Earley

Martin Rau

December 9, 2023

Abstract

In 1968 Earley [1] introduced his parsing algorithm capable of parsing all context-free grammars in cubic space and time. This entry contains a formalization of an executable Earley parser. We base our development on Jones' [2] extensive paper proof of Earley's recognizer and the formalization of context-free grammars and derivations of Obua [4] [3]. We implement and prove correct a functional recognizer modeling Earley's original imperative implementation and extend it with the necessary data structures to enable the construction of parse trees following the work of Scott [5]. We then develop a functional algorithm that builds a single parse tree and prove its correctness. Finally, we generalize this approach to an algorithm for a complete parse forest and prove soundness.

Contents

theory *Limit*
imports *Main*
begin

1 Slightly adjusted content from AFP/LocalLexing

fun *funpower* :: $('a \Rightarrow 'a) \Rightarrow nat \Rightarrow ('a \Rightarrow 'a)$ **where**
funpower f 0 x = *x*
| *funpower f* (*Suc n*) *x* = *f* (*funpower f n x*)

definition *natUnion* :: $(nat \Rightarrow 'a\ set) \Rightarrow 'a\ set$ **where**
natUnion f = $\bigcup$ { *f n* | *n*. *True* }

definition *limit* :: $('a\ set \Rightarrow 'a\ set) \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
limit f x = *natUnion* (λ *n*. *funpower f n x*)

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

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

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

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

definition *chain* :: (*nat* ⇒ *'a set*) ⇒ *bool*
where
chain C = (∀ *i*. *C i* ⊆ *C* (*i* + *1*))

definition *continuous* :: (*'a set* ⇒ *'b set*) ⇒ *bool*
where
continuous f = (∀ *C*. *chain C* ⟶ (*chain* (*f o C*) ∧ *f* (*natUnion C*) = *natUnion* (*f o C*)))

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 *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*)

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

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
    by (metis (no-types, lifting) Limit.chain-def chain natUnion-eq)
  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 }

end
theory CFG
  imports Main
begin
```

2 Adjusted content from AFP/LocalLexing

```
type-synonym 'a rule = 'a × 'a list

type-synonym 'a rules = 'a rule list

type-synonym 'a sentence = 'a list

datatype 'a cfg =
```

CFG ($\mathfrak{N}$: *'a list*) ($\mathfrak{T}$: *'a list*) ($\mathfrak{R}$: *'a rules*) ($\mathfrak{S}$: *'a*)

definition *disjunct-symbols* :: *'a cfg* ⇒ *bool* **where**
disjunct-symbols $\mathcal{G}$ ≡ *set* ($\mathfrak{N}$ $\mathcal{G}$) ∩ *set* ($\mathfrak{T}$ $\mathcal{G}$) = {}

definition *valid-startsymbol* :: *'a cfg* ⇒ *bool* **where**
valid-startsymbol $\mathcal{G}$ ≡ $\mathfrak{S}$ $\mathcal{G}$ ∈ *set* ($\mathfrak{N}$ $\mathcal{G}$)

definition *valid-rules* :: *'a cfg* ⇒ *bool* **where**
valid-rules $\mathcal{G}$ ≡ ∀ (*N*, *α*) ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$). *N* ∈ *set* ($\mathfrak{N}$ $\mathcal{G}$) ∧ (∀ *s* ∈ *set α*. *s* ∈ *set* ($\mathfrak{N}$ $\mathcal{G}$) ∪ *set* ($\mathfrak{T}$ $\mathcal{G}$))

definition *distinct-rules* :: *'a cfg* ⇒ *bool* **where**
distinct-rules $\mathcal{G}$ ≡ *distinct* ($\mathfrak{R}$ $\mathcal{G}$)

definition *wf-*$\mathcal{G}$ :: *'a cfg* ⇒ *bool* **where**
wf-$\mathcal{G}$ $\mathcal{G}$ ≡ *disjunct-symbols* $\mathcal{G}$ ∧ *valid-startsymbol* $\mathcal{G}$ ∧ *valid-rules* $\mathcal{G}$ ∧ *distinct-rules* $\mathcal{G}$

lemmas *wf-*$\mathcal{G}$*-defs* = *wf-*$\mathcal{G}$*-def valid-rules-def valid-startsymbol-def disjunct-symbols-def distinct-rules-def*

definition *is-terminal* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
is-terminal $\mathcal{G}$ *x* ≡ *x* ∈ *set* ($\mathfrak{T}$ $\mathcal{G}$)

definition *is-nonterminal* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
is-nonterminal $\mathcal{G}$ *x* ≡ *x* ∈ *set* ($\mathfrak{N}$ $\mathcal{G}$)

definition *is-symbol* :: *'a cfg* ⇒ *'a* ⇒ *bool* **where**
is-symbol $\mathcal{G}$ *x* ≡ *is-terminal* $\mathcal{G}$ *x* ∨ *is-nonterminal* $\mathcal{G}$ *x*

definition *wf-sentence* :: *'a cfg* ⇒ *'a sentence* ⇒ *bool* **where**
wf-sentence $\mathcal{G}$ *ω* ≡ ∀ *x* ∈ *set ω*. *is-symbol* $\mathcal{G}$ *x*

lemma *is-nonterminal-startsymbol*:
wf-$\mathcal{G}$ $\mathcal{G}$ ⟹ *is-nonterminal* $\mathcal{G}$ ($\mathfrak{S}$ $\mathcal{G}$)
by (*simp add*: *is-nonterminal-def wf-*$\mathcal{G}$*-defs*)

definition *is-word* :: *'a cfg* ⇒ *'a sentence* ⇒ *bool* **where**
is-word $\mathcal{G}$ *ω* ≡ ∀ *x* ∈ *set ω*. *is-terminal* $\mathcal{G}$ *x*

definition *derives1* :: *'a cfg* ⇒ *'a sentence* ⇒ *'a sentence* ⇒ *bool* **where**
derives1 $\mathcal{G}$ *u v* ≡ ∃ *x y N α*.
u = *x* @ [*N*] @ *y* ∧
v = *x* @ *α* @ *y* ∧
(*N*, *α*) ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$)

definition *derivations1* :: *'a cfg* ⇒ (*'a sentence* × *'a sentence*) *set* **where**
derivations1 $\mathcal{G}$ ≡ { (*u*,*v*) | *u v*. *derives1* $\mathcal{G}$ *u v* }

definition *derivations* :: *'a cfg* ⇒ (*'a sentence* × *'a sentence*) *set* **where**
derivations $\mathcal{G}$ ≡ (*derivations1* $\mathcal{G}$)^*

definition *derives* :: *'a cfg* ⇒ *'a sentence* ⇒ *'a sentence* ⇒ *bool* **where**
derives $\mathcal{G}$ *u v* ≡ ((*u*, *v*) ∈ *derivations* $\mathcal{G}$)

end
theory *Derivations*
imports
CFG
begin

3 Adjusted content from AFP/LocalLexing

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

lemma *is-word-empty*: *is-word* $\mathcal{G}$ [] **by** (*auto simp add*: *is-word-def*)

lemma *derives1-implies-derives*[*simp*]:
derives1 $\mathcal{G}$ *a b* ⟹ *derives* $\mathcal{G}$ *a b*
by (*auto simp add*: *derives-def derivations-def derivations1-def*)

lemma *derives-trans*:
derives $\mathcal{G}$ *a b* ⟹ *derives* $\mathcal{G}$ *b c* ⟹ *derives* $\mathcal{G}$ *a c*
by (*auto simp add*: *derives-def derivations-def*)

lemma *derives1-eq-derivations1*:
derives1 $\mathcal{G}$ *x y* = ((*x*, *y*) ∈ *derivations1* $\mathcal{G}$)
by (*simp add*: *derivations1-def*)

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

definition *Derives1* :: *'a cfg* ⇒ *'a sentence* ⇒ *nat* ⇒ *'a rule* ⇒ *'a sentence* ⇒ *bool* **where**
Derives1 $\mathcal{G}$ *u i r v* ≡ ∃ *x y N* α.
u = *x* @ [*N*] @ *y* ∧
v = *x* @ α @ *y* ∧
(*N*, α) ∈ *set* (ℜ $\mathcal{G}$) ∧ *r* = (*N*, α) ∧ *i* = *length x*

lemma *Derives1-split*:
Derives1 $\mathcal{G}$ u i r v $\Longrightarrow \exists$ *x y. u = x @ [fst r] @ y* $\wedge$ *v = x @ (snd r) @ y* $\wedge$ *length x = i*
by (*metis Derives1-def fst-conv snd-conv*)

lemma *Derives1-implies-derives1*: *Derives1 $\mathcal{G}$ u i r v* $\Longrightarrow$ *derives1 $\mathcal{G}$ u v*
by (*auto simp add*: *Derives1-def derives1-def*)

lemma *derives1-implies-Derives1*: *derives1 $\mathcal{G}$ u v* $\Longrightarrow \exists$ *i r. Derives1 $\mathcal{G}$ u i r v*
by (*auto simp add*: *Derives1-def derives1-def*)

fun *Derivation* :: *'a cfg* $\Rightarrow$ *'a sentence* $\Rightarrow$ *'a derivation* $\Rightarrow$ *'a sentence* $\Rightarrow$ *bool*
where
Derivation - a [] b = (a = b)
| *Derivation $\mathcal{G}$ a (d#D) b = (*$\exists$ *x. Derives1 $\mathcal{G}$ a (fst d) (snd d) x* $\wedge$ *Derivation $\mathcal{G}$ x D b)*

lemma *Derivation-implies-derives*: *Derivation $\mathcal{G}$ a D b* $\Longrightarrow$ *derives $\mathcal{G}$ 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** $\exists$ *x. Derives1 $\mathcal{G}$ a (fst d) (snd d) x* $\wedge$ *Derivation $\mathcal{G}$ x D b* **by** *auto*
then obtain *x* **where** *Derives1 $\mathcal{G}$ a (fst d) (snd d) x* **and** *xb*: *Derivation $\mathcal{G}$ x D b* **by** *blast*
with *Derives1-implies-derives1* **have** *d1*: *derives $\mathcal{G}$ a x* **by** *fastforce*
from *ihyps xb* **have** *d2*:*derives $\mathcal{G}$ x b* **by** *simp*
show *derives $\mathcal{G}$ a b* **by** (*rule derives-trans*[*OF d1 d2*])
qed

lemma *Derivation-Derives1*: *Derivation $\mathcal{G}$ a S y* $\Longrightarrow$ *Derives1 $\mathcal{G}$ y i r z* $\Longrightarrow$ *Derivation $\mathcal{G}$ 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 $\mathcal{G}$ a b* $\Longrightarrow \exists$ *D. Derivation $\mathcal{G}$ 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* $\mathcal{G}$ *a D y* **by** *blast*
from *ihyps derives1-implies-Derives1* **obtain** *i r* **where** *yz*: *Derives1* $\mathcal{G}$ *y i r z*
by *blast*
from *Derivation-Derives1*[*OF ay yz*] **show** *?case* **by** *auto*
qed

lemma *rule-nonterminal-type*[*simp*]: *wf-*$\mathcal{G}$ $\mathcal{G} \Longrightarrow (N, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G}) \Longrightarrow$ *is-nonterminal* $\mathcal{G}$ *N*
by (*auto simp add*: *is-nonterminal-def wf-*$\mathcal{G}$*-defs*)

lemma *Derives1-rule* [*elim*]: *Derives1* $\mathcal{G}$ *a i r b* $\Longrightarrow r \in set\ (\mathfrak{R}\ \mathcal{G})$
using *Derives1-def* **by** *metis*

lemma *is-terminal-nonterminal*: *wf-*$\mathcal{G}$ $\mathcal{G} \Longrightarrow$ *is-terminal* $\mathcal{G}$ *x* $\Longrightarrow$ *is-nonterminal* $\mathcal{G}$ *x* $\Longrightarrow$ *False*
by (*auto simp*: *wf-*$\mathcal{G}$*-defs disjoint-iff is-nonterminal-def is-terminal-def*)

lemma *is-word-is-terminal*: *i* < *length u* $\Longrightarrow$ *is-word* $\mathcal{G}$ *u* $\Longrightarrow$ *is-terminal* $\mathcal{G}$ (*u* ! *i*)
using *is-word-def* **by** *force*

lemma *Derivation-append*: *Derivation* $\mathcal{G}$ *a* (*D*@*E*) *c* = ($\exists$ *b*. *Derivation* $\mathcal{G}$ *a D b* $\wedge$ *Derivation* $\mathcal{G}$ *b E c*)
by (*induct D arbitrary*: *a c E*) *auto*

lemma *Derivation-implies-append*:
Derivation $\mathcal{G}$ *a D b* $\Longrightarrow$ *Derivation* $\mathcal{G}$ *b E c* $\Longrightarrow$ *Derivation* $\mathcal{G}$ *a* (*D*@*E*) *c*
using *Derivation-append* **by** *blast*

4 Additional derivation lemmas

lemma *Derives1-prepend*:
assumes *Derives1* $\mathcal{G}$ *u i r v*
shows *Derives1* $\mathcal{G}$ (*w*@*u*) (*i* + *length w*) *r* (*w*@*v*)
proof −
obtain *x y N* α **where** ∗:
u = *x* @ [*N*] @ *y v* = *x* @ α @ *y*
(*N*, α) ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$) *r* = (*N*, α) *i* = *length x*
using *assms Derives1-def* **by** (*smt* (*verit*))
hence *w*@*u* = *w* @ *x* @ [*N*] @ *y w*@*v* = *w* @ *x* @ α @ *y*
by *auto*
thus *?thesis*
unfolding *Derives1-def* **using** ∗
apply (*rule-tac exI*[**where** *x*=*w*@*x*])
apply (*rule-tac exI*[**where** *x*=*y*])
by *simp*
qed

lemma *Derivation-prepend*:
Derivation $\mathcal{G}$ b D $b' \Longrightarrow$ *Derivation* $\mathcal{G}$ $(a@b)$ $(map\ (\lambda(i,\ r).\ (i + length\ a,\ r))\ D)$ $(a@b')$
using *Derives1-prepend* **by** (*induction D arbitrary*: b b') (*auto*, *fast*)

lemma *Derives1-append*:
assumes *Derives1* $\mathcal{G}$ u i r v
shows *Derives1* $\mathcal{G}$ $(u@w)$ i r $(v@w)$
proof −
obtain x y N α **where** $*$:
$u = x\ @\ [N]\ @\ y$ $v = x\ @\ \alpha\ @\ y$
$(N,\ \alpha) \in set\ (\mathfrak{R}\ \mathcal{G})$ $r = (N,\ \alpha)$ $i = length\ x$
using *assms Derives1-def* **by** (*smt* (*verit*))
hence $u@w = x\ @\ [N]\ @\ y\ @\ w$ $v@w = x\ @\ \alpha\ @\ y\ @\ w$
by *auto*
thus *?thesis*
unfolding *Derives1-def* **using** $*$
apply (*rule-tac exI*[**where** x=x])
apply (*rule-tac exI*[**where** x=$y@w$])
by *blast*
qed

lemma *Derivation-append'*:
Derivation $\mathcal{G}$ a D $a' \Longrightarrow$ *Derivation* $\mathcal{G}$ $(a@b)$ D $(a'@b)$
using *Derives1-append* **by** (*induction D arbitrary*: a a') (*auto*, *fast*)

lemma *Derivation-append-rewrite*:
assumes *Derivation* $\mathcal{G}$ a D $(b\ @\ c\ @\ d)$ *Derivation* $\mathcal{G}$ c E c'
shows $\exists F.$ *Derivation* $\mathcal{G}$ a F $(b\ @\ c'\ @\ d)$
using *assms Derivation-append' Derivation-prepend Derivation-implies-append* **by** *fast*

lemma *derives1-if-valid-rule*:
$(N,\ \alpha) \in set\ (\mathfrak{R}\ \mathcal{G}) \Longrightarrow$ *derives1* $\mathcal{G}$ $[N]$ α
unfolding *derives1-def*
apply (*rule-tac exI*[**where** x=[]])
apply (*rule-tac exI*[**where** x=[]])
by *simp*

lemma *derives-if-valid-rule*:
$(N,\ \alpha) \in set\ (\mathfrak{R}\ \mathcal{G}) \Longrightarrow$ *derives* $\mathcal{G}$ $[N]$ α
using *derives1-if-valid-rule* **by** *fastforce*

lemma *Derivation-from-empty*:
Derivation $\mathcal{G}$ [] D $a \Longrightarrow a = []$
by (*cases D*) (*auto simp*: *Derives1-def*)

lemma *Derivation-concat-split*:
Derivation $\mathcal{G}$ $(a@b)$ D $c \Longrightarrow \exists E\ F\ a'\ b'.$ *Derivation* $\mathcal{G}$ a E $a' \wedge$ *Derivation* $\mathcal{G}$ b

