add: L′ ladder-stepdown-L-def index-lower-bound ladder-ix-def
ladder-shift-n-drop)
      apply (subst drop-2-shift)
      apply (simp add: index-lower-bound)
      using length-L-greater-1 apply (simp add: ladder-shift-n-length)
      apply (simp add: deriv-ix-def)
      apply (simp add: ladder-shift-n-at[OF index-bound′])
      using index-lower-bound by arith
qed

lemma Derive-Derive:
  assumes Derivation α (D@E) γ
  shows Derive (Derive α D) E = Derive α (D@E)
using Derivation-append Derive assms by fastforce

lemma drop-at-shift:
  assumes n ≤ index
  assumes index < length D
  shows drop n D ! (index − n) = D ! index
```

using *assms*(*1*) *assms*(*2*) **by** *auto*

theorem *LeftDerivationLadder-stepdown*:
assumes *ldl*: *LeftDerivationLadder* α *D L* γ
assumes *length-L*: *length L* > *1*
shows $\exists$ *L'*. *LeftDerivationLadder* (*ladder-stepdown-*α*-0* α *D L*) (*drop* (*ladder-stepdown-diff L*) *D*)
L' γ $\wedge$ *length L'* = *length L* − *1* $\wedge$ *ladder-i L' 0* = *ladder-i L 1* + *ladder-ix L 1* $\wedge$
ladder-last-j L' = *ladder-last-j L*
proof −
obtain *L'* **where** *L'*: *L'* = *ladder-stepdown L* **by** *blast*
have *ldl1*: *LeftDerivation* (*ladder-stepdown-*α*-0* α *D L*) (*drop* (*ladder-stepdown-diff L*) *D*) γ
proof −
have *D-split*: *D* = (*take* (*ladder-stepdown-diff L*) *D*) @ (*drop* (*ladder-stepdown-diff L*) *D*)
by *simp*
show *?thesis* **using** *D-split ldl*
proof −
obtain *sss* :: *symbol list* ⇒ (*nat* × *symbol* × *symbol list*) *list* ⇒ (*nat* × *symbol* × *symbol list*) *list* ⇒ *symbol list* ⇒ *symbol list* **where**
∀ *x0 x1 x2 x3*. (∃ *v4*. *LeftDerivation x3 x2 v4* ∧ *LeftDerivation v4 x1 x0*) = (*LeftDerivation x3 x2* (*sss x0 x1 x2 x3*) ∧ *LeftDerivation* (*sss x0 x1 x2 x3*) *x1 x0*)
by *moura*
then have (¬ *LeftDerivation* α (*take* (*ladder-stepdown-diff L*) *D* @ *drop* (*ladder-stepdown-diff L*) *D*) γ ∨ *LeftDerivation* α (*take* (*ladder-stepdown-diff L*) *D*) (*sss* γ (*drop* (*ladder-stepdown-diff L*) *D*) (*take* (*ladder-stepdown-diff L*) *D*) α) ∧ *LeftDerivation* (*sss* γ (*drop* (*ladder-stepdown-diff L*) *D*) (*take* (*ladder-stepdown-diff L*) *D*) α) (*drop* (*ladder-stepdown-diff L*) *D*) γ) ∧ (*LeftDerivation* α (*take* (*ladder-stepdown-diff L*) *D* @ *drop* (*ladder-stepdown-diff L*) *D*) γ ∨ (∀ *ss*. ¬ *LeftDerivation* α (*take* (*ladder-stepdown-diff L*) *D*) *ss* ∨ ¬ *LeftDerivation ss* (*drop* (*ladder-stepdown-diff L*) *D*) γ))
using *LeftDerivation-append* **by** *blast*
then show *?thesis*
by (*metis* (*no-types*) *D-split Derivation-take-derive Derivation-unique-dest LeftDerivationLadder-def LeftDerivation-implies-Derivation ladder-stepdown-*α*-0-def ldl*)
qed
qed
have *L'-nonempty*: *L'* ≠ [] **using** *L' ladder-stepdown-length length-L* **by** *fastforce*
{
fix *u* :: *nat*
assume *u'*: *u* < *length L'*
then have *Suc-u*: *Suc u* < *length L* **using** *L' ladder-stepdown-length length-L* **by** *auto*
have *ladder-n L* (*Suc u*) ≤ *length D*
using *ldl Suc-u* **by** (*simp add*: *LeftDerivationLadder-ladder-n-bound*)
then have *ladder-n L' u* ≤ *length D* − *ladder-stepdown-diff L*

```
    apply (subst ladder-stepdown-n[OF length-L L' u'])
    by auto
  }
  note is-ladder-prop1 = this
  {
    fix u :: nat
    fix v :: nat
    assume u-less-v: u < v
    assume v-L': v < length L'
    from u-less-v v-L' have u-L': u < length L' by arith
    have ladder-n L (Suc u) < ladder-n L (Suc v)
     using ldl by (metis (no-types, lifting) L' LeftDerivationLadder-def One-nat-def
Suc-diff-1
        Suc-lessD Suc-mono is-ladder-def ladder-stepdown-length length-L u-less-v
v-L')
    then have ladder-n L' u < ladder-n L' v
      apply (simp add: ladder-stepdown-n[OF length-L L'] u-L' v-L')
     by (metis (no-types, lifting) L' LeftDerivationLadder-def Suc-eq-plus1 Suc-leI
        diff-less-mono is-ladder-def ladder-stepdown-diff-def ladder-stepdown-length
ldl
       length-L less-diff-conv u-L' zero-less-Suc)
  }
  note is-ladder-prop2 = this
  have is-ladder-L': is-ladder (drop (ladder-stepdown-diff L) D) L'
    apply (auto simp add: is-ladder-def)
    using L'-nonempty apply blast
    using is-ladder-prop1 apply blast
    using is-ladder-prop2 apply blast
   using ladder-last-n-def ladder-stepdown-n L' LeftDerivationLadder-def Suc-diff-Suc
diff-Suc-1
     ladder-n-last-is-length ladder-stepdown-length ldl length-L lessI by auto
  have ldfix: LeftDerivationFix (ladder-stepdown-α-0 α D L) (ladder-i L' 0)
    (take (ladder-n L' 0) (drop (ladder-stepdown-diff L) D)) (ladder-j L' 0)
     (ladder-γ (ladder-stepdown-α-0 α D L) (drop (ladder-stepdown-diff L) D) L'
0)
  proof −
    have introsAt-L-1: LeftDerivationIntrosAt α D L 1
      using LeftDerivationIntros-def LeftDerivationLadder-def ldl length-L by blast
    thm LeftDerivationIntrosAt-def
    obtain n where n: n = ladder-n L 0 by blast
    obtain m where m: m = ladder-n L 1 by blast
    have LeftDerivationIntro (ladder-α α D L 1) (ladder-i L 1) (snd (D ! n))
     (ladder-ix L 1) (drop (Suc n) (take m D)) (ladder-j L 1) (ladder-γ α D L 1)
       using n m introsAt-L-1 by (metis LeftDerivationIntrosAt-def One-nat-def
diff-Suc-1)
    from iffD1[OF LeftDerivationIntro-def this] obtain β where β:
      LeftDerives1 (ladder-α α D L 1) (ladder-i L 1) (snd (D ! n)) β ∧
       ladder-ix L 1 < length (snd (snd (D ! n))) ∧
       snd (snd (D ! n)) ! ladder-ix L 1 = ladder-γ α D L 1 ! ladder-j L 1 ∧
```

LeftDerivationFix β (*ladder-i L 1* + *ladder-ix L 1*) (*drop* (*Suc n*) (*take m D*)) (*ladder-j L 1*)
(*ladder-*γ α *D L 1*)
by *blast*
have β = *Derive* (*ladder-*α α *D L 1*) [*D* ! *n*]
by (*metis* (*no-types*, *opaque-lifting*) *LeftDerivationIntrosAt-def LeftDerives1-Derive* β
cancel-comm-monoid-add-class.diff-cancel introsAt-L-1 n prod.collapse)
then have β*-def*: β = *ladder-stepdown-*α*-0* α *D L*
proof −
obtain *sss* :: *nat* ⇒ *symbol list* ⇒ *symbol list* **and** *ss* :: *nat* ⇒ *symbol list* ⇒ *symbol* **and** *sssa* :: *nat* ⇒ *symbol list* ⇒ *symbol list* **where**
∀ *x2 x3*. (∃ *v4 v5 v6*. *splits-at x3 x2 v4 v5 v6*) = *splits-at x3 x2* (*sss x2 x3*) (*ss x2 x3*) (*sssa x2 x3*)
by *moura*
then have *f1*: ∀ *ssa n p ssb*. ¬ *Derives1 ssa n p ssb* ∨ *splits-at ssa n* (*sss n ssa*) (*ss n ssa*) (*sssa n ssa*)
using *splits-at-ex* **by** *presburger*
then have β = *sss* (*ladder-i L 1*) (*ladder-*α α *D L 1*) @ *snd* (*snd* (*D* ! *n*)) @ *sssa* (*ladder-i L 1*) (*ladder-*α α *D L 1*)
by (*meson LeftDerives1-implies-Derives1* β *splits-at-combine-dest*)
then show *?thesis*
using *f1* **by** (*metis* (*no-types*) *LeftDerives1-implies-Derives1* β *ladder-stepdown-*α*-0-altdef ldl length-L n*)
qed
have *ladder-i-L′-0*: *ladder-i L′ 0* = *ladder-i L 1* + *ladder-ix L 1*
using *L′ ladder-stepdown-i-0 length-L* **by** *blast*
have *derivation-eq*: (*take* (*ladder-n L′ 0*) (*drop* (*ladder-stepdown-diff L*) *D*)) =

(*drop* (*Suc n*) (*take m D*)) **using** *n m*
by (*metis L′ L′-nonempty One-nat-def drop-take ladder-stepdown-diff-def ladder-stepdown-n*
length-L length-greater-0-conv)
have *ladder-j-L′-0*: *ladder-j L′ 0* = *ladder-j L 1*
using *L′ L′-nonempty ladder-stepdown-j length-L* **by** *auto*
have *ladder-*γ: (*ladder-*γ (*ladder-stepdown-*α*-0* α *D L*) (*drop* (*ladder-stepdown-diff L*) *D*) *L′ 0*) =
ladder-γ α *D L 1*
by (*metis Derivation-take-derive Derivation-unique-dest LeftDerivationFix-def*

LeftDerivation-implies-Derivation β β*-def derivation-eq ladder-*γ*-def ldl1*)
from β*-def* β *ladder-i-L′-0 derivation-eq ladder-j-L′-0 ladder-*γ
show *?thesis* **by** *auto*
qed
{
fix *index* :: *nat*
assume *index-lower-bound*: *Suc 0* ≤ *index*
assume *index-upper-bound*: *index* < *length L′*
then have *Suc-index-upper-bound*: *Suc index* < *length L*

using *L′ Suc-diff-Suc Suc-less-eq diff-Suc-1 ladder-stepdown-length length-L less-Suc-eq*
by *auto*
then have *intros-at-Suc-index*: *LeftDerivationIntrosAt* α *D L* (*Suc index*)
by (*metis LeftDerivationIntros-def LeftDerivationLadder-def Suc-eq-plus1-left ldl le-add1*)
from *iffD1*[*OF LeftDerivationIntrosAt-def this*] **have** *ldintro*:
let α' = *ladder-*α α *D L* (*Suc index*); *i* = *ladder-i L* (*Suc index*); *j* = *ladder-j L* (*Suc index*);
ix = *ladder-ix L* (*Suc index*); γ = *ladder-*γ α *D L* (*Suc index*); *n* = *ladder-n L* (*Suc index* − *1*);
m = *ladder-n L* (*Suc index*); *e* = *D* ! *n*; *E* = *drop* (*Suc n*) (*take m D*)
in i = *fst e* ∧ *LeftDerivationIntro* α' *i* (*snd e*) *ix E j* γ **by** *blast*
have *index-minus-Suc-0-bound*: *index* − *Suc 0* < *length L′*
by (*simp add*: *index-upper-bound less-imp-diff-less*)
note *helpers* = *length-L L′ index-minus-Suc-0-bound*
have *ladder-i-L′-index*:
ladder-i L′ index = *fst* (*drop* (*ladder-stepdown-diff L*) *D* ! *ladder-n L′* (*index* − *Suc 0*))
apply (*auto simp add*: *ladder-i-def*)
using *index-lower-bound* **apply** *arith*
apply (*simp add*: *ladder-stepdown-n*[*OF helpers*] *ladder-stepdown-j*[*OF helpers*])
apply (*subst drop-at-shift*)
using *LeftDerivationLadder-def Suc-index-upper-bound Suc-leI Suc-lessD is-ladder-def*
ladder-stepdown-diff-def ldl **apply** *presburger*
apply (*metis LeftDerivationLadder-def One-nat-def Suc-eq-plus1 Suc-index-upper-bound*

add.commute add-diff-cancel-right′ ladder-n-minus-1-bound ldl le-add1)
by (*metis LeftDerivationIntrosAt-def intros-at-Suc-index diff-Suc-1 ladder-i-def nat.simps*(*3*))
have *intro-at-index*:
LeftDerivationIntro (*ladder-*α (*ladder-stepdown-*α*-0* α *D L*) (*drop* (*ladder-stepdown-diff L*) *D*) *L′ index*)
(*ladder-i L′ index*) (*snd* (*drop* (*ladder-stepdown-diff L*) *D* ! *ladder-n L′* (*index* − *Suc 0*)))
(*ladder-ix L′ index*)
(*drop* (*Suc* (*ladder-n L′* (*index* − *Suc 0*)))
(*take* (*ladder-n L′ index*) (*drop* (*ladder-stepdown-diff L*) *D*)))
(*ladder-j L′ index*) (*ladder-*γ (*ladder-stepdown-*α*-0* α *D L*)
(*drop* (*ladder-stepdown-diff L*) *D*) *L′ index*)
proof −
have *arg1*: (*ladder-*α (*ladder-stepdown-*α*-0* α *D L*)
(*drop* (*ladder-stepdown-diff L*) *D*) *L′ index*) = *ladder-*α α *D L* (*Suc index*)
apply (*auto simp add*: *ladder-*α*-def ladder-*γ*-def*)
using *index-lower-bound* **apply** *arith*
apply (*simp add*: *ladder-stepdown-n*[*OF helpers*] *ladder-stepdown-*α*-0-def*)
apply (*subst Derive-Derive*[**where** γ=*ladder-*γ α *D L index*])
apply (*metis* (*no-types*, *lifting*) *Derivation-take-derive LeftDerivationLad-*

der-def
LeftDerivation-implies-Derivation Suc-index-upper-bound Suc-leI Suc-lessD

add.commute is-ladder-def ladder-γ-def ladder-stepdown-diff-def ldl
le-add-diff-inverse2 take-add)
by (*metis LeftDerivationLadder-def Suc-index-upper-bound Suc-leI Suc-lessD add.commute*
is-ladder-def ladder-stepdown-diff-def ldl le-add-diff-inverse2 take-add)
have *arg2*: *ladder-i L′ index* = *ladder-i L* (*Suc index*)
using *L′ index-lower-bound index-minus-Suc-0-bound ladder-i-def ladder-stepdown-j*
length-L **by** *auto*
obtain *n* **where** *n*: *n* = *ladder-n L* (*Suc index* − *1*) **by** *blast*
have *arg3*: (*snd* (*drop* (*ladder-stepdown-diff L*) *D* ! *ladder-n L′* (*index* − *Suc 0*))) =
snd (*D* ! *n*)
apply (*simp add*: *ladder-stepdown-n*[*OF helpers*] *index-lower-bound*)
apply (*subst drop-at-shift*)
using *index-lower-bound*
apply (*metis* (*no-types*, *opaque-lifting*) *L′ LeftDerivationLadder-def One-nat-def Suc-eq-plus1*
add.commute diff-Suc-1 index-upper-bound is-ladder-def ladder-stepdown-diff-def

ladder-stepdown-length ldl le-add-diff-inverse2 length-L less-or-eq-imp-le n
nat.simps(*3*) *neq0-conv not-less not-less-eq-eq*)
using *index-lower-bound*
apply (*metis LeftDerivationLadder-def One-nat-def Suc-index-upper-bound Suc-le-lessD*
Suc-pred diff-Suc-1 ladder-n-minus-1-bound ldl le-imp-less-Suc less-imp-le)

using *index-lower-bound n* **by** (*simp add*: *Suc-diff-le*)
have *arg4*: *ladder-ix L′ index* = *ladder-ix L* (*Suc index*)
using *ladder-stepdown-ix L′ Suc-le-lessD index-lower-bound index-upper-bound length-L*
by *auto*
obtain *m* **where** *m*: *m* = *ladder-n L* (*Suc index*) **by** *blast*
have *Suc-index-Suc*: *Suc* (*index* − *Suc 0*) = *index*
using *index-lower-bound* **by** *arith*
have *arg5*: (*drop* (*Suc* (*ladder-n L′* (*index* − *Suc 0*))) (*take* (*ladder-n L′ index*)
(*drop* (*ladder-stepdown-diff L*) *D*))) = *drop* (*Suc n*) (*take m D*)
apply (*simp add*: *ladder-stepdown-n*[*OF helpers*]
ladder-stepdown-n[*OF length-L L′ index-upper-bound*] *n m Suc-index-Suc*)
by (*metis* (*no-types*, *lifting*) *LeftDerivationLadder-def Suc-eq-plus1-left*
Suc-index-upper-bound Suc-leI Suc-le-lessD Suc-lessD drop-drop drop-take
index-lower-bound is-ladder-def ladder-stepdown-diff-def ldl le-add-diff-inverse2)
have *arg6*: *ladder-j L′ index* = *ladder-j L* (*Suc index*)
using *L′ index-upper-bound ladder-stepdown-j length-L* **by** *blast*
have *arg7*: (*ladder-γ* (*ladder-stepdown-α-0 α D L*)

(*drop* (*ladder-stepdown-diff L*) *D*) *L′ index*) = *ladder-γ α D L* (*Suc index*)
apply (*simp add*: *ladder-γ-def*)
apply (*simp add*: *ladder-stepdown-n*[*OF length-L L′ index-upper-bound*]
ladder-stepdown-α-0-def)
apply (*subst Derive-Derive*[**where** *γ*=*ladder-γ α D L* (*Suc index*)])
apply (*metis* (*no-types*, *lifting*) *L′ LeftDerivationLadder-def*
LeftDerivation-implies-Derivation LeftDerivation-take-derive Suc-le-lessD
add-diff-inverse-nat diff-is-0-eq index-lower-bound index-upper-bound
is-ladder-L′
is-ladder-def ladder-γ-def ladder-stepdown-n ldl le-0-eq length-L less-numeral-extra(*3*)

less-or-eq-imp-le take-add)
by (*metis* (*no-types*, *lifting*) *L′ One-nat-def add-diff-inverse-nat diff-is-0-eq*
index-lower-bound index-upper-bound is-ladder-L′ is-ladder-def ladder-stepdown-n
le-0-eq
le-neq-implies-less length-L less-numeral-extra(*3*) *less-or-eq-imp-le take-add*
zero-less-one)
from *ldintro arg1 arg2 arg3 arg4 arg5 arg6 arg7* **show** *?thesis*
by (*metis m n*)
qed
have *LeftDerivationIntrosAt* (*ladder-stepdown-α-0 α D L*) (*drop* (*ladder-stepdown-diff*
L) *D*)
L′ index
apply (*auto simp add*: *LeftDerivationIntrosAt-def Let-def*)
using *ladder-i-L′-index* **apply** *blast*
using *intro-at-index* **by** *blast*
}
note *introsAt* = *this*
show *?thesis*
apply (*rule-tac x*=*L′* **in** *exI*)
apply *auto*
defer *1*
using *L′ ladder-stepdown-length length-L* **apply** *auto*[*1*]
using *ladder-stepdown-i-0 length-L L′* **apply** *auto*[*1*]
using *ladder-stepdown-last-j L′ length-L* **apply** *auto*[*1*]
apply (*auto simp add*: *LeftDerivationLadder-def*)
using *ldl1* **apply** *blast*
using *is-ladder-L′* **apply** *blast*
using *ldfix* **apply** *blast*
apply (*auto simp add*: *LeftDerivationIntros-def*)
apply (*simp add*: *introsAt*)
done
qed

fun *ladder-shift-j* :: *nat* ⇒ *ladder* ⇒ *ladder* **where**
ladder-shift-j d [] = []
| *ladder-shift-j d* ((*n*, *j*, *i*)#*L*) = ((*n*, *j* − *d*, *i*)#(*ladder-shift-j d L*))

definition *ladder-cut-prefix* :: *nat* ⇒ *ladder* ⇒ *ladder*

where
ladder-cut-prefix d L =
(*ladder-shift-j d L*)[*0* := (*ladder-n L 0*, *ladder-j L 0* − *d*, *ladder-i L 0* − *d*)]

lemma *ladder-shift-j-length*:
length (*ladder-shift-j d L*) = *length L*
by (*induct L*, *auto*)

lemma *ladder-cut-prefix-length*:
shows *length* (*ladder-cut-prefix d L*) = *length L*
apply (*simp add*: *ladder-cut-prefix-def*)
apply (*simp add*: *ladder-shift-j-length*)
done

lemma *ladder-shift-j-cons*: *ladder-shift-j d* (*x*#*L*) = (*fst x*, *fst* (*snd x*) − *d*, *snd*(*snd x*))#
(*ladder-shift-j d L*)
apply (*induct L*)
by (*cases x*, *simp*)+

lemma *deriv-j-ladder-shift-j*:
index < *length L* ⟹ *deriv-j* (*ladder-shift-j d L* ! *index*) = *deriv-j* (*L* ! *index*) − *d*
proof (*induct L arbitrary*: *index*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x L*)
show *?case*
apply (*subst ladder-shift-j-cons*)
apply (*cases index*)
using *Cons* **by** (*auto simp add*: *deriv-j-def*)
qed

lemma *deriv-n-ladder-shift-j*:
index < *length L* ⟹ *deriv-n* (*ladder-shift-j d L* ! *index*) = *deriv-n* (*L* ! *index*)
proof (*induct L arbitrary*: *index*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x L*)
show *?case*
apply (*subst ladder-shift-j-cons*)
apply (*cases index*)
using *Cons* **by** (*auto simp add*: *deriv-n-def*)
qed

lemma *deriv-ix-ladder-shift-j*:
index < *length L* ⟹ *deriv-ix* (*ladder-shift-j d L* ! *index*) = *deriv-ix* (*L* ! *index*)

proof (*induct L arbitrary*: *index*)
  case *Nil*
    then show *?case* **by** *auto*
next
  case (*Cons x L*)
    show *?case*
      apply (*subst ladder-shift-j-cons*)
      apply (*cases index*)
      using *Cons* **by** (*auto simp add*: *deriv-ix-def*)
qed

lemma *ladder-cut-prefix-j*:
  assumes *index-bound*: *index* < *length L*
  assumes *length-L*: *length L* > *0*
  shows *ladder-j* (*ladder-cut-prefix d L*) *index* = *ladder-j L index* − *d*
  apply (*simp add*: *ladder-j-def ladder-cut-prefix-def*)
  apply (*cases index*)
  apply (*auto simp add*: *length-L*)
  apply (*subst nth-list-update-eq*)
  apply (*simp only*: *ladder-shift-j-length length-L*)
  apply (*simp add*: *deriv-j-def*)
  apply (*subst deriv-j-ladder-shift-j*)
  using *index-bound* **apply** *arith*
  by *blast*

lemma *hd-0-subst*: *length L* > *0* ⟹ *hd* (*L* [*0* := *x*]) = *x*
  using *hd-conv-nth* **by** (*simp add*: *upd-conv-take-nth-drop*)

lemma *ladder-cut-prefix-i*:
  assumes *index-bound*: *index* < *length L*
  assumes *length-L*: *length L* > *0*
  shows *ladder-i* (*ladder-cut-prefix d L*) *index* = *ladder-i L index* − *d*
  apply (*simp add*: *ladder-i-def ladder-cut-prefix-def*)
  apply (*cases index*)
  apply *auto*[*1*]
  apply (*subst hd-0-subst*)
  using *length-L ladder-shift-j-length* **apply** *metis*
  apply (*auto simp add*: *deriv-i-def*)
  apply (*case-tac nat*)
  apply (*simp add*: *ladder-j-def deriv-j-def*)
  apply *auto*
  apply (*subst nth-list-update-eq*)
  using *length-L ladder-shift-j-length* **apply** *auto*[*1*]
  apply *simp*
  apply (*simp add*: *ladder-j-def*)
  apply (*subst deriv-j-ladder-shift-j*)
  using *index-bound* **apply** *arith*
  apply *simp*
  done