```
F b' ∧
    c = a' @ b' ∧ length E ≤ length D ∧ length F ≤ length D
proof (induction D arbitrary: a b)
 case Nil
 thus ?case
   by (metis Derivation.simps(1) order-refl)
next
 case (Cons d D)
 then obtain ab where ∗: Derives1 G (a@b) (fst d) (snd d) ab Derivation G ab
D c
   by auto
 then obtain x y N α where #:
   a@b = x @ [N] @ y ab = x @ α @ y (N,α) ∈ set (ℜ G) snd d = (N,α) fst d
= length x
   using ∗ unfolding Derives1-def by blast
 show ?case
 proof (cases length a ≤ length x)
   case True
   hence ab-def:
     a = take (length a) x
     b = drop (length a) x @ [N] @ y
     ab = take (length a) x @ drop (length a) x @ α @ y
     using #(1,2) True by (metis append-eq-append-conv-if)+
   then obtain E F a' b' where IH:
     Derivation G (take (length a) x) E a'
     Derivation G (drop (length a) x @ α @ y) F b'
     c = a' @ b'
     length E ≤ length D
     length F ≤ length D
     using Cons ∗(2) by blast
   have Derives1 G b (fst d − length a) (snd d) (drop (length a) x @ α @ y)
    unfolding Derives1-def using ∗(1) #(3−5) ab-def(2) by (metis length-drop)
   hence Derivation G b ((fst d − length a, snd d) # F) b'
     using IH(2) by force
   moreover have Derivation G a E a'
     using IH(1) ab-def(1) by fastforce
   ultimately show ?thesis
     using IH(3−5) by fastforce
 next
   case False
   hence a-def: a = x @ [N] @ take (length a − length x − 1) y
     using #(1) append-eq-conv-conj[of a b x @ [N] @ y] take-all-iff take-append
     by (metis append-Cons append-Nil diff-is-0-eq le-cases take-Cons')
   hence b-def: b = drop (length a − length x − 1) y
    using #(1) by (metis List.append.assoc append-take-drop-id same-append-eq)
   have ab = x @ α @ take (length a − length x − 1) y @ drop (length a − length
x − 1) y
     using #(2) by force
   then obtain E F a' b' where IH:
```

Derivation $\mathcal{G}$ (*x* @ α @ *take* (*length a* − *length x* − *1*) *y*) *E a′*
Derivation $\mathcal{G}$ (*drop* (*length a* − *length x* − *1*) *y*) *F b′*
c = *a′* @ *b′*
length E ≤ *length D*
length F ≤ *length D*
using *Cons.IH*[*of x* @ α @ *take* (*length a* − *length x* − *1*) *y drop* (*length a* − *length x* − *1*) *y*] ∗(*2*) **by** *auto*
have *Derives1* $\mathcal{G}$ *a* (*fst d*) (*snd d*) (*x* @ α @ *take* (*length a* − *length x* − *1*) *y*)
unfolding *Derives1-def* **using** #(*3*−*5*) *a-def* **by** *blast*
hence *Derivation* $\mathcal{G}$ *a* ((*fst d*, *snd d*) # *E*) *a′*
using *IH*(*1*) **by** *fastforce*
moreover have *Derivation* $\mathcal{G}$ *b F b′*
using *b-def IH*(*2*) **by** *blast*
ultimately show *?thesis*
using *IH*(*3*−*5*) **by** *fastforce*
qed
qed

lemma *Derivation-*$\mathfrak{S}$*1*:
assumes *Derivation* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] *D* ω *is-word* $\mathcal{G}$ ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows ∃α *E*. *Derivation* $\mathcal{G}$ α *E* ω ∧ ($\mathfrak{S}$ $\mathcal{G}$,α) ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$)
proof (*cases D*)
case *Nil*
thus *?thesis*
using *assms is-nonterminal-startsymbol is-terminal-nonterminal* **by** (*metis Derivation.simps*(*1*) *is-word-def list.set-intros*(*1*))
next
case (*Cons d D*)
then obtain α **where** *Derives1* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] (*fst d*) (*snd d*) α *Derivation* $\mathcal{G}$ α *D* ω
using *assms* **by** *auto*
hence ($\mathfrak{S}$ $\mathcal{G}$, α) ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$)
unfolding *Derives1-def*
by (*metis List.append.right-neutral List.list.discI append-eq-Cons-conv append-is-Nil-conv nth-Cons-0 self-append-conv2*)
thus *?thesis*
using ‹*Derivation* $\mathcal{G}$ α *D* ω› **by** *auto*
qed

end
theory *Earley*
imports
Derivations
begin

5 Slices

fun *slice* :: *nat* ⇒ *nat* ⇒ *'a list* ⇒ *'a list* **where**
slice - - [] = []
| *slice - 0* (*x*#*xs*) = []

```
| slice 0 (Suc b) (x#xs) = x # slice 0 b xs
| slice (Suc a) (Suc b) (x#xs) = slice a b xs

lemma slice-drop-take:
  slice a b xs = drop a (take b xs)
  by (induction a b xs rule: slice.induct) auto

lemma slice-append-aux:
  Suc b ≤ c ⟹ slice (Suc b) c (x # xs) = slice b (c−1) xs
  using Suc-le-D by fastforce

lemma slice-concat:
  a ≤ b ⟹ b ≤ c ⟹ slice a b xs @ slice b c xs = slice a c xs
proof (induction a b xs arbitrary: c rule: slice.induct)
  case (3 b x xs)
  then show ?case
    using Suc-le-D by(fastforce simp: slice-append-aux)
qed (auto simp: slice-append-aux)

lemma slice-concat-Ex:
  a ≤ c ⟹ slice a c xs = ys @ zs ⟹ ∃ b. ys = slice a b xs ∧ zs = slice b c xs ∧
a ≤ b ∧ b ≤ c
proof (induction a c xs arbitrary: ys zs rule: slice.induct)
  case (3 b x xs)
  show ?case
  proof (cases ys)
    case Nil
    then obtain zs' where x # slice 0 b xs = x # zs' x # zs' = zs
      using 3.prems(2) by auto
    thus ?thesis
      using Nil by force
  next
    case (Cons y ys')
    then obtain ys' where x # slice 0 b xs = x # ys' @ zs x # ys' = ys
      using 3.prems(2) by auto
    thus ?thesis
      using 3.IH[of ys' zs] by force
  qed
next
  case (4 a b x xs)
  thus ?case
    by (auto, metis slice.simps(4) Suc-le-mono)
qed auto

lemma slice-nth:
  a < length xs ⟹ slice a (a+1) xs = [xs!a]
  unfolding slice-drop-take
 by (metis Cons-nth-drop-Suc One-nat-def diff-add-inverse drop-take take-Suc-Cons
take-eq-Nil)
```

lemma *slice-append-nth*:
a ≤ *b* ⟹ *b* < *length xs* ⟹ *slice a b xs* @ [*xs*!*b*] = *slice a* (*b*+*1*) *xs*
by (*metis le-add1 slice-concat slice-nth*)

lemma *slice-empty*:
b ≤ *a* ⟹ *slice a b xs* = []
by (*simp add*: *slice-drop-take*)

lemma *slice-id*[*simp*]:
slice 0 (*length xs*) *xs* = *xs*
by (*simp add*: *slice-drop-take*)

lemma *slice-singleton*:
b ≤ *length xs* ⟹ [*x*] = *slice a b xs* ⟹ *b* = *a* + *1*
by (*induction a b xs rule*: *slice.induct*) (*auto simp*: *slice-drop-take*)

6 Earley recognizer

6.1 Earley items

definition *rule-head* :: *'a rule* ⇒ *'a* **where**
rule-head ≡ *fst*

definition *rule-body* :: *'a rule* ⇒ *'a list* **where**
rule-body ≡ *snd*

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

definition *item-rule-head* :: *'a item* ⇒ *'a* **where**
item-rule-head x ≡ *rule-head* (*item-rule x*)

definition *item-rule-body* :: *'a item* ⇒ *'a sentence* **where**
item-rule-body x ≡ *rule-body* (*item-rule x*)

definition *item-α* :: *'a item* ⇒ *'a sentence* **where**
item-α x ≡ *take* (*item-dot x*) (*item-rule-body x*)

definition *item-β* :: *'a item* ⇒ *'a sentence* **where**
item-β x ≡ *drop* (*item-dot x*) (*item-rule-body x*)

definition *is-complete* :: *'a item* ⇒ *bool* **where**
is-complete x ≡ *item-dot x* ≥ *length* (*item-rule-body x*)

definition *next-symbol* :: *'a item* ⇒ *'a option* **where**
next-symbol x ≡ *if is-complete x then None else Some* (*item-rule-body x* ! *item-dot x*)

lemmas *item-defs = item-rule-head-def item-rule-body-def item-α-def item-β-def rule-head-def rule-body-def*

definition *is-finished* :: *'a cfg ⇒ 'a sentence ⇒ 'a item ⇒ bool* **where**
is-finished $\mathcal{G}$ ω $x \equiv$
item-rule-head $x = \mathfrak{S}$ $\mathcal{G}$ $\wedge$
item-origin $x = 0$ $\wedge$
item-end $x =$ *length* ω $\wedge$
is-complete x

definition *recognizing* :: *'a item set ⇒ 'a cfg ⇒ 'a sentence ⇒ bool* **where**
recognizing I $\mathcal{G}$ $\omega \equiv \exists\, x \in I.$ *is-finished* $\mathcal{G}$ ω x

inductive-set *Earley* :: *'a cfg ⇒ 'a sentence ⇒ 'a item set*
for $\mathcal{G}$ :: *'a cfg* **and** ω :: *'a sentence* **where**
Init: $r \in$ *set* $(\mathfrak{R}\ \mathcal{G}) \Longrightarrow$ *fst* $r = \mathfrak{S}\ \mathcal{G} \Longrightarrow$
Item $r\ 0\ 0\ 0 \in$ *Earley* $\mathcal{G}$ ω
| *Scan*: $x =$ *Item* $r\ b\ i\ j \Longrightarrow x \in$ *Earley* $\mathcal{G}$ $\omega \Longrightarrow$
$\omega!j = a \Longrightarrow j <$ *length* $\omega \Longrightarrow$ *next-symbol* $x =$ *Some* $a \Longrightarrow$
Item $r\ (b + 1)\ i\ (j + 1) \in$ *Earley* $\mathcal{G}$ ω
| *Predict*: $x =$ *Item* $r\ b\ i\ j \Longrightarrow x \in$ *Earley* $\mathcal{G}$ $\omega \Longrightarrow$
$r' \in$ *set* $(\mathfrak{R}\ \mathcal{G}) \Longrightarrow$ *next-symbol* $x =$ *Some* (*rule-head* r') $\Longrightarrow$
Item $r'\ 0\ j\ j \in$ *Earley* $\mathcal{G}$ ω
| *Complete*: $x =$ *Item* $r_x\ b_x\ i\ j \Longrightarrow x \in$ *Earley* $\mathcal{G}$ $\omega \Longrightarrow y =$ *Item* $r_y\ b_y\ j\ k \Longrightarrow$
$y \in$ *Earley* $\mathcal{G}$ $\omega \Longrightarrow$
is-complete $y \Longrightarrow$ *next-symbol* $x =$ *Some* (*item-rule-head* y) $\Longrightarrow$
Item $r_x\ (b_x + 1)\ i\ k \in$ *Earley* $\mathcal{G}$ ω

6.2 Well-formedness

definition *wf-item* :: *'a cfg ⇒ 'a sentence => 'a item ⇒ bool* **where**
wf-item $\mathcal{G}$ ω $x \equiv$
item-rule $x \in$ *set* $(\mathfrak{R}\ \mathcal{G}) \wedge$
item-dot $x \leq$ *length* (*item-rule-body* x) $\wedge$
item-origin $x \leq$ *item-end* x $\wedge$
item-end $x \leq$ *length* ω

lemma *wf-Init*:
assumes $r \in$ *set* $(\mathfrak{R}\ \mathcal{G})$ *fst* $r = \mathfrak{S}\ \mathcal{G}$
shows *wf-item* $\mathcal{G}$ ω (*Item* $r\ 0\ 0\ 0$)
using *assms* **unfolding** *wf-item-def* **by** *simp*

lemma *wf-Scan*:
assumes $x =$ *Item* $r\ b\ i\ j$ *wf-item* $\mathcal{G}$ ω x $\omega!j = a$ $j <$ *length* ω *next-symbol* $x =$ *Some* a
shows *wf-item* $\mathcal{G}$ ω (*Item* $r\ (b + 1)\ i\ (j+1)$)
using *assms* **unfolding** *wf-item-def* **by** (*auto simp*: *item-defs is-complete-def next-symbol-def split*: *if-splits*)

lemma *wf-Predict*:
assumes $x = Item\ r\ b\ i\ j$ *wf-item* $\mathcal{G}\ \omega\ x\ r' \in set\ (\mathfrak{R}\ \mathcal{G})$ *next-symbol* $x = Some\ (rule\text{-}head\ r')$
shows *wf-item* $\mathcal{G}\ \omega\ (Item\ r'\ 0\ j\ j)$
using *assms* **unfolding** *wf-item-def* **by** *simp*

lemma *wf-Complete*:
assumes $x = Item\ r_x\ b_x\ i\ j$ *wf-item* $\mathcal{G}\ \omega\ x\ y = Item\ r_y\ b_y\ j\ k$ *wf-item* $\mathcal{G}\ \omega\ y$
assumes *is-complete y next-symbol* $x = Some\ (item\text{-}rule\text{-}head\ y)$
shows *wf-item* $\mathcal{G}\ \omega\ (Item\ r_x\ (b_x + 1)\ i\ k)$
using *assms* **unfolding** *wf-item-def is-complete-def next-symbol-def item-rule-body-def*
by (*auto split*: *if-splits*)

lemma *wf-Earley*:
assumes $x \in Earley\ \mathcal{G}\ \omega$
shows *wf-item* $\mathcal{G}\ \omega\ x$
using *assms wf-Init wf-Scan wf-Predict wf-Complete*
by (*induction rule*: *Earley.induct*) *fast+*

6.3 Soundness

definition *sound-item* :: *'a cfg* ⇒ *'a sentence* ⇒ *'a item* ⇒ *bool* **where**
sound-item $\mathcal{G}\ \omega\ x \equiv derives\ \mathcal{G}\ [item\text{-}rule\text{-}head\ x]\ (slice\ (item\text{-}origin\ x)\ (item\text{-}end\ x)\ \omega\ @\ item\text{-}\beta\ x)$

lemma *sound-Init*:
assumes $r \in set\ (\mathfrak{R}\ \mathcal{G})$ *fst* $r = \mathfrak{S}\ \mathcal{G}$
shows *sound-item* $\mathcal{G}\ \omega\ (Item\ r\ 0\ 0\ 0)$
proof −
let *?x* = $Item\ r\ 0\ 0\ 0$
have (*item-rule-head ?x*, *item-β ?x*) ∈ *set* $(\mathfrak{R}\ \mathcal{G})$
using *assms*(*1*) **by** (*simp add*: *item-defs*)
hence *derives* $\mathcal{G}$ [*item-rule-head ?x*] (*item-β ?x*)
using *derives-if-valid-rule* **by** *metis*
thus *sound-item* $\mathcal{G}\ \omega$ *?x*
unfolding *sound-item-def* **by** (*simp add*: *slice-empty*)
qed

lemma *sound-Scan*:
assumes $x = Item\ r\ b\ i\ j$ *wf-item* $\mathcal{G}\ \omega\ x$ *sound-item* $\mathcal{G}\ \omega\ x$
assumes $\omega!j = a\ j < length\ \omega$ *next-symbol* $x = Some\ a$
shows *sound-item* $\mathcal{G}\ \omega\ (Item\ r\ (b+1)\ i\ (j+1))$
proof −
define x' **where** [*simp*]: $x' = Item\ r\ (b+1)\ i\ (j+1)$
obtain *item-β′* **where** ∗: *item-β x* = a # *item-β′ item-β x′* = *item-β′*
using *assms*(*1*,*6*) **apply** (*auto simp*: *item-defs next-symbol-def is-complete-def split*: *if-splits*)
by (*metis Cons-nth-drop-Suc leI*)
have *slice i j ω @ item-β x* = *slice i* (*j*+*1*) *ω @ item-β′*

using ∗ *assms*(*1*,*2*,*4*,*5*) **by** (*auto simp*: *slice-append-nth wf-item-def*)
moreover have *derives* $\mathcal{G}$ [*item-rule-head x*] (*slice i j* ω @ *item-*β *x*)
using *assms*(*1*,*3*) *sound-item-def* **by** *force*
ultimately show *?thesis*
using *assms*(*1*) ∗ **by** (*auto simp*: *item-defs sound-item-def*)
qed

lemma *sound-Predict*:
assumes x = *Item r b i j wf-item* $\mathcal{G}$ ω x *sound-item* $\mathcal{G}$ ω x
assumes $r' \in$ *set* ($\mathfrak{R}$ $\mathcal{G}$) *next-symbol x* = *Some* (*rule-head r'*)
shows *sound-item* $\mathcal{G}$ ω (*Item r' 0 j j*)
using *assms* **by** (*auto simp*: *sound-item-def derives-if-valid-rule slice-empty item-defs*)

lemma *sound-Complete*:
assumes x = *Item* r_x b_x *i j wf-item* $\mathcal{G}$ ω x *sound-item* $\mathcal{G}$ ω x
assumes y = *Item* r_y b_y *j k wf-item* $\mathcal{G}$ ω y *sound-item* $\mathcal{G}$ ω y
assumes *is-complete y next-symbol x* = *Some* (*item-rule-head y*)
shows *sound-item* $\mathcal{G}$ ω (*Item* r_x (b_x + *1*) *i k*)
proof −
have *derives* $\mathcal{G}$ [*item-rule-head y*] (*slice j k* ω)
using *assms*(*4*,*6*,*7*) **by** (*auto simp*: *sound-item-def is-complete-def item-defs*)
then obtain *E* **where** *E*: *Derivation* $\mathcal{G}$ [*item-rule-head y*] *E* (*slice j k* ω)
using *derives-implies-Derivation* **by** *blast*
have *derives* $\mathcal{G}$ [*item-rule-head x*] (*slice i j* ω @ *item-*β *x*)
using *assms*(*1*,*3*,*4*) **by** (*auto simp*: *sound-item-def*)
moreover have *0*: *item-*β *x* = (*item-rule-head y*) # *tl* (*item-*β *x*)
using *assms*(*8*) **apply** (*auto simp*: *next-symbol-def is-complete-def item-defs split*: *if-splits*)
by (*metis drop-eq-Nil hd-drop-conv-nth leI list.collapse*)
ultimately obtain *D* **where** *D*:
Derivation $\mathcal{G}$ [*item-rule-head x*] *D* (*slice i j* ω @ [*item-rule-head y*] @ (*tl* (*item-*β *x*)))
using *derives-implies-Derivation* **by** (*metis append-Cons append-Nil*)
obtain *F* **where** *F*:
Derivation $\mathcal{G}$ [*item-rule-head x*] *F* (*slice i j* ω @ *slice j k* ω @ *tl* (*item-*β *x*))
using *Derivation-append-rewrite D E* **by** *blast*
moreover have $i \leq j$
using *assms*(*1*,*2*) *wf-item-def* **by** *force*
moreover have $j \leq k$
using *assms*(*4*,*5*) *wf-item-def* **by** *force*
ultimately have *derives* $\mathcal{G}$ [*item-rule-head x*] (*slice i k* ω @ *tl* (*item-*β *x*))
by (*metis Derivation-implies-derives append.assoc slice-concat*)
thus *sound-item* $\mathcal{G}$ ω (*Item* r_x (b_x + *1*) *i k*)
using *assms*(*1*,*4*) **by** (*auto simp*: *sound-item-def item-defs drop-Suc tl-drop*)
qed

lemma *sound-Earley*:
assumes $x \in$ *Earley* $\mathcal{G}$ ω *wf-item* $\mathcal{G}$ ω x
shows *sound-item* $\mathcal{G}$ ω x

using *assms*
proof (*induction rule*: *Earley.induct*)
case (*Init r*)
thus *?case*
using *sound-Init* **by** *blast*
next
case (*Scan x r b i j a*)
thus *?case*
using *wf-Earley sound-Scan* **by** *fast*
next
case (*Predict x r b i j r'*)
thus *?case*
using *wf-Earley sound-Predict* **by** *blast*
next
case (*Complete x* r_x b_x *i j y* r_y b_y *k*)
thus *?case*
using *wf-Earley sound-Complete* **by** *metis*
qed

theorem *soundness-Earley*:
assumes *recognizing* (*Earley* $\mathcal{G}$ ω) $\mathcal{G}$ ω
shows *derives* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω
proof −
obtain *x* **where** *x*: $x \in$ *Earley* $\mathcal{G}$ ω *is-finished* $\mathcal{G}$ ω *x*
using *assms recognizing-def* **by** *blast*
hence *sound-item* $\mathcal{G}$ ω *x*
using *wf-Earley sound-Earley* **by** *blast*
thus *?thesis*
unfolding *sound-item-def* **using** *x* **by** (*auto simp*: *is-finished-def is-complete-def item-defs*)
qed

6.4 Completeness

definition *partially-completed* :: *nat* ⇒ *'a cfg* ⇒ *'a sentence* ⇒ *'a item set* ⇒ (*'a derivation* ⇒ *bool*) ⇒ *bool* **where**
partially-completed k $\mathcal{G}$ ω *E P* ≡ ∀ *r b i' i j x a D*.
$i \leq j \wedge j \leq k \wedge k \leq$ *length* $\omega \wedge$
$x =$ *Item r b i' i* $\wedge$ $x \in E \wedge$ *next-symbol x* = *Some a* $\wedge$
Derivation $\mathcal{G}$ [*a*] *D* (*slice i j* ω) $\wedge$ *P D* ⟶
Item r (*b*+*1*) *i' j* ∈ *E*

lemma *partially-completed-upto*:
assumes $j \leq k$ $k \leq$ *length* ω
assumes $x =$ *Item* (*N*,α) *d i j x* ∈ *I* ∀ $x \in I$. *wf-item* $\mathcal{G}$ ω *x*
assumes *Derivation* $\mathcal{G}$ (*item-*β *x*) *D* (*slice j k* ω)
assumes *partially-completed k* $\mathcal{G}$ ω *I* ($\lambda D'$. *length* $D' \leq$ *length D*)
shows *Item* (*N*,α) (*length* α) *i k* ∈ *I*
using *assms*

proof (*induction item-β x arbitrary*: *d i j k N α x D*)
case *Nil*
have *item-α x* = *α*
using *Nil*(*1*,*4*) **unfolding** *item-α-def item-β-def item-rule-body-def rule-body-def*
by *simp*
hence *x* = *Item* (*N*,*α*) (*length α*) *i j*
using *Nil.hyps Nil.prems*(*3*−*5*) **unfolding** *wf-item-def item-defs* **by** *auto*
have *Derivation G* [] *D* (*slice j k ω*)
using *Nil.hyps Nil.prems*(*6*) **by** *auto*
hence *slice j k ω* = []
using *Derivation-from-empty* **by** *blast*
hence *j* = *k*
unfolding *slice-drop-take* **using** *Nil.prems*(*1*,*2*) **by** *simp*
thus *?case*
using ‹*x* = *Item* (*N*, *α*) (*length α*) *i j*› *Nil.prems*(*4*) **by** *blast*
next
case (*Cons b bs*)
obtain *j′ E F* **where** ∗:
Derivation G [*b*] *E* (*slice j j′ ω*)
Derivation G bs F (*slice j′ k ω*)
j ≤ *j′ j′* ≤ *k length E* ≤ *length D length F* ≤ *length D*
using *Derivation-concat-split*[*of G* [*b*] *bs D slice j k ω*] *slice-concat-Ex*
using *Cons.hyps*(*2*) *Cons.prems*(*1*,*6*)
by (*smt* (*verit*, *ccfv-threshold*) *Cons-eq-appendI append-self-conv2*)
have *next-symbol x* = *Some b*
using *Cons.hyps*(*2*) **unfolding** *item-defs*(*4*) *next-symbol-def is-complete-def*
by (*auto*, *metis nth-via-drop*)
hence *Item* (*N*, *α*) (*d*+*1*) *i j′* ∈ *I*
using *Cons.prems*(*7*) **unfolding** *partially-completed-def*
using *Cons.prems*(*2*,*3*,*4*) ∗(*1*,*3*−*5*) **by** *blast*
moreover have *partially-completed k G ω I* (*λD′. length D′* ≤ *length F*)
using *Cons.prems*(*7*) ∗(*6*) **unfolding** *partially-completed-def* **by** *fastforce*
moreover have *bs* = *item-β* (*Item* (*N*,*α*) (*d*+*1*) *i j′*)
using *Cons.hyps*(*2*) *Cons.prems*(*3*) **unfolding** *item-defs*(*4*) *item-rule-body-def*

by (*auto*, *metis List.list.sel*(*3*) *drop-Suc drop-tl*)
ultimately show *?case*
using *Cons.hyps*(*1*) ∗(*2*,*4*) *Cons.prems*(*2*,*3*,*5*) *wf-item-def* **by** *blast*
qed

lemma *partially-completed-Earley-k*:
assumes *wf-G G*
shows *partially-completed k G ω* (*Earley G ω*) (*λ-. True*)
unfolding *partially-completed-def*
proof (*standard*, *standard*, *standard*, *standard*, *standard*, *standard*, *standard*, *standard*, *standard*)
fix *r b i′ i j x a D*
assume
i ≤ *j* ∧ *j* ≤ *k* ∧ *k* ≤ *length ω* ∧