```
lemma ladder-cut-prefix-n:
  assumes index-bound: index < length L
  assumes length-L: length L > 0
  shows ladder-n (ladder-cut-prefix d L) index = ladder-n L index
  apply (simp add: ladder-cut-prefix-def)
  apply (cases index)
  apply (auto simp add: ladder-n-def)
  apply (subst nth-list-update-eq)
  apply (simp add: ladder-shift-j-length)
  using length-L apply blast
  apply (simp add: deriv-n-def )
  apply (rule-tac deriv-n-ladder-shift-j)
  using index-bound by arith

lemma ladder-cut-prefix-ix:
  assumes index-bound: index < length L
  assumes length-L: length L > 0
  shows ladder-ix (ladder-cut-prefix d L) index = ladder-ix L index
  apply (simp add: ladder-cut-prefix-def)
  apply (cases index)
  apply (auto simp add: ladder-ix-def)
  apply (rule-tac deriv-ix-ladder-shift-j)
  using index-bound by arith

lemma LeftDerivationFix-derivation-ge-is-nonterminal:
  assumes ldfix: LeftDerivationFix α i D j γ
  assumes derivation-ge-d: derivation-ge D d
  assumes is-nonterminal: is-nonterminal (γ ! j)
  shows (D = [] ∧ α = γ ∧ i = j) ∨ (i > d ∧ j ≥ d)
proof −
  have is-nonterminal (α ! i) using ldfix is-nonterminal
    by (simp add: LeftDerivationFix-def)
  from LeftDerivationFix-splits-at-nonterminal[OF ldfix this] obtain U a1 a2 b1
where U:
    splits-at α i a1 U a2 ∧ splits-at γ j b1 U a2 ∧ LeftDerivation a1 D b1 by blast
  have D = [] ∨ D ≠ [] by auto
  then show ?thesis
  proof (induct rule: disjCases2)
    case 1
      then have a1 = b1 using U by auto
      then have i-eq-j: i = j using U
      by (metis dual-order.strict-implies-order length-take min.absorb2 splits-at-def)

      from 1 have α = γ using ldfix LeftDerivationFix-def by auto
      with 1 i-eq-j show ?case by blast
  next
    case 2
      have ∃ a1'. LeftDerives1 a1 (fst (hd D)) (snd (hd D)) a1' using U 2
```

by (*metis LeftDerivation.elims*(*2*) *list.sel*(*1*))
then obtain *a1′* **where** *a1′*: *LeftDerives1 a1* (*fst* (*hd D*)) (*snd* (*hd D*)) *a1′*
by *blast*
then have (*fst* (*hd D*)) < *length a1* **using** *Derives1-bound LeftDerives1-implies-Derives1*
by *blast*
then have *fst-less-i*: (*fst* (*hd D*)) < *i* **using** *U*
by (*simp add*: *leD min.absorb2 nat-le-linear splits-at-def*)
have *d-le-fst*: *d* ≤ *fst* (*hd D*) **using** *derivation-ge-d 2* **by** (*simp add*: *derivation-ge-def*)
with *fst-less-i* **have** *d-less-i*: *d* < *i* **using** *le-less-trans* **by** *blast*
have ∃ *b1′*. *LeftDerives1 b1′* (*fst* (*last D*)) (*snd* (*last D*)) *b1* **using** *U 2*
by (*metis Derive LeftDerivation-Derive-take-LeftDerives1 LeftDerivation-implies-Derivation*

last-conv-nth length-0-conv order-refl take-all)
then obtain *b1′* **where** *b1′*: *LeftDerives1 b1′* (*fst* (*last D*)) (*snd* (*last D*)) *b1*
by *blast*
then have *fst* (*last D*) ≤ *length b1*
using *Derives1-bound′ LeftDerives1-implies-Derives1* **by** *blast*
then have *fst-le-j*: *fst* (*last D*) ≤ *j* **using** *U splits-at-def* **by** *auto*
have *d* ≤ *fst* (*last D*) **using** *derivation-ge-d 2* **using** *derivation-ge-def last-in-set* **by** *blast*
with *fst-le-j* **have** *d* ≤ *j* **using** *order.trans* **by** *blast*
with *d-less-i* **show** *?thesis* **by** *auto*
qed
qed

lemma *LeftDerivationFix-derivation-ge*:
assumes *ldfix*: *LeftDerivationFix* α *i D j* γ
assumes *derivation-ge-d*: *derivation-ge D d*
shows *i* = *j* ∨ (*i* > *d* ∧ *j* ≥ *d*)
proof −
from *LeftDerivationFix-splits-at-symbol*[*OF ldfix*] **obtain** *U a1 a2 b1 b2 n* **where** *U*:
splits-at α *i a1 U a2* ∧
splits-at γ *j b1 U b2* ∧
n ≤ *length D* ∧
LeftDerivation a1 (*take n D*) *b1* ∧
derivation-ge (*drop n D*) (*Suc* (*length b1*)) ∧
LeftDerivation a2 (*derivation-shift* (*drop n D*) (*Suc* (*length b1*)) *0*) *b2* ∧
(*n* = *length D* ∨ *n* < *length D* ∧ *is-word* (*b1* @ [*U*])) **by** *blast*
have *n* = *0* ∨ *n* > *0* **by** *auto*
then show *?thesis*
proof (*induct rule*: *disjCases2*)
case *1*
then have *a1* = *b1* **using** *U* **by** *auto*
then have *i-eq-j*: *i* = *j* **using** *U*
by (*metis dual-order.strict-implies-order length-take min.absorb2 splits-at-def*)

then show *?case* **by** *blast*

```
  next
    case 2
      obtain E where E: E = take n D by blast
      have E-nonempty: E ≠ [] using E 2
        using U less-nat-zero-code list.size(3) take-eq-Nil by auto
        have ∃ a1'. LeftDerives1 a1 (fst (hd E)) (snd (hd E)) a1' using U E
E-nonempty
        by (metis LeftDerivation.simps(2) list.exhaust list.sel(1))
      then obtain a1' where a1': LeftDerives1 a1 (fst (hd E)) (snd (hd E)) a1'
by blast
    then have (fst (hd E)) < length a1 using Derives1-bound LeftDerives1-implies-Derives1
by blast
      then have fst-less-i: (fst (hd E)) < i using U
        by (simp add: leD min.absorb2 nat-le-linear splits-at-def)
      have d-le-fst: d ≤ fst (hd E) using derivation-ge-d E-nonempty E
        by (simp add: LeftDerivation.elims(2) U derivation-ge-def hd-conv-nth)
      with fst-less-i have d-less-i: d < i using le-less-trans by blast
    have ∃ b1'. LeftDerives1 b1' (fst (last E)) (snd (last E)) b1 using E-nonempty
U E
          by (metis LeftDerivation-append1 append-butlast-last-id prod.collapse)

     then obtain b1' where b1': LeftDerives1 b1' (fst (last E)) (snd (last E)) b1
by blast
      then have fst (last E) ≤ length b1
        using Derives1-bound' LeftDerives1-implies-Derives1 by blast
      then have fst-le-j: fst (last E) ≤ j using U splits-at-def by auto
      have d ≤ fst (last E) using derivation-ge-d E-nonempty E
        using derivation-ge-d last-in-set by (metis derivation-ge-def set-take-subset
subsetCE)
      with fst-le-j have d ≤ j using order.trans by blast
      with d-less-i show ?thesis by auto
  qed
qed

lemma LeftDerivationIntro-derivation-ge:
  assumes ldintro: LeftDerivationIntro α i r ix D j γ
  assumes i-ge-d: i ≥ d
  assumes derivation-ge-d: derivation-ge D d
  shows j ≥ d
proof −
  from iffD1[OF LeftDerivationIntro-def ldintro] obtain β where β:
    LeftDerives1 α i r β ∧ ix < length (snd r) ∧ snd r ! ix = γ ! j ∧
     LeftDerivationFix β (i + ix) D j γ by blast
  then have (i + ix = j) ∨ (i + ix > d ∧ j ≥ d)
    using LeftDerivationFix-derivation-ge derivation-ge-d by blast
  then show ?thesis
  proof (induct rule: disjCases2)
    case 1 then show ?case using i-ge-d trans-le-add1 by blast
  next
```

```
    case 2 then show ?case by simp
  qed
qed

lemma derivation-ge-LeftDerivationLadder:
  assumes derivation-ge-d: derivation-ge D d
  assumes ladder: LeftDerivationLadder α D L γ
  assumes ladder-i-0: ladder-i L 0 ≥ d
  shows index < length L ⟹ ladder-i L index ≥ d ∧ ladder-j L index ≥ d
proof (induct index)
  case 0
    from iffD1[OF LeftDerivationLadder-def ladder]
    have ldfix: LeftDerivationFix α (ladder-i L 0)
      (take (ladder-n L 0) D) (ladder-j L 0) (ladder-γ α D L 0) by blast
    have derivation-ge (take (ladder-n L 0) D) d
      using derivation-ge-d by (metis append-take-drop-id derivation-ge-append)
    from ladder-i-0 derivation-ge-d LeftDerivationFix-derivation-ge[OF ldfix this]
    show ?case by linarith
next
  case (Suc n)
    have ladder-i-Suc: ladder-i L (Suc n) ≥ d
      apply (auto simp add: ladder-i-def)
      using Suc by auto
     from iffD1[OF LeftDerivationLadder-def ladder] have LeftDerivationIntros α
D L
      by blast
    then have LeftDerivationIntrosAt α D L (Suc n)
      using Suc.prems
      by (metis LeftDerivationIntros-def Suc-eq-plus1-left le-add1)
    from iffD1[OF LeftDerivationIntrosAt-def this]
    show ?case using ladder-i-Suc LeftDerivationIntro-derivation-ge
      by (metis append-take-drop-id derivation-ge-append derivation-ge-d)
qed

lemma derivation-shift-append:
  derivation-shift (A@B) left right =
    (derivation-shift A left right) @ (derivation-shift B left right)
by (induct A, simp+)

lemma derivation-shift-right-left-subtract:
  right ≥ left ⟹ derivation-shift (derivation-shift L 0 right) left 0 =
  derivation-shift L 0 (right − left)
by (induct L, simp+)

lemma LeftDerivationFix-cut-prefix:
  assumes LeftDerivationFix (δ@α) i D j γ
  assumes derivation-ge D (length δ)
  assumes i ≥ length δ
  assumes is-word-δ: is-word δ
```

shows ∃ γ'. $\gamma = \delta$ @ γ' ∧
LeftDerivationFix α (i − *length* δ) (*derivation-shift D* (*length* δ) *0*) (j − *length* δ) γ'
proof −
have *j-ge-d*: $j \geq$ *length* δ
using *assms*(*3*) *LeftDerivationFix-derivation-ge*[*OF assms*(*1*) *assms*(*2*)] **by** *arith*
obtain γ' **where** γ': γ' = *drop* (*length* δ) γ **by** *blast*
from *iffD1*[*OF LeftDerivationFix-def assms*(*1*)] **obtain** *E F* **where** *EF*:
is-sentence (δ @ α) ∧
is-sentence γ ∧
LeftDerivation (δ @ α) *D* γ ∧
$i <$ *length* (δ @ α) ∧
$j <$ *length* γ ∧
(δ @ α) ! i = γ ! j ∧
$D = E$ @ *derivation-shift F 0* (*Suc j*) ∧
LeftDerivation (*take i* (δ @ α)) *E* (*take j* γ) ∧
LeftDerivation (*drop* (*Suc i*) (δ @ α)) *F* (*drop* (*Suc j*) γ) **by** *blast*
then have *LeftDerivation* (δ @ α) *D* γ **by** *blast*
from *LeftDerivation-skip-prefixword-ex*[*OF this is-word-*δ]
obtain γ' **where** γ': $\gamma = \delta$ @ γ' ∧ *LeftDerivation* α (*derivation-shift D* (*length* δ) *0*) γ' **by** *blast*
have *ldf1*: *is-sentence* α **using** *EF is-sentence-concat* **by** *blast*
have *ldf2*: *is-sentence* γ' **using** *EF* γ' *is-sentence-concat* **by** *blast*
have *ldf3*: i − *length* $\delta <$ *length* α
by (*metis EF append-Nil assms*(*3*) *drop-append drop-eq-Nil not-le*)
have *ldf4*: j − *length* $\delta <$ *length* γ'
by (*metis EF append-Nil j-ge-d* γ' *drop-append drop-eq-Nil not-le*)
have *ldf5*: α ! (i − *length* δ) = γ' ! (j − *length* δ)
by (*metis* γ' *EF assms*(*3*) *j-ge-d leD nth-append*)
have *D-split*: $D = E$ @ *derivation-shift F 0* (*Suc j*) **using** *EF* **by** *blast*
show *?thesis*
apply (*rule-tac x*=γ' **in** *exI*)
apply (*auto simp add*: γ')
apply (*auto simp add*: *LeftDerivationFix-def*)
using *ldf1* **apply** *blast*
using *ldf2* **apply** *blast*
using γ' **apply** *blast*
using *ldf3* **apply** *blast*
using *ldf4* **apply** *blast*
using *ldf5* **apply** *blast*
apply (*rule-tac x*=*derivation-shift E* (*length* δ) *0* **in** *exI*)
apply (*rule-tac x*=*F* **in** *exI*)
apply *auto*
apply (*subst D-split*)
apply (*simp add*: *derivation-shift-append*)
apply (*subst derivation-shift-right-left-subtract*)
apply (*simp add*: *j-ge-d le-Suc-eq*)
using *j-ge-d* **apply** (*simp add*: *Suc-diff-le*)

apply (*metis EF LeftDerivation-implies-Derivation LeftDerivation-skip-prefix* γ'
append-eq-conv-conj assms(*3*) *drop-take is-word-Derivation-derivation-ge is-word-*δ

take-all take-append)
using *EF Suc-diff-le* γ' *assms*(*3*) *j-ge-d* **by** *auto*
qed

lemma *LeftDerives1-propagate-prefix*:
LeftDerives1 (δ @ α) *i r* $\beta \Longrightarrow i \geq$ *length* $\delta \Longrightarrow$ *is-prefix* δ β
proof −
assume *a1*: *LeftDerives1* (δ @ α) *i r* β
assume *a2*: *length* $\delta \leq i$
have *f3*: *take i* (δ @ α) = *take i* β
using *a1 Derives1-take LeftDerives1-implies-Derives1* **by** *blast*
then have *f4*: *length* (*take i* β) = *i*
using *a1* **by** (*metis* (*no-types*) *Derives1-bound LeftDerives1-implies-Derives1 dual-order.strict-implies-order length-take min.absorb2*)
have *take* (*length* δ) (*take i* β) = δ
using *f3 a2* **by** (*simp add*: *append-eq-conv-conj*)
then show *?thesis*
using *f4 a2* **by** (*metis* (*no-types*) *append-Nil2 append-eq-conv-conj diff-is-0-eq*$'$ *is-prefix-take take-0 take-append*)
qed

lemma *LeftDerivationIntro-cut-prefix*:
assumes *LeftDerivationIntro* (δ@α) *i r ix D j* γ
assumes *derivation-ge D* (*length* δ)
assumes $i \geq$ *length* δ
assumes *is-word-*δ: *is-word* δ
shows $\exists\ \gamma'.\ \gamma = \delta$ @ $\gamma' \wedge$
LeftDerivationIntro α (*i* − *length* δ) *r ix* (*derivation-shift D* (*length* δ) *0*) (*j* − *length* δ) γ'
proof −
from *iffD1*[*OF LeftDerivationIntro-def assms*(*1*)] **obtain** β **where** β:
LeftDerives1 (δ @ α) *i r* $\beta \wedge$
ix < *length* (*snd r*) $\wedge$ *snd r* ! *ix* = γ ! *j* $\wedge$ *LeftDerivationFix* β (*i* + *ix*) *D j* γ **by** *blast*
have $\exists\ \beta'.\ \beta = \delta$ @ β'
using *LeftDerives1-propagate-prefix* β *assms*(*3*) **by** (*metis append-dropped-prefix*)

then obtain β' **where** β': $\beta = \delta$ @ β' **by** *blast*
with β **have** *LeftDerives1* (δ @ α) *i r* (δ @ β') **by** *simp*
from *LeftDerives1-skip-prefix*[*OF assms*(*3*) *this*]
have α-β': *LeftDerives1* α (*i* − *length* δ) *r* β' **by** *blast*
have *ldfix*: *LeftDerivationFix* (δ @ β') (*i* + *ix*) *D j* γ **using** β β' **by** *auto*
have δ*-le-i-plus-ix*: *length* $\delta \leq i$ + *ix* **using** *assms*(*3*) **by** *arith*
from *LeftDerivationFix-cut-prefix*[*OF ldfix assms*(*2*) δ*-le-i-plus-ix assms*(*4*)]
obtain γ' **where** γ': $\gamma = \delta$ @ $\gamma' \wedge$

LeftDerivationFix β' (i + ix − *length* δ) (*derivation-shift* D (*length* δ) *0*) (j − *length* δ) γ'
by *blast*
have *same-symbol*: γ ! j = γ' ! (j − *length* δ)
by (*metis LeftDerivationFix-def* β β' δ*-le-i-plus-ix* γ' *leD nth-append*)
have β'-γ': *LeftDerivationFix* β' (i − *length* δ + ix)
(*derivation-shift* D (*length* δ) *0*) (j − *length* δ) γ' **by** (*simp add*: γ' *assms*(*3*))

show *?thesis*
apply (*simp add*: *LeftDerivationIntro-def*)
apply (*rule-tac x*=γ' **in** *exI*)
apply (*auto simp add*: γ')
apply (*rule-tac x*=β' **in** *exI*)
by (*auto simp add*: β α-β' *same-symbol* β'-γ')
qed

lemma *LeftDerivationLadder-implies-LeftDerivation-at-index*:
assumes *LeftDerivationLadder* α D L γ
assumes *index* < *length* L
shows *LeftDerivation* α (*take* (*ladder-n* L *index*) D) (*ladder-*γ α D L *index*)
using *LeftDerivationLadder-def LeftDerivation-take-derive assms*(*1*) *ladder-*γ*-def*
by *auto*

lemma *LeftDerivationLadder-cut-prefix-propagate*:
assumes *ladder*: *LeftDerivationLadder* (δ@α) D L γ
assumes *is-word-*δ: *is-word* δ
assumes *derivation-ge-*δ: *derivation-ge* D (*length* δ)
assumes *ladder-i-0*: *ladder-i* L *0* $\geq$ *length* δ
assumes L': L' = *ladder-cut-prefix* (*length* δ) L
assumes D': D' = *derivation-shift* D (*length* δ) *0*
shows *index* < *length* L $\Longrightarrow$
LeftDerivation α (*take* (*ladder-n* L' *index*) D') (*ladder-*γ α D' L' *index*) $\wedge$
ladder-α (δ@α) D L *index* = δ@(*ladder-*α α D' L' *index*) $\wedge$
ladder-γ (δ@α) D L *index* = δ@(*ladder-*γ α D' L' *index*)
proof (*induct index*)
case *0*
have *ladder-*α: *ladder-*α (δ@α) D L *0* = δ@(*ladder-*α α D' L' *0*)
by (*simp add*: *ladder-*α*-def*)
have *ldfix*: *LeftDerivationFix* (δ@α) (*ladder-i* L *0*) (*take* (*ladder-n* L *0*) D)
(*ladder-j* L *0*) (*ladder-*γ (δ@α) D L *0*) **using** *ladder LeftDerivationLadder-def*
by *blast*
have *dge-take*: *derivation-ge* (*take* (*ladder-n* L *0*) D) (*length* δ)
using *derivation-ge-*δ **by** (*metis append-take-drop-id derivation-ge-append*)
from *LeftDerivationFix-cut-prefix*[*OF ldfix dge-take ladder-i-0 is-word-*δ]
obtain γ' **where** γ': *ladder-*γ (δ @ α) D L *0* = δ @ γ' $\wedge$
LeftDerivationFix α (*ladder-i* L *0* − *length* δ) (*derivation-shift* (*take* (*ladder-n* L *0*) D) (*length* δ) *0*)
(*ladder-j* L *0* − *length* δ) γ' **by** *blast*
have *ladder-*γ: *ladder-*γ (δ@α) D L *0* = δ@(*ladder-*γ α D' L' *0*)

using γ' **by** (*metis 0.prems* D' *Derive* L' *LeftDerivationFix-def*
LeftDerivation-implies-Derivation ladder-γ*-def ladder-cut-prefix-n take-derivation-shift*)
have *LeftDerivation* α (*take* (*ladder-n* L' *0*) D') (*ladder-*γ α D' L' *0*)
proof −
have *LeftDerivation* ($\delta @ \alpha$) (*take* (*ladder-n L 0*) *D*) (*ladder-*γ ($\delta @ \alpha$) *D L 0*)
using *LeftDerivationLadder-implies-LeftDerivation-at-index ladder 0.prems*
by *blast*
then show *?thesis*
by (*metis* D' *LeftDerivationLadder-def LeftDerivation-skip-prefix*
LeftDerivation-take-derive derivation-ge-δ *ladder ladder-*γ*-def*)
qed
then show *?case* **using** *ladder-*α *ladder-*γ **by** *auto*
next
case (*Suc index*)
have *index-less-L*: *index* < *length L* **using** *Suc*(*2*) **by** *arith*
then have *induct*: *ladder-*γ ($\delta @ \alpha$) *D L index* = $\delta @$(*ladder-*γ α D' L' *index*)
using *Suc* **by** *blast*
then have *ladder-*α: *ladder-*α ($\delta @ \alpha$) *D L* (*Suc index*) = $\delta @$(*ladder-*α α D' L'
(*Suc index*))
by (*simp add*: *ladder-*α*-def*)
have *introsAt*: *LeftDerivationIntrosAt* ($\delta @ \alpha$) *D L* (*Suc index*)
using *Suc*(*2*) *ladder*
by (*metis LeftDerivationIntros-def LeftDerivationLadder-def Suc-eq-plus1-left
le-add1*)
obtain *n m e E* **where** *n*: *n* = *ladder-n L* (*Suc index* − *1*) **and**
m: *m* = *ladder-n L* (*Suc index*) **and** *e*: *e* = *D* ! *n* **and** *E*: *E* = *drop* (*Suc n*)
(*take m D*)
by *blast*
from *iffD1*[*OF LeftDerivationIntrosAt-def introsAt*] **have**
LeftDerivationIntro (*ladder-*α (δ @ α) *D L* (*Suc index*)) (*ladder-i L* (*Suc
index*)) (*snd e*)
(*ladder-ix L* (*Suc index*)) *E* (*ladder-j L* (*Suc index*)) (*ladder-*γ (δ @ α) *D L*
(*Suc index*))
using *n m e E Let-def* **by** *meson*
then have *ldintro*:
LeftDerivationIntro ($\delta @$(*ladder-*α α D' L' (*Suc index*))) (*ladder-i L* (*Suc
index*)) (*snd e*)
(*ladder-ix L* (*Suc index*)) *E* (*ladder-j L* (*Suc index*)) (*ladder-*γ (δ @ α) *D L*
(*Suc index*))
by (*simp add*: *ladder-*α)
have *dge-E-*δ: *derivation-ge E* (*length* δ)
apply (*simp add*: *E*)
using *derivation-ge-*δ
by (*metis append-take-drop-id derivation-ge-append*)
have *ladder-i-Suc*: *length* $\delta \leq$ *ladder-i L* (*Suc index*)
using *Suc.prems derivation-ge-LeftDerivationLadder derivation-ge-*δ *ladder
ladder-i-0*
by *blast*
from *LeftDerivationIntro-cut-prefix*[*OF ldintro dge-E-*δ *ladder-i-Suc is-word-*δ]

obtain γ' **where** γ': *ladder-*γ (δ @ α) *D L* (*Suc index*) = δ @ γ' $\wedge$
LeftDerivationIntro (*ladder-*α α D' L' (*Suc index*)) (*ladder-i L* (*Suc index*) − *length* δ) (*snd e*)
(*ladder-ix L* (*Suc index*)) (*derivation-shift E* (*length* δ) *0*) (*ladder-j L* (*Suc index*) − *length* δ) γ'
by *blast*
then have *LeftDerivation* (*ladder-*α α D' L' (*Suc index*))
((*ladder-i L* (*Suc index*) − *length* δ, *snd e*) # (*derivation-shift E* (*length* δ) *0*)) γ'
using *LeftDerivationIntro-implies-LeftDerivation* **by** *blast*
then have *LeftDerivation* (*ladder-*γ α D' L' *index*)
((*ladder-i L* (*Suc index*) − *length* δ, *snd e*) # (*derivation-shift E* (*length* δ) *0*)) γ'
by (*auto simp add*: *ladder-*α*-def*)
have *ld*: *LeftDerivation* α (*take* (*ladder-n* L' (*Suc index*)) D') (*ladder-*γ α D' L' (*Suc index*))
proof −
have *LeftDerivation* (δ@α) (*take* (*ladder-n L* (*Suc index*)) *D*) (*ladder-*γ (δ@α) *D L* (*Suc index*))
using *LeftDerivationLadder-implies-LeftDerivation-at-index ladder Suc.prems* **by** *blast*
then show *?thesis*
by (*metis* D' *LeftDerivationLadder-def LeftDerivation-skip-prefix*
LeftDerivation-take-derive derivation-ge-δ *ladder ladder-*γ*-def*)
qed
then show *?case*
using γ' D' *Derive* L' *LeftDerivationIntro-def n m e E ld*
LeftDerivation-implies-Derivation ladder-γ*-def ladder-cut-prefix-n take-derivation-shift*
by (*metis* (*no-types*, *lifting*) *LeftDerivationLadder-implies-LeftDerivation-at-index*

LeftDerivation-skip-prefixword-ex Suc.prems Suc-leI index-less-L is-word-δ *ladder*
ladder-α *le-0-eq neq0-conv*)
qed

theorem *LeftDerivationLadder-cut-prefix*:
assumes *ladder*: *LeftDerivationLadder* (δ@α) *D L* γ
assumes *is-word-*δ: *is-word* δ
assumes *ladder-i-0*: *ladder-i L 0* $\geq$ *length* δ
shows $\exists$ D' L' γ'. γ = δ @ γ' $\wedge$
LeftDerivationLadder α D' L' γ' $\wedge$
D' = *derivation-shift D* (*length* δ) *0* $\wedge$
length L' = *length L* $\wedge$ *ladder-i* L' *0* + *length* δ = *ladder-i L 0* $\wedge$
ladder-last-j L' + *length* δ = *ladder-last-j L*
proof −
obtain D' **where** D': D' = *derivation-shift D* (*length* δ) *0* **by** *blast*
obtain L' **where** L': L' = *ladder-cut-prefix* (*length* δ) *L* **by** *blast*
obtain γ' **where** γ': γ' = *drop* (*length* δ) γ **by** *blast*
have *ladder-last-j-upper-bound*: *ladder-last-j L* < *length* γ **using** *ladder*

using *ladder-last-j-bound* **by** *blast*
have *derivation-ge-*δ: *derivation-ge D* (*length* δ) **using** *is-word-*δ *LeftDerivation-Ladder-def*
LeftDerivation-implies-Derivation is-word-Derivation-derivation-ge ladder **by** *blast*
note *derivation-ge-ladder* =
derivation-ge-LeftDerivationLadder[*OF derivation-ge-*δ *ladder ladder-i-0*]
have *ladder-last-j-lower-bound*: *ladder-last-j L* $\geq$ *length* δ
using *LeftDerivationLadder-def derivation-ge-ladder is-ladder-def ladder*
ladder-last-j-def **by** *auto*
from *ladder-last-j-upper-bound ladder-last-j-lower-bound*
have δ*-less-*γ: *length* δ < *length* γ **by** *arith*
then have γ*-def*: $\gamma = \delta$ @ γ'
by (*metis LeftDerivation.simps*(*1*) *LeftDerivationLadder-def LeftDerivation-ge-take* γ'
append-eq-conv-conj derivation-ge-δ *ladder*)
have *length-L-nonzero*: *length L* $\neq$ *0*
using *LeftDerivationLadder-def is-ladder-def ladder* **by** *auto*
have *ladder-i-L'-thm*: $\bigwedge$ *index. index* < *length L* $\Longrightarrow$ *ladder-i L' index* + *length* δ = *ladder-i L index*
apply (*simp add*: *L'*)
apply (*subst ladder-cut-prefix-i*)
apply *simp*
using *length-L-nonzero* **apply** *blast*
using *derivation-ge-ladder* **by** *auto*
have *ladder-j-L'-thm*: $\bigwedge$ *index. index* < *length L* $\Longrightarrow$ *ladder-j L' index* + *length* δ = *ladder-j L index*
apply (*simp add*: *L'*)
apply (*subst ladder-cut-prefix-j*)
using *LeftDerivationLadder-def is-ladder-def ladder* **apply** *blast*
using *LeftDerivationLadder-def is-ladder-def ladder* **apply** *blast*
using *derivation-ge-ladder* **by** *auto*
have *length-L'*: *length L'* = *length L* **using** *L' ladder-cut-prefix-length* **by** *simp*
have α*-*γ': *LeftDerivation* α *D'* γ'
using *D' LeftDerivationLadder-def LeftDerivation-skip-prefix* γ' *derivation-ge-*δ *ladder*
by *blast*
have *length-D'*: *length D'* = *length D* **by** (*simp add*: *D'*)
have *is-ladder-D-L*: *is-ladder D L* **using** *LeftDerivationLadder-def ladder* **by** *blast*
{
fix *u* :: *nat*
assume *u-bound-L'*: *u* < *length L'*
have *u-bound-L*: *u* < *length L* **using** *length-L'* **using** *u-bound-L'* **by** *simp*
have *ladder-n L' u* $\leq$ *length D'*
apply (*simp add*: *length-D' L'*)
apply (*subst ladder-cut-prefix-n*)
apply (*simp add*: *u-bound-L*)
using *length-L-nonzero* **apply** *arith*

```
    using u-bound-L is-ladder-D-L
    by (simp add: is-ladder-def)
  }
  note is-ladder-1 = this
  {
   fix u :: nat
   fix v :: nat
   assume u-less-v: u < v
   assume v-bound-L': v < length L'
   then have v-bound-L: v < length L by (simp add: length-L')
   with u-less-v have u-bound-L: u < length L by arith
   have ladder-n L' u < ladder-n L' v
    apply (simp add: L')
    apply (subst ladder-cut-prefix-n)
    using u-bound-L apply blast
    using length-L-nonzero apply blast
    apply (subst ladder-cut-prefix-n)
    using v-bound-L apply blast
    using length-L-nonzero apply blast
    using u-less-v v-bound-L is-ladder-D-L by (simp add: is-ladder-def)
  }
  note is-ladder-2 = this
  have is-ladder-3: ladder-last-n L' = length D'
   apply (simp add: length-D' ladder-last-n-def L')
   apply (subst ladder-cut-prefix-n)
   apply (simp add: ladder-cut-prefix-length)
   using length-L-nonzero apply auto[1]
   using length-L-nonzero apply blast
   apply (simp add: ladder-cut-prefix-length)
   using is-ladder-D-L by (simp add: is-ladder-def ladder-last-n-def)
  have is-ladder-4: LeftDerivationFix α (ladder-i L' 0) (take (ladder-n L' 0) D')
   (ladder-j L' 0) (ladder-γ α D' L' 0)
  proof -
   have ldfix: LeftDerivationFix (δ@α) (ladder-i L 0) (take (ladder-n L 0) D)
   (ladder-j L 0) (ladder-γ (δ@α) D L 0)
   using ladder LeftDerivationLadder-def by blast
   have dge: derivation-ge (take (ladder-n L 0) D) (length δ)
    using derivation-ge-δ by (metis append-take-drop-id derivation-ge-append)
   from LeftDerivationFix-cut-prefix[OF ldfix dge ladder-i-0 is-word-δ]
   obtain γ' where γ': ladder-γ (δ @ α) D L 0 = δ @ γ' ∧
     LeftDerivationFix α (ladder-i L 0 − length δ) (derivation-shift (take (ladder-n
L 0) D) (length δ) 0)
     (ladder-j L 0 − length δ) γ' by blast
   then show ?thesis
     using LeftDerivationLadder-cut-prefix-propagate D' L' append-eq-conv-conj
derivation-ge-δ
          is-word-δ ladder ladder-cut-prefix-i ladder-cut-prefix-j ladder-cut-prefix-n
ladder-i-0
       length-0-conv length-L-nonzero length-greater-0-conv take-derivation-shift by
```

auto
qed
{
fix *index* :: *nat*
assume *index-lower-bound*: *Suc 0* ≤ *index*
assume *index-upper-bound*: *index* < *length L′*
have *introsAt*: *LeftDerivationIntrosAt* (*δ*@*α*) *D L index*
by (*metis LeftDerivationIntros-def LeftDerivationLadder-def One-nat-def index-lower-bound*
index-upper-bound ladder length-L′)
then have *ladder-i-L*: *ladder-i L index* = *fst* (*D* ! *ladder-n L* (*index* − *Suc 0*))
by (*metis LeftDerivationIntrosAt-def One-nat-def* ‹*LeftDerivationIntrosAt* (*δ* @ *α*) *D L index*›)
have *ladder-i-L′-as-L*: *ladder-i L′ index* = *ladder-i L index* − (*length δ*)
using *ladder-cut-prefix-i L′ index-upper-bound is-ladder-D-L is-ladder-not-empty length-L′*
length-greater-0-conv **by** *auto*
have *length-L-gr-0*: *length L* > *0* **using** *length-L′ length-L-nonzero* **by** *arith*
have *index-Suc-upper-bound-L*: *index* − *Suc 0* < *length L* **using** *index-upper-bound length-L′* **by** *arith*
have *fst* (*D′* ! *ladder-n L′* (*index* − *Suc 0*)) = *fst* (*D* ! *ladder-n L* (*index* − *Suc 0*)) − (*length δ*)
apply (*subst D′*, *subst L′*)
apply (*subst ladder-cut-prefix-n*[*OF index-Suc-upper-bound-L length-L-gr-0*])
apply (*simp add*: *derivation-shift-def*)
using *index-lower-bound index-upper-bound is-ladder-D-L ladder-n-minus-1-bound length-L′* **by** *auto*
then have *ladder-i-L′*: *ladder-i L′ index* = *fst* (*D′* ! *ladder-n L′* (*index* − *Suc 0*))
using *ladder-i-L ladder-i-L′-as-L* **by** *auto*
have *LeftDerivationIntro* (*ladder-α α D′ L′ index*) (*ladder-i L′ index*)
(*snd* (*D′* ! *ladder-n L′* (*index* − *Suc 0*))) (*ladder-ix L′ index*)
(*drop* (*Suc* (*ladder-n L′* (*index* − *Suc 0*))) (*take* (*ladder-n L′ index*) *D′*)) (*ladder-j L′ index*)
(*ladder-γ α D′ L′ index*)
proof −
have *LeftDerivationIntro* (*ladder-α* (*δ*@*α*) *D L index*) (*ladder-i L index*)
(*snd* (*D* ! *ladder-n L* (*index* − *Suc 0*))) (*ladder-ix L index*)
(*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) (*take* (*ladder-n L index*) *D*)) (*ladder-j L index*)
(*ladder-γ* (*δ*@*α*) *D L index*) **using** *introsAt*
by (*metis LeftDerivationIntrosAt-def One-nat-def*)
then have *ldintro*: *LeftDerivationIntro* (*δ*@(*ladder-α α D′ L′ index*)) (*ladder-i L index*)
(*snd* (*D* ! *ladder-n L* (*index* − *Suc 0*))) (*ladder-ix L index*)
(*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) (*take* (*ladder-n L index*) *D*)) (*ladder-j L index*)
(*ladder-γ* (*δ*@*α*) *D L index*)
using *D′ L′ LeftDerivationLadder-cut-prefix-propagate derivation-ge-δ in-*

dex-upper-bound
is-word-δ ladder ladder-i-0 length-L′ **by** *auto*
have *dge*: *derivation-ge* (*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*)))
(*take* (*ladder-n L index*) *D*)) (*length δ*) **using** *derivation-ge-δ*
by (*metis append-take-drop-id derivation-ge-append*)
have *δ-le-i-L*: *length δ* ≤ *ladder-i L index*
using *derivation-ge-ladder index-upper-bound length-L′* **by** *auto*
from *LeftDerivationIntro-cut-prefix*[*OF ldintro dge δ-le-i-L is-word-δ*] **obtain**
γ′ **where** *γ′*:
ladder-γ (*δ* @ *α*) *D L index* = *δ* @ *γ′* ∧
LeftDerivationIntro (*ladder-α α D′ L′ index*) (*ladder-i L index* − *length*
δ)
(*snd* (*D* ! *ladder-n L* (*index* − *Suc 0*))) (*ladder-ix L index*)
(*derivation-shift* (*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) (*take* (*ladder-n*
L index) *D*))
(*length δ*) *0*) (*ladder-j L index* − *length δ*) *γ′* **by** *blast*
have *h1*: *ladder-i L′ index* = *ladder-i L index* − *length δ*
using *L′ ladder-cut-prefix-i ladder-i-L′-as-L* **by** *blast*
have *h2*: (*snd* (*D′* ! *ladder-n L′* (*index* − *Suc 0*))) = (*snd* (*D* ! *ladder-n L*
(*index* − *Suc 0*)))
apply (*subst L′*, *subst ladder-cut-prefix-n*)
apply (*simp add*: *index-Suc-upper-bound-L*)
using *length-L-gr-0* **apply** *auto*[*1*]
apply (*subst D′*)
apply (*simp add*: *derivation-shift-def*)
using *index-lower-bound index-upper-bound is-ladder-D-L ladder-n-minus-1-bound*

length-L′ **by** *auto*
have *h3*: *ladder-ix L′ index* = *ladder-ix L index*
using *ladder-cut-prefix-ix L′ index-upper-bound length-L′ length-L-gr-0* **by**
auto
have *h4*: (*drop* (*Suc* (*ladder-n L′* (*index* − *Suc 0*))) (*take* (*ladder-n L′ index*)
D′)) =
(*derivation-shift* (*drop* (*Suc* (*ladder-n L* (*index* − *Suc 0*))) (*take* (*ladder-n*
L index) *D*))
(*length δ*) *0*)
apply (*subst D′*)
apply (*subst L′*)+
apply (*subst ladder-cut-prefix-n*, *simp add*: *index-Suc-upper-bound-L*)
using *length-L-gr-0* **apply** *blast*
apply (*subst ladder-cut-prefix-n*)
using *index-upper-bound length-L′* **apply** *arith*
using *length-L-gr-0* **apply** *blast*
apply (*simp add*: *derivation-shift-def*)
by (*simp add*: *drop-map take-map*)
have *h5*: *ladder-j L′ index* = *ladder-j L index* − *length δ*
using *ladder-cut-prefix-j L′ index-upper-bound length-L′ length-L-gr-0* **by**
auto
have *h6*: *ladder-γ α D′ L′ index* = *γ′*

using *D′ L′ LeftDerivationLadder-cut-prefix-propagate γ′ derivation-ge-δ index-upper-bound*
is-word-δ ladder ladder-i-0 length-L′ **by** *auto*
show *?thesis* **using** *h1 h2 h3 h4 h5 h6 γ′* **by** *simp*
qed
then have *LeftDerivationIntrosAt α D′ L′ index*
apply (*auto simp add*: *LeftDerivationIntrosAt-def Let-def*)
using *ladder-i-L′* **by** *blast*
}
note *is-ladder-5* = *this*
show *?thesis*
apply (*rule-tac x=D′* **in** *exI*)
apply (*rule-tac x=L′* **in** *exI*)
apply (*rule-tac x=γ′* **in** *exI*)
apply *auto*
using *γ-def* **apply** *blast*
defer *1*
using *D′* **apply** *blast*
using *L′ ladder-cut-prefix-length* **apply** *auto*[*1*]
apply (*subst ladder-i-L′-thm*)
using *LeftDerivationLadder-def is-ladder-def ladder* **apply** *blast*
apply *simp*
apply (*simp add*: *ladder-last-j-def*)
apply (*subst ladder-j-L′-thm*)
apply (*simp add*: *length-L′*)
using *length-L-nonzero* **apply** *arith*
apply (*simp add*: *length-L′*)
apply (*auto simp add*: *LeftDerivationLadder-def*)
using *α-γ′* **apply** *blast*
apply (*auto simp add*: *is-ladder-def*)
using *length-L-nonzero length-L′* **apply** *auto*[*1*]
using *is-ladder-1* **apply** *blast*
using *is-ladder-2* **apply** *blast*
using *is-ladder-3* **apply** *blast*
using *is-ladder-4* **apply** *blast*
by (*auto simp add*: *LeftDerivationIntros-def is-ladder-5*)
qed

end

end
theory *TheoremD10*
imports *TheoremD9 Ladder*
begin

context *LocalLexing* **begin**

lemma *𝒫-wellformed*: $p \in \mathcal{P}\ k\ u \Longrightarrow$ *wellformed-tokens p*
using *𝒫-are-admissible admissible-wellformed-tokens* **by** *blast*

lemma *$\mathcal{X}$-token-length*: $t \in \mathcal{X}\ k \Longrightarrow k + length\ (chars\text{-}of\text{-}token\ t) \leq length\ Doc$
by (*metis le-diff-conv2 $\mathcal{X}$-is-prefix add.commute chars-of-token-def empty-$\mathcal{X}$ empty-iff is-prefix-length le-neq-implies-less length-drop linear*)

lemma *mono-Scan*: *mono* (*Scan T k*)
by (*simp add*: *Scan-regular regular-implies-mono*)

lemma *π-apply-setmonotone*: $x \in I \Longrightarrow x \in \pi\ k\ T\ I$
using *Complete-subset-π LocalLexing.Complete-def LocalLexing-axioms* **by** *blast*

lemma *Scan-apply-setmonotone*: $x \in I \Longrightarrow x \in Scan\ T\ k\ I$
by (*simp add*: *Scan-def*)

lemma *leftderives-padfront*:
assumes *leftderives α β*
assumes *is-word u*
shows *leftderives* (*u*@*α*) (*u*@*β*)
using *LeftDerivation-append-prefix LeftDerivation-implies-leftderives assms*(*1*) *assms*(*2*)

leftderives-implies-LeftDerivation **by** *blast*

lemma *leftderives-padback*:
assumes *leftderives α β*
assumes *is-sentence u*
shows *leftderives* (*α*@*u*) (*β*@*u*)
using *LeftDerivation-append-suffix LeftDerivation-implies-leftderives assms*(*1*) *assms*(*2*)

leftderives-implies-LeftDerivation **by** *blast*

lemma *leftderives-pad*:
assumes *α-β*: *leftderives α β*
assumes *is-word*: *is-word u*
assumes *is-sentence*: *is-sentence v*
shows *leftderives* (*u*@*α*@*v*) (*u*@*β*@*v*)
by (*simp add*: *α-β is-sentence is-word leftderives-padback leftderives-padfront*)

lemma *leftderives-rule*:
assumes $(N, w) \in \mathfrak{R}$
shows *leftderives* [*N*] *w*
by (*metis append-Nil append-Nil2 assms is-sentence-def is-word-terminals leftderives1-def*

leftderives1-implies-leftderives list.pred-inject(*1*) *terminals-empty wellformed-tokens-empty-path*)

lemma *leftderives-rule-step*:
assumes *ld*: *leftderives a* (*u*@[*N*]@*v*)
assumes *rule*: $(N, w) \in \mathfrak{R}$
assumes *is-word*: *is-word u*

assumes *is-sentence*: *is-sentence v*
shows *leftderives a* $(u@w@v)$
proof –
have *N-w*: *leftderives* $[N]$ *w* **using** *rule leftderives-rule* **by** *blast*
then have *leftderives* $(u@[N]@v)$ $(u@w@v)$ **using** *leftderives-pad is-word is-sentence*
by *blast*
then show *leftderives a* $(u@w@v)$ **using** *leftderives-trans ld* **by** *blast*
qed

lemma *leftderives-trans-step*:
assumes *ld*: *leftderives a* $(u@b@v)$
assumes *rule*: *leftderives b c*
assumes *is-word*: *is-word u*
assumes *is-sentence*: *is-sentence v*
shows *leftderives a* $(u@c@v)$
proof –
have *leftderives* $(u@b@v)$ $(u@c@v)$ **using** *leftderives-pad ld rule is-word is-sentence*
by *blast*
then show *?thesis* **using** *leftderives-trans ld* **by** *blast*
qed

lemma *charslength-of-prefix*:
assumes *is-prefix a b*
shows *charslength a* $\leq$ *charslength b*
by (*simp add*: *assms is-prefix-chars is-prefix-length*)

lemma *item-rhs-simp*[*simp*]: *item-rhs* (*Item* (N, α) *d i j*) $= \alpha$
by (*simp add*: *item-rhs-def*)

definition *Prefixes* :: $'a$ *list* $\Rightarrow$ $'a$ *list set*
where
Prefixes p = { *q* . *is-prefix q p* }

lemma $\mathfrak{P}$*-wellformed*: $p \in \mathfrak{P} \Longrightarrow$ *wellformed-tokens p*
by (*simp add*: $\mathfrak{P}$*-are-admissible admissible-wellformed-tokens*)

lemma *Prefixes-reflexive*[*simp*]: $p \in$ *Prefixes p*
by (*simp add*: *Prefixes-def is-prefix-def*)

lemma *Prefixes-is-prefix*: $q \in$ *Prefixes p* = *is-prefix q p*
by (*simp add*: *Prefixes-def*)

lemma *prefixes-are-paths'*: $p \in \mathfrak{P} \Longrightarrow$ *is-prefix q p* $\Longrightarrow q \in \mathfrak{P}$
using $\mathcal{P}$*.simps(3)* $\mathfrak{P}$*-def prefixes-are-paths* **by** *blast*

lemma *thmD10-ladder*:
$p \in \mathfrak{P} \Longrightarrow$
charslength $p = k \Longrightarrow$
$X \in T \Longrightarrow$

$T \subseteq \mathcal{X}\ k \Longrightarrow$
$(N,\ \alpha @ \beta) \in \mathfrak{R} \Longrightarrow$
$r \leq$ *length p* $\Longrightarrow$
leftderives $[\mathfrak{S}]$ ((*terminals* (*take r p*))@$[N]$@γ) $\Longrightarrow$
LeftDerivationLadder α *D L* (*terminals* ((*drop r p*)@$[X]$)) $\Longrightarrow$
ladder-last-j L = *length* (*drop r p*) $\Longrightarrow$
$k' = k +$ *length* (*chars-of-token X*) $\Longrightarrow$
$x =$ *Item* $(N,\ \alpha @ \beta)$ (*length* α) (*charslength* (*take r p*)) $k' \Longrightarrow$
$I =$ *items-le* k' (π k' {} (*Scan T k* (*Gen* (*Prefixes p*))))
$\Longrightarrow x \in I$
proof (*induct length L arbitrary*: *N* α β *r* γ *D L x rule*: *less-induct*)
  case *less*
    have *item-origin-x-def*: *item-origin x* = (*charslength* (*take r p*))
      by (*simp add*: *less.prems*(*11*))
    then have *x-k*: *item-origin x* $\leq$ *k*
    using *charslength.simps is-prefix-chars is-prefix-length is-prefix-take less.prems*(*2*)
by *force*
    have *item-end-x-def*: *item-end x* = k' **by** (*simp add*: *less.prems*(*11*))
    have *item-dot-x-def*: *item-dot x* = *length* α **by** (*simp add*: *less.prems*(*11*))
    have k'*-upperbound*: $k' \leq$ *length Doc*
      using $\mathcal{X}$*-token-length less.prems*(*10*) *less.prems*(*3*) *less.prems*(*4*) **by** *blast*
    note *item-def* = *less.prems*(*11*)
    note k' = *less.prems*(*10*)
    note *rule-dom* = *less.prems*(*5*)
    note *p-charslength* = *less.prems*(*2*)
    note *p-dom* = *less.prems*(*1*)
    note *r* = *less.prems*(*6*)
    note *leftderives-start* = *less.prems*(*7*)
    note *X-dom* = *less.prems*(*3*)
    have *wellformed-x*: *wellformed-item x*
      apply (*auto simp add*: *wellformed-item-def item-def rule-dom p-charslength*)
      apply (*subst* k')
      apply (*subst charslength.simps*[*symmetric*])
      apply (*subst p-charslength*[*symmetric*])
      using *item-origin-x-def p-charslength x-k* **apply** *linarith*
      apply (*rule* k'*-upperbound*)
      done
    have *leftderives-*α: *leftderives* α (*terminals* ((*drop r p*)@$[X]$))
    using *LeftDerivationLadder-def LeftDerivation-implies-leftderives less.prems*(*8*)
by *blast*
   have *is-sentence-drop-pX*: *is-sentence* (*drop r* (*terminals p*) @ [*terminal-of-token*
X])
    by (*metis derives-is-sentence is-sentence-concat leftderives-*α *leftderives-implies-derives*

      rule-α*-type rule-dom terminals-append terminals-drop terminals-singleton*)
    have *snd-item-rule-x*: *snd* (*item-rule x*) = $\alpha @ \beta$ **by** (*simp add*: *item-def*)
    from *less* **have** *is-ladder D L* **using** *LeftDerivationLadder-def* **by** *blast*
    then have *length L* $\neq$ *0* **by** (*simp add*: *is-ladder-not-empty*)
    then have *length L* = *1* $\vee$ *length L* > *1* **by** *arith*

then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have ∃ *i*. *LeftDerivationFix* α *i D* (*length* (*drop r p*)) (*terminals* ((*drop r p*)@[*X*]))
using *1.hyps LeftDerivationLadder-L-0 less.prems*(*8*) *less.prems*(*9*) **by** *fastforce*
then obtain *i* **where** *LDF*:
LeftDerivationFix α *i D* (*length* (*drop r p*)) (*terminals* ((*drop r p*)@[*X*])) **by** *blast*
from *LeftDerivationFix-splits-at-derives*[*OF this*] **obtain** *U a1 a2 b1 b2* **where** *decompose*:
splits-at α *i a1 U a2* ∧ *splits-at* (*terminals* (*drop r p* @ [*X*]))
(*length* (*drop r p*)) *b1 U b2* ∧ *derives a1 b1* ∧ *derives a2 b2* **by** *blast*
then have *b1*: *b1* = *terminals* (*drop r p*)
by (*simp add*: *append-eq-conv-conj splits-at-def*)
with *decompose* **have** *U*: *U* = *fst X*
by (*metis length-terminals nth-append-length splits-at-def terminal-of-token-def*
terminals-append terminals-singleton)
from *decompose b1 U* **have** *b2*: *b2* = []
by (*simp add*: *splits-at-combine splits-at-def*)
have *D*: *LeftDerivation* α *D* (*terminals* ((*drop r p*)@[*X*]))
using *LDF LeftDerivationLadder-def less.prems*(*8*) **by** *blast*
let *?y* = *Item* (*item-rule x*) (*length a1*) (*item-origin x*) *k*
have *wellformed-y*: *wellformed-item ?y*
using *wellformed-x*
apply (*auto simp add*: *wellformed-item-def x-k*)
using *k′ k′-upperbound* **apply** *arith*
apply (*simp add*: *item-rhs-def snd-item-rule-x*)
using *decompose splits-at-def*
by (*simp add*: *is-prefix-length trans-le-add1*)
have *y-nonterminal*: *item-nonterminal ?y* = *N*
by (*simp add*: *item-def item-nonterminal-def*)
have *item-α-y*: *item-α ?y* = *a1*
by (*metis append-assoc append-eq-conv-conj append-take-drop-id decompose item.sel*(*1*)
item.sel(*2*) *item-α-def item-rhs-def snd-item-rule-x splits-at-def*)
have *pvalid-y*: *pvalid p ?y*
apply (*auto simp add*: *pvalid-def*)
using *p-dom* 𝔓*-wellformed* **apply** *blast*
using *wellformed-y* **apply** *auto*[*1*]
apply (*rule-tac x*=*r* **in** *exI*)
apply (*auto simp add*: *r*)
using *p-charslength* **apply** *simp*
using *item-def* **apply** *simp*
apply (*rule-tac x*=γ **in** *exI*)
using *y-nonterminal* **apply** *simp*
using *is-derivation-def leftderives-start* **apply** *auto*[*1*]

apply (*simp add*: *item-α-y*)
using *b1 decompose* **by** *auto*
let *?z* = *inc-item ?y k′*
have *item-rhs-y*: *item-rhs ?y* = *α@β*
by (*simp add*: *item-def item-rhs-def*)
have *split-α*: *α* = *a1*@[*U*]@*a2* **using** *decompose splits-at-combine* **by** *blast*
have *next-symbol-y*: *next-symbol ?y* = *Some*(*fst X*)
by (*auto simp add*: *next-symbol-def is-complete-def item-rhs-y split-α U*)
have *z-in-Scan-y*: *?z* ∈ *Scan T k* {*?y*}
apply (*simp add*: *Scan-def*)
apply (*rule disjI2*)
apply (*rule-tac x*=*?y* **in** *exI*)
apply (*rule-tac x*=*fst X* **in** *exI*)
apply (*rule-tac x*=*snd X* **in** *exI*)
apply (*auto simp add*: *bin-def X-dom*)
using *k′ chars-of-token-def* **apply** *simp*
using *next-symbol-y* **apply** *simp*
done
from *pvalid-y* **have** *?y* ∈ *Gen*(*Prefixes p*)
apply (*simp add*: *Gen-def*)
apply (*rule-tac x*=*p* **in** *exI*)
by (*auto simp add*: *paths-le-def p-dom*)
then have *Scan T k* {*?y*} ⊆ *Scan T k* (*Gen*(*Prefixes p*))
apply (*rule-tac monoD*[*OF mono-Scan*])
apply *blast*
done
with *z-in-Scan-y* **have** *z-in-Scan-Gen*: *?z* ∈ *Scan T k* (*Gen*(*Prefixes p*))
using *rev-subsetD* **by** *blast*
have *wellformed-z*: *wellformed-item ?z*
using *k′ k′-upperbound next-symbol-y wellformed-inc-item wellformed-y* **by**
auto
have *item-β-z*: *item-β ?z* = *a2@β*
apply (*simp add*: *item-β-def*)
apply (*simp add*: *item-rhs-y split-α*)
done
have *item-end-z*: *item-end ?z* = *k′* **by** *simp*
have *x-via-z*: *x* = *inc-dot* (*length a2*) *?z*
by (*simp add*: *inc-dot-def less.prems*(*11*) *split-α*)
have *x-in-z*: *x* ∈ *π k′* {} {*?z*}
apply (*subst x-via-z*)
apply (*rule-tac thmD6*[*OF wellformed-z item-β-z item-end-z*])
using *decompose b2* **by** *blast*
have *π k′* {} {*?z*} ⊆ *π k′* {} (*Scan T k* (*Gen*(*Prefixes p*)))
apply (*rule-tac monoD*[*OF mono-π*])
using *z-in-Scan-Gen* **using** *empty-subsetI insert-subset* **by** *blast*
then have *x-in-Scan-Gen*: *x* ∈ *π k′* {} (*Scan T k* (*Gen*(*Prefixes p*)))
using *x-in-z* **by** *blast*
have *item-end x* = *k′* **by** (*simp add*: *item-end-x-def*)
with *x-in-Scan-Gen* **show** *?case*