```
    x = Item r b i' i ∧ x ∈ Earley G ω ∧
    next-symbol x = Some a ∧
    Derivation G [a] D (slice i j ω) ∧ True
  thus Item r (b + 1) i' j ∈ Earley G ω
  proof (induction length D arbitrary: r b i' i j x a D rule: nat-less-induct)
    case 1
    show ?case
    proof cases
      assume D = []
      hence [a] = slice i j ω
        using 1.prems by force
      moreover have j ≤ length ω
        using le-trans 1.prems by blast
      ultimately have j = i+1
        using slice-singleton by metis
      hence i < length ω
        using ‹j ≤ length ω› by simp
      hence ω!i = a
        using slice-nth ‹[a] = slice i j ω› ‹j = i + 1› by fastforce
      hence Item r (b + 1) i' j ∈ Earley G ω
        using Earley.Scan 1.prems ‹i < length ω› ‹j = i + 1› by metis
      thus ?thesis
        by (simp add: ‹j = i + 1›)
    next
      assume ¬ D = []
      then obtain d D' where D = d # D'
        by (meson List.list.exhaust)
      then obtain α where ∗: Derives1 G [a] (fst d) (snd d) α Derivation G α D'
(slice i j ω)
        using 1.prems by auto
      hence rule: (a, α) ∈ set (ℜ G) fst d = 0 snd d = (a ,α)
        using ∗(1) unfolding Derives1-def by (simp add: Cons-eq-append-conv)+
      show ?thesis
      proof cases
        assume is-terminal G a
        have is-nonterminal G a
          using rule by (simp add: assms)
        thus ?thesis
          using ‹is-terminal G a› is-terminal-nonterminal by (metis assms)
      next
        assume ¬ is-terminal G a
        define y where y-def: y = Item (a ,α) 0 i i
        have length D' < length D
          using ‹D = d # D'› by fastforce
        hence partially-completed k G ω (Earley G ω) (λE. length E ≤ length D')
         unfolding partially-completed-def using 1.hyps order-le-less-trans by (smt
(verit, best))
        hence partially-completed j G ω (Earley G ω) (λE. length E ≤ length D')
          unfolding partially-completed-def using 1.prems by force
```

```
      moreover have Derivation 𝒢 (item-β y) D' (slice i j ω)
        using ∗(2) by (auto simp: item-defs y-def)
      moreover have y ∈ Earley 𝒢 ω
        using y-def 1.prems rule by (auto simp: item-defs Earley.Predict)
      moreover have j ≤ length ω
        using 1.prems by simp
      ultimately have Item (a,α) (length α) i j ∈ Earley 𝒢 ω
        using partially-completed-upto 1.prems wf-Earley y-def by metis
      moreover have x: x = Item r b i' i x ∈ Earley 𝒢 ω
        using 1.prems by blast+
      moreover have next-symbol x = Some a
        using 1.prems by linarith
      ultimately show ?thesis
        using Earley.Complete[OF x] by (auto simp: is-complete-def item-defs)
    qed
  qed
 qed
qed

lemma partially-completed-Earley:
 wf-𝒢 𝒢 ⟹ partially-completed (length ω) 𝒢 ω (Earley 𝒢 ω) (λ-. True)
 by (simp add: partially-completed-Earley-k)

theorem completeness-Earley:
 assumes derives 𝒢 [𝔖 𝒢] ω is-word 𝒢 ω wf-𝒢 𝒢
 shows recognizing (Earley 𝒢 ω) 𝒢 ω
proof −
 obtain α D where ∗: (𝔖 𝒢 ,α) ∈ set (ℜ 𝒢) Derivation 𝒢 α D ω
   using Derivation-𝔖1 assms derives-implies-Derivation by metis
 define x where x-def: x = Item (𝔖 𝒢, α) 0 0 0
 have partially-completed (length ω) 𝒢 ω (Earley 𝒢 ω) (λ-. True)
   using assms(3) partially-completed-Earley by blast
  hence 0: partially-completed (length ω) 𝒢 ω (Earley 𝒢 ω) (λD'. length D' ≤
length D)
   unfolding partially-completed-def by blast
 have 1: x ∈ Earley 𝒢 ω
   using x-def Earley.Init ∗(1) by fastforce
 have 2: Derivation 𝒢 (item-β x) D (slice 0 (length ω) ω)
   using ∗(2) x-def by (simp add: item-defs)
 have Item (𝔖 𝒢,α) (length α) 0 (length ω) ∈ Earley 𝒢 ω
   using partially-completed-upto[OF - - - - - 2 0] wf-Earley 1 x-def by auto
 then show ?thesis
  unfolding recognizing-def is-finished-def by (auto simp: is-complete-def item-defs,
force)
qed
```

6.5 Correctness

theorem *correctness-Earley*:

assumes *wf-𝒢 𝒢 is-word 𝒢 ω*
shows *recognizing* (*Earley 𝒢 ω*) *𝒢 ω* ⟷ *derives 𝒢* [𝔖 *𝒢*] *ω*
using *assms soundness-Earley completeness-Earley* **by** *blast*

6.6 Finiteness

lemma *finiteness-empty*:
set (ℜ *𝒢*) = {} ⟹ *finite* { *x* | *x*. *wf-item 𝒢 ω x* }
unfolding *wf-item-def* **by** *simp*

fun *item-intro* :: *'a rule* × *nat* × *nat* × *nat* ⇒ *'a item* **where**
item-intro (*rule*, *dot*, *origin*, *ends*) = *Item rule dot origin ends*

lemma *finiteness-nonempty*:
assumes *set* (ℜ *𝒢*) ≠ {}
shows *finite* { *x* | *x*. *wf-item 𝒢 ω x* }
proof −
define *M* **where** *M* = *Max* { *length* (*rule-body r*) | *r*. *r* ∈ *set* (ℜ *𝒢*) }
define *Top* **where** *Top* = (*set* (ℜ *𝒢*) × {*0*..*M*} × {*0*..*length ω*} × {*0*..*length ω*})
hence *finite Top*
using *finite-cartesian-product finite* **by** *blast*
have *inj-on item-intro Top*
unfolding *Top-def inj-on-def* **by** *simp*
hence *finite* (*item-intro* ‘ *Top*)
using *finite-image-iff* ‹*finite Top*› **by** *auto*
have { *x* | *x*. *wf-item 𝒢 ω x* } ⊆ *item-intro* ‘ *Top*
proof *standard*
fix *x*
assume *x* ∈ { *x* | *x*. *wf-item 𝒢 ω x* }
then obtain *rule dot origin endp* **where** ∗: *x* = *Item rule dot origin endp*
rule ∈ *set* (ℜ *𝒢*) *dot* ≤ *length* (*item-rule-body x*) *origin* ≤ *length ω endp* ≤ *length ω*
unfolding *wf-item-def* **using** *item.exhaust-sel le-trans* **by** *blast*
hence *length* (*rule-body rule*) ∈ { *length* (*rule-body r*) | *r*. *r* ∈ *set* (ℜ *𝒢*) }
using ∗(*1*,*2*) *item-rule-body-def* **by** *blast*
moreover have *finite* { *length* (*rule-body r*) | *r*. *r* ∈ *set* (ℜ *𝒢*) }
using *finite finite-image-set*[*of* λ*x*. *x* ∈ *set* (ℜ *𝒢*)] **by** *fastforce*
ultimately have *M* ≥ *length* (*rule-body rule*)
unfolding *M-def* **by** *simp*
hence *dot* ≤ *M*
using ∗(*1*,*3*) *item-rule-body-def* **by** (*metis item.sel*(*1*) *le-trans*)
hence (*rule*, *dot*, *origin*, *endp*) ∈ *Top*
using ∗(*2*,*4*,*5*) **unfolding** *Top-def* **by** *simp*
thus *x* ∈ *item-intro* ‘ *Top*
using ∗(*1*) **by** *force*
qed
thus *?thesis*
using ‹*finite* (*item-intro* ‘ *Top*)› *rev-finite-subset* **by** *auto*

qed

lemma *finiteness-UNIV-wf-item*:
finite { *x* | *x*. *wf-item* $\mathcal{G}$ ω *x* }
using *finiteness-empty finiteness-nonempty* **by** *fastforce*

theorem *finiteness-Earley*:
finite (*Earley* $\mathcal{G}$ ω)
using *finiteness-UNIV-wf-item wf-Earley rev-finite-subset* **by** (*metis mem-Collect-eq subsetI*)

end
theory *Earley-Fixpoint*
imports
Earley
Limit
begin

7 Earley recognizer

7.1 Earley fixpoint

definition *init-item* :: *'a rule* ⇒ *nat* ⇒ *'a item* **where**
init-item r k ≡ *Item r 0 k k*

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

definition *bin* :: *'a item set* ⇒ *nat* ⇒ *'a item set* **where**
bin I k ≡ { *x* . *x* ∈ *I* ∧ *item-end x* = *k* }

definition $Init_F$ :: *'a cfg* ⇒ *'a item set* **where**
$Init_F$ $\mathcal{G}$ ≡ { *init-item r 0* | *r*. *r* ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$) ∧ *fst r* = ($\mathfrak{S}$ $\mathcal{G}$) }

definition $Scan_F$ :: *nat* ⇒ *'a sentence* ⇒ *'a item set* ⇒ *'a item set* **where**
$Scan_F$ *k* ω *I* ≡ { *inc-item x* (*k*+*1*) | *x a*.
x ∈ *bin I k* ∧
ω!*k* = *a* ∧
k < *length* ω ∧
next-symbol x = *Some a* }

definition $Predict_F$ :: *nat* ⇒ *'a cfg* ⇒ *'a item set* ⇒ *'a item set* **where**
$Predict_F$ *k* $\mathcal{G}$ *I* ≡ { *init-item r k* | *r x*.
r ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$) ∧
x ∈ *bin I k* ∧
next-symbol x = *Some* (*rule-head r*) }

definition $Complete_F$ :: *nat* ⇒ *'a item set* ⇒ *'a item set* **where**
$Complete_F$ *k I* ≡ { *inc-item x k* | *x y*.

$x \in bin\ I\ (item\text{-}origin\ y) \wedge$
$y \in bin\ I\ k \wedge$
$is\text{-}complete\ y \wedge$
$next\text{-}symbol\ x = Some\ (item\text{-}rule\text{-}head\ y)\ \}$

definition $Earley_F\text{-}bin\text{-}step :: nat \Rightarrow 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow 'a\ item\ set \Rightarrow 'a\ item\ set$ **where**
$Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ I \equiv I \cup Scan_F\ k\ \omega\ I \cup Complete_F\ k\ I \cup Predict_F\ k\ \mathcal{G}\ I$

definition $Earley_F\text{-}bin :: nat \Rightarrow 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow 'a\ item\ set \Rightarrow 'a\ item\ set$ **where**
$Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I \equiv limit\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega)\ I$

fun $Earley_F\text{-}bins :: nat \Rightarrow 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow 'a\ item\ set$ **where**
$Earley_F\text{-}bins\ 0\ \mathcal{G}\ \omega = Earley_F\text{-}bin\ 0\ \mathcal{G}\ \omega\ (Init_F\ \mathcal{G})$
$| Earley_F\text{-}bins\ (Suc\ n)\ \mathcal{G}\ \omega = Earley_F\text{-}bin\ (Suc\ n)\ \mathcal{G}\ \omega\ (Earley_F\text{-}bins\ n\ \mathcal{G}\ \omega)$

definition $Earley_F :: 'a\ cfg \Rightarrow 'a\ sentence \Rightarrow 'a\ item\ set$ **where**
$Earley_F\ \mathcal{G}\ \omega \equiv Earley_F\text{-}bins\ (length\ \omega)\ \mathcal{G}\ \omega$

7.2 Monotonicity and Absorption

lemma $Earley_F\text{-}bin\text{-}step\text{-}empty$:
$Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ \{\} = \{\}$
unfolding $Earley_F\text{-}bin\text{-}step\text{-}def\ Scan_F\text{-}def\ Complete_F\text{-}def\ Predict_F\text{-}def\ bin\text{-}def$ **by** *blast*

lemma $Earley_F\text{-}bin\text{-}step\text{-}setmonotone$:
$setmonotone\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega)$
by (*simp add*: $Un\text{-}assoc\ Earley_F\text{-}bin\text{-}step\text{-}def\ setmonotone\text{-}def$)

lemma $Earley_F\text{-}bin\text{-}step\text{-}continuous$:
$continuous\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega)$
unfolding *continuous-def*
proof (*standard*, *standard*, *standard*)
fix $C :: nat \Rightarrow 'a\ item\ set$
assume *chain C*
thus $chain\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega \circ C)$
unfolding $chain\text{-}def\ Earley_F\text{-}bin\text{-}step\text{-}def$ **by** (*auto simp*: $Scan_F\text{-}def\ Predict_F\text{-}def\ Complete_F\text{-}def\ bin\text{-}def\ subset\text{-}eq$)
next
fix $C :: nat \Rightarrow 'a\ item\ set$
assume $*$: *chain C*
show $Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ (natUnion\ C) = natUnion\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega \circ C)$
unfolding *natUnion-def*
proof *standard*
show $Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ (\bigcup\ \{C\ n\ |n.\ True\}) \subseteq \bigcup\ \{(Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega \circ C)\ n\ |n.\ True\}$

```
    proof standard
      fix x
      assume #: x ∈ Earley_F-bin-step k G ω (⋃ {C n |n. True})
      show x ∈ ⋃ {(Earley_F-bin-step k G ω ∘ C) n |n. True}
      proof (cases x ∈ Complete_F k (⋃ {C n |n. True}))
        case True
        then show ?thesis
          using ∗ unfolding chain-def Earley_F-bin-step-def Complete_F-def bin-def
        proof clarsimp
          fix y :: 'a item and z :: 'a item and n :: nat and m :: nat
          assume a1: is-complete z
          assume a2: item-end y = item-origin z
          assume a3: y ∈ C n
          assume a4: z ∈ C m
          assume a5: next-symbol y = Some (item-rule-head z)
          assume ∀ i. C i ⊆ C (Suc i)
          hence f6: ⋀n m. ¬ n ≤ m ∨ C n ⊆ C m
            by (meson lift-Suc-mono-le)
          hence f7: ⋀n. ¬ m ≤ n ∨ z ∈ C n
            using a4 by blast
          have ∃ n ≥ m. y ∈ C n
            using f6 a3 by (meson le-sup-iff subset-eq sup-ge1)
          thus ∃ I.
                (∃ n. I = C n ∪
                        Scan_F (item-end z) ω (C n) ∪
                        {inc-item i (item-end z) |i.
                           i ∈ C n ∧
                           (∃ j.
                             item-end i = item-origin j ∧
                             j ∈ C n ∧
                             item-end j = item-end z ∧
                             is-complete j ∧
                             next-symbol i = Some (item-rule-head j))} ∪
                        Predict_F (item-end z) G (C n))
                ∧ inc-item y (item-end z) ∈ I
            using f7 a5 a2 a1 by blast
        qed
      next
        case False
        thus ?thesis
        using # Un-iff by (auto simp: Earley_F-bin-step-def Scan_F-def Predict_F-def
bin-def; blast)
      qed
    qed
  next
    show ⋃ {(Earley_F-bin-step k G ω ∘ C) n |n. True} ⊆ Earley_F-bin-step k G ω
(⋃ {C n |n. True})
      unfolding Earley_F-bin-step-def
        using ∗ by (auto simp: Scan_F-def Predict_F-def Complete_F-def chain-def
```

bin-def, *metis+*)
qed
qed

lemma $Earley_F$*-bin-step-regular*:
regular (*Earley*$_F$*-bin-step k* $\mathcal{G}$ ω)
by (*simp add*: *Earley*$_F$*-bin-step-continuous Earley*$_F$*-bin-step-setmonotone regular-def*)

lemma $Earley_F$*-bin-idem*:
Earley$_F$*-bin k* $\mathcal{G}$ ω (*Earley*$_F$*-bin k* $\mathcal{G}$ ω *I*) = *Earley*$_F$*-bin k* $\mathcal{G}$ ω *I*
by (*simp add*: *Earley*$_F$*-bin-def Earley*$_F$*-bin-step-regular limit-is-idempotent*)

lemma $Scan_F$*-bin-absorb*:
$Scan_F$ *k* ω (*bin I k*) = $Scan_F$ *k* ω *I*
unfolding $Scan_F$*-def bin-def* **by** *simp*

lemma $Predict_F$*-bin-absorb*:
$Predict_F$ *k* $\mathcal{G}$ (*bin I k*) = $Predict_F$ *k* $\mathcal{G}$ *I*
unfolding $Predict_F$*-def bin-def* **by** *simp*

lemma $Scan_F$*-Un*:
$Scan_F$ *k* ω (*I* $\cup$ *J*) = $Scan_F$ *k* ω *I* $\cup$ $Scan_F$ *k* ω *J*
unfolding $Scan_F$*-def bin-def* **by** *blast*

lemma $Predict_F$*-Un*:
$Predict_F$ *k* $\mathcal{G}$ (*I* $\cup$ *J*) = $Predict_F$ *k* $\mathcal{G}$ *I* $\cup$ $Predict_F$ *k* $\mathcal{G}$ *J*
unfolding $Predict_F$*-def bin-def* **by** *blast*

lemma $Scan_F$*-sub-mono*:
I $\subseteq$ *J* $\Longrightarrow$ $Scan_F$ *k* ω *I* $\subseteq$ $Scan_F$ *k* ω *J*
unfolding $Scan_F$*-def bin-def* **by** *blast*

lemma $Predict_F$*-sub-mono*:
I $\subseteq$ *J* $\Longrightarrow$ $Predict_F$ *k* $\mathcal{G}$ *I* $\subseteq$ $Predict_F$ *k* $\mathcal{G}$ *J*
unfolding $Predict_F$*-def bin-def* **by** *blast*

lemma $Complete_F$*-sub-mono*:
I $\subseteq$ *J* $\Longrightarrow$ $Complete_F$ *k I* $\subseteq$ $Complete_F$ *k J*
unfolding $Complete_F$*-def bin-def* **by** *blast*

lemma $Earley_F$*-bin-step-sub-mono*:
I $\subseteq$ *J* $\Longrightarrow$ *Earley*$_F$*-bin-step k* $\mathcal{G}$ ω *I* $\subseteq$ *Earley*$_F$*-bin-step k* $\mathcal{G}$ ω *J*
unfolding *Earley*$_F$*-bin-step-def* **using** $Scan_F$*-sub-mono* $Predict_F$*-sub-mono* $Complete_F$*-sub-mono* **by** (*metis sup.mono*)

lemma *funpower-sub-mono*:
I $\subseteq$ *J* $\Longrightarrow$ *funpower* (*Earley*$_F$*-bin-step k* $\mathcal{G}$ ω) *n I* $\subseteq$ *funpower* (*Earley*$_F$*-bin-step k* $\mathcal{G}$ ω) *n J*

by (*induction n*) (*auto simp*: $Earley_F$*-bin-step-sub-mono*)

lemma $Earley_F$*-bin-sub-mono*:
$I \subseteq J \Longrightarrow Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I \subseteq Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ J$
proof *standard*
fix *x*
assume $I \subseteq J\ x \in Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I$
then obtain *n* **where** $x \in funpower\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega)\ n\ I$
unfolding $Earley_F$*-bin-def limit-def natUnion-def* **by** *blast*
hence $x \in funpower\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega)\ n\ J$
using ‹$I \subseteq J$› *funpower-sub-mono* **by** *blast*
thus $x \in Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ J$
unfolding $Earley_F$*-bin-def limit-def natUnion-def* **by** *blast*
qed

lemma $Scan_F$*-*$Earley_F$*-bin-step-mono*:
$Scan_F\ k\ \omega\ I \subseteq Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ I$
using $Earley_F$*-bin-step-def* **by** *blast*

lemma $Predict_F$*-*$Earley_F$*-bin-step-mono*:
$Predict_F\ k\ \mathcal{G}\ I \subseteq Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ I$
using $Earley_F$*-bin-step-def* **by** *blast*

lemma $Complete_F$*-*$Earley_F$*-bin-step-mono*:
$Complete_F\ k\ I \subseteq Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ I$
using $Earley_F$*-bin-step-def* **by** *blast*

lemma $Earley_F$*-bin-step-*$Earley_F$*-bin-mono*:
$Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ I \subseteq Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I$
proof −
have $Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega\ I \subseteq funpower\ (Earley_F\text{-}bin\text{-}step\ k\ \mathcal{G}\ \omega)\ 1\ I$
by *simp*
thus *?thesis*
by (*metis* $Earley_F$*-bin-def limit-elem subset-eq*)
qed

lemma $Scan_F$*-*$Earley_F$*-bin-mono*:
$Scan_F\ k\ \omega\ \ I \subseteq Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I$
using $Scan_F$*-*$Earley_F$*-bin-step-mono* $Earley_F$*-bin-step-*$Earley_F$*-bin-mono* **by** *force*

lemma $Predict_F$*-*$Earley_F$*-bin-mono*:
$Predict_F\ k\ \mathcal{G}\ I \subseteq Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I$
using $Predict_F$*-*$Earley_F$*-bin-step-mono* $Earley_F$*-bin-step-*$Earley_F$*-bin-mono* **by** *force*

lemma $Complete_F$*-*$Earley_F$*-bin-mono*:
$Complete_F\ k\ I \subseteq Earley_F\text{-}bin\ k\ \mathcal{G}\ \omega\ I$
using $Complete_F$*-*$Earley_F$*-bin-step-mono* $Earley_F$*-bin-step-*$Earley_F$*-bin-mono* **by** *force*

lemma *Earley$_F$-bin-mono*:
I ⊆ *Earley$_F$-bin k* 𝒢 *ω I*
using *Earley$_F$-bin-step-Earley$_F$-bin-mono Earley$_F$-bin-step-def* **by** *blast*

lemma *Init$_F$-sub-Earley$_F$-bins*:
Init$_F$ 𝒢 ⊆ *Earley$_F$-bins n* 𝒢 *ω*
by (*induction n*) (*use Earley$_F$-bin-mono* **in** *fastforce*)+

7.3 Soundness

lemma *Init$_F$-sub-Earley*:
Init$_F$ 𝒢 ⊆ *Earley* 𝒢 *ω*
unfolding *Init$_F$-def init-item-def* **using** *Init* **by** *blast*

lemma *Scan$_F$-sub-Earley*:
assumes *I* ⊆ *Earley* 𝒢 *ω*
shows *Scan$_F$ k ω I* ⊆ *Earley* 𝒢 *ω*
unfolding *Scan$_F$-def inc-item-def bin-def* **using** *assms Scan*
by (*smt* (*verit*, *ccfv-SIG*) *item.exhaust-sel mem-Collect-eq subsetD subsetI*)

lemma *Predict$_F$-sub-Earley*:
assumes *I* ⊆ *Earley* 𝒢 *ω*
shows *Predict$_F$ k* 𝒢 *I* ⊆ *Earley* 𝒢 *ω*
unfolding *Predict$_F$-def init-item-def bin-def* **using** *assms Predict*
using *item.exhaust-sel* **by** *blast*

lemma *Complete$_F$-sub-Earley*:
assumes *I* ⊆ *Earley* 𝒢 *ω*
shows *Complete$_F$ k I* ⊆ *Earley* 𝒢 *ω*
unfolding *Complete$_F$-def inc-item-def bin-def* **using** *assms Complete*
by (*smt* (*verit*, *del-insts*) *item.exhaust-sel mem-Collect-eq subset-eq*)

lemma *Earley$_F$-bin-step-sub-Earley*:
assumes *I* ⊆ *Earley* 𝒢 *ω*
shows *Earley$_F$-bin-step k* 𝒢 *ω I* ⊆ *Earley* 𝒢 *ω*
unfolding *Earley$_F$-bin-step-def* **using** *assms Complete$_F$-sub-Earley Predict$_F$-sub-Earley Scan$_F$-sub-Earley* **by** (*metis le-supI*)

lemma *Earley$_F$-bin-sub-Earley*:
assumes *I* ⊆ *Earley* 𝒢 *ω*
shows *Earley$_F$-bin k* 𝒢 *ω I* ⊆ *Earley* 𝒢 *ω*
using *assms Earley$_F$-bin-step-sub-Earley* **by** (*metis Earley$_F$-bin-def limit-upperbound*)

lemma *Earley$_F$-bins-sub-Earley*:
shows *Earley$_F$-bins n* 𝒢 *ω* ⊆ *Earley* 𝒢 *ω*
by (*induction n*) (*auto simp*: *Earley$_F$-bin-sub-Earley Init$_F$-sub-Earley*)

lemma *Earley$_F$-sub-Earley*:

shows $Earley_F\ \mathcal{G}\ \omega \subseteq Earley\ \mathcal{G}\ \omega$
by (*simp add*: $Earley_F$*-bins-sub-Earley* $Earley_F$*-def*)

theorem *soundness-*$Earley_F$:
assumes *recognizing* ($Earley_F\ \mathcal{G}\ \omega$) $\mathcal{G}\ \omega$
shows *derives* $\mathcal{G}$ [$\mathfrak{S}\ \mathcal{G}$] ω
using *soundness-Earley* $Earley_F$*-sub-Earley assms recognizing-def* **by** (*metis subsetD*)

7.4 Completeness

definition *prev-symbol* :: *'a item* ⇒ *'a option* **where**
prev-symbol x ≡ *if item-dot x* = *0 then None else Some* (*item-rule-body x* ! (*item-dot x* − *1*))

definition *base* :: *'a sentence* ⇒ *'a item set* ⇒ *nat* ⇒ *'a item set* **where**
base ω *I k* ≡ { *x* . *x* ∈ *I* ∧ *item-end x* = *k* ∧ *k* > *0* ∧ *prev-symbol x* = *Some* (ω!(*k*−*1*)) }

lemma $Earley_F$*-bin-sub-*$Earley_F$*-bin*:
assumes $Init_F\ \mathcal{G} \subseteq I$
assumes $\forall\, k' < k.\ bin\ (Earley\ \mathcal{G}\ \omega)\ k' \subseteq I$
assumes *base* ω (*Earley* $\mathcal{G}$ ω) *k* ⊆ *I*
shows *bin* (*Earley* $\mathcal{G}$ ω) *k* ⊆ *bin* ($Earley_F$*-bin k* $\mathcal{G}$ ω *I*) *k*
proof *standard*
fix *x*
assume ∗: *x* ∈ *bin* (*Earley* $\mathcal{G}$ ω) *k*
hence *x* ∈ *Earley* $\mathcal{G}$ ω
using *bin-def* **by** *blast*
thus *x* ∈ *bin* ($Earley_F$*-bin k* $\mathcal{G}$ ω *I*) *k*
using *assms* ∗
proof (*induction rule*: *Earley.induct*)
case (*Init r*)
thus *?case*
unfolding $Init_F$*-def init-item-def bin-def* **using** $Earley_F$*-bin-mono* **by** *fast*
next
case (*Scan x r b i j a*)
have *j*+*1* = *k*
using *Scan.prems*(*4*) *bin-def* **by** (*metis* (*mono-tags*, *lifting*) *CollectD item.sel*(*4*))
have *prev-symbol* (*Item r* (*b*+*1*) *i* (*j*+*1*)) = *Some* (ω!(*k*−*1*))
using *Scan.hyps*(*1*,*3*,*5*) ‹*j*+*1* = *k*› **by** (*auto simp*: *next-symbol-def prev-symbol-def item-rule-body-def split*: *if-splits*)
hence *Item r* (*b*+*1*) *i* (*j*+*1*) ∈ *base* ω (*Earley* $\mathcal{G}$ ω) *k*
unfolding *base-def* **using** *Scan.prems*(*4*) *bin-def* **by** *fastforce*
hence *Item r* (*b*+*1*) *i* (*j*+*1*) ∈ *I*
using *Scan.prems*(*3*) **by** *blast*
hence *Item r* (*b*+*1*) *i* (*j*+*1*) ∈ $Earley_F$*-bin k* $\mathcal{G}$ ω *I*
using $Earley_F$*-bin-mono* **by** *blast*
thus *?case*