```
      using items-le-def less.prems(12) mem-Collect-eq nat-le-linear by blast
  next
    case 2
     obtain i where i: i = ladder-i L 0 by blast
     obtain i′ where i′: i′ = ladder-j L 0 by blast
     obtain α′ where α′: α′ = ladder-γ α D L 0 by blast
     obtain n where n: n = ladder-n L 0 by blast
     have ldfix: LeftDerivationFix α i (take n D)  i′ α′
       using LeftDerivationLadder-def α′ i i′ n less.prems(8) by blast
     have α′-alt: α′ = ladder-α α D L 1 using 2 by (simp add: α′ ladder-α-def)

     have i′-alt: i′ = ladder-i L 1 using 2 by (simp add: i′ ladder-i-def)
     obtain e where e: e = D ! n by blast
     obtain ix where ix: ix = ladder-ix L 1 by blast
     obtain m where m: m = ladder-n L 1 by blast
     obtain E where E: E = drop (Suc n) (take m D) by blast
     have ldintro: LeftDerivationIntro α′ i′ (snd e) ix E (ladder-j L 1) (ladder-γ
α D L 1)
       by (metis 2.hyps LeftDerivationIntrosAt-def LeftDerivationIntros-def
         LeftDerivationLadder-def One-nat-def α′-alt E diff-Suc-1 e i′-alt ix leI
         less.prems(8) m n not-less-eq zero-less-one)
     have is-nonterminal-α′-at-i′: is-nonterminal (α′ ! i′)
       using LeftDerivationIntro-implies-nonterminal ldintro by blast
     then have is-nonterminal-α-at-i: is-nonterminal (α ! i)
       using LeftDerivationFix-def ldfix by auto
     then have ∃ A a1 a2 a1′. splits-at α i a1 A a2 ∧ splits-at α′ i′ a1′ A a2 ∧
       LeftDerivation a1 (take n D) a1′
       using LeftDerivationFix-splits-at-nonterminal ldfix by auto
     then obtain A a1 a2 a1′ where A: splits-at α i a1 A a2 ∧ splits-at α′ i′
a1′ A a2 ∧
       LeftDerivation a1 (take n D) a1′ by blast
     have A-def: A = α′ ! i′ using A splits-at-def by blast
     have is-nonterminal-A: is-nonterminal A using A-def is-nonterminal-α′-at-i′
by blast
     have ∃ rule. e = (i′, rule)
       by (metis e 2.hyps LeftDerivationIntrosAt-def LeftDerivationIntros-def
       LeftDerivationLadder-def One-nat-def Suc-leI diff-Suc-1 i′-alt less.prems(8)

         n prod.collapse zero-less-one)
     then obtain rule where rule: e = (i′, rule) by blast
     obtain w where w: w = snd rule by blast
     obtain α″ where α″: α″ = a1′ @ w @ a2 by blast
     have α′-α″: LeftDerives1 α′ i′ rule α″
       by (metis A w LeftDerivationFix-is-sentence LeftDerivationIntro-def
             LeftDerivationIntro-examine-rule LeftDerives1-def α″ ldfix ldintro
prod.collapse
         rule snd-conv splits-at-implies-Derives1)
     then have is-word-a1′: is-word a1′ using LeftDerives1-splits-at-is-word A
by blast
```

have *A-eq-fst-rule*: *A* = *fst rule*
using *A LeftDerivationIntro-examine-rule ldintro rule* **by** *fastforce*
have *ix-bound*: *ix* < *length w* **using** *ix w rule ldintro LeftDerivationIntro-def snd-conv*
by *auto*
then have ∃ *w1 W w2*. *splits-at w ix w1 W w2* **using** *splits-at-def* **by** *blast*

then obtain *w1 W w2* **where** *W*: *splits-at w ix w1 W w2* **by** *blast*
have *a1′-w-a2*: *a1′@w@a2* = *ladder-stepdown-α-0 α D L*
using *ladder-stepdown-α-0-altdef 2.hyps A α′-alt e i′-alt less.prems*(*8*) *n rule*
snd-conv w **by** *force*
from *LeftDerivationLadder-stepdown*[*OF less.prems*(*8*) *2*] **obtain** *L′* **where** *L′*:
LeftDerivationLadder (*a1′*@(*w*@*a2*)) (*drop* (*ladder-stepdown-diff L*) *D*) *L′*
(*terminals* (*drop r p* @ [*X*])) ∧
length L′ = *length L* − *1* ∧
ladder-i L′ 0 = *ladder-i L 1* + *ladder-ix L 1* ∧ *ladder-last-j L′* = *ladder-last-j L*
using *a1′-w-a2* **by** *auto*
have *ladder-i-L′-0*: *ladder-i L′ 0* = *i′* + *ix* **using** *L′ i′-alt ix* **by** *auto*
have *ladder-last-j-L′*: *ladder-last-j L′* = *length* (*drop r p*) **using** *L′ less.prems* **by** *auto*
from *L′* **have** *this1*: *LeftDerivationLadder* (*a1′*@(*w*@*a2*)) (*drop* (*ladder-stepdown-diff L*) *D*) *L′*
(*terminals* (*drop r p* @ [*X*])) **by** *blast*
have *this2*: *length a1′* ≤ *ladder-i L′ 0* **using** *A ladder-i-L′-0 splits-at-def* **by** *auto*
from *LeftDerivationLadder-cut-prefix*[*OF this1 is-word-a1′ this2*]
obtain *D′ L′′ γ′* **where** *L′′*:
terminals (*drop r p* @ [*X*]) = *a1′* @ *γ′* ∧
LeftDerivationLadder (*w* @ *a2*) *D′ L′′ γ′* ∧
D′ = *derivation-shift* (*drop* (*ladder-stepdown-diff L*) *D*) (*length a1′*) *0* ∧
length L′′ = *length L′* ∧
ladder-i L′′ 0 + *length a1′* = *ladder-i L′ 0* ∧
ladder-last-j L′′ + *length a1′* = *ladder-last-j L′* **by** *blast*
have *length-a1′-bound*: *length a1′* ≤ *length* (*drop r p*) **using** *L′′ ladder-last-j-L′*
by *linarith*
then have *is-prefix-a1′-drop-r-p*: *is-prefix a1′* (*terminals* (*drop r p*))
proof −
have *f1*: ∀ *ss ssa ssb*. ¬ *is-prefix* (*ss*::*symbol list*) (*ssa* @ *ssb*) ∨ *is-prefix ss ssa* ∨ (∃ *ssc*. *ssc* ≠ [] ∧ *is-prefix ssc ssb* ∧ *ss* = *ssa* @ *ssc*)
by (*simp add*: *is-prefix-of-append*)
have *f2*: ⋀*ss ssa*. *is-prefix* ((*ss*::*symbol list*) @ *ssa*) *ss* ∨ ¬ *is-prefix ssa* []
by (*metis* (*no-types*) *append-Nil2 is-prefix-cancel*)
have *f3*: ⋀*ss*. *is-prefix ss* [] ∨ ¬ *is-prefix* (*terminals* (*drop r p*) @ *ss*) *a1′*
by (*metis* (*no-types*) *drop-eq-Nil is-prefix-append length-a1′-bound length-terminals*)

have *is-prefix a1′* (*a1′* @ *γ′*) ∧ *is-prefix a1′ a1′*
by (*metis* (*no-types*) *append-Nil2 is-prefix-cancel is-prefix-empty*)
then show *?thesis*
using *f3 f2 f1* **by** (*metis L″ terminals-append*)
qed
obtain *r′* **where** *r′*: *r′* = *r* + *i′* **by** *blast*
have *length-a1′-eq-i′*: *length a1′* = *i′*
using *A less-or-eq-imp-le min.absorb2 splits-at-def* **by** *auto*
have *a1′-r′*: *r* ≤ *r′* ∧ *r′* ≤ *length p* ∧
terminals (*drop r p*) = *a1′* @ (*terminals* (*drop r′ p*))
using *is-prefix-a1′-drop-r-p r′*
proof −
have ∃ *q*. *terminals* (*drop r p*) = *a1′* @ *q*
using *is-prefix-a1′-drop-r-p* **by** (*metis is-prefix-unsplit*)
then obtain *q* **where** *q*: *terminals* (*drop r p*) = *a1′* @ *q* **by** *blast*
have *q* = *drop i′* (*terminals* (*drop r p*))
using *length-a1′-eq-i′ q* **by** (*simp add*: *append-eq-conv-conj*)
then have *q* = *terminals* (*drop i′* (*drop r p*)) **by** *simp*
then have *q* = *terminals* (*drop r′ p*) **by** (*simp add*: *r′ add.commute*)
with *q* **show** *?thesis*
using *add.commute diff-add-inverse le-Suc-ex le-add1 le-diff-conv length-a1′-bound*
length-a1′-eq-i′ length-drop r r′ **by** *auto*
qed
have *ladder-i-L″*: *ladder-i L″ 0* = *ix* **using** *L″ ladder-i-L′-0 length-a1′-eq-i′*

add.commute add-left-cancel **by** *linarith*
have *L″-ladder*: *LeftDerivationLadder* (*w* @ *a2*) *D′ L″ γ′* **using** *L″* **by** *blast*
have *ladder-i L″ 0* < *length w* **using** *ladder-i-L″ ix-bound* **by** *blast*
from *LeftDerivationLadder-cut-appendix*[*OF L″-ladder this*]
obtain *E′ F′ γ1′ γ2′ L‴* **where** *L‴*:
D′ = *E′* @ *F′* ∧
γ′ = *γ1′* @ *γ2′* ∧
LeftDerivationLadder w E′ L‴ γ1′ ∧
derivation-ge F′ (*length γ1′*) ∧
LeftDerivation a2 (*derivation-shift F′* (*length γ1′*) *0*) *γ2′* ∧
length L‴ = *length L″* ∧ *ladder-i L‴ 0* = *ladder-i L″ 0* ∧
ladder-last-j L‴ = *ladder-last-j L″*
by *blast*
obtain *z* **where** *z*: *z* = *Item* (*A*, *w*) (*length w*) (*charslength* (*take r′ p*)) *k′* **by** *blast*
have *z1*: *length L‴* < *length L* **using** *2.hyps L′ L″ L‴* **by** *linarith*
have *z2*: *p* ∈ $\mathfrak{P}$ **by** (*simp add*: *p-dom*)
have *z3*: *charslength p* = *k* **using** *p-charslength* **by** *auto*
have *z4*: *X* ∈ *T* **by** (*simp add*: *X-dom*)
have *z5*: *T* ⊆ $\mathcal{X}$ *k* **by** (*simp add*: *less.prems*(*4*))
have *rule* ∈ $\mathfrak{R}$
using *Derives1-rule LeftDerives1-implies-Derives1 α′-α″* **by** *blast*

then have *z6*: $(A,\ w\ @\ []) \in \mathfrak{R}$ **using** *w A-eq-fst-rule* **by** *auto*
have *z7*: $r' \leq$ *length p* **using** *a1'-r'* **by** *linarith*
have *leftderives* $[\mathfrak{S}]$ ((*terminals* (*take r p*))@[*N*]@γ)
using *leftderives-start* **by** *blast*
then have *leftderives* $[\mathfrak{S}]$ ((*terminals* (*take r p*))@(α@β)@γ)
by (*metis* $\mathfrak{P}$*-wellformed is-derivation-def is-derivation-is-sentence is-sentence-concat*

is-word-terminals-take leftderives-implies-derives leftderives-rule-step
p-dom rule-dom)
then have *leftderives* $[\mathfrak{S}]$ ((*terminals* (*take r p*))@*a1*@([*A*]@*a2*@β)@γ)
using *A splits-at-combine append-assoc* **by** *fastforce*
then have *z8-helper*: *leftderives* $[\mathfrak{S}]$ ((*terminals* (*take r p*))@*a1'*@([*A*]@*a2*@β)@γ)
by (*meson A LeftDerivation-implies-leftderives* $\mathfrak{P}$*-wellformed is-derivation-def*

is-derivation-is-sentence is-sentence-concat is-word-terminals-take
leftderives-implies-derives leftderives-trans-step p-dom)
have *join-terminals*: (*terminals* (*take r p*))@*a1'* = *terminals* (*take r' p*)
by (*metis is-prefix-a1'-drop-r-p length-a1'-eq-i' r' take-add take-prefix*
terminals-drop terminals-take)
from *z8-helper join-terminals* **have** *z8*:
leftderives $[\mathfrak{S}]$ (*terminals* (*take r' p*) @ [*A*] @ *a2* @ β @ γ)
by (*metis append-assoc*)
have γ'*-altdef*: γ' = *terminals* (*drop r' p* @ [*X*])
using *L'' a1'-r'* **by** *auto*
have *ladder-last-j L'''* + *length a1'* = *length* (*drop r p*)
using *L'' L''' ladder-last-j-L'* **by** *linarith*
then have *ladder-last-j-L'''-*γ': *ladder-last-j L'''* = *length* $\gamma' - 1$
by (*simp add*: γ'*-altdef length-a1'-eq-i' r'*)
then have *length* $\gamma' - 1 <$ *length* $\gamma 1'$
using *L''' ladder-last-j-bound* **by** *fastforce*
then have *length* $\gamma 1'$ + *length* $\gamma 2' - 1 <$ *length* $\gamma 1'$
using *L'''* **by** *simp*
then have *length* $\gamma 2' = 0$ **by** *arith*
then have $\gamma 2'$: $\gamma 2' = []$ **by** *simp*
then have $\gamma 1'$: $\gamma 1'$ = *terminals* (*drop r' p* @ [*X*]) **using** γ'*-altdef L'''* **by**
auto
then have *z9*: *LeftDerivationLadder w E' L'''* (*terminals* (*drop r' p* @ [*X*]))
using *L'''* **by** *blast*
have *z10*: *ladder-last-j L'''* = *length* (*drop r' p*)
using γ'*-altdef ladder-last-j-L'''-*γ' **by** *auto*
note *z11* = k'
have *z12*: *z* = *Item* (*A*, *w* @ []) (*length w*) (*charslength* (*take r' p*)) k'
using *z* **by** *simp*
note *z13* = *less.prems*(*12*)
note *induct* = *less.hyps*(*1*)[*of L''' A w* [] *r' a2*@β@γ *E' z*]
note *z-in-I* = *induct*[*OF z1 z2 z3 z4 z5 z6 z7 z8 z9 z10 z11 z12 z13*]
have *a2-derives-empty*: *derives a2* [] **using** *L'''* $\gamma 2'$
using *LeftDerivation-implies-leftderives leftderives-implies-derives* **by** *blast*

obtain *x1* **where** *x1*: *x1* = *Item* (*N*, *α@β*) (*length a1*)
(*charslength* (*take r p*)) (*charslength* (*take r′ p*)) **by** *blast*
obtain *q* **where** *q*: *q* = *take r′ p* **by** *blast*
then have *is-prefix-q-p*: *is-prefix q p* **by** *simp*
then have *q-in-Prefixes*: *q* ∈ *Prefixes p* **using** *Prefixes-is-prefix* **by** *blast*
then have *wellformed-q*: *wellformed-tokens q*
using *p-dom is-prefix-q-p prefixes-are-paths′ 𝔓-wellformed* **by** *blast*
have *item-rule-x1*: *item-rule x1* = (*N*, *α@β*)
using *x1* **by** *simp*
have *is-prefix-r-r′*: *is-prefix* (*take r p*) (*take r′ p*)
by (*metis append-eq-conv-conj is-prefix-take r′ take-add*)
then have *charslength-le-r-r′*: *charslength* (*take r p*) ≤ *charslength* (*take r′ p*)
using *charslength-of-prefix* **by** *blast*
have *is-prefix* (*take r′ p*) *p* **by** *auto*
then have *charslength-r′-p*: *charslength* (*take r′ p*) ≤ *charslength p*
using *charslength-of-prefix* **by** *blast*
have *charslength p* ≤ *length Doc*
using *less.prems*(*1*) *add-leE k′ k′-upperbound z3* **by** *blast*
with *charslength-r′-p* **have** *charslength-r′-Doc*:
charslength (*take r′ p*) ≤ *length Doc* **by** *arith*
have *α-decompose*: *α* = *a1* @ [*A*] @ *a2* **using** *A splits-at-combine* **by** *blast*
have *wellformed-x1*: *wellformed-item x1*
apply (*auto simp add*: *wellformed-item-def*)
using *item-rule-x1 less.prems* **apply** *auto*[*1*]
using *charslength-le-r-r′ x1* **apply** *auto*[*1*]
using *charslength-r′-Doc x1* **apply** *auto*[*1*]
using *x1 α-decompose* **by** *simp*
have *item-nonterminal-x1*: *item-nonterminal x1* = *N*
by (*simp add*: *x1 item-nonterminal-def*)
have *r-q-p*: *take r* (*terminals q*) = *terminals* (*take r p*)
by (*metis q is-prefix-r-r′ length-take min.absorb2 r take-prefix terminals-take*)

have *item-α-x1*: *item-α x1* = *a1* **by** (*simp add*: *α-decompose item-α-def x1*)
have *a1′*: *a1′* = *drop r* (*terminals q*)
by (*metis append-eq-conv-conj join-terminals length-take length-terminals min.absorb2 q r*)
have *pvalid-q-x1*: *pvalid q x1*
apply (*simp add*: *pvalid-def wellformed-q wellformed-x1 item-nonterminal-x1*)
apply (*rule-tac x*=*r* **in** *exI*)
apply *auto*
apply (*simp add*: *a1′-r′ min.absorb2 q*)
apply (*simp add*: *q x1*)
apply (*simp add*: *q x1 r′*)
using *r-q-p less.prems*(*7*) *append-Cons is-leftderivation-def*
leftderivation-implies-derivation **apply** *fastforce*
apply (*simp add*: *item-α-x1*)
using *a1′ A LeftDerivation-implies-leftderives leftderives-implies-derives* **by** *blast*