```
    using ‹j+1 = k› bin-def by fastforce
  next
    case (Predict x r b i j r')
    have j = k
      using Predict.prems(4) bin-def by (metis (mono-tags, lifting) CollectD
item.sel(4))
    hence x ∈ bin (Earley 𝒢 ω) k
      using Predict.hyps(1,2) bin-def by fastforce
    hence x ∈ bin (Earley_F-bin k 𝒢 ω I) k
      using Predict.IH Predict.prems(1−3) by blast
    hence Item r' 0 j j ∈ Predict_F k 𝒢 (Earley_F-bin k 𝒢 ω I)
      unfolding Predict_F-def init-item-def using Predict.hyps(1,3,4) ‹j = k› by
blast
    hence Item r' 0 j j ∈ Earley_F-bin-step k 𝒢 ω (Earley_F-bin k 𝒢 ω I)
      using Predict_F-Earley_F-bin-step-mono by blast
    hence Item r' 0 j j ∈ Earley_F-bin k 𝒢 ω I
      using Earley_F-bin-idem Earley_F-bin-step-Earley_F-bin-mono by blast
    thus ?case
      by (simp add: ‹j = k› bin-def)
  next
    case (Complete x r_x b_x i j y r_y b_y l)
    have l = k
      using Complete.prems(4) bin-def by (metis (mono-tags, lifting) CollectD
item.sel(4))
    hence y ∈ bin (Earley 𝒢 ω) l
      using Complete.hyps(3,4) bin-def by fastforce
    hence 0: y ∈ bin (Earley_F-bin k 𝒢 ω I) k
      using Complete.IH(2) Complete.prems(1−3) ‹l = k› by blast
    have 1: x ∈ bin (Earley_F-bin k 𝒢 ω I) (item-origin y)
    proof (cases j = k)
      case True
      hence x ∈ bin (Earley 𝒢 ω) k
        using Complete.hyps(1,2) bin-def by fastforce
      hence x ∈ bin (Earley_F-bin k 𝒢 ω I) k
        using Complete.IH(1) Complete.prems(1−3) by blast
      thus ?thesis
        using Complete.hyps(3) True by simp
    next
      case False
      hence j < k
        using ‹l = k› wf-Earley wf-item-def Complete.hyps(3,4) by force
      moreover have x ∈ bin (Earley 𝒢 ω) j
        using Complete.hyps(1,2) bin-def by force
      ultimately have x ∈ I
        using Complete.prems(2) by blast
      hence x ∈ bin (Earley_F-bin k 𝒢 ω I) j
        using Complete.hyps(1) Earley_F-bin-mono bin-def by fastforce
      thus ?thesis
        using Complete.hyps(3) by simp
```

```
    qed
    have Item r_x (b_x + 1) i k ∈ Complete_F k (Earley_F-bin k G ω I)
      unfolding Complete_F-def inc-item-def using 0 1 Complete.hyps(1,5,6) by
force
    hence Item r_x (b_x + 1) i k ∈ Earley_F-bin-step k G ω (Earley_F-bin k G ω I)
      unfolding Earley_F-bin-step-def by blast
    hence Item r_x (b_x + 1) i k ∈ Earley_F-bin k G ω I
      using Earley_F-bin-idem Earley_F-bin-step-Earley_F-bin-mono by blast
    thus ?case
      using bin-def ‹l = k› by fastforce
  qed
qed

lemma Earley-base-sub-Earley_F-bin:
  assumes Init_F G ⊆ I
  assumes ∀ k' < k. bin (Earley G ω) k' ⊆ I
  assumes base ω (Earley G ω) k ⊆ I
  assumes wf-G G is-word G ω
  shows base ω (Earley G ω) (k+1) ⊆ bin (Earley_F-bin k G ω I) (k+1)
proof standard
  fix x
  assume ∗: x ∈ base ω (Earley G ω) (k+1)
  hence x ∈ Earley G ω
    using base-def by blast
  thus x ∈ bin (Earley_F-bin k G ω I) (k+1)
    using assms ∗
  proof (induction rule: Earley.induct)
    case (Init r)
    have k = 0
      using Init.prems(6) unfolding base-def by simp
    hence False
      using Init.prems(6) unfolding base-def by simp
    thus ?case
      by blast
  next
    case (Scan x r b i j a)
    have j = k
    using Scan.prems(6) base-def by (metis (mono-tags, lifting) CollectD add-right-cancel
item.sel(4))
    hence x ∈ bin (Earley_F-bin k G ω I) k
      using Earley_F-bin-sub-Earley_F-bin Scan.prems Scan.hyps(1,2) bin-def
      by (metis (mono-tags, lifting) CollectI item.sel(4) subsetD)
    hence Item r (b+1) i (j+1) ∈ Scan_F k ω (Earley_F-bin k G ω I)
      unfolding Scan_F-def inc-item-def using Scan.hyps ‹j = k› by force
    hence Item r (b+1) i (j+1) ∈ Earley_F-bin-step k G ω (Earley_F-bin k G ω I)
      using Scan_F-Earley_F-bin-step-mono by blast
    hence Item r (b+1) i (j+1) ∈ Earley_F-bin k G ω I
      using Earley_F-bin-idem Earley_F-bin-step-Earley_F-bin-mono by blast
    thus ?case
```

```
    using ‹j = k› bin-def by fastforce
  next
    case (Predict x r b i j r')
    have False
      using Predict.prems(6) unfolding base-def by (auto simp: prev-symbol-def)
    thus ?case
      by blast
  next
    case (Complete x r_x b_x i j y r_y b_y l)
    have l−1 < length ω
      using Complete.prems(6) base-def wf-Earley wf-item-def
     by (metis (mono-tags, lifting) CollectD add.right-neutral add-Suc-right add-diff-cancel-right'
item.sel(4) less-eq-Suc-le plus-1-eq-Suc)
    hence is-terminal G (ω!(l−1))
      using Complete.prems(5) is-word-is-terminal by blast
    moreover have is-nonterminal G (item-rule-head y)
      using Complete.hyps(3,4) Complete.prems(4) wf-Earley wf-item-def
     by (metis item-rule-head-def prod.collapse rule-head-def rule-nonterminal-type)
    moreover have prev-symbol (Item r_x (b_x+1) i l) = next-symbol x
      using Complete.hyps(1,6)
     by (auto simp: next-symbol-def prev-symbol-def is-complete-def item-rule-body-def
split: if-splits)
    moreover have prev-symbol (Item r_x (b_x+1) i l) = Some (ω!(l−1))
       using Complete.prems(6) base-def by (metis (mono-tags, lifting) CollectD
item.sel(4))
    ultimately have False
     using Complete.hyps(6) Complete.prems(4) is-terminal-nonterminal by fast-
force
    thus ?case
      by blast
  qed
qed

lemma Earley_F-bin-k-sub-Earley_F-bins:
  assumes wf-G G is-word G ω k ≤ n
  shows bin (Earley G ω) k ⊆ Earley_F-bins n G ω
  using assms
proof (induction n arbitrary: k)
  case 0
  have bin (Earley G ω) 0 ⊆ bin (Earley_F-bin 0 G ω (Init_F G)) 0
    using Earley_F-bin-sub-Earley_F-bin base-def by fastforce
  thus ?case
    unfolding bin-def using 0.prems(3) by auto
next
  case (Suc n)
  show ?case
  proof (cases k ≤ n)
    case True
    thus ?thesis
```