```
    have item-end-x1-le-k': item-end x1 ≤ k'
      using charslength-r'-p
      apply (simp add: x1)
      using less.prems by auto
    have x1-in-I: x1 ∈ I
      apply (subst less.prems(12))
      apply (auto simp add: items-le-def item-end-x1-le-k')
      apply (rule π-apply-setmonotone)
      apply (rule Scan-apply-setmonotone)
      apply (simp add: Gen-def)
      apply (rule-tac x=q in exI)
      by (simp add: pvalid-q-x1 paths-le-def q-in-Prefixes)
    obtain x2 where x2: x2 = inc-item x1 k' by blast
    have x1-in-bin: x1 ∈ bin I (item-origin z)
      using bin-def x1 x1-in-I z12 by auto
    have x2-in-Complete: x2 ∈ Complete k' I
      apply (simp add: Complete-def)
      apply (rule disjI2)
      apply (rule-tac x=x1 in exI)
      apply (simp add: x2)
      apply (rule-tac x=z in exI)
      apply auto
      using x1-in-bin apply blast
      using bin-def z12 z-in-I apply auto[1]
      apply (simp add: is-complete-def z12)
   by (simp add: α-decompose is-complete-def item-nonterminal-def next-symbol-def
x1 z12)
    have wf-I': wellformed-items (π k' {} (Scan T k (Gen (Prefixes p))))
   by (simp add: wellformed-items-Gen wellformed-items-Scan wellformed-items-π
z5)
    from items-le-Complete[OF this] x2-in-Complete
    have x2-in-I: x2 ∈ I by (metis Complete-π-fix z13)
    have wellformed-items I
      using wf-I' items-le-is-filter wellformed-items-def z13 by auto
    with x2-in-I have wellformed-x2: wellformed-item x2
      by (simp add: wellformed-items-def)
    have item-dot-x2: item-dot x2 = Suc (length a1)
      by (simp add: x2 x1)
    have item-β-x2: item-β x2 = a2 @ β
      apply (simp add: item-β-def item-dot-x2)
      apply (simp add: item-rhs-def x2 x1 α-decompose)
      done
    have item-end-x2: item-end x2 = k' by (simp add: x2)
     note inc-dot-x2-by-a2 = thmD6[OF wellformed-x2 item-β-x2 item-end-x2
a2-derives-empty]
    have x = inc-dot (length a2) x2
      by (simp add: α-decompose inc-dot-def less.prems(11) x1 x2)
    with inc-dot-x2-by-a2 have x ∈ π k' {} {x2} by auto
    then have x ∈ π k' {} I using x2-in-I
```

by (*meson contra-subsetD empty-subsetI insert-subset monoD mono-*π)
then show $x \in I$
by (*metis* (*no-types*, *lifting*) π*-subset-elem-trans dual-order.refl item-end-x-def*

items-le-def items-le-is-filter mem-Collect-eq z13)
qed
qed

theorem *thmD10*:
assumes *p-dom*: $p \in \mathfrak{P}$
assumes *p-charslength*: *charslength* $p = k$
assumes *X-dom*: $X \in T$
assumes *T-dom*: $T \subseteq \mathcal{X}\ k$
assumes *rule-dom*: $(N, \alpha @ \beta) \in \mathfrak{R}$
assumes *r*: $r \leq$ *length p*
assumes *leftderives-start*: *leftderives* $[\mathfrak{S}]$ ((*terminals* (*take r p*))@$[N]$@γ)
assumes *leftderives-*α: *leftderives* α (*terminals* ((*drop r p*)@$[X]$))
assumes *k'*: $k' = k +$ *length* (*chars-of-token X*)
assumes *item-def*: $x =$ *Item* $(N, \alpha @ \beta)$ (*length* α) (*charslength* (*take r p*)) k'
assumes *I*: $I =$ *items-le* k' (π k' {} (*Scan T k* (*Gen* (*Prefixes p*))))
shows $x \in I$
proof −
have $\exists$ *D*. *LeftDerivation* α *D* (*terminals* ((*drop r p*)@$[X]$))
using *leftderives-*α *leftderives-implies-LeftDerivation* **by** *blast*
then obtain *D* **where** *D*: *LeftDerivation* α *D* (*terminals* ((*drop r p*)@$[X]$)) **by**
blast
have *is-sentence*: *is-sentence* (*terminals* (*drop r p* @ $[X]$))
using *derives-is-sentence is-sentence-concat leftderives-*α *leftderives-implies-derives*

rule-α*-type rule-dom* **by** *blast*
have $\exists$ *L*. *LeftDerivationLadder* α *D L* (*terminals* ((*drop r p*)@$[X]$)) $\wedge$
ladder-last-j L = *length* (*drop r p*)
apply (*rule LeftDerivationLadder-exists*)
apply (*rule D*)
apply (*rule is-sentence*)
by *auto*
then obtain *L* **where** *L*: *LeftDerivationLadder* α *D L* (*terminals* ((*drop r*
p)@$[X]$)) **and**
L-ladder-last-j: *ladder-last-j L* = *length* (*drop r p*) **by** *blast*
from *thmD10-ladder*[*OF assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*)
assms(*7*)
L L-ladder-last-j k' item-def I]
show *?thesis* **.**
qed

end

end
theory *TheoremD11*

imports *TheoremD10*
begin

context *LocalLexing* **begin**

lemma *LeftDerivationLadder-length-1*:
assumes *ladder*: *LeftDerivationLadder* α *D L* γ
assumes *singleton-L*: *length L* = *1*
shows *LeftDerivationFix* α (*ladder-i L 0*) *D* (*ladder-last-j L*) γ
proof −
have *ldfix*: *LeftDerivationFix* α (*ladder-i L 0*) (*take* (*ladder-n L 0*) *D*) (*ladder-j L 0*)
(*ladder-*γ α *D L 0*)
using *ladder LeftDerivationLadder-def* **by** *blast*
have *ladder-n-0*: *ladder-n L 0* = *length D*
using *ladder singleton-L*
by (*metis LeftDerivationLadder-def One-nat-def diff-Suc-1 is-ladder-def ladder-last-n-intro*)
from *ldfix ladder-n-0 ladder singleton-L* **show** *?thesis*
by (*metis Derivation-unique-dest LeftDerivationLadder-def*
LeftDerivationLadder-implies-LeftDerivation-at-index LeftDerivationLadder-ladder-n-bound

LeftDerivation-implies-Derivation One-nat-def diff-Suc-1 ladder-last-j-def take-all

zero-less-one)
qed

lemma *LeftDerivationFix-from-singleton-helper*:
assumes *LeftDerivationFix* [*A*] *0 D* (*length u*) (*u* @ [*B*] @ *v*)
shows *D* = []
proof −
from *iffD1*[*OF LeftDerivationFix-def assms*] **obtain** *E F* **where** *EF*:
is-sentence [*A*] ∧
is-sentence (*u* @ [*B*] @ *v*) ∧
LeftDerivation [*A*] *D* (*u* @ [*B*] @ *v*) ∧
0 < *length* [*A*] ∧
length u < *length* (*u* @ [*B*] @ *v*) ∧
[*A*] ! *0* = (*u* @ [*B*] @ *v*) ! *length u* ∧
D = *E* @ *derivation-shift F 0* (*Suc* (*length u*)) ∧
LeftDerivation (*take 0* [*A*]) *E* (*take* (*length u*) (*u* @ [*B*] @ *v*)) ∧
LeftDerivation (*drop* (*Suc 0*) [*A*]) *F* (*drop* (*Suc* (*length u*)) (*u* @ [*B*] @ *v*))
by *blast*
from *EF* **have** *E*: *E* = []
by (*metis Derivation.elims*(*2*) *Derives1-split LeftDerivation-implies-Derivation*

Nil-is-append-conv list.distinct(*1*) *take-0*)
from *EF* **have** *F*: *F* = []
by (*metis E LeftDerivation.simps*(*1*) *LeftDerivation-ge-take LeftDerivation-implies-Derivation*

append-eq-conv-conj derivation-ge-shift is-word-Derivation length-Cons length-derivation-shift

list.size(3) nth-Cons-0 nth-append self-append-conv2 take-0)
from *EF E F* **show** $D = []$ **by** *auto*
qed

lemma *LeftDerivationFix-from-singleton*:
assumes *LeftDerivationFix* [*A*] *i D j* γ
shows $D = []$
proof −
have $\exists$ *u B v. splits-at* γ *j u B v* **using** *assms*
using *LeftDerivationFix-splits-at-derives* **by** *blast*
then obtain *u B v* **where** *s*: *splits-at* γ *j u B v* **by** *blast*
from *s* **have** *s1*: $\gamma = u$ @ [*B*] @ *v* **using** *splits-at-combine* **by** *blast*
from *s* **have** *s2*: *j* = *length u* **using** *splits-at-def* **by** *auto*
from *assms s1 s2 LeftDerivationFix-from-singleton-helper*
show *?thesis* **by** (*metis LeftDerivationFix-def length-Cons less-Suc0 list.size*(*3*))
qed

lemma *LeftDerivationLadder-ladder-*γ*-last*:
assumes *LeftDerivationLadder* α *D L* γ
shows γ = *ladder-*γ α *D L* (*length L* − *1*)
by (*metis Derive LeftDerivationLadder-def LeftDerivation-implies-Derivation One-nat-def assms*
is-ladder-def last-ladder-γ)

theorem *thmD11-helper*:
$p \in \mathfrak{P} \Longrightarrow$
charslength $p = k \Longrightarrow$
$X \in T \Longrightarrow$
$T \subseteq \mathcal{X}\ k \Longrightarrow$
$q = p$ @ $[X] \Longrightarrow$
$(N, \alpha@\beta) \in \mathfrak{R} \Longrightarrow$
$r \leq$ *length* $q \Longrightarrow$
LeftDerivation [$\mathfrak{S}$] *D* ((*terminals* (*take r q*))@[*N*]@γ) $\Longrightarrow$
leftderives α (*terminals* (*drop r q*)) $\Longrightarrow$
$k' = k$ + *length* (*chars-of-token X*) $\Longrightarrow$
x = *Item* (*N*, $\alpha@\beta$) (*length* α) (*charslength* (*take r q*)) $k' \Longrightarrow$
I = *items-le* k' (π k' {} (*Scan T k* (*Gen* (*Prefixes p*)))) $\Longrightarrow$
$x \in I$
proof (*induct length D arbitrary*: *D r N* γ α β *x rule*: *less-induct*)
case *less*
have $D = [] \vee D \neq []$ **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
then have *r*: $r = 0$
by (*metis LeftDerivation.simps*(*1*) *diff-add-0 diff-add-inverse2 le-0-eq length-0-conv*

length-append length-terminals less.prems(*7*) *less.prems*(*8*) *list.size*(*4*) *take-eq-Nil*)
with *1* **have** γ: γ = []
using *LeftDerivation.simps*(*1*) *append-Cons append-self-conv2 less.prems*(*8*) *list.inject*
take-eq-Nil terminals-empty **by** *auto*
from *r* γ *1* **have** *start-is-N*: $\mathfrak{S}$ = *N*
using *LeftDerivation.simps*(*1*) *append-eq-Cons-conv less.prems*(*8*) *list.inject take-eq-Nil*
terminals-empty **by** *auto*
have *h1*: *r* ≤ *length p* **using** *r* **by** *auto*
have *h2*: *leftderives* [$\mathfrak{S}$] (*terminals* (*take r p*) @ [*N*] @ γ)
by (*simp add*: *r* γ *start-is-N*)
have *h3*: *leftderives* α (*terminals* (*drop r p* @ [*X*]))
using *less.prems* **by** (*simp add*: *r less.prems*)
have *h4*: *x* = *Item* (*N*, α @ β) (*length* α) (*charslength* (*take r p*)) *k*′
using *less.prems* **by** (*simp add*: *r less.prems*)
from *thmD10*[*OF less.prems*(*1*, *2*, *3*, *4*, *6*) *h1 h2 h3 less.prems*(*10*) *h4 less.prems*(*12*)]
show *?case* **.**
next
case *2*
note *D-non-empty* = *2*
have *r* < *length q* ∨ *r* = *length q* **using** *less* **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have *h1*: *r* ≤ *length p* **using** *less.prems 1* **by** *auto*
have *take-q-p*: *take r q* = *take r p*
using *1 less.prems*
by (*simp add*: *drop-keep-last le-neq-implies-less nat-le-linear not-less-eq not-less-eq-eq*)
have *h2*: *leftderives* [$\mathfrak{S}$] (*terminals* (*take r p*) @ [*N*] @ γ)
apply (*simp only*: *take-q-p*[*symmetric*])
using *less.prems LeftDerivation-implies-leftderives* **by** *blast*
have *h3*: *leftderives* α (*terminals* (*drop r p* @ [*X*]))
using *less.prems*(*5*, *9*) *h1* **by** *simp*
have *h4*: *k*′ = *k* + *length* (*chars-of-token X*) **using** *less.prems* **by** *blast*
have *h5*: *x* = *Item* (*N*, α @ β) (*length* α) (*charslength* (*take r p*)) *k*′
using *less.prems take-q-p* **by** *simp*
from *thmD10*[*OF less.prems*(*1*, *2*, *3*, *4*, *6*) *h1 h2 h3 h4 h5 less.prems*(*12*)]
show *?case* **.**
next
case *2*
from *2* **have** *ld*: *LeftDerivation* [$\mathfrak{S}$] *D* (*terminals q* @ [*N*] @ γ)
using *less.prems*(*8*) **by** *auto*
from *2* **have** α*-derives-empty*: *derives* α []
using *less.prems*(*9*) **by** *auto*
have *is-sentence-p*: *is-sentence* (*terminals p*)

using *less.prems(1)* $\mathcal{L}_P$*-derives* $\mathfrak{P}$*-are-admissible admissible-def is-derivation-def*
is-derivation-is-sentence is-sentence-concat **by** *blast*
have *is-symbol-X*: *is-symbol (terminal-of-token X)*
using *less.prems(3, 4)* $\mathcal{X}$*-are-terminals is-symbol-def rev-subsetD* **by** *blast*
have *is-sentence-q*: *is-sentence (terminals q)* **using** *is-sentence-p is-symbol-X*
less.prems LeftDerivation-implies-leftderives is-derivation-def
is-derivation-is-sentence is-sentence-concat ld leftderives-implies-derives
by *blast*
have *is-symbol-N*: *is-symbol N*
using *less.prems(6) is-symbol-def rule-nonterminal-type* **by** *blast*
have *is-sentence-γ*: *is-sentence γ*
by *(meson LeftDerivation-implies-leftderives is-derivation-def is-derivation-is-sentence*

is-sentence-concat ld leftderives-implies-derives)
have *ld-exists-h1*: *is-sentence (terminals q @ [N] @ γ)*
using *is-sentence-q is-sentence-γ is-symbol-N is-sentence-concat*
LeftDerivation-implies-leftderives is-derivation-def is-derivation-is-sentence ld
leftderives-implies-derives **by** *blast*
have *ld-exists-h2*: *length q < length (terminals q @ [N] @ γ)* **by** *simp*
from *LeftDerivationLadder-exists[OF ld ld-exists-h1 ld-exists-h2]* **obtain** *L* **where**
L: *LeftDerivationLadder [𝔖] D L (terminals q @ [N] @ γ)* **and**
L-last-j: *ladder-last-j L = length q* **by** *blast*
note *r-eq-length-q = 2*
have *ladder-i-0-eq-0*: *ladder-i L 0 = 0* **using** *L append-Nil ladder-i-0-bound*

length-append-singleton less-Suc0 list.size(3) **by** *fastforce*
have *length L = 1 ∨ length L > 1* **using** *L*
by *(metis LeftDerivationLadder-def Suc-eq-plus1 Suc-eq-plus1-left butlast-conv-take*
butlast-snoc diff-add-inverse2 is-ladder-def le-add1 le-neq-implies-less
length-append-singleton old.nat.exhaust take-0)
then show *?case*
proof *(induct rule: disjCases2)*
case *1*
from *LeftDerivationLadder-length-1[OF L 1] ladder-i-0-eq-0*
have *ldfix*: *LeftDerivationFix [𝔖] 0 D (ladder-last-j L) (terminals q @ [N] @ γ)*
by *auto*
with *LeftDerivationFix-from-singleton* **have** *D = []* **by** *blast*
with *D-non-empty* **have** *False* **by** *auto*
then show *?case* **by** *blast*
next
case *2*
obtain *a* **where** *a*: *a = ladder-α [𝔖] D L (length L − 1)* **by** *blast*

then have *a-as-γ*: $a = ladder\text{-}\gamma$ [𝔊] *D L* (*length L* − *2*) **using** *2 ladder-α-def*
by (*metis diff-diff-left diff-is-0-eq not-le one-add-one*)
have *introsAt*: *LeftDerivationIntrosAt* [𝔊] *D L* (*length L* − *1*) **using** *L*
by (*metis 2.hyps LeftDerivationIntros-def LeftDerivationLadder-def One-nat-def*
Suc-leI Suc-lessD diff-less less-not-refl not-less-eq zero-less-diff)
obtain *i* **where** *i*: *i* = *ladder-i L* (*length L* − *1*) **by** *blast*
obtain *j* **where** *j*: *j* = *ladder-j L* (*length L* − *1*) **by** *blast*
obtain *ix* **where** *ix*: *ix* = *ladder-ix L* (*length L* − *1*) **by** *blast*
obtain *c* **where** *c*: *c* = *ladder-γ* [𝔊] *D L* (*length L* − *1*) **by** *blast*
obtain *n* **where** *n*: *n* = *ladder-n L* (*length L* − *1* − *1*) **by** *blast*
obtain *m* **where** *m*: *m* = *ladder-n L* (*length L* − *1*) **by** *blast*
obtain *e* **where** *e*: *e* = *D* ! *n* **by** *blast*
obtain *E* **where** *E*: *E* = *drop* (*Suc n*) (*take m D*) **by** *blast*
from *iffD1*[*OF LeftDerivationIntrosAt-def introsAt*]
have *i* = *fst e* ∧ *LeftDerivationIntro a i* (*snd e*) *ix E j c*
by (*metis E a c e i ix j m n*)
then have *i-eq-fst-e*: *i* = *fst e* **and**
ldintro: *LeftDerivationIntro a i* (*snd e*) *ix E j c* **by** *auto*
have *c-def*: *c* = *terminals q* @ [*N*] @ *γ*
using *c L LeftDerivationLadder-ladder-γ-last* **by** *simp*
from *iffD1*[*OF LeftDerivationIntro-def ldintro*] **obtain** *b* **where** *b*:
LeftDerives1 a i (*snd e*) *b* ∧ *ix* < *length* (*snd* (*snd e*)) ∧
snd (*snd e*) ! *ix* = *c* ! *j* ∧ *LeftDerivationFix b* (*i* + *ix*) *E j c* **by** *blast*
obtain *M ω* **where** *Mω*: (*M*, *ω*) = *snd e* **using** *prod.collapse* **by** *blast*
have *j-q*: *j* = *length q* **using** *L-last-j j ladder-last-j-def* **by** *auto*
with *c-def* **have** *c-at-j*: *c* ! *j* = *N*
by (*metis append-Cons length-terminals nth-append-length*)
with *b Mω* **have** *ω-at-ix*: *ω* ! *ix* = *N* **by** (*metis snd-conv*)
obtain *μ1 μ2* **where** *split-ω*: *splits-at ω ix μ1 N μ2*
by (*metis Mω ω-at-ix b snd-conv splits-at-def*)
obtain *a1 a2* **where** *split-a*: *splits-at a i a1 M a2* **using** *b*
by (*metis LeftDerivationIntro-bounds-ij LeftDerivationIntro-examine-rule Mω*
fst-conv ldintro splits-at-def)
then have *is-word-a1*: *is-word a1*
using *LeftDerives1-splits-at-is-word b* **by** *blast*
have *b* = *a1* @ *ω* @ *a2* **using** *split-a b Mω*
by (*metis LeftDerives1-implies-Derives1 snd-conv splits-at-combine-dest*)

then have *b-def*: *b* = *a1* @ *μ1* @ [*N*] @ *μ2* @ *a2* **using** *split-ω splits-at-combine*
by *simp*
have *is-nonterminal-N*: *is-nonterminal N*
using *less.prems*(*6*) *rule-nonterminal-type* **by** *blast*
with *LeftDerivationFix-splits-at-nonterminal split-a b*

have ∃ *U b1 b2 c1*. *splits-at b* (*i* + *ix*) *b1 U b2* ∧ *splits-at c j c1 U b2* ∧
LeftDerivation b1 E c1 **by** (*simp add*: *LeftDerivationFix-def c-at-j*)
then obtain *b1 b2 c1* **where** *b1b2c1*:
splits-at b (*i* + *ix*) *b1 N b2* ∧ *splits-at c j c1 N b2* ∧
LeftDerivation b1 E c1 **using** *c-at-j splits-at-def* **by** *auto*
then have *c1-q*: *c1* = *terminals q* **using** *c-def j-q*
by (*simp add*: *append-eq-conv-conj splits-at-def*)
have *length-a1-eq-i*: *length a1* = *i* **using** *split-a splits-at-def* **by** *auto*
have *length-μ1-eq-ix*: *length μ1* = *ix* **using** *split-ω splits-at-def* **by** *auto*
have *b1* = *take* (*i* + *ix*) *b* **using** *b1b2c1 splits-at-def* **by** *blast*
then have *b1-eq-a1-μ1*: *b1* = *a1* @ *μ1* **using** *b-def length-a1-eq-i length-μ1-eq-ix*
by (*simp add*: *append-eq-conv-conj take-add*)
have *LeftDerivation* (*a1* @ *μ1*) *E c1* **using** *b1b2c1 b1-eq-a1-μ1* **by** *blast*
from *LeftDerivation-skip-prefixword-ex*[*OF this is-word-a1*]
obtain *w* **where** *w*: *c1* = *a1* @ *w* ∧
LeftDerivation μ1 (*derivation-shift E* (*length a1*) *0*) *w* **by** *blast*
have *a1-eq-take-i*: *a1* = *take i* (*terminals q*)
using *w c1-q split-a append-eq-conv-conj length-a1-eq-i* **by** *blast*
have *μ1-leftderives*: *leftderives μ1* (*terminals* (*drop i q*)) **using** *w c1-q split-a*
LeftDerivation-implies-leftderives append-eq-conv-conj length-a1-eq-i **by** *auto*
have *LeftDerivation* [𝔖] (*take n D*) *a*
by (*metis 2.hyps L LeftDerivationLadder-implies-LeftDerivation-at-index*
One-nat-def Suc-lessD a-as-γ diff-Suc-eq-diff-pred diff-Suc-less n numeral-2-eq-2)
then have *LD-to-M*: *LeftDerivation* [𝔖] (*take n D*) ((*terminals* (*take i q*))@[*M*]@*a2*)
using *split-a splits-at-combine a1-eq-take-i terminals-take* **by** *auto*
have *is-ladder*: *is-ladder D L* **using** *L* **by** (*simp add*: *LeftDerivation-Ladder-def*)
then have *n-less-m*: *n* < *m* **using** *n m is-ladder-def*
by (*metis* (*no-types*, *lifting*) *2.hyps One-nat-def diff-Suc-less*
length-greater-0-conv zero-less-diff)
have *m-le-D*: *m* ≤ *length D* **using** *m is-ladder-def is-ladder dual-order.refl*
ladder-n-last-is-length **by** *auto*
have *length* (*take n D*) = *n* **using** *n-less-m m-le-D*
using *length-take less-irrefl less-le-trans linear min.absorb2* **by** *auto*
then have *length-take-n-D*: *length* (*take n D*) < *length D*
using *n-less-m m-le-D less-le-trans* **by** *linarith*
have *ω-decompose*: *ω* = *μ1*@(*N*#*μ2*) **using** *split-ω splits-at-combine* **by** *simp*
have (*M*, *ω*) ∈ ℜ **by** (*metis Derives1-rule LeftDerives1-implies-Derives1*

*M*ω *b*)
with *ω-decompose* **have** *M-rule*: (*M*, *μ1*@(*N*#*μ2*)) ∈ 𝔑 **by** *simp*
have *i-le-q*: *i* ≤ *length q* **using** *a1-eq-take-i length-a1-eq-i* **by** *auto*
obtain *y* **where**
y: *y* = *Item* (*M*, *μ1* @ *N* # *μ2*) (*length μ1*) (*charslength* (*take i q*)) *k′* **by** *blast*
from *less.hyps*[*OF length-take-n-D less.prems*(*1*, *2*, *3*, *4*, *5*) *M-rule i-le-q LD-to-M*
μ1-leftderives less.prems(*10*) *y less.prems*(*12*)]
have *y-in-I*: *y* ∈ *I* **by** *blast*
obtain *z* **where** *z*: *z* = *Item* (*N*, *α*@*β*) *0 k′ k′* **by** *blast*
then have *z-is-init-item*: *z* = *init-item* (*N*, *α*@*β*) *k′* **by** (*simp add*: *init-item-def*)
have *z* ∈ *Predict k′* {*y*}
apply (*simp add*: *z-is-init-item*)
apply (*rule next-symbol-predicts*)
apply (*simp add*: *is-complete-def next-symbol-def y*)
apply (*simp add*: *less.prems*(*6*))
apply (*simp add*: *y item-end-def*)
done
then have *z* ∈ *Predict k′ I* **using** *Predict-def bin-def y-in-I* **by** *auto*
then have *z-in-I*: *z* ∈ *I* **by** (*metis Predict-π-fix items-le-Predict less.prems*(*12*))
have *length-chars-q*: *length* (*chars q*) = *k′* **using** *less.prems* **by** *simp*
have *x-is-inc-dot*: *x* = *inc-dot* (*length α*) *z*
by (*simp add*: *less.prems*(*11*) *r-eq-length-q length-chars-q z inc-dot-def*)
have *wellformed-items-I*: *wellformed-items I*
apply (*subst less.prems*(*12*))
by (*meson LocalLexing.items-le-is-filter LocalLexing.wellformed-items-Gen*
LocalLexing-axioms empty-subsetI less.prems(*4*) *subsetCE wellformed-items-Scan*
wellformed-items-π wellformed-items-def)
with *z-in-I* **have** *wellformed-z*: *wellformed-item z*
using *wellformed-items-def* **by** *blast*
have *item-β-z*: *item-β z* = *α*@*β* **by** (*simp add*: *z-is-init-item*)
have *item-end-z*: *item-end z* = *k′* **by** (*simp add*: *z-is-init-item*)

have *x* ∈ *π k′* {} {*z*}
apply (*simp add*: *x-is-inc-dot*)
apply (*rule thmD6*)
apply (*rule wellformed-z*)
apply (*rule item-β-z*)
apply (*rule item-end-z*)
by (*simp add*: *α-derives-empty*)
then have *x* ∈ *π k′* {} *I* **using** *z-in-I*
by (*meson contra-subsetD empty-subsetI insert-subset monoD mono-π*)

then show *?case*

by (*metis* (*no-types*, *lifting*) *LocalLexing.wellformed-item-def* *LocalLexing-axioms*
π-subset-elem-trans item.sel(*3*) *item.sel*(*4*) *items-le-def items-le-is-filter*

less.prems(*11*) *less.prems*(*12*) *mem-Collect-eq wellformed-z z*)
qed
qed
qed
qed

theorem *thmD11*:
assumes *p-dom*: $p \in \mathfrak{P}$
assumes *p-charslength*: *charslength* $p = k$
assumes *X-dom*: $X \in T$
assumes *T-dom*: $T \subseteq \mathcal{X}\ k$
assumes *q-def*: $q = p$ @ $[X]$
assumes *rule-dom*: $(N, \alpha@\beta) \in \mathfrak{R}$
assumes *r*: $r \leq$ *length q*
assumes *leftderives-start*: *leftderives* [𝔖] ((*terminals* (*take r q*))@[*N*]@γ)
assumes *leftderives-α*: *leftderives* α (*terminals* (*drop r q*))
assumes *k′*: $k' = k +$ *length* (*chars-of-token X*)
assumes *item-def*: $x =$ *Item* (*N*, α@β) (*length* α) (*charslength* (*take r q*)) *k′*
assumes *I*: $I =$ *items-le k′* (π *k′* {} (*Scan T k* (*Gen* (*Prefixes p*))))
shows $x \in I$
proof −
have ∃ *D*. *LeftDerivation* [𝔖] *D* ((*terminals* (*take r q*))@[*N*]@γ)
using *leftderives-start leftderives-implies-LeftDerivation* **by** *blast*
then obtain *D* **where** *D*: *LeftDerivation* [𝔖] *D* ((*terminals* (*take r q*))@[*N*]@γ) **by** *blast*
from *thmD11-helper*[*OF assms*(*1*, *2*, *3*, *4*, *5*, *6*, *7*) *D assms*(*9*, *10*, *11*, *12*)]
show *?thesis* **.**
qed

end

end
theory *TheoremD12*
imports *TheoremD11*
begin

context *LocalLexing* **begin**

lemma *charslength-appendix-is-empty*:
charslength (*p*@*ts*) = *charslength p* ⟹ (⋀ *t*. *t* ∈ *set ts* ⟹ *chars-of-token t* = [])
proof (*induct ts*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons s ts*)

have *charslength* (*p* @ *s* # *ts*) = *charslength* (*p* @ *ts*) + *length* (*chars-of-token* *s*)
by *simp*
then have *charslength* (*p* @ *s* # *ts*) = *charslength* *p* + *charslength* *ts* + *length* (*chars-of-token* *s*)
by *simp*
with *Cons.prems*(*1*) **have** *charslength* *ts* + *length* (*chars-of-token* *s*) = *0* **by** *arith*
then show *?case* **using** *Cons.prems*(*2*) *charslength-0* **by** *auto*
qed

lemma *empty-tokens-have-charslength-0*:
($\bigwedge$ *t*. *t* ∈ *set* *ts* ⟹ *chars-of-token* *t* = []) ⟹ *charslength* *ts* = *0*
proof (*induct* *ts*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons* *t* *ts*)
then show *?case* **by** *auto*
qed

lemma *π-idempotent′*: π *k* {} (π *k* *T* *I*) = π *k* *T* *I*
apply (*simp* *add*: *π-no-tokens*)
by (*simp* *add*: *Complete-π-fix* *Predict-π-fix* *fix-is-fix-of-limit*)

theorem *thmD12*:
assumes *induct*: *items-le* *k* ($\mathcal{J}$ *k* *u*) = *Gen* (*paths-le* *k* ($\mathcal{P}$ *k* *u*))
assumes *induct-tokens*: $\mathcal{T}$ *k* *u* = $\mathcal{Z}$ *k* *u*
shows *items-le* *k* ($\mathcal{J}$ *k* (*Suc* *u*)) ⊇ *Gen* (*paths-le* *k* ($\mathcal{P}$ *k* (*Suc* *u*)))
proof −
{
fix *x* :: *item*
assume *x-dom*: *x* ∈ *Gen* (*paths-le* *k* ($\mathcal{P}$ *k* (*Suc* *u*)))
have ∃ *q*. *pvalid* *q* *x* ∧ *q* ∈ $\mathcal{P}$ *k* (*Suc* *u*) ∧ *charslength* *q* ≤ *k*
proof −
have $\bigwedge$*i* *I* *n*. *i* ∈ *I* ∨ *i* ∉ *items-le* *n* *I*
by (*meson* *items-le-is-filter* *subsetCE*)
then show *?thesis*
by (*metis* *Gen-implies-pvalid* *x-dom* *items-le-fix-D* *items-le-idempotent* *items-le-paths-le*
pvalid-item-end)
qed
then obtain *q* **where** *q*: *pvalid* *q* *x* ∧ *q* ∈ $\mathcal{P}$ *k* (*Suc* *u*) ∧ *charslength* *q* ≤ *k* **by** *blast*
have *q* ∈ $\mathcal{P}$ *k* *u* ∨ *q* ∉ $\mathcal{P}$ *k* *u* **by** *blast*
then have *x* ∈ *items-le* *k* ($\mathcal{J}$ *k* (*Suc* *u*))
proof (*induct* *rule*: *disjCases2*)
case *1*
with *q* **have** *x* ∈ *Gen* (*paths-le* *k* ($\mathcal{P}$ *k* *u*))
apply (*auto* *simp* *add*: *Gen-def*)

apply (*rule-tac* $x=q$ **in** *exI*)
by (*simp add*: *paths-le-def*)
with *induct* **have** $x \in$ *items-le* k ($\mathcal{J}$ k u) **by** *simp*
then show *?case*
using *LocalLexing.items-le-def LocalLexing-axioms* $\mathcal{J}$*-subset-Suc-u* **by** *fastforce*
next
case *2*
have *q-is-limit*: $q \in$ *limit* (*Append* ($\mathcal{Y}$ ($\mathcal{Z}$ k u) ($\mathcal{P}$ k u) k) k) ($\mathcal{P}$ k u) **using** q **by** *auto*
from *limit-Append-path-nonelem-split*[*OF q-is-limit 2*] **obtain** p *ts* **where** *p-ts*:
$q = p$ @ *ts* $\wedge$
$p \in \mathcal{P}$ k u $\wedge$
charslength $p = k$ $\wedge$
admissible (p @ *ts*) $\wedge$
($\forall$ $t$$\in$*set ts*. $t \in \mathcal{Y}$ ($\mathcal{Z}$ k u) ($\mathcal{P}$ k u) k) $\wedge$ ($\forall$ $t$$\in$*set* (*butlast ts*). *chars-of-token* t = [])
by *blast*
then have *ts-nonempty*: *ts* $\neq$ [] **using** *2* **using** *self-append-conv* **by** *auto*
obtain T **where** T: $T = \mathcal{Y}$ ($\mathcal{Z}$ k u) ($\mathcal{P}$ k u) k **by** *blast*
obtain J **where** J: $J = \pi$ k T (*Gen* (*paths-le* k ($\mathcal{P}$ k u))) **by** *blast*
from q *p-ts* **have** *chars-of-token-is-empty*: $\bigwedge$ t. $t$$\in$*set ts* $\Longrightarrow$ *chars-of-token* t = []
using *charslength-appendix-is-empty chars-append charslength.simps le-add1 le-imp-less-Suc*
le-neq-implies-less length-append not-less-eq **by** *auto*
{
fix *sss* :: *token list*
have *is-prefix sss ts* $\Longrightarrow$ *pvalid* (p @ *sss*) x $\Longrightarrow$ $x \in J$
proof (*induct length sss arbitrary*: *sss x rule*: *less-induct*)
case *less*
have *sss* = [] $\vee$ *sss* $\neq$ [] **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
with *less* **have** *pvalid-p-x*: *pvalid* p x **by** *auto*
have *charslength-p*: *charslength* $p \leq k$ **using** *p-ts* **by** *blast*
with *p-ts* **have** $p \in$ *paths-le* k ($\mathcal{P}$ k u)
by (*simp add*: *paths-le-def*)
with *pvalid-p-x* **have** $x \in$ *Gen* (*paths-le* k ($\mathcal{P}$ k u))
using *Gen-def mem-Collect-eq* **by** *blast*
then have $x \in \pi$ k T (*Gen* (*paths-le* k ($\mathcal{P}$ k u))) **using** π*-apply-setmonotone*
by *blast*
then show $x \in J$ **using** *pvalid-item-end q J LocalLexing.items-le-def*

LocalLexing-axioms charslength-p mem-Collect-eq pvalid-p-x **by** *auto*

next
case *2*
then have ∃ *a ss. sss* = *ss*@[*a*] **using** *rev-exhaust* **by** *blast*
then obtain *a ss* **where** *snoc*: *sss* = *ss*@[*a*] **by** *blast*
obtain *p′* **where** *p′*: *p′* = *p* @ *ss* **by** *auto*
then have *pvalid-left* (*p′*@[*a*]) *x* **using** *snoc less pvalid-left* **by** *simp*
from *iffD1*[*OF pvalid-left-def this*] **obtain** *r ω* **where** *pvalid*:
wellformed-tokens (*p′* @ [*a*]) ∧
wellformed-item x ∧
r ≤ *length* (*p′* @ [*a*]) ∧
charslength (*p′* @ [*a*]) = *item-end x* ∧
charslength (*take r* (*p′* @ [*a*])) = *item-origin x* ∧
is-leftderivation (*terminals* (*take r* (*p′* @ [*a*])) @ [*item-nonterminal x*] @ *ω*) ∧
leftderives (*item-α x*) (*terminals* (*drop r* (*p′* @ [*a*]))) **by** *blast*
obtain *q′* **where** *q′*: *q′* = *p′*@[*a*] **by** *blast*
have *is-prefix-ss-ts*: *is-prefix ss ts* **using** *snoc less*
by (*simp add*: *is-prefix-append*)
then have *is-prefix* (*p*@*ss*) *q* **using** *p′ snoc p-ts* **by** *simp*
then have *is-prefix p′ q* **using** *p′* **by** *simp*
then have *h1*: *p′* ∈ 𝔓 **using** *q* 𝔓*-covers-*𝒫 *prefixes-are-paths′ subsetCE* **by** *blast*
have *charslength-ss*: *charslength ss* = *0*
apply (*rule empty-tokens-have-charslength-0*)
by (*metis is-prefix-ss-ts append-is-Nil-conv chars-append chars-of-token-is-empty*
charslength.simps charslength-0 is-prefix-def length-greater-0-conv list.size(*3*))
then have *h2*: *charslength p′* = *k* **using** *p′ p-ts* **by** *auto*
have *a-in-ts*: *a* ∈ *set ts*
by (*metis in-set-dropD is-prefix-append is-prefix-cons list.set-intros*(*1*)

snoc less(*2*))
then have *h3*: *a* ∈ *T* **using** *T p-ts* **by** *blast*
have *h4*: *T* ⊆ 𝒳 *k*
using *LocalLexing.*𝒵*.simps*(*2*) *LocalLexing-axioms T* 𝒵*-subset-*𝒳 **by** *blast*
note *h5* = *q′*
obtain *N* **where** *N*: *N* = *item-nonterminal x* **by** *blast*
obtain *α* **where** *α*: *α* = *item-α x* **by** *blast*
obtain *β* **where** *β*: *β* = *item-β x* **by** *blast*
have *item-rule-x*: *item-rule x* = (*N*, *α* @ *β*)
using *N α β item-nonterminal-def item-rhs-def item-rhs-split* **by** *auto*
have *wellformed-item x* **using** *pvalid* **by** *blast*
then have *h6*: (*N*, *α*@*β*) ∈ ℜ **using** *item-rule-x*
by (*simp add*: *wellformed-item-def*)
have *h7*: *r* ≤ *length q′* **using** *pvalid q′* **by** *blast*
have *h8*: *leftderives* [𝔖] (*terminals* (*take r q′*) @ [*N*] @ *ω*)

using *N is-leftderivation-def pvalid q′* **by** *blast*
have *h9*: *leftderives* α (*terminals* (*drop r q′*)) **using** α *pvalid q′* **by** *blast*
have *h10*: $k = k + $ *length* (*chars-of-token a*)
by (*simp add*: *a-in-ts chars-of-token-is-empty*)
have *h11*: $x =$ *Item* (N, α @ β) (*length* α) (*charslength* (*take r q′*)) k
by (*metis* α *charslength-ss a-in-ts append-Nil2 chars.simps*(*2*) *chars-append*
chars-of-token-is-empty charslength.simps h2 item.collapse item-dot-is-α*-length*
item-rule-x length-greater-0-conv list.size(*3*) *pvalid q′*)
have *x-dom*: $x \in$ *items-le k* (π *k* {} (*Scan T k* (*Gen* (*Prefixes p′*))))
using *thmD11*[*OF h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11*] **by** *auto*
{
fix *y* :: *item*
fix *toks* :: *token list*
assume *pvalid-toks-y*: *pvalid toks y*
assume *is-prefix-toks-p′*: *is-prefix toks p′*
then have *charslength-toks*: *charslength toks* $\leq k$
using *charslength-of-prefix h2* **by** *blast*
then have *item-end-y*: *item-end y* $\leq k$ **using** *pvalid-item-end pvalid-toks-y*
by *auto*
have *is-prefix toks p* $\vee$ ($\exists$ *ss′*. *is-prefix ss′ ss* $\wedge$ *toks* = *p*@*ss′*)
using *is-prefix-of-append is-prefix-toks-p′ p′* **by** *auto*
then have $y \in J$
proof (*induct rule*: *disjCases2*)
case *1*
then have *toks* $\in \mathcal{P}$ *k u* **using** *p-ts prefixes-are-paths* **by** *blast*
with *charslength-toks* **have** *toks* $\in$ *paths-le k* ($\mathcal{P}$ *k u*)
by (*simp add*: *paths-le-def*)
then have $y \in$ *Gen* (*paths-le k* ($\mathcal{P}$ *k u*)) **using** *pvalid-toks-y*
Gen-def mem-Collect-eq **by** *blast*
then have $y \in \pi$ *k T* (*Gen* (*paths-le k* ($\mathcal{P}$ *k u*))) **using** π*-apply-setmonotone*
by *blast*
then show $y \in J$ **by** (*simp add*: *J items-le-def item-end-y*)
next
case *2*
then obtain *ss′* **where** *ss′*: *is-prefix ss′ ss* $\wedge$ *toks* = *p*@*ss′* **by** *blast*
then have *l1*: *length ss′* < *length sss*
using *append-eq-conv-conj append-self-conv is-prefix-length length-append*
less-le-trans nat-neq-iff not-Cons-self2 not-add-less1 snoc **by** *fastforce*
have *l2*: *is-prefix ss′ ts* **using** *ss′ is-prefix-ss-ts*
by (*metis append-dropped-prefix is-prefix-append*)