```
    using Suc Earley_F-bin-mono by force
  next
    case False
    hence k = n+1
      using Suc.prems(3) by force
    have 0: ∀ k' < k. bin (Earley G ω) k' ⊆ Earley_F-bins n G ω
      using Suc by simp
    moreover have base ω (Earley G ω) k ⊆ Earley_F-bins n G ω
    proof −
      have ∀ k' < k−1. bin (Earley G ω) k' ⊆ Earley_F-bins n G ω
        using Suc ‹k = n + 1› by auto
      moreover have base ω (Earley G ω) (k−1) ⊆ Earley_F-bins n G ω
        using 0 bin-def base-def False ‹k = n+1›
          by (smt (verit) Suc-eq-plus1 diff-Suc-1 linorder-not-less mem-Collect-eq
subsetD subsetI)
    ultimately have base ω (Earley G ω) k ⊆ bin (Earley_F-bin n G ω (Earley_F-bins
n G ω)) k
      using Suc.prems(1,2) Earley-base-sub-Earley_F-bin ‹k = n + 1› Init_F-sub-Earley_F-bins
by (metis add-diff-cancel-right')
      hence base ω (Earley G ω) k ⊆ bin (Earley_F-bins n G ω) k
        by (metis Earley_F-bins.elims Earley_F-bin-idem)
      thus ?thesis
        using bin-def by blast
    qed
    ultimately have bin (Earley G ω) k ⊆ bin (Earley_F-bin k G ω (Earley_F-bins
n G ω)) k
      using Earley_F-bin-sub-Earley_F-bin Init_F-sub-Earley_F-bins by metis
    thus ?thesis
      using Earley_F-bins.simps(2) ‹k = n + 1› bin-def by auto
  qed
qed

lemma Earley-sub-Earley_F:
  assumes wf-G G is-word G ω
  shows Earley G ω ⊆ Earley_F G ω
proof −
  have ∀ k ≤ length ω. bin (Earley G ω) k ⊆ Earley_F G ω
    by (simp add: Earley_F-bin-k-sub-Earley_F-bins Earley_F-def assms)
  thus ?thesis
    using wf-Earley wf-item-def bin-def by blast
qed

theorem completeness-Earley_F:
  assumes derives G [S G] ω is-word G ω wf-G G
  shows recognizing (Earley_F G ω) G ω
   using assms Earley-sub-Earley_F Earley_F-sub-Earley completeness-Earley by
(metis subset-antisym)
```

7.5 Correctness

theorem *Earley-eq-Earley*$_F$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *is-word* $\mathcal{G}$ ω
shows *Earley* $\mathcal{G}$ ω = *Earley*$_F$ $\mathcal{G}$ ω
using *Earley-sub-Earley*$_F$ *Earley*$_F$*-sub-Earley assms* **by** *blast*

theorem *correctness-Earley*$_F$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *is-word* $\mathcal{G}$ ω
shows *recognizing* (*Earley*$_F$ $\mathcal{G}$ ω) $\mathcal{G}$ ω $\longleftrightarrow$ *derives* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω
using *assms Earley-eq-Earley*$_F$ *correctness-Earley* **by** *fastforce*

end
theory *Earley-Recognizer*
imports
Earley-Fixpoint
begin

8 Earley recognizer

8.1 List auxilaries

fun *filter-with-index*$'$ :: *nat* $\Rightarrow$ ($'a$ $\Rightarrow$ *bool*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ ($'a$ $\times$ *nat*) *list* **where**
filter-with-index$'$ - - [] = []
| *filter-with-index*$'$ *i P* (*x*#*xs*) = (
if P x then (*x*,*i*) # *filter-with-index*$'$ (*i*+*1*) *P xs*
else filter-with-index$'$ (*i*+*1*) *P xs*)

definition *filter-with-index* :: ($'a$ $\Rightarrow$ *bool*) $\Rightarrow$ $'a$ *list* $\Rightarrow$ ($'a$ $\times$ *nat*) *list* **where**
filter-with-index P xs = *filter-with-index*$'$ *0 P xs*

lemma *filter-with-index*$'$*-P*:
(*x*, *n*) $\in$ *set* (*filter-with-index*$'$ *i P xs*) $\Longrightarrow$ *P x*
by (*induction xs arbitrary*: *i*) (*auto split*: *if-splits*)

lemma *filter-with-index-P*:
(*x*, *n*) $\in$ *set* (*filter-with-index P xs*) $\Longrightarrow$ *P x*
by (*metis filter-with-index*$'$*-P filter-with-index-def*)

lemma *filter-with-index*$'$*-cong-filter*:
map fst (*filter-with-index*$'$ *i P xs*) = *filter P xs*
by (*induction xs arbitrary*: *i*) *auto*

lemma *filter-with-index-cong-filter*:
map fst (*filter-with-index P xs*) = *filter P xs*
by (*simp add*: *filter-with-index*$'$*-cong-filter filter-with-index-def*)

lemma *size-index-filter-with-index*$'$:
(*x*, *n*) $\in$ *set* (*filter-with-index*$'$ *i P xs*) $\Longrightarrow$ *n* $\geq$ *i*

by (*induction xs arbitrary*: *i*) (*auto simp*: *Suc-leD split*: *if-splits*)

lemma *index-filter-with-index'-lt-length*:
(*x*, *n*) ∈ *set* (*filter-with-index' i P xs*) ⟹ *n*−*i* < *length xs*
by (*induction xs arbitrary*: *i*)(*auto simp*: *less-Suc-eq-0-disj split*: *if-splits*; *metis Suc-diff-Suc leI*)+

lemma *index-filter-with-index-lt-length*:
(*x*, *n*) ∈ *set* (*filter-with-index P xs*) ⟹ *n* < *length xs*
by (*metis filter-with-index-def index-filter-with-index'-lt-length minus-nat.diff-0*)

lemma *filter-with-index'-nth*:
(*x*, *n*) ∈ *set* (*filter-with-index' i P xs*) ⟹ *xs* ! (*n*−*i*) = *x*
proof (*induction xs arbitrary*: *i*)
case (*Cons y xs*)
show *?case*
proof (*cases x* = *y*)
case *True*
thus *?thesis*
using *Cons* **by** (*auto simp*: *nth-Cons' split*: *if-splits*)
next
case *False*
hence (*x*, *n*) ∈ *set* (*filter-with-index'* (*i*+*1*) *P xs*)
using *Cons.prems* **by** (*cases xs*) (*auto split*: *if-splits*)
hence *n* ≥ *i* + *1 xs* ! (*n* − *i* − *1*) = *x*
by (*auto simp*: *size-index-filter-with-index' Cons.IH*)
thus *?thesis*
by *simp*
qed
qed *simp*

lemma *filter-with-index-nth*:
(*x*, *n*) ∈ *set* (*filter-with-index P xs*) ⟹ *xs* ! *n* = *x*
by (*metis diff-zero filter-with-index'-nth filter-with-index-def*)

lemma *filter-with-index-nonempty*:
x ∈ *set xs* ⟹ *P x* ⟹ *filter-with-index P xs* ≠ []
by (*metis filter-empty-conv filter-with-index-cong-filter list.map*(*1*))

lemma *filter-with-index'-Ex-first*:
(∃ *x i xs'*. *filter-with-index' n P xs* = (*x*, *i*)#*xs'*) ⟷ (∃ *x* ∈ *set xs*. *P x*)
by (*induction xs arbitrary*: *n*) *auto*

lemma *filter-with-index-Ex-first*:
(∃ *x i xs'*. *filter-with-index P xs* = (*x*, *i*)#*xs'*) ⟷ (∃ *x* ∈ *set xs*. *P x*)
using *filter-with-index'-Ex-first filter-with-index-def* **by** *metis*

8.2 Definitions

datatype *pointer* =
  Null
  | *Pre nat* — pre
  | *PreRed nat* × *nat* × *nat* (*nat* × *nat* × *nat*) *list* — k', pre, red

datatype *'a entry* =
  Entry (*item* : *'a item*) (*pointer* : *pointer*)

type-synonym *'a bin* = *'a entry list*
type-synonym *'a bins* = *'a bin list*

definition *items* :: *'a bin* ⇒ *'a item list* **where**
  items b ≡ *map item b*

lemma *length-items-eq*:
  ‹*length* (*items b*) = *length b*›
  by (*simp add*: *items-def*)

definition *pointers* :: *'a bin* ⇒ *pointer list* **where**
  pointers b ≡ *map pointer b*

definition *bins-eq-items* :: *'a bins* ⇒ *'a bins* ⇒ *bool* **where**
  bins-eq-items bs0 bs1 ≡ *map items bs0* = *map items bs1*

definition *bins* :: *'a bins* ⇒ *'a item set* **where**
  bins bs ≡ ⋃ { *set* (*items* (*bs*!*k*)) | *k*. *k* < *length bs* }

definition *bin-upto* :: *'a bin* ⇒ *nat* ⇒ *'a item set* **where**
  bin-upto b i ≡ { *items b* ! *j* | *j*. *j* < *i* ∧ *j* < *length* (*items b*) }

definition *bins-upto* :: *'a bins* ⇒ *nat* ⇒ *nat* ⇒ *'a item set* **where**
  bins-upto bs k i ≡ ⋃ { *set* (*items* (*bs* ! *l*)) | *l*. *l* < *k* } ∪ *bin-upto* (*bs* ! *k*) *i*

definition *wf-bin-items* :: *'a cfg* ⇒ *'a sentence* ⇒ *nat* ⇒ *'a item list* ⇒ *bool* **where**
  wf-bin-items 𝒢 *ω k xs* ≡ ∀ *x* ∈ *set xs*. *wf-item* 𝒢 *ω x* ∧ *item-end x* = *k*

definition *wf-bin* :: *'a cfg* ⇒ *'a sentence* ⇒ *nat* ⇒ *'a bin* ⇒ *bool* **where**
  wf-bin 𝒢 *ω k b* ≡ *distinct* (*items b*) ∧ *wf-bin-items* 𝒢 *ω k* (*items b*)

definition *wf-bins* :: *'a cfg* ⇒ *'a list* ⇒ *'a bins* ⇒ *bool* **where**
  wf-bins 𝒢 *ω bs* ≡ ∀ *k* < *length bs*. *wf-bin* 𝒢 *ω k* (*bs*!*k*)

definition *nonempty-derives* :: *'a cfg* ⇒ *bool* **where**
  nonempty-derives 𝒢 ≡ ∀ *N*. *N* ∈ *set* (𝔑 𝒢) ⟶ ¬ *derives* 𝒢 [*N*] []

definition $Init_L$ :: *'a cfg* ⇒ *'a sentence* ⇒ *'a bins* **where**
  $Init_L$ 𝒢 *ω* ≡
    let rs = *filter* (λ*r*. *rule-head r* = 𝔖 𝒢) (𝔑 𝒢) *in*

```
    let b0 = map (λr. (Entry (init-item r 0) Null)) rs in
    let bs = replicate (length ω + 1) ([]) in
    bs[0 := b0]

definition Scan_L :: nat ⇒ 'a sentence ⇒ 'a  ⇒ 'a item ⇒ nat ⇒ 'a entry list
where
  Scan_L k ω a x pre ≡
    if ω!k = a then
      let x' = inc-item x (k+1) in
      [Entry x' (Pre pre)]
    else []

definition Predict_L :: nat ⇒ 'a cfg ⇒ 'a ⇒ 'a entry list where
  Predict_L k G X ≡
    let rs = filter (λr. rule-head r = X) (ℜ G) in
    map (λr. (Entry (init-item r k) Null)) rs

definition Complete_L :: nat ⇒ 'a item ⇒ 'a bins ⇒ nat ⇒ 'a entry list where
  Complete_L k y bs red ≡
    let orig = bs ! (item-origin y) in
    let is = filter-with-index (λx. next-symbol x = Some (item-rule-head y)) (items
orig) in
    map (λ(x, pre). (Entry (inc-item x k) (PreRed (item-origin y, pre, red) []))) is

fun bin-upd :: 'a entry ⇒ 'a bin ⇒ 'a bin where
  bin-upd e' [] = [e']
| bin-upd e' (e#es) = (
    case (e', e) of
      (Entry x (PreRed px xs), Entry y (PreRed py ys)) ⇒
        if x = y then Entry x (PreRed py (px#xs@ys)) # es
        else e # bin-upd e' es
      | - ⇒
        if item e' = item e then e # es
        else e # bin-upd e' es)

fun bin-upds :: 'a entry list ⇒ 'a bin ⇒ 'a bin where
  bin-upds [] b = b
| bin-upds (e#es) b = bin-upds es (bin-upd e b)

definition bins-upd :: 'a bins ⇒ nat ⇒ 'a entry list ⇒ 'a bins where
  bins-upd bs k es ≡ bs[k := bin-upds es (bs!k)]

partial-function (tailrec) Earley_L-bin' :: nat ⇒ 'a cfg ⇒ 'a sentence ⇒ 'a bins
⇒ nat ⇒ 'a bins where
  Earley_L-bin' k G ω bs i = (
    if i ≥ length (items (bs ! k)) then bs
    else
      let x = items (bs!k) ! i in
      let bs' =
```

```
      case next-symbol x of
        Some a ⇒
          if is-terminal 𝒢 a then
            if k < length ω then bins-upd bs (k+1) (Scan_L k ω a x i)
            else bs
          else bins-upd bs k (Predict_L k 𝒢 a)
      | None ⇒ bins-upd bs k (Complete_L k x bs i)
    in Earley_L-bin' k 𝒢 ω bs' (i+1))
```

declare *Earley$_L$-bin'.simps*[*code*]

definition *Earley$_L$-bin* :: *nat* ⇒ *'a cfg* ⇒ *'a sentence* ⇒ *'a bins* ⇒ *'a bins* **where**
Earley$_L$-bin k 𝒢 ω bs ≡ *Earley$_L$-bin' k 𝒢 ω bs 0*

fun *Earley$_L$-bins* :: *nat* ⇒ *'a cfg* ⇒ *'a sentence* ⇒ *'a bins* **where**
Earley$_L$-bins 0 𝒢 ω = *Earley$_L$-bin 0 𝒢 ω* (*Init$_L$ 𝒢 ω*)
| *Earley$_L$-bins* (*Suc n*) *𝒢 ω* = *Earley$_L$-bin* (*Suc n*) *𝒢 ω* (*Earley$_L$-bins n 𝒢 ω*)

definition *Earley$_L$* :: *'a cfg* ⇒ *'a sentence* ⇒ *'a bins* **where**
Earley$_L$ 𝒢 ω ≡ *Earley$_L$-bins* (*length ω*) *𝒢 ω*

8.3 Bin lemmas

lemma *length-bins-upd*[*simp*]:
length (*bins-upd bs k es*) = *length bs*
unfolding *bins-upd-def* **by** *simp*

lemma *length-bin-upd*:
length (*bin-upd e b*) ≥ *length b*
by (*induction e b rule*: *bin-upd.induct*) (*auto split*: *pointer.splits entry.splits*)

lemma *length-bin-upds*:
length (*bin-upds es b*) ≥ *length b*
by (*induction es arbitrary*: *b*) (*auto*, *meson le-trans length-bin-upd*)

lemma *length-nth-bin-bins-upd*:
length (*bins-upd bs k es* ! *n*) ≥ *length* (*bs* ! *n*)
unfolding *bins-upd-def* **using** *length-bin-upds*
by (*metis linorder-not-le list-update-beyond nth-list-update-eq nth-list-update-neq order-refl*)

lemma *nth-idem-bins-upd*:
k ≠ *n* ⟹ *bins-upd bs k es* ! *n* = *bs* ! *n*
unfolding *bins-upd-def* **by** *simp*

lemma *items-nth-idem-bin-upd*:
n < *length b* ⟹ *items* (*bin-upd e b*) ! *n* = *items b* ! *n*
by (*induction b arbitrary*: *e n*) (*auto simp*: *items-def less-Suc-eq-0-disj split*!: *entry.split pointer.split*)

lemma *items-nth-idem-bin-upds*:
n < *length b* ⟹ *items* (*bin-upds es b*) ! *n* = *items b* ! *n*
by (*induction es arbitrary*: *b*)
(*auto*, *metis items-def items-nth-idem-bin-upd length-bin-upd nth-map order.strict-trans2*)

lemma *items-nth-idem-bins-upd*:
n < *length* (*bs* ! *k*) ⟹ *items* (*bins-upd bs k es* ! *k*) ! *n* = *items* (*bs* ! *k*) ! *n*
unfolding *bins-upd-def* **using** *items-nth-idem-bin-upds*
by (*metis linorder-not-less list-update-beyond nth-list-update-eq*)

lemma *bin-upto-eq-set-items*:
i ≥ *length b* ⟹ *bin-upto b i* = *set* (*items b*)
by (*auto simp*: *bin-upto-def items-def*, *metis in-set-conv-nth nth-map order-le-less order-less-trans*)

lemma *bins-upto-empty*:
bins-upto bs 0 0 = {}
unfolding *bins-upto-def bin-upto-def* **by** *simp*

lemma *set-items-bin-upd*:
set (*items* (*bin-upd e b*)) = *set* (*items b*) ∪ {*item e*}
proof (*induction b arbitrary*: *e*)
case (*Cons b bs*)
show *?case*
proof (*cases* ∃ *x xp xs y yp ys*. *e* = *Entry x* (*PreRed xp xs*) ∧ *b* = *Entry y* (*PreRed yp ys*))
case *True*
then obtain *x xp xs y yp ys* **where** *e* = *Entry x* (*PreRed xp xs*) *b* = *Entry y* (*PreRed yp ys*)
by *blast*
thus *?thesis*
using *Cons.IH* **by** (*auto simp*: *items-def*)
next
case *False*
then show *?thesis*
proof *cases*
assume ∗: *item e* = *item b*
hence *bin-upd e* (*b* # *bs*) = *b* # *bs*
using *False* **by** (*auto split*: *pointer.splits entry.splits*)
thus *?thesis*
using ∗ **by** (*auto simp*: *items-def*)
next
assume ∗: ¬ *item e* = *item b*
hence *bin-upd e* (*b* # *bs*) = *b* # *bin-upd e bs*
using *False* **by** (*auto split*: *pointer.splits entry.splits*)
thus *?thesis*
using ∗ *Cons.IH* **by** (*auto simp*: *items-def*)
qed

qed
qed (*auto simp*: *items-def*)

lemma *set-items-bin-upds*:
set (*items* (*bin-upds es b*)) = *set* (*items b*) ∪ *set* (*items es*)
using *set-items-bin-upd* **by** (*induction es arbitrary*: *b*) (*auto simp*: *items-def*, *blast*, *force*+)

lemma *bins-bins-upd*:
assumes *k* < *length bs*
shows *bins* (*bins-upd bs k es*) = *bins bs* ∪ *set* (*items es*)
proof −
let *?bs* = *bins-upd bs k es*
have *bins* (*bins-upd bs k es*) = ⋃ {*set* (*items* (*?bs* ! *k*)) |*k*. *k* < *length ?bs*}
unfolding *bins-def* **by** *blast*
also have ... = ⋃ {*set* (*items* (*bs* ! *l*)) |*l*. *l* < *length bs* ∧ *l* ≠ *k*} ∪ *set* (*items* (*?bs* ! *k*))
unfolding *bins-upd-def* **using** *assms* **by** (*auto*, *metis nth-list-update*)
also have ... = ⋃ {*set* (*items* (*bs* ! *l*)) |*l*. *l* < *length bs* ∧ *l* ≠ *k*} ∪ *set* (*items* (*bs* ! *k*)) ∪ *set* (*items es*)
using *set-items-bin-upds*[*of es bs*!*k*] **by** (*simp add*: *assms bins-upd-def sup-assoc*)
also have ... = ⋃ {*set* (*items* (*bs* ! *k*)) |*k*. *k* < *length bs*} ∪ *set* (*items es*)
using *assms* **by** *blast*
also have ... = *bins bs* ∪ *set* (*items es*)
unfolding *bins-def* **by** *blast*
finally show *?thesis* **.**
qed

lemma *kth-bin-sub-bins*:
k < *length bs* ⟹ *set* (*items* (*bs* ! *k*)) ⊆ *bins bs*
unfolding *bins-def bins-upto-def bin-upto-def* **by** *blast*+

lemma *bin-upto-Cons-0*:
bin-upto (*e*#*es*) *0* = {}
by (*auto simp*: *bin-upto-def*)

lemma *bin-upto-Cons*:
assumes *0* < *n*
shows *bin-upto* (*e*#*es*) *n* = { *item e* } ∪ *bin-upto es* (*n*−*1*)
proof −
have *bin-upto* (*e*#*es*) *n* = { *items* (*e*#*es*) ! *j* | *j*. *j* < *n* ∧ *j* < *length* (*items* (*e*#*es*)) }
unfolding *bin-upto-def* **by** *blast*
also have ... = { *item e* } ∪ { *items es* ! *j* | *j*. *j* < (*n*−*1*) ∧ *j* < *length* (*items es*) }
using *assms* **by** (*cases n*) (*auto simp*: *items-def nth-Cons'*, *metis One-nat-def Zero-not-Suc diff-Suc-1 not-less-eq nth-map*)
also have ... = { *item e* } ∪ *bin-upto es* (*n*−*1*)
unfolding *bin-upto-def* **by** *blast*

```
  finally show ?thesis .
qed

lemma bin-upto-nth-idem-bin-upd:
  n < length b ⟹ bin-upto (bin-upd e b) n = bin-upto b n
proof (induction b arbitrary: e n)
  case (Cons b bs)
  show ?case
  proof (cases ∃ x xp xs y yp ys. e = Entry x (PreRed xp xs) ∧ b = Entry y (PreRed
yp ys))
    case True
    then obtain x xp xs y yp ys where e = Entry x (PreRed xp xs) b = Entry y
(PreRed yp ys)
      by blast
    thus ?thesis
      using Cons bin-upto-Cons-0
      by (cases n) (auto simp: items-def bin-upto-Cons, blast+)
  next
    case False
    then show ?thesis
    proof cases
      assume ∗: item e = item b
      hence bin-upd e (b # bs) = b # bs
        using False by (auto split: pointer.splits entry.splits)
      thus ?thesis
        using ∗ by (auto simp: items-def)
    next
      assume ∗: ¬ item e = item b
      hence bin-upd e (b # bs) = b # bin-upd e bs
        using False by (auto split: pointer.splits entry.splits)
      thus ?thesis
        using ∗ Cons bin-upto-Cons-0
        by (cases n) (auto simp: items-def bin-upto-Cons, blast+)
    qed
  qed
qed (auto simp: items-def)

lemma bin-upto-nth-idem-bin-upds:
  n < length b ⟹ bin-upto (bin-upds es b) n = bin-upto b n
  using bin-upto-nth-idem-bin-upd length-bin-upd
  apply (induction es arbitrary: b)
   apply auto
  using order.strict-trans2 order.strict-trans1 by blast+

lemma bins-upto-kth-nth-idem:
  assumes l < length bs k ≤ l n < length (bs ! k)
  shows bins-upto (bins-upd bs l es) k n = bins-upto bs k n
proof −
  let ?bs = bins-upd bs l es
```

have *bins-upto ?bs k n* = $\bigcup$ {*set* (*items* (*?bs* ! *l*)) |*l*. *l* < *k*} ∪ *bin-upto* (*?bs* ! *k*) *n*
unfolding *bins-upto-def* **by** *blast*
also have ... = $\bigcup$ {*set* (*items* (*bs* ! *l*)) |*l*. *l* < *k*} ∪ *bin-upto* (*?bs* ! *k*) *n*
unfolding *bins-upd-def* **using** *assms*(*1*,*2*) **by** *auto*
also have ... = $\bigcup$ {*set* (*items* (*bs* ! *l*)) |*l*. *l* < *k*} ∪ *bin-upto* (*bs* ! *k*) *n*
unfolding *bins-upd-def* **using** *assms*(*1*,*3*) *bin-upto-nth-idem-bin-upds*
by (*metis* (*no-types*, *lifting*) *nth-list-update*)
also have ... = *bins-upto bs k n*
unfolding *bins-upto-def* **by** *blast*
finally show *?thesis* **.**
qed

lemma *bins-upto-sub-bins*:
k < *length bs* ⟹ *bins-upto bs k n* ⊆ *bins bs*
unfolding *bins-def bins-upto-def bin-upto-def* **using** *less-trans* **by** (*auto*, *blast*)

lemma *bins-upto-Suc-Un*:
n < *length* (*bs* ! *k*) ⟹ *bins-upto bs k* (*n*+*1*) = *bins-upto bs k n* ∪ { *items* (*bs* ! *k*) ! *n* }
unfolding *bins-upto-def bin-upto-def* **using** *less-Suc-eq* **by** (*auto simp*: *items-def*, *metis nth-map*)

lemma *bins-bin-exists*:
x ∈ *bins bs* ⟹ ∃ *k* < *length bs*. *x* ∈ *set* (*items* (*bs* ! *k*))
unfolding *bins-def* **by** *blast*

lemma *distinct-bin-upd*:
distinct (*items b*) ⟹ *distinct* (*items* (*bin-upd e b*))
proof (*induction b arbitrary*: *e*)
case (*Cons b bs*)
show *?case*
proof (*cases* ∃ *x xp xs y yp ys*. *e* = *Entry x* (*PreRed xp xs*) ∧ *b* = *Entry y* (*PreRed yp ys*))
case *True*
then obtain *x xp xs y yp ys* **where** *e* = *Entry x* (*PreRed xp xs*) *b* = *Entry y* (*PreRed yp ys*)
by *blast*
thus *?thesis*
using *Cons*
apply (*auto simp*: *items-def*)
by (*metis Un-insert-right entry.sel*(*1*) *imageI items-def list.set-map list.simps*(*15*) *set-ConsD set-items-bin-upd sup-bot-right*)
next
case *False*
then show *?thesis*
proof *cases*
assume ∗: *item e* = *item b*
hence *bin-upd e* (*b* # *bs*) = *b* # *bs*

```
      using False by (auto split: pointer.splits entry.splits)
    thus ?thesis
      using ∗ Cons.prems by (auto