```
              have l3: pvalid (p @ ss') y using ss' pvalid-toks-y by simp
              show ?case using less.hyps[OF l1 l2 l3] by blast
            qed
          }
          then have Gen (Prefixes p') ⊆ J
            by (meson Gen-implies-pvalid Prefixes-is-prefix subsetI)
          with x-dom have r0: x ∈ items-le k (π k {} (Scan T k J))
              by (metis (no-types, lifting) LocalLexing.items-le-def LocalLex-
ing-axioms
               mem-Collect-eq mono-Scan mono-π mono-subset-elem subsetI)
          then have x-in-π: x ∈ π k {} (Scan T k J)
             using LocalLexing.items-le-is-filter LocalLexing-axioms subsetCE
by blast
          have r1: Scan T k J = J
            by (simp add: J Scan-π-fix)
          have r2: π k {} J = J using π-idempotent' using J by blast
          from x-in-π r1 r2 show x ∈ J by auto
        qed
      qed
    }
    note th = this
    have x-in-J: x ∈ J
      apply (rule th[of ts])
      apply (simp add: is-prefix-eq-proper-prefix)
      using p-ts q by blast
    have 𝒯-eq-𝒵: 𝒯 k (Suc u) = 𝒵 k (Suc u)
      using induct induct-tokens 𝒯-equals-𝒵-induct-step by blast
    have T-alt: T = 𝒯 k (Suc u) using 𝒯-eq-𝒵 T by simp
    have J = π k T (items-le k (𝒥 k u)) using induct J by simp
      then have J ⊆ π k T (𝒥 k u) by (simp add: items-le-is-filter monoD
mono-π)
    with T-alt have J ⊆ 𝒥 k (Suc u) using 𝒥.simps(2) by blast
    with x-in-J have x ∈ 𝒥 k (Suc u) by blast
    then show ?case
       using LocalLexing.items-le-def LocalLexing-axioms pvalid-item-end q by
auto
  qed
 }
 then show ?thesis by auto
qed

end

end
theory TheoremD13
imports TheoremD12
begin

context LocalLexing begin
```

lemma *pointwise-natUnion-swap*:
  assumes *pointwise-f*: *pointwise f*
  shows *f* (*natUnion G*) = *natUnion* (λ *u*. *f* (*G u*))
proof −
  note *f-simp* = *pointwise-simp*[*OF pointwise-f*]
  have *h1*: *f* (*natUnion G*) = ⋃ {*f* {*x*} |*x*. *x* ∈ (*natUnion G*)} **using** *f-simp* **by** *blast*
  have *h2*: ⋀ *x*. *f* (*G x*) = ⋃ {*f* {*y*} |*y*. *y* ∈ *G x*} **using** *f-simp* **by** *metis*
  show *?thesis*
    apply (*subst h1*)
    apply (*subst h2*)
    apply (*simp add*: *natUnion-def*)
    by *blast*
qed

lemma *pointwise-Gen*: *pointwise Gen*
  by (*simp add*: *pointwise-def Gen-def*, *blast*)

lemma *thmD13-part1*:
  assumes *start*: *items-le k* (𝒥 *k 0*) = *Gen* (*paths-le k* (𝒫 *k 0*))
  assumes *valid-k*: *k* ≤ *length Doc*
  shows *items-le k* (𝒥 *k u*) = *Gen* (*paths-le k* (𝒫 *k u*)) ∧ 𝒯 *k u* = 𝒵 *k u*
proof (*induct u*)
  case *0*
    then show *?case* **using** *start* **by** *auto*
next
  case (*Suc u*)
    from *Suc* **have** *sub*: *items-le k* (𝒥 *k* (*Suc u*)) ⊆ *Gen* (*paths-le k* (𝒫 *k* (*Suc u*)))
      using *thmD9 valid-k* **by** *blast*
    from *Suc* **have** *sup*: *items-le k* (𝒥 *k* (*Suc u*)) ⊇ *Gen* (*paths-le k* (𝒫 *k* (*Suc u*)))
      using *thmD12* **by** *blast*
    from *Suc* **have** *tokens*: 𝒯 *k* (*Suc u*) = 𝒵 *k* (*Suc u*)
      using 𝒯*-equals-*𝒵*-induct-step* **by** *blast*
    from *sub sup tokens* **show** *?case* **by** *blast*
qed

lemma *thmD13-part2*:
  assumes *start*: *items-le k* (𝒥 *k 0*) = *Gen* (*paths-le k* (𝒫 *k 0*))
  assumes *valid-k*: *k* ≤ *length Doc*
  shows *items-le k* (ℐ *k*) = *Gen* (*paths-le k* (𝒬 *k*))
proof −
  note *part1* = *thmD13-part1*[*OF start valid-k*]
  have *e1*: *items-le k* (ℐ *k*) = *natUnion* (λ *u*. *items-le k* (𝒥 *k u*))
    using *items-le-pointwise pointwise-natUnion-swap* **by** *auto*
  have *e2*: *natUnion* (λ *u*. *items-le k* (𝒥 *k u*)) = *natUnion* (λ *u*. *Gen* (*paths-le k* (𝒫 *k u*)))
    using *part1* **by** *auto*
  have *e3*: *natUnion* (λ *u*. *Gen* (*paths-le k* (𝒫 *k u*))) = *Gen* (*natUnion* (λ *u*.

paths-le k ($\mathcal{P}$ *k u*)))
  using *pointwise-Gen pointwise-natUnion-swap* **by** *fastforce*
 have *e4*: *natUnion* (λ *u*. *paths-le k* ($\mathcal{P}$ *k u*)) = *paths-le k* (*natUnion* (λ *u*. $\mathcal{P}$ *k u*))
  using *paths-le-pointwise pointwise-natUnion-swap* **by** *fastforce*
 from *e1 e2 e3 e4* **show** *?thesis* **by** *simp*
qed

theorem *thmD13*:
 assumes *start*: *items-le k* ($\mathcal{J}$ *k 0*) = *Gen* (*paths-le k* ($\mathcal{P}$ *k 0*))
 assumes *valid-k*: *k* ≤ *length Doc*
 shows *items-le k* ($\mathcal{J}$ *k u*) = *Gen* (*paths-le k* ($\mathcal{P}$ *k u*)) ∧ $\mathcal{T}$ *k u* = $\mathcal{Z}$ *k u*
  ∧ *items-le k* ($\mathcal{I}$ *k*) = *Gen* (*paths-le k* ($\mathcal{Q}$ *k*))
using *thmD13-part1*[*OF start valid-k*] *thmD13-part2*[*OF start valid-k*] **by** *blast*

end

end
theory *TheoremD14*
imports *TheoremD13*
begin

context *LocalLexing* **begin**

lemma *empty-tokens-of-empty*[*simp*]: *empty-tokens* {} = {}
 using *empty-tokens-is-filter* **by** *blast*

lemma *items-le-split-via-eq*: *items-le* (*Suc k*) *J* = *items-le k J* ∪ *items-eq* (*Suc k*) *J*
 by (*auto simp add*: *items-le-def items-eq-def*)

lemma *paths-le-split-via-eq*: *paths-le* (*Suc k*) *P* = *paths-le k P* ∪ *paths-eq* (*Suc k*) *P*
 by (*auto simp add*: *paths-le-def paths-eq-def*)

lemma *natUnion-superset*:
 shows *g i* ⊆ *natUnion g*
by (*meson natUnion-elem subset-eq*)

definition *indexle* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
 indexle k' u' k u = ((*indexlt k' u' k u*) ∨ (*k'* = *k* ∧ *u'* = *u*))

definition *produced-by-scan-step* :: *item* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
 produced-by-scan-step x k u = (∃ *k' u' y X*. *indexle k' u' k u* ∧ *y* ∈ $\mathcal{J}$ *k' u'* ∧
  item-end y = *k'* ∧ *X* ∈ ($\mathcal{T}$ *k' u'*) ∧ *x* = *inc-item y* (*k'* + *length* (*chars-of-token X*)) ∧
  next-symbol y = *Some* (*terminal-of-token X*))

lemma *indexle-trans*: *indexle k'' u'' k' u'* ⟹ *indexle k' u' k u* ⟹ *indexle k'' u''*

k u
using *indexle-def indexlt-trans*
proof −
assume *a1*: *indexle* k'' u'' k' u'
assume *a2*: *indexle* k' u' *k u*
then have *f3*: $\bigwedge n\ na.\ u' = u \vee$ *indexlt n na k u* $\vee \neg$ *indexlt n na* k' u'
by (*meson indexle-def indexlt-trans*)
have $\bigwedge n\ na.\ k' = k \vee$ *indexlt n na k u* $\vee \neg$ *indexlt n na* k' u'
using *a2* **by** (*meson indexle-def indexlt-trans*)
then show *?thesis*
using *f3 a2 a1 indexle-def* **by** *auto*
qed

lemma *produced-by-scan-step-trans*:
assumes *indexle* k' u' *k u*
assumes *produced-by-scan-step x* k' u'
shows *produced-by-scan-step x k u*
proof −
from *iffD1*[*OF produced-by-scan-step-def assms*(*2*)] **obtain** $k'a$ $u'a$ *y X* **where** *produced-*k'*-*u':
indexle $k'a$ $u'a$ k' u' $\wedge$
$y \in \mathcal{J}$ $k'a$ $u'a$ $\wedge$
item-end y $= k'a \wedge$
$X \in \mathcal{T}$ $k'a$ $u'a$ $\wedge$
$x =$ *inc-item y* ($k'a$ + *length* (*chars-of-token X*)) $\wedge$ *next-symbol y* = *Some* (*terminal-of-token X*)
by *blast*
then show *?thesis* **using** *indexle-trans assms*(*1*) *produced-by-scan-step-def* **by** *blast*
qed

lemma $\mathcal{J}$*-induct*[*consumes 1*, *case-names Induct*]:
assumes $x \in \mathcal{J}$ *k u*
assumes *induct*: $\bigwedge$ *x k u* . ($\bigwedge$ x' k' u'. $x' \in \mathcal{J}$ k' u' $\Longrightarrow$ *indexlt* k' u' *k u* $\Longrightarrow$ *P* x' k' u')
$\Longrightarrow x \in \mathcal{J}$ *k u* $\Longrightarrow$ *P x k u*
shows *P x k u*
proof −
let *?R* = *indexlt-rel* <*∗lex∗*> {}
have *wf-R*: *wf ?R* **by** (*auto simp add*: *wf-indexlt-rel*)
let *?P* = λ *a*. *snd a* $\in \mathcal{J}$ (*fst* (*fst a*)) (*snd* (*fst a*)) $\longrightarrow$ *P* (*snd a*) (*fst* (*fst a*)) (*snd* (*fst a*))
have $x \in \mathcal{J}$ *k u* $\longrightarrow$ *P x k u*
apply (*rule wf-induct*[*OF wf-R*, **where** *P* = *?P* **and** *a* = ((*k*, *u*), *x*), *simplified*])
apply (*auto simp add*: *indexlt-def*[*symmetric*])
apply (*rule-tac x=ba* **and** *k=a* **and** *u=b* **in** *induct*)
by *auto*
thus *?thesis* **using** *assms* **by** *auto*
qed

```
lemma π-no-tokens-item-end:
  assumes x-in-π: x ∈ π k {} I
  shows item-end x = k ∨ x ∈ I
proof −
  have x-in-limit: x ∈ limit (λI. Complete k (Predict k I)) I
    using x-in-π π-no-tokens by auto
  then show ?thesis
  proof (induct rule: limit-induct)
    case (Init x) then show ?case by auto
  next
    case (Iterate x J)
      from Iterate(2) have item-end x = k ∨ x ∈ Predict k J
        using Complete-item-end by auto
      then show ?case
      proof (induct rule: disjCases2)
        case 1 then show ?case by blast
      next
        case 2
          then have item-end x = k ∨ x ∈ J
            using Predict-item-end by auto
          then show ?case
          proof (induct rule: disjCases2)
            case 1 then show ?case by blast
          next
            case 2 then show ?case using Iterate(1)[OF 2] by blast
          qed
      qed
  qed
qed

lemma natUnion-ex: x ∈ natUnion f ⟹ ∃ i. x ∈ f i
  by (metis (no-types, opaque-lifting) mk-disjoint-insert natUnion-superset natU-
nion-upperbound
    subsetCE subset-insert)

lemma locate-in-limit:
  assumes x-in-limit: x ∈ limit f X
  assumes x-notin-X: x ∉ X
  shows ∃ n. x ∈ funpower f (Suc n) X ∧ x ∉ funpower f n X
proof −
  have ∃ N. x ∈ funpower f N X using x-in-limit limit-def natUnion-ex by fastforce
  then obtain N where N: x ∈ funpower f N X by blast
  {
    fix n :: nat
    have x ∈ funpower f n X ⟹ ∃ m < n. x ∈ funpower f (Suc m) X ∧ x ∉
funpower f m X
    proof (induct n)
      case 0
```

with *x-notin-X* **show** *?case* **by** *auto*
next
case (*Suc n*)
have $x \notin funpower\ f\ n\ X \vee x \in funpower\ f\ n\ X$ **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
then show *?case* **using** *Suc* **by** *fastforce*
next
case *2*
from *Suc*(*1*)[*OF 2*] **show** *?case* **using** *less-SucI* **by** *blast*
qed
qed
}
with *N* **show** *?thesis* **by** *auto*
qed

lemma *produced-by-scan-step*:
$x \in \mathcal{J}\ k\ u \Longrightarrow item\text{-}end\ x > k \Longrightarrow produced\text{-}by\text{-}scan\text{-}step\ x\ k\ u$
proof (*induct rule*: $\mathcal{J}$*-induct*)
case (*Induct x k u*)
have $(k = 0 \wedge u = 0) \vee (k > 0 \wedge u = 0) \vee (u > 0)$ **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases3*)
case *1*
with *Induct* **have** $item\text{-}end\ x = 0$ **using** $\mathcal{J}$*-0-0-item-end* **by** *blast*
with *Induct* **have** *False* **by** *arith*
then show *?case* **by** *blast*
next
case *2*
then obtain k' **where** k': $k = Suc\ k'$ **using** *Suc-pred'* **by** *blast*
with *Induct 2* **have** $x \in \mathcal{J}\ (Suc\ k')\ 0$ **by** *auto*
then have $x \in \pi\ k\ \{\}\ (\mathcal{I}\ k')$ **by** (*simp add*: k')
then have $item\text{-}end\ x = k \vee x \in \mathcal{I}\ k'$ **using** π*-no-tokens-item-end* **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
with *Induct* **have** *False* **by** *auto*
then show *?case* **by** *blast*
next
case *2*
then have $\exists\ u'.\ x \in \mathcal{J}\ k'\ u'$ **using** $\mathcal{I}$*.simps natUnion-ex* **by** *fastforce*
then obtain u' **where** u': $x \in \mathcal{J}\ k'\ u'$ **by** *blast*
have k'*-bound*: $k' < item\text{-}end\ x$ **using** k' *Induct* **by** *arith*
have *indexlt*: $indexlt\ k'\ u'\ k\ u$ **by** (*simp add*: *indexlt-simp* k')
from *Induct*(*1*)[*OF* u' *this* k'*-bound*]
have *pred-produced*: $produced\text{-}by\text{-}scan\text{-}step\ x\ k'\ u'$ **.**
then show *?case* **using** *indexlt produced-by-scan-step-trans indexle-def*
by *blast*

qed
next
case *3*
then have *ex-u'*: $\exists\ u'.\ u = Suc\ u'$ **by** *arith*
then obtain u' **where** u': $u = Suc\ u'$ **by** *blast*
with *Induct* **have** $x \in \mathcal{J}\ k\ (Suc\ u')$ **by** *metis*
then have *x-in-π*: $x \in \pi\ k\ (\mathcal{T}\ k\ u)\ (\mathcal{J}\ k\ u')$ **using** u' *𝒥.simps* **by** *metis*
have $x \in \mathcal{J}\ k\ u' \vee x \notin \mathcal{J}\ k\ u'$ **by** *blast*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have *indexlt*: *indexlt k u' k u* **by** (*simp add*: *indexlt-simp u'*)
with *Induct*(*1*)[*OF 1 indexlt Induct*(*3*)] **show** *?case*
using *indexle-def produced-by-scan-step-trans* **by** *blast*
next
case *2*
have *item-end-x*: $k <$ *item-end x* **using** *Induct* **by** *auto*
obtain *f* **where** *f*: $f = Scan\ (\mathcal{T}\ k\ u)\ k \circ Complete\ k \circ Predict\ k$ **by** *blast*
have $x \in limit\ f\ (\mathcal{J}\ k\ u')$
using *x-in-π π-functional f* **by** *simp*
from *locate-in-limit*[*OF this 2*] **obtain** *n* **where** *n*:
$x \in funpower\ f\ (Suc\ n)\ (\mathcal{J}\ k\ u') \wedge$
$x \notin funpower\ f\ n\ (\mathcal{J}\ k\ u')$ **by** *blast*
obtain *Y* **where** *Y*: $Y = funpower\ f\ n\ (\mathcal{J}\ k\ u')$
by *blast*
have *x-f-Y*: $x \in f\ Y \wedge x \notin Y$ **using** *Y n* **by** *auto*
then have $x \in Scan\ (\mathcal{T}\ k\ u)\ k\ (Complete\ k\ (Predict\ k\ Y))$ **using** *comp-apply f* **by** *simp*
then have $x \in (Complete\ k\ (Predict\ k\ Y)) \vee$
$x \in \{$ *inc-item x'* (k + *length c*) $\mid x'\ t\ c.\ x' \in bin\ (Complete\ k\ (Predict\ k\ Y))\ k \wedge$
$(t,\ c) \in (\mathcal{T}\ k\ u) \wedge$ *next-symbol* $x' = Some\ t$ $\}$ **using** *Scan-def* **by** *simp*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
then have *False* **using** *item-end-x x-f-Y Complete-item-end Predict-item-end*
using *less-not-refl3* **by** *blast*
then show *?case* **by** *auto*
next
case *2*
have $Y \subseteq limit\ f\ (\mathcal{J}\ k\ u')$ **using** *Y limit-def natUnion-superset* **by** *fastforce*
then have $Y \subseteq \pi\ k\ (\mathcal{T}\ k\ u)\ (\mathcal{J}\ k\ u')$ **using** *f* **by** (*simp add*: *π-functional*)
then have *Y-in-𝒥*: $Y \subseteq \mathcal{J}\ k\ u$ **using** u' **by** *simp*
then have *in-𝒥*: $Complete\ k\ (Predict\ k\ Y) \subseteq \mathcal{J}\ k\ u$
proof −

have *f1*: $\forall$ *f I Ia i*. ($\neg$ *mono f* $\vee$ $\neg$ (*I*::*item set*) $\subseteq$ *Ia* $\vee$ (*i*::*item*) $\notin$ *f I*) $\vee$ *i* $\in$ *f Ia*
by (*meson mono-subset-elem*)
obtain *ii* :: *item set* $\Rightarrow$ *item set* $\Rightarrow$ *item* **where**
$\forall$ *x0 x1*. ($\exists$ *v2*. *v2* $\in$ *x1* $\wedge$ *v2* $\notin$ *x0*) = (*ii x0 x1* $\in$ *x1* $\wedge$ *ii x0 x1* $\notin$ *x0*)
by *moura*
then have *f2*: $\forall$ *I Ia*. *ii Ia I* $\in$ *I* $\wedge$ *ii Ia I* $\notin$ *Ia* $\vee$ *I* $\subseteq$ *Ia*
by *blast*
obtain *nn* :: *nat* **where**
f3: *u* = *Suc nn*
using *ex-u′* **by** *presburger*
moreover
{ assume *ii* ($\mathcal{J}$ *k u*) (*Complete k* (*Predict k Y*)) $\in$ *Complete k* (π *k* ($\mathcal{T}$ *k* (*Suc nn*)) ($\mathcal{J}$ *k nn*))
then have *?thesis*
using *f3 f2 Complete-π-fix* **by** *auto* **}**
ultimately show *?thesis*
using *f2 f1* **by** (*metis* (*full-types*) *Complete-regular Predict-π-fix Predict-regular*
$\mathcal{J}$*.simps*(*2*) *Y-in-*$\mathcal{J}$ *regular-implies-mono*)
qed
from *2* **obtain** *x′ t c* **where** *x′-t-c*:
x = *inc-item x′* (*k* + *length c*) $\wedge$ *x′* $\in$ *bin* (*Complete k* (*Predict k Y*)) *k* $\wedge$
(*t*, *c*) $\in$ $\mathcal{T}$ *k u* $\wedge$ *next-symbol x′* = *Some t* **by** *blast*
show *?case*
apply (*simp add*: *produced-by-scan-step-def*)
apply (*rule-tac x*=*k* **in** *exI*)
apply (*rule-tac x*=*u* **in** *exI*)
apply (*simp add*: *indexle-def*)
apply (*rule-tac x*=*x′* **in** *exI*)
apply *auto*
using *x′-t-c bin-def in-*$\mathcal{J}$ **apply** *auto*[*1*]
using *x′-t-c bin-def* **apply** *blast*
apply (*rule-tac x*=*t* **in** *exI*)
apply (*rule-tac x*=*c* **in** *exI*)
using *x′-t-c* **by** *auto*
qed
qed
qed
qed

lemma *limit-single-step*:
assumes *x* $\in$ *f X*
shows *x* $\in$ *limit f X*
by (*metis assms elem-limit-simp funpower.simps*(*1*) *funpower.simps*(*2*))

lemma *Gen-union*: *Gen* (*A* $\cup$ *B*) = *Gen A* $\cup$ *Gen B*

by (*simp add*: *Gen-def*, *blast*)

lemma *is-prefix-Prefixes-subset*:
assumes *is-prefix q p*
shows *Prefixes q* $\subseteq$ *Prefixes p*
proof −
show *?thesis*
apply (*auto simp add*: *Prefixes-def*)
using *assms* **by** (*metis is-prefix-append is-prefix-def*)
qed

lemma *Prefixes-subset-*$\mathcal{P}$:
assumes $p \in \mathcal{P}\ k\ u$
shows *Prefixes p* $\subseteq \mathcal{P}\ k\ u$
using *Prefixes-is-prefix assms prefixes-are-paths* **by** *blast*

lemma *Prefixes-subset-paths-le*:
assumes *Prefixes p* $\subseteq P$
shows *Prefixes p* $\subseteq$ *paths-le* (*charslength p*) *P*
using *Prefixes-is-prefix assms charslength-of-prefix paths-le-def* **by** *auto*

lemma *Scan-*$\mathcal{J}$*-subset-*$\mathcal{J}$:
Scan ($\mathcal{T}$ *k* (*Suc u*)) *k* ($\mathcal{J}$ *k u*) $\subseteq \mathcal{J}$ *k* (*Suc u*)
by (*metis* (*no-types*, *lifting*) *Scan-*π*-fix* $\mathcal{J}$*.simps*(*2*) $\mathcal{J}$*-subset-Suc-u monoD mono-Scan*)

lemma *subset-*$\mathcal{J}$*k*: $u \leq v \Longrightarrow \mathcal{J}\ k\ u \subseteq \mathcal{J}\ k\ v$
thm $\mathcal{J}$*-subset-Suc-u*
by (*rule subset-fSuc*, *rule* $\mathcal{J}$*-subset-Suc-u*)

lemma *subset-*$\mathcal{J}\mathcal{I}$*k*: $\mathcal{J}\ k\ u \subseteq \mathcal{I}\ k$ **by** (*auto simp add*: *natUnion-def*)

lemma *subset-*$\mathcal{I}\mathcal{J}$*Suc*: $\mathcal{I}\ k \subseteq \mathcal{J}$ (*Suc k*) *u*
proof −
have *a*: $\mathcal{I}\ k \subseteq \mathcal{J}$ (*Suc k*) *0*
apply (*simp only*: $\mathcal{J}$*.simps*)
using π*-apply-setmonotone* **by** *blast*
show *?thesis*
apply (*case-tac u* = *0*)
apply (*simp only*: *a*)
apply (*rule subset-trans*[*OF a subset-*$\mathcal{J}$*k*])
by *auto*
qed

lemma *subset-*$\mathcal{I}$*Suc*: $\mathcal{I}\ k \subseteq \mathcal{I}$ (*Suc k*)
by (*rule subset-trans*[*OF subset-*$\mathcal{I}\mathcal{J}$*Suc subset-*$\mathcal{J}\mathcal{I}$*k*])

lemma *subset-*$\mathcal{I}$: $i \leq j \Longrightarrow \mathcal{I}\ i \subseteq \mathcal{I}\ j$
by (*rule subset-fSuc*[**where** *u*=*i* **and** *v*=*j* **and** *f* = $\mathcal{I}$, *OF subset-*$\mathcal{I}$*Suc*])

lemma *subset-𝒥* :
  assumes *leq*: $k' < k \vee (k' = k \wedge u' \leq u)$
  shows $\mathcal{J}\ k'\ u' \subseteq \mathcal{J}\ k\ u$
proof −
  from *leq* **show** *?thesis*
  proof (*induct rule*: *disjCases2*)
    case *1*
    have *s1*: $\mathcal{J}\ k'\ u' \subseteq \mathcal{I}\ k'$ **by** (*rule-tac subset-𝒥ℐk*)
    have *s2*: $\mathcal{I}\ k' \subseteq \mathcal{I}\ (k - 1)$
      apply (*rule-tac subset-ℐ*)
      using *1* **by** *arith*
    from *subset-ℐ𝒥Suc*[**where** $k{=}k - 1$] *1* **have** *s3*: $\mathcal{I}\ (k - 1) \subseteq \mathcal{J}\ k\ 0$
      by *simp*
    have *s4*: $\mathcal{J}\ k\ 0 \subseteq \mathcal{J}\ k\ u$ **by** (*rule-tac subset-𝒥k*, *simp*)
    from *s1 s2 s3 s4 subset-trans* **show** *?case* **by** *blast*
  next
    case *2* **thus** *?case* **by** (*simp add* : *subset-𝒥k*)
  qed
qed

lemma *𝒥-subset*:
  assumes *indexle* $k'\ u'\ k\ u$
  shows $\mathcal{J}\ k'\ u' \subseteq \mathcal{J}\ k\ u$
using *subset-𝒥 indexle-def indexlt-simp*
by (*metis assms less-imp-le-nat order-refl*)

lemma *Scan-items-le*:
  assumes *bounded-T*: $\bigwedge\ t\ .\ t \in T \Longrightarrow$ *length* (*chars-of-token t*) $\leq l$
  shows *Scan T k* (*items-le k P*) $\subseteq$ *items-le* $(k + l)$ (*Scan T k P*)
proof −
  {
    fix *x* :: *item*
    assume *x-dom*: $x \in$ *Scan T k* (*items-le k P*)
    then have *x-dom′*: $x \in$ *Scan T k P*
      by (*meson items-le-is-filter mono-Scan mono-subset-elem*)
    from *x-dom* **have** $x \in$ *items-le k P* $\vee$
      ($\exists$ *y t c*. $x =$ *inc-item y* $(k +$ *length c*$)$ $\wedge$ $y \in$ *bin* (*items-le k P*) $k \wedge (t, c) \in$
T
       $\wedge$ *next-symbol* $y =$ *Some t*)
      using *Scan-def* **using** *UnE mem-Collect-eq* **by** *auto*
    then have *item-end* $x \leq k + l$
    proof (*induct rule*: *disjCases2*)
      case *1* **then show** *?case*
        by (*meson items-le-fix-D items-le-idempotent trans-le-add1*)
    next
      case *2*
        then obtain *y t c* **where** *y*: $x =$ *inc-item y* $(k +$ *length c*$)$ $\wedge$ $y \in$ *bin*
(*items-le k P*) $k \wedge$
        $(t, c) \in T \wedge$ *next-symbol* $y =$ *Some t* **by** *blast*

then have *item-end-x*: *item-end* $x = (k + \textit{length } c)$ **by** *simp*
from *bounded-T y* **have** *length* $c \le l$
using *chars-of-token-simp* **by** *auto*
with *item-end-x* **show** *?case* **by** *arith*
qed
with *x-dom′* **have** $x \in$ *items-le* $(k + l)$ (*Scan T k P*)
using *items-le-def mem-Collect-eq* **by** *blast*
}
then show *?thesis* **by** *blast*
qed

lemma *Scan-mono-tokens*:
$P \subseteq Q \Longrightarrow$ *Scan P k I* $\subseteq$ *Scan Q k I*
by (*auto simp add*: *Scan-def*)

theorem *thmD14*: $k \le$ *length Doc* $\Longrightarrow$ *items-le k* $(\mathcal{J}\ k\ u)$ = *Gen* (*paths-le k* $(\mathcal{P}\ k\ u)) \wedge \mathcal{T}\ k\ u = \mathcal{Z}\ k\ u$
$\wedge$ *items-le k* $(\mathcal{I}\ k)$ = *Gen* (*paths-le k* $(\mathcal{Q}\ k)$)
proof (*induct k arbitrary*: *u rule*: *less-induct*)
case (*less k*)
have $k = 0 \vee k \neq 0$ **by** *arith*
then show *?case*
proof (*induct rule*: *disjCases2*)
case *1*
have $\mathcal{J}$*-eq-*$\mathcal{P}$: *items-le k* $(\mathcal{J}\ k\ 0)$ = *Gen* (*paths-le k* $(\mathcal{P}\ k\ 0)$)
by (*simp only*: *1 thmD8 items-le-paths-le*)
show *?case* **using** *thmD13*[*OF* $\mathcal{J}$*-eq-*$\mathcal{P}$ *less.prems*] **by** *blast*
next
case *2*
have $\exists\ k'.\ k = Suc\ k'$ **using** *2* **by** *arith*
then obtain k' **where** k': $k = Suc\ k'$ **by** *blast*
have k'*-less-k*: $k' < k$ **using** k' **by** *arith*
have *items-le k* $(\mathcal{J}\ k\ 0)$ = *Gen* (*paths-le k* $(\mathcal{P}\ k\ 0)$)
proof −
have *simp-left*: *items-le k* $(\mathcal{J}\ k\ 0) = \pi\ k\ \{\}$ (*items-le k* $(\mathcal{I}\ k')$)
using *items-le-*π*-swap* k' *wellformed-items-*$\mathcal{I}$ **by** *auto*
have *simp-right*: *Gen* (*paths-le k* $(\mathcal{P}\ k\ 0)$) = *natUnion* ($\lambda\ v.$ *Gen* (*paths-le k* $(\mathcal{P}\ k'\ v)$))
by (*simp add*: k' *paths-le-pointwise pointwise-Gen pointwise-natUnion-swap*)

{
fix v :: *nat*
have *split-*$\mathcal{J}$: *items-le k* $(\mathcal{J}\ k'\ v)$ = *items-le* k' $(\mathcal{J}\ k'\ v) \cup$ *items-eq k* $(\mathcal{J}\ k'\ v)$
using k' *items-le-split-via-eq* **by** *blast*
have *sub1*: *items-le* k' $(\mathcal{J}\ k'\ v) \subseteq$ *natUnion* ($\lambda\ v.$ *Gen* (*paths-le k* $(\mathcal{P}\ k'\ v)$))
proof −
have *h*: *items-le* k' $(\mathcal{J}\ k'\ v) \subseteq$ *Gen* (*paths-le k* $(\mathcal{P}\ k'\ v)$)

proof −
have *f1*: $items\text{-}le\ k'\ (Gen\ (\mathcal{P}\ k'\ v)) \cup items\text{-}eq\ (Suc\ k')\ (Gen\ (\mathcal{P}\ k'\ v)) =$
$Gen\ (paths\text{-}le\ k\ (\mathcal{P}\ k'\ v))$
using *LocalLexing.items-le-split-via-eq LocalLexing-axioms items-le-paths-le* k'
by *blast*
have $k' \leq length\ Doc$
by (*metis* (*no-types*) *dual-order.trans* k' *less.prems lessI less-imp-le-nat*)
then have $items\text{-}le\ k'\ (\mathcal{J}\ k'\ v) = items\text{-}le\ k'\ (Gen\ (\mathcal{P}\ k'\ v))$
by (*simp add*: *items-le-paths-le* k' *less.hyps*)
then show *?thesis*
using *f1* **by** *blast*
qed
have $Gen\ (paths\text{-}le\ k\ (\mathcal{P}\ k'\ v)) \subseteq natUnion\ (\lambda\ v.\ Gen\ (paths\text{-}le\ k\ (\mathcal{P}\ k'\ v)))$
using *natUnion-superset* **by** *fastforce*
then show *?thesis* **using** *h* **by** *blast*
qed
{
fix $x :: item$
assume *x-dom*: $x \in items\text{-}eq\ k\ (\mathcal{J}\ k'\ v)$
have *x-in-*$\mathcal{J}$: $x \in \mathcal{J}\ k'\ v$ **using** *x-dom items-eq-def* **by** *auto*
have *item-end-x*: $item\text{-}end\ x = k$ **using** *x-dom items-eq-def* **by** *auto*
then have k'*-bound*: $k' < item\text{-}end\ x$ **using** k' **by** *arith*
from *produced-by-scan-step*[*OF x-in-*$\mathcal{J}$ k'*-bound*]
have *produced-by-scan-step* $x\ k'\ v$ **.**
from *iffD1*[*OF produced-by-scan-step-def this*] **obtain** $k''\ v''\ y\ X$ **where** *scan-step*:
$indexle\ k''\ v''\ k'\ v \wedge y \in \mathcal{J}\ k''\ v'' \wedge item\text{-}end\ y = k'' \wedge X \in \mathcal{T}\ k''\ v'' \wedge$
$x = inc\text{-}item\ y\ (k'' + length\ (chars\text{-}of\text{-}token\ X)) \wedge$
$next\text{-}symbol\ y = Some\ (terminal\text{-}of\text{-}token\ X)$ **by** *blast*
then have *y-in-items-le*: $y \in items\text{-}le\ k''\ (\mathcal{J}\ k''\ v'')$
using *items-le-def LocalLexing-axioms le-refl mem-Collect-eq* **by** *blast*
have *y-in-Gen*: $y \in Gen(paths\text{-}le\ k''\ (\mathcal{P}\ k''\ v''))$
proof −
have *f1*: $\bigwedge n.\ k' < n \vee \neg\ k < n$
using *Suc-lessD* k' **by** *blast*
have *f2*: $k'' = k' \vee k'' < k'$
using *indexle-def indexlt-simp scan-step* **by** *force*
have *f3*: $k' < k$
using k' **by** *blast*
have *f4*: $k' \leq length\ Doc$
using *f1* **by** (*meson less.prems less-Suc-eq-le*)
have $k'' \leq length\ Doc \vee k' = k''$
using *f2 f1* **by** (*meson Suc-lessD less.prems less-Suc-eq-le less-trans-Suc*)
then show *?thesis*

using *f4 f3 f2 Suc-lessD y-in-items-le less.hyps less-trans-Suc* **by** *blast*
qed
then have ∃ *p*. *p* ∈ 𝒫 *k''* *v''* ∧ *pvalid p y*
by (*meson Gen-implies-pvalid paths-le-is-filter rev-subsetD*)
then obtain *p* **where** *p*: *p* ∈ 𝒫 *k''* *v''* ∧ *pvalid p y* **by** *blast*
then have *charslength-p*: *charslength p* = *k''* **using** *pvalid-item-end scan-step* **by** *auto*
have *pvalid-p-y*: *pvalid p y* **using** *p* **by** *blast*
have *admissible* (*p*@[(*fst X*, *snd X*)])
apply (*rule pvalid-next-terminal-admissible*)
apply (*rule pvalid-p-y*)
using *scan-step* **apply** (*simp add*: *terminal-of-token-def*)
using *scan-step* **by** (*metis TokensAt-subset-𝒳 𝒯-subset-TokensAt 𝒳-are-terminals*
rev-subsetD terminal-of-token-def)
then have *admissible-p-X*: *admissible* (*p*@[*X*]) **by** *simp*
have *X-in-𝒵*: *X* ∈ 𝒵 *k''* (*Suc v''*) **by** (*metis* (*no-types*, *lifting*) *Suc-lessD 𝒵-subset-Suc*
k'-bound dual-order.trans indexle-def indexlt-simp item-end-of-inc-item item-end-x
le-add1 le-neq-implies-less less.hyps less.prems not-less-eq scan-step subsetCE)
have *pX-in-𝒫-k''-v''*: *p*@[*X*] ∈ 𝒫 *k''* (*Suc v''*)
apply (*simp only*: *𝒫.simps*)
apply (*rule limit-single-step*)
apply (*auto simp only*: *Append-def*)
apply (*rule-tac x*=*p* **in** *exI*)
apply (*rule-tac x*=*X* **in** *exI*)
apply (*simp only*: *admissible-p-X X-in-𝒵*)
using *charslength-p p* **by** *auto*
have *indexle k'' v'' k' v* **using** *scan-step* **by** *simp*
then have *indexle k''* (*Suc v''*) *k'* (*Suc v*)
by (*simp add*: *indexle-def indexlt-simp*)
then have 𝒫 *k''* (*Suc v''*) ⊆ 𝒫 *k'* (*Suc v*)
by (*metis indexle-def indexlt-simp less-or-eq-imp-le subset-𝒫*)
with *pX-in-𝒫-k''-v''* **have** *pX-in-𝒫-k'*: *p*@[*X*] ∈ 𝒫 *k'* (*Suc v*) **by** *blast*
have *charslength* (*p*@[*X*]) = *k''* + *length* (*chars-of-token X*)
using *charslength-p* **by** *auto*
then have *charslength* (*p*@[*X*]) = *item-end x* **using** *scan-step* **by** *simp*
then have *charslength-p-X*: *charslength* (*p*@[*X*]) = *k* **using** *item-end-x* **by** *simp*
then have *pX-dom*: *p*@[*X*] ∈ *paths-le k* (𝒫 *k'* (*Suc v*))
using *lessI less-Suc-eq-le mem-Collect-eq pX-in-𝒫-k' paths-le-def* **by** *auto*
have *wellformed-x*: *wellformed-item x*
using *item-end-x less.prems scan-step wellformed-inc-item wellformed-items-𝒥*
wellformed-items-def **by** *auto*

```
have wellformed-p-X: wellformed-tokens (p@[X])
  using 𝒫-wellformed pX-in-𝒫-k''-v'' by blast
from iffD1[OF pvalid-def pvalid-p-y] obtain r γ where r-γ:
  wellformed-tokens p ∧
   wellformed-item y ∧
   r ≤ length p ∧
   charslength p = item-end y ∧
   charslength (take r p) = item-origin y ∧
   is-derivation (terminals (take r p) @ [item-nonterminal y] @ γ) ∧
   derives (item-α y) (terminals (drop r p)) by blast
have r-le-p: r ≤ length p by (simp add: r-γ)
have item-nonterminal-x: item-nonterminal x = item-nonterminal y
  by (simp add: scan-step)
have item-α-x: item-α x = (item-α y) @ [terminal-of-token X]
  by (simp add: item-α-of-inc-item r-γ scan-step)
have pvalid-x: pvalid (p@[X]) x
  apply (auto simp add: pvalid-def wellformed-x wellformed-p-X)
  apply (rule-tac x=r in exI)
  apply auto
  apply (simp add: le-SucI r-γ)
  using r-γ scan-step apply auto[1]
  using r-γ scan-step apply auto[1]
  apply (rule-tac x=γ in exI)
  apply (simp add: r-le-p item-nonterminal-x)
  using r-γ apply simp
  apply (simp add: r-le-p item-α-x)
  by (metis terminals-singleton append-Nil2
    derives-implies-leftderives derives-is-sentence is-sentence-concat
          is-sentence-cons is-symbol-def is-word-append is-word-cons
is-word-terminals
    is-word-terminals-drop leftderives-implies-derives leftderives-padback
    leftderives-refl r-γ terminals-append terminals-drop wellformed-p-X)
then have x ∈ Gen (paths-le k (𝒫 k' (Suc v))) using pX-dom Gen-def
  LocalLexing-axioms mem-Collect-eq by auto
}
then have sub2: items-eq k (𝒥 k' v) ⊆ natUnion (λ v. Gen (paths-le k
(𝒫 k' v)))
by (meson dual-order.trans natUnion-superset subsetI)
have suffices3: items-le k (𝒥 k' v) ⊆ natUnion (λ v. Gen (paths-le k (𝒫
k' v)))
  using split-𝒥 sub1 sub2 by blast
have items-le k (𝒥 k' v) ⊆ Gen (paths-le k (𝒫 k 0))
  using suffices3 simp-right by blast
}
note suffices2 = this
have items-le-natUnion-swap: items-le k (ℐ k') = natUnion(λ v. items-le
k (𝒥 k' v))
  by (simp add: items-le-pointwise pointwise-natUnion-swap)
then have suffices1: items-le k (ℐ k') ⊆ Gen (paths-le k (𝒫 k 0))
```

using *suffices2 natUnion-upperbound* **by** *metis*
have *sub-lemma*: *items-le* k ($\mathcal{J}$ k *0*) $\subseteq$ *Gen* (*paths-le* k ($\mathcal{P}$ k *0*))
proof −
have *items-le* k ($\mathcal{J}$ k *0*) $\subseteq$ *Gen* ($\mathcal{P}$ k *0*)
apply (*subst simp-left*)
apply (*rule thmD5*)
apply (*auto simp only*: *less*)
using *suffices1 items-le-is-filter items-le-paths-le subsetCE* **by** *blast*
then show *?thesis*
by (*simp add*: *items-le-idempotent remove-paths-le-in-subset-Gen*)
qed
have *eq1*: π k {} (*items-le* k ($\mathcal{I}$ k')) = π k {} (*items-le* k (*natUnion* ($\mathcal{J}$ k'))) **by** *simp*
then have *eq2*: π k {} (*items-le* k (*natUnion* ($\mathcal{J}$ k'))) =
π k {} (*natUnion* (λ v. *items-le* k ($\mathcal{J}$ k' v)))
using *items-le-natUnion-swap* **by** *auto*
from *simp-left eq1 eq2*
have *simp-left'*: *items-le* k ($\mathcal{J}$ k *0*) = π k {} (*natUnion* (λ v. *items-le* k ($\mathcal{J}$ k' v)))
by *metis*
{
fix v :: *nat*
fix q :: *token list*
fix x :: *item*
assume *q-dom*: $q \in$ *paths-eq* k ($\mathcal{P}$ k' v)
assume *pvalid-q-x*: *pvalid* q x
have *q-in-*$\mathcal{P}$: $q \in \mathcal{P}$ k' v **using** *q-dom paths-eq-def* **by** *auto*
have *charslength-q*: *charslength* $q = k$ **using** *q-dom paths-eq-def* **by** *auto*
with k'*-less-k* **have** *q-nonempty*: $q \neq$ []
using *2.hyps chars.simps*(*1*) *charslength.simps list.size*(*3*) **by** *auto*
then have $\exists$ p X. $q = p$ @ [X] **by** (*metis append-butlast-last-id*)
then obtain p X **where** *pX*: $q = p$ @ [X] **by** *blast*
from *last-step-of-path*[*OF q-in-*$\mathcal{P}$ *pX*] **obtain** k'' v'' **where** k'':
indexlt k'' v'' k' v $\wedge$ $q \in \mathcal{P}$ k'' (*Suc* v'') $\wedge$ *charslength* $p = k''$ $\wedge$
$X \in \mathcal{Z}$ k'' (*Suc* v'') **by** *blast*
have *h1*: $p \in \mathfrak{P}$
by (*metis* (*no-types*, *lifting*) *LocalLexing.*$\mathfrak{P}$*-covers-*$\mathcal{P}$ *LocalLexing-axioms*

append-Nil2 is-prefix-cancel is-prefix-empty pX prefixes-are-paths q-in-$\mathcal{P}$ *subsetCE*)
have *h2*: *charslength* $p = k''$ **using** k'' **by** *blast*
obtain T **where** T: $T = \{X\}$ **by** *blast*
have *h3*: $X \in T$ **using** T **by** *blast*
have *h4*: $T \subseteq \mathcal{X}$ k'' **using** $\mathcal{Z}$*-subset-*$\mathcal{X}$ T k'' **by** *blast*
obtain N **where** N: N = *item-nonterminal* x **by** *blast*
obtain α **where** α: α = *item-*α x **by** *blast*
obtain β **where** β: β = *item-*β x **by** *blast*
have *wellformed-x*: *wellformed-item* x **using** *pvalid-def pvalid-q-x* **by** *blast*

then have $h5$: $(N, \alpha @ \beta) \in \mathfrak{R}$
using N α β *item-nonterminal-def item-rhs-def item-rhs-split prod.collapse*

wellformed-item-def **by** *auto*
have *pvalid-left-q-x*: *pvalid-left q x* **using** *pvalid-q-x* **by** (*simp add*: *pvalid-left*)
from *iffD1*[*OF pvalid-left-def pvalid-left-q-x*] **obtain** r γ **where** r-γ:
wellformed-tokens q $\wedge$
wellformed-item x $\wedge$
$r \leq$ *length q* $\wedge$
charslength q = *item-end x* $\wedge$
charslength (*take r q*) = *item-origin x* $\wedge$
is-leftderivation (*terminals* (*take r q*) @ [*item-nonterminal x*] @ γ) $\wedge$
leftderives (*item-*α *x*) (*terminals* (*drop r q*)) **by** *blast*
have *h6*: $r \leq$ *length q* **using** r-γ **by** *blast*
have *h7*: *leftderives* [$\mathfrak{S}$] (*terminals* (*take r q*) @ [N] @ γ)
using r-γ N *is-leftderivation-def* **by** *blast*
have *h8*: *leftderives* α (*terminals* (*drop r q*)) **using** r-γ α **by** *metis*
have *h9*: $k = k'' +$ *length* (*chars-of-token X*) **using** r-γ
using *charslength-q h2 pX* **by** *auto*
have *h10*: x = *Item* (N, α @ β) (*length* α) (*charslength* (*take r q*)) k
by (*metis* N α β *charslength-q item.collapse item-dot-is-*α*-length item-nonterminal-def*
item-rhs-def item-rhs-split prod.collapse r-γ)
from *thmD11*[*OF h1 h2 h3 h4 pX h5 h6 h7 h8 h9 h10*]
have $x \in$ *items-le* k (π k {} (*Scan T* k'' (*Gen* (*Prefixes p*))))
by *blast*
then have *x-in*: $x \in \pi$ k {} (*Scan T* k'' (*Gen* (*Prefixes p*)))
using *items-le-is-filter* **by** *blast*
have *subset1*: *Prefixes p* $\subseteq$ *Prefixes q*
apply (*rule is-prefix-Prefixes-subset*)
by (*simp add*: *pX is-prefix-def*)
have *subset2*: *Prefixes q* $\subseteq \mathcal{P}$ k'' (*Suc* v'')
apply (*rule Prefixes-subset-*$\mathcal{P}$)
using k'' **by** *blast*
from *subset1 subset2* **have** *Prefixes p* $\subseteq \mathcal{P}$ k'' (*Suc* v'') **by** *blast*
then have *Prefixes p* $\subseteq$ *paths-le* k'' ($\mathcal{P}$ k'' (*Suc* v''))
using k'' *Prefixes-subset-paths-le* **by** *blast*
then have *subset3*: *Gen* (*Prefixes p*) $\subseteq$ *Gen* (*paths-le* k'' ($\mathcal{P}$ k'' (*Suc* v'')))
using *Gen-def LocalLexing-axioms* **by** *auto*
have k''*-less-k*: $k'' < k$ **using** k'' k' **using** *indexlt-simp less-Suc-eq* **by** *auto*
then have k''*-Doc-bound*: $k'' \leq$ *length Doc* **using** *less* **by** *auto*
from *less*(*1*)[*OF* k''*-less-k* k''*-Doc-bound*, *of Suc* v'']
have *induct1*: *items-le* k'' ($\mathcal{J}$ k'' (*Suc* v'')) = *Gen* (*paths-le* k'' ($\mathcal{P}$ k'' (*Suc* v'')))
by *blast*
from *less*(*1*)[*OF* k''*-less-k* k''*-Doc-bound*, *of Suc*(*Suc* v'')]

have *induct2*: $\mathcal{T}$ k'' (*Suc* (*Suc* v'')) = $\mathcal{Z}$ k'' (*Suc* (*Suc* v'')) **by** *blast*
have *subset4*: *Gen* (*Prefixes p*) $\subseteq$ *items-le* k'' ($\mathcal{J}$ k'' (*Suc* v''))
using *subset3 induct1* **by** *auto*
from *induct1 subset4*
have *subset6*: *Scan T* k'' (*Gen* (*Prefixes p*)) $\subseteq$
Scan T k'' (*items-le* k'' ($\mathcal{J}$ k'' (*Suc* v'')))
apply (*rule-tac monoD*[*OF mono-Scan*])
by *blast*
have k'' + *length* (*chars-of-token X*) = *k*
by (*simp add*: *h9*)
have $\bigwedge$ *t*. *t* $\in$ *T* $\Longrightarrow$ *length* (*chars-of-token t*) $\leq$ *length* (*chars-of-token X*)
using *T* **by** *auto*
from *Scan-items-le*[*of T*, *OF this*, *simplified*, *of* k'' $\mathcal{J}$ k'' (*Suc* v'')] *h9*
have *subset7*: *Scan T* k'' (*items-le* k'' ($\mathcal{J}$ k'' (*Suc* v'')))
$\subseteq$ *items-le k* (*Scan T* k'' ($\mathcal{J}$ k'' (*Suc* v''))) **by** *simp*
have *T* $\subseteq$ $\mathcal{Z}$ k'' (*Suc* (*Suc* v'')) **using** *T* k''
using $\mathcal{Z}$*-subset-Suc rev-subsetD singletonD subsetI* **by** *blast*
then have *T-subset-*$\mathcal{T}$: *T* $\subseteq$ $\mathcal{T}$ k'' (*Suc* (*Suc* v'')) **using** *induct2* **by** *auto*
have *subset8*: *Scan T* k'' ($\mathcal{J}$ k'' (*Suc* v'')) $\subseteq$
Scan ($\mathcal{T}$ k'' (*Suc* (*Suc* v''))) k'' ($\mathcal{J}$ k'' (*Suc* v''))
using *T-subset-*$\mathcal{T}$ *Scan-mono-tokens* **by** *blast*
have *subset9*: *Scan* ($\mathcal{T}$ k'' (*Suc* (*Suc* v''))) k'' ($\mathcal{J}$ k'' (*Suc* v'')) $\subseteq$ $\mathcal{J}$ k'' (*Suc* (*Suc* v''))
by (*rule Scan-*$\mathcal{J}$*-subset-*$\mathcal{J}$)
have *subset10*: (*Scan T* k'' ($\mathcal{J}$ k'' (*Suc* v''))) $\subseteq$ $\mathcal{J}$ k'' (*Suc* (*Suc* v''))
using *subset8 subset9* **by** *blast*
have $k'' \leq k'$ **using** k'' *indexlt-simp* **by** *auto*
then have *indexle* k'' (*Suc* (*Suc* v'')) k' (*Suc* (*Suc* v'')) **using** *indexlt-simp*
using *indexle-def le-neq-implies-less* **by** *auto*
then have *subset11*: $\mathcal{J}$ k'' (*Suc* (*Suc* v'')) $\subseteq$ $\mathcal{J}$ k' (*Suc* (*Suc* v''))
using $\mathcal{J}$*-subset* **by** *blast*
have *subset12*: *Scan T* k'' ($\mathcal{J}$ k'' (*Suc* v'')) $\subseteq$ $\mathcal{J}$ k' (*Suc* (*Suc* v''))
using *subset8 subset9 subset10 subset11* **by** *blast*
then have *subset13*: *items-le k* (*Scan T* k'' ($\mathcal{J}$ k'' (*Suc* v''))) $\subseteq$
items-le k ($\mathcal{J}$ k' (*Suc* (*Suc* v'')))
using *items-le-def mem-Collect-eq rev-subsetD subsetI* **by** *auto*
have *subset14*: *Scan T* k'' (*Gen* (*Prefixes p*)) $\subseteq$ *items-le k* ($\mathcal{J}$ k' (*Suc* (*Suc* v'')))
using *subset6 subset7 subset13* **by** *blast*
then have *x-in'*: $x \in \pi$ *k* {} (*items-le k* ($\mathcal{J}$ k' (*Suc* (*Suc* v''))))
using *x-in*
by (*meson* π*-apply-setmonotone* π*-subset-elem-trans subsetCE subsetI*)
from *x-in'* **have** $x \in \pi$ *k* {} (*natUnion* (λ *v*. *items-le k* ($\mathcal{J}$ k' *v*)))
by (*meson* k' *mono-*π *mono-subset-elem natUnion-superset*)
}
note *suffices6* = *this*
{
fix *v* :: *nat*

have *Gen* (*paths-eq k* ($\mathcal{P}$ *k′ v*)) ⊆ π *k* {} (*natUnion* (λ *v*. *items-le k* ($\mathcal{J}$ *k′ v*)))
using *suffices6* **by** (*meson Gen-implies-pvalid subsetI*)
}
note *suffices5* = *this*
{
fix *v* :: *nat*
have *paths-le k* ($\mathcal{P}$ *k′ v*) = *paths-le k′* ($\mathcal{P}$ *k′ v*) ∪ *paths-eq k* ($\mathcal{P}$ *k′ v*)
using *paths-le-split-via-eq k′* **by** *metis*
then have *Gen-split*: *Gen* (*paths-le k* ($\mathcal{P}$ *k′ v*)) =
Gen (*paths-le k′* ($\mathcal{P}$ *k′ v*)) ∪ *Gen*(*paths-eq k* ($\mathcal{P}$ *k′ v*)) **using** *Gen-union* **by** *metis*
have *case-le*: *Gen* (*paths-le k′* ($\mathcal{P}$ *k′ v*)) ⊆ π *k* {} (*natUnion* (λ *v*. *items-le k* ($\mathcal{J}$ *k′ v*)))
proof −
from *less k′-less-k* **have** *k′* ≤ *length Doc* **by** *arith*
from *less*(*1*)[*OF k′-less-k this*]
have *items-le k′* ($\mathcal{J}$ *k′ v*) = *Gen* (*paths-le k′* ($\mathcal{P}$ *k′ v*)) **by** *blast*
then have *Gen* (*paths-le k′* ($\mathcal{P}$ *k′ v*)) ⊆ *natUnion* (λ *v*. *items-le k* ($\mathcal{J}$ *k′ v*))
using *items-le-def LocalLexing-axioms k′-less-k natUnion-superset* **by** *fastforce*
then show *?thesis* **using** *π-apply-setmonotone* **by** *blast*
qed
have *Gen* (*paths-le k* ($\mathcal{P}$ *k′ v*)) ⊆ π *k* {} (*natUnion* (λ *v*. *items-le k* ($\mathcal{J}$ *k′ v*)))
using *Gen-split case-le suffices5 UnE rev-subsetD subsetI* **by** *blast*
}
note *suffices4* = *this*
have *super-lemma*: *Gen* (*paths-le k* ($\mathcal{P}$ *k 0*)) ⊆ *items-le k* ($\mathcal{J}$ *k 0*)
apply (*subst simp-right*)
apply (*subst simp-left′*)
using *suffices4* **by** (*meson natUnion-ex rev-subsetD subsetI*)
from *super-lemma sub-lemma* **show** *?thesis* **by** *blast*
qed
then show *?case* **using** *thmD13 less.prems* **by** *blast*
qed
qed

end

end
theory *PathLemmas*
imports *TheoremD14*
begin

context *LocalLexing* **begin**

lemma *characterize-*$\mathcal{P}$:

(∀ *i* < *length p*. ∃ *u*. *p* ! *i* ∈ 𝒵 (*charslength* (*take i p*)) *u*) ⟹ *admissible p* ⟹
∃ *u*. *p* ∈ 𝒫 (*charslength p*) *u*
proof (*induct p rule*: *rev-induct*)
case *Nil*
show *?case* **by** *simp*
next
case (*snoc a p*)
from *snoc.prems* **have** *admissible-p*: *admissible p*
by (*metis append-Nil2 is-prefix-admissible is-prefix-cancel is-prefix-empty*)
{
fix *i* :: *nat*
assume *ilen*: *i* < *length p*
then have *i* < *length* (*p*@[*a*])
by (*simp add*: *Suc-leI Suc-le-lessD trans-le-add1*)
with *snoc* **have** ∃ *u*. (*p* @ [*a*]) ! *i* ∈ 𝒵 (*charslength* (*take i* (*p* @ [*a*]))) *u*
by *blast*
then obtain *u* **where** *u*: (*p* @ [*a*]) ! *i* ∈ 𝒵 (*charslength* (*take i* (*p* @ [*a*]))) *u*
by *blast*
from *ilen* **have** *p-at*: (*p* @ [*a*]) ! *i* = *p* ! *i* **by** (*simp add*: *nth-append*)
from *ilen* **have** *p-take*: *take i* (*p* @ [*a*]) = *take i p* **by** (*simp add*: *less-or-eq-imp-le*)

from *u p-at p-take* **have** *p-i*: *p* ! *i* ∈ 𝒵 (*charslength* (*take i p*)) *u* **by** *simp*
then have ∃ *u*. *p* ! *i* ∈ 𝒵 (*charslength* (*take i p*)) *u* **by** *blast*
}
then have ∀ *i* < *length p*. ∃ *u*. *p* ! *i* ∈ 𝒵 (*charslength* (*take i p*)) *u* **by** *auto*
with *admissible-p snoc.hyps* **obtain** *u* **where** *u*: *p* ∈ 𝒫 (*charslength p*) *u* **by**
blast
have ∃ *u*. (*p* @ [*a*]) ! (*length p*) ∈ 𝒵 (*charslength* (*take* (*length p*) (*p* @ [*a*]))) *u*
using *snoc*
by (*metis* (*no-types*, *opaque-lifting*) *add-Suc-right append-Nil2 length-Cons*
length-append
less-Suc-eq-le less-or-eq-imp-le)
then obtain *v* **where** (*p* @ [*a*]) ! (*length p*) ∈ 𝒵 (*charslength* (*take* (*length p*)
(*p* @ [*a*]))) *v*
by *blast*
then have *v*: *a* ∈ 𝒵 (*charslength p*) *v* **by** *simp*
{
assume *v-leq-u*: *v* ≤ *u*
then have *a* ∈ 𝒵 (*charslength p*) (*Suc u*) **using** *v*
by (*meson LocalLexing.subset-fSuc LocalLexing-axioms 𝒵-subset-Suc sub-
setCE*)
from *path-append-token*[*OF u this snoc.prems*(*2*)]
have *p* @ [*a*] ∈ 𝒫 (*charslength p*) (*Suc u*) **by** *blast*
then have *?case* **using** *in-𝒫-charslength* **by** *blast*
}
note *case-v-leq-u* = *this*
{
assume *u-less-v*: *u* < *v*
then obtain *w* **where** *w*: *v* = *Suc w* **using** *less-imp-Suc-add* **by** *blast*

with *u-less-v* **have** $u \leq w$ **by** *arith*
with *u* **have** $p \in \mathcal{P}$ (*charslength p*) *w* **by** (*meson subsetCE subset-*$\mathcal{P}$*k*)
from *v w path-append-token*[*OF this - snoc.prems*(*2*)]
have *p* @ [*a*] $\in \mathcal{P}$ (*charslength p*) (*Suc w*) **by** *blast*
then have *?case* **using** *in-*$\mathcal{P}$*-charslength* **by** *blast*
}
note *case-u-less-v* = *this*

show *?case* **using** *case-v-leq-u case-u-less-v not-le* **by** *blast*
qed

lemma *drop-empty-tokens*:
assumes *p*: $p \in \mathfrak{P}$
assumes *r*: $r \leq$ *length p*
assumes *empty*: *charslength* (*take r p*) = *0*
assumes *admissible*: *admissible* (*drop r p*)
shows *drop r p* $\in \mathfrak{P}$
proof −
have *p-*$\mathcal{Z}$: $\forall$ *i*<*length p*. $\exists$ *u*. *p* ! *i* $\in \mathcal{Z}$ (*charslength* (*take i p*)) *u* **using** *p*
using *tokens-nth-in-*$\mathcal{Z}$ **by** *blast*
obtain *q* **where** *q*: *q* = *drop r p* **by** *blast*
{
fix *j* :: *nat*
assume *j* : *j* < *length q*
have *length-p-q-r*: *length p* = *length q* + *r*
using *r q add.commute diff-add-inverse le-Suc-ex length-drop* **by** *simp*
have *j-plus-r-bound*: *j* + *r* < *length p* **by** (*simp add*: *j length-p-q-r*)
with *p-*$\mathcal{Z}$ **have** $\exists$ *u*. *p* ! (*j* + *r*) $\in \mathcal{Z}$ (*charslength* (*take* (*j* + *r*) *p*)) *u* **by** *blast*
then obtain *u* **where** *u*: *p* ! (*j* + *r*) $\in \mathcal{Z}$ (*charslength* (*take* (*j* + *r*) *p*)) *u* **by** *blast*
have *p-at-is-q-at*: *p* ! (*j* + *r*) = *q* ! *j* **by** (*simp add*: *add.commute q r*)
have *take* (*j* + *r*) *p* = (*take r p*) @ (*take j q*) **by** (*metis add.commute q take-add*)
with *empty* **have** *charslength* (*take* (*j* + *r*) *p*) = *charslength* (*take j q*) **by** *auto*
with *u p-at-is-q-at* **have** *q* ! *j* $\in \mathcal{Z}$ (*charslength* (*take j q*)) *u* **by** *simp*
then have $\exists$ *u*. *q* ! *j* $\in \mathcal{Z}$ (*charslength* (*take j q*)) *u* **by** *auto*
}
then have $\forall$ *i*<*length q*. $\exists$ *u*. *q* ! *i* $\in \mathcal{Z}$ (*charslength* (*take i q*)) *u* **by** *blast*
from *characterize-*$\mathcal{P}$[*OF this*] **have** $\exists$ *u*. $q \in \mathcal{P}$ (*charslength q*) *u* **using** *admissible q* **by** *auto*
then show *?thesis* **using** $\mathfrak{P}$*-covers-*$\mathcal{P}$ *q* **by** *blast*
qed

end

end
theory *MainTheorems*
imports *PathLemmas*
begin

context *LocalLexing* **begin**

theorem $\mathfrak{I}$-*is-generated-by*-$\mathfrak{P}$: $\mathfrak{I} = Gen\ \mathfrak{P}$
proof −
  have *wellformed-items* ($\mathcal{I}$ (*length Doc*))
    using *wellformed-items-*$\mathcal{I}$ **by** *auto*
  then have $\bigwedge x.\ x \in \mathcal{I}$ (*length Doc*) $\Longrightarrow$ *item-end* $x \leq$ *length Doc*
    using *wellformed-item-def wellformed-items-def* **by** *blast*
  then have $\mathcal{I}$ (*length Doc*) $\subseteq$ *items-le* (*length Doc*) ($\mathcal{I}$ (*length Doc*))
    by (*auto simp only*: *items-le-def*)
  then have $\mathcal{I}$ (*length Doc*) = *items-le* (*length Doc*) ($\mathcal{I}$ (*length Doc*))
    using *items-le-is-filter* **by** *blast*
  then have $\mathfrak{I}$-*altdef*: $\mathfrak{I}$ = *items-le* (*length Doc*) ($\mathcal{I}$ (*length Doc*)) **using** $\mathfrak{I}$-*def* **by** *auto*
  have $\bigwedge p.\ p \in (\mathcal{Q}$ (*length Doc*)) $\Longrightarrow$ *charslength* $p \leq$ *length Doc*
    using $\mathfrak{P}$-*are-doc-tokens* $\mathfrak{P}$-*def* *doc-tokens-length* **by** *auto*
  then have $\mathcal{Q}$ (*length Doc*) $\subseteq$ *paths-le* (*length Doc*) ($\mathcal{Q}$ (*length Doc*))
    by (*auto simp only*: *paths-le-def*)
  then have $\mathcal{Q}$ (*length Doc*) = *paths-le* (*length Doc*) ($\mathcal{Q}$ (*length Doc*))
    using *paths-le-is-filter* **by** *blast*
  then have $\mathfrak{P}$-*altdef*: $\mathfrak{P}$ = *paths-le* (*length Doc*) ($\mathcal{Q}$ (*length Doc*)) **using** $\mathfrak{P}$-*def* **by** *auto*
  show *?thesis* **using** $\mathfrak{I}$-*altdef* $\mathfrak{P}$-*altdef* *thmD14* **by** *auto*
qed

definition *finished-item* :: *symbol list* ⇒ *item*
where
  finished-item α = *Item* ($\mathfrak{S}$, α) (*length* α) *0* (*length Doc*)

lemma *item-rule-finished-item*[*simp*]: *item-rule* (*finished-item* α) = ($\mathfrak{S}$, α)
  by (*simp add*: *finished-item-def*)

lemma *item-origin-finished-item*[*simp*]: *item-origin* (*finished-item* α) = *0*
  by (*simp add*: *finished-item-def*)

lemma *item-end-finished-item*[*simp*]: *item-end* (*finished-item* α) = *length Doc*
  by (*simp add*: *finished-item-def*)

lemma *item-dot-finished-item*[*simp*]: *item-dot* (*finished-item* α) = *length* α
  by (*simp add*: *finished-item-def*)

lemma *item-rhs-finished-item*[*simp*]: *item-rhs* (*finished-item* α) = α
  by (*simp add*: *finished-item-def*)

lemma *item-*α*-finished-item*[*simp*]: *item-*α (*finished-item* α) = α
  by (*simp add*: *finished-item-def item-*α*-def*)

lemma *item-nonterminal-finished-item*[*simp*]: *item-nonterminal* (*finished-item* α) = $\mathfrak{S}$

by (*simp add*: *finished-item-def item-nonterminal-def*)

lemma *Derives1-of-singleton*:
assumes *Derives1* [*N*] *i r* α
shows $i = 0 \wedge r = (N, \alpha)$
proof –
from *assms* **have** $i = 0$ **using** *Derives1-bound*
using *length-Cons less-Suc0 list.size*(*3*) **by** *fastforce*
then show *?thesis* **using** *assms*
using *Derives1-def append-Cons append-self-conv append-self-conv2 length-0-conv*

list.inject **by** *auto*
qed

definition *pvalid-with* :: *tokens* ⇒ *item* ⇒ *nat* ⇒ *symbol list* ⇒ *bool*
where
pvalid-with p x u γ =
(*wellformed-tokens p* ∧
wellformed-item x ∧
u ≤ *length p* ∧
charslength p = *item-end x* ∧
charslength (*take u p*) = *item-origin x* ∧
is-derivation (*terminals* (*take u p*) @ [*item-nonterminal x*] @ γ) ∧
derives (*item-*α *x*) (*terminals* (*drop u p*)))

lemma *pvalid-with*: *pvalid p x* = (∃ *u* γ. *pvalid-with p x u* γ)
using *pvalid-def pvalid-with-def* **by** *blast*

theorem *Completeness*:
assumes *p-in-ll*: *p* ∈ *ll*
shows ∃ α. *pvalid-with p* (*finished-item* α) *0* [] ∧ *finished-item* $\alpha \in \mathfrak{I}$
proof –
have *p*: *p* ∈ $\mathfrak{P}$ ∧ *charslength p* = *length Doc* ∧ *terminals p* ∈ $\mathcal{L}$
using *p-in-ll ll-def* **by** *auto*
then have *derives-p*: *derives* [$\mathfrak{S}$] (*terminals p*)
using $\mathcal{L}$*-def is-derivation-def mem-Collect-eq* **by** *blast*
then have ∃ *D*. *Derivation* [$\mathfrak{S}$] *D* (*terminals p*)
by (*simp add*: *derives-implies-Derivation*)
then obtain *D* **where** *D*: *Derivation* [$\mathfrak{S}$] *D* (*terminals p*) **by** *blast*
have *is-word-p*: *is-word* (*terminals p*) **using** *leftlang p* **by** *blast*
have *not-is-word-*$\mathfrak{S}$: ¬ (*is-word* [$\mathfrak{S}$]) **using** *is-nonterminal-startsymbol is-terminal-nonterminal*

is-word-cons **by** *blast*
have *D* ≠ [] **using** *D is-word-p not-is-word-*$\mathfrak{S}$ **using** *Derivation.simps*(*1*) **by**
force
then have ∃ *d D'*. *D* = *d#D'* **using** *D Derivation.elims*(*2*) **by** *blast*
then obtain *d D'* **where** *d*: *D* = *d#D'* **by** *blast*
have ∃ α. *Derives1* [$\mathfrak{S}$] (*fst d*) (*snd d*) α ∧ *derives* α (*terminals p*)
using *d D Derivation.simps*(*2*) *Derivation-implies-derives* **by** *blast*

then obtain *α* **where** *α*: *Derives1* [𝔖] (*fst d*) (*snd d*) *α* ∧ *derives α* (*terminals p*) **by** *blast*
then have *snd-d-in-*ℜ: *snd d* ∈ ℜ **using** *Derives1-rule* **by** *blast*
with *α* **have** *snd-d*: *snd d* = (𝔖, *α*) **using** *Derives1-of-singleton* **by** *blast*
have *wellformed-p*: *wellformed-tokens p* **by** (*simp add*: 𝔓*-wellformed p*)
have *wellformed-finished-item*: *wellformed-item* (*finished-item α*)
apply (*auto simp add*: *wellformed-item-def*)
using *snd-d snd-d-in-*ℜ **by** *metis*
have *pvalid-with*: *pvalid-with p* (*finished-item α*) *0* []
apply (*auto simp add*: *pvalid-with-def*)
using *wellformed-p* **apply** *blast*
using *wellformed-finished-item* **apply** *blast*
using *p* **apply** (*simp add*: *finished-item-def*)
apply (*simp add*: *is-derivation-def*)
by (*simp add*: *α*)
then have *pvalid p* (*finished-item α*) **using** *pvalid-def pvalid-with-def* **by** *blast*
then have *finished-item α* ∈ *Gen* 𝔓 **using** *Gen-def mem-Collect-eq p* **by** *blast*
then have *finished-item α* ∈ ℑ **using** ℑ*-is-generated-by-*𝔓 **by** *blast*
with *pvalid-with* **show** *?thesis* **by** *blast*
qed

theorem *Soundness*:
assumes *finished-item-α*: *finished-item α* ∈ ℑ
shows ∃ *p*. *pvalid-with p* (*finished-item α*) *0* [] ∧ *p* ∈ *ll*
proof −
have *finished-item α* ∈ *Gen* 𝔓
using ℑ*-is-generated-by-*𝔓 *finished-item-α* **by** *auto*
then obtain *p* **where** *p*: *p* ∈ 𝔓 ∧ *pvalid p* (*finished-item α*)
using *Gen-implies-pvalid* **by** *blast*
have *pvalid-p-finished-item*: *pvalid p* (*finished-item α*) **using** *p* **by** *blast*
from *iffD1*[*OF pvalid-def this*, *simplified*] **obtain** *r γ* **where** *pvalid*:
wellformed-tokens p ∧
wellformed-item (*finished-item α*) ∧
r ≤ *length p* ∧
length (*chars p*) = *length Doc* ∧
chars (*take r p*) = [] ∧
is-derivation (*take r* (*terminals p*) @ 𝔖 # *γ*) ∧ *derives α* (*drop r* (*terminals p*))
by *blast*
have *item-rule* (*finished-item α*) ∈ ℜ **using** *pvalid*
using *wellformed-item-def* **by** *blast*
then have (𝔖, *α*) ∈ ℜ **by** *simp*
then have *is-derivation-α*: *is-derivation α* **by** (*simp add*: *is-derivation-def left-derives-rule*)
have *drop-r-p-in-*𝔓: *drop r p* ∈ 𝔓
apply (*rule drop-empty-tokens*)
using *p* **apply** *blast*
using *pvalid* **apply** *blast*
using *pvalid* **apply** *simp*

by (*metis append-Nil2 derives-trans is-derivation-α is-derivation-def*
is-derivation-implies-admissible is-word-terminals-drop pvalid terminals-drop)
then have *in-ll*: *drop r p* ∈ *ll*
apply (*auto simp add*: *ll-def*)
apply (*metis append-Nil append-take-drop-id chars-append pvalid*)
using *is-derivation-α pvalid*
by (*metis* (*no-types*, *lifting*) *ℒ-def derives-trans is-derivation-def*
is-word-terminals-drop mem-Collect-eq terminals-drop)
have *pvalid-with* (*drop r p*) (*finished-item α*) *0* []
apply (*auto simp add*: *pvalid-with-def*)
using 𝔓*-wellformed drop-r-p-in-*𝔓 **apply** *blast*
using *pvalid* **apply** *blast*
apply (*metis append-Nil append-take-drop-id chars-append pvalid*)
apply (*simp add*: *is-derivation-def*)
using *pvalid* **by** *blast*
with *in-ll* **show** *?thesis* **by** *auto*
qed

lemma *is-finished-and-finished-item*:
assumes *wellformed-x*: *wellformed-item x*
shows *is-finished x* = (∃ *α*. *x* = *finished-item α*)
proof −
{
assume *is-finished-x*: *is-finished x*
obtain *α* **where** *α*: *α* = *item-rhs x* **by** *blast*
have *x* = *finished-item α*
apply (*rule item.expand*)
apply *auto*
using *α is-finished-def is-finished-x item-nonterminal-def item-rhs-def* **apply** *auto*[*1*]
using *α assms is-complete-def is-finished-def is-finished-x wellformed-item-def* **apply** *auto*[*1*]
using *is-finished-def is-finished-x* **apply** *blast*
using *is-finished-def is-finished-x* **by** *auto*
then have ∃ *α*. *x* = *finished-item α* **by** *blast*
}
note *left-implies-right* = *this*
{
assume ∃ *α*. *x* = *finished-item α*
then obtain *α* **where** *α*: *x* = *finished-item α* **by** *blast*
have *is-finished x* **by** (*simp add*: *α is-finished-def is-complete-def*)
}
note *right-implies-left* = *this*
show *?thesis* **using** *left-implies-right right-implies-left* **by** *blast*
qed

theorem *Correctness*:
shows (*ll* ≠ {}) = *earley-recognised*
proof −

have *1*: $(ll \neq \{\}) = (\exists\ \alpha.\ \textit{finished-item}\ \alpha \in \mathfrak{I})$
using *Soundness Completeness ex-in-conv* **by** *fastforce*
have *2*: $(\exists\ \alpha.\ \textit{finished-item}\ \alpha \in \mathfrak{I}) = (\exists\ x \in \mathfrak{I}.\ \textit{is-finished}\ x)$
using $\mathfrak{I}$*-def is-finished-and-finished-item wellformed-items-*$\mathcal{I}$ *wellformed-items-def*
by *auto*
show *?thesis* **using** *earley-recognised-def 1 2* **by** *blast*
qed

end

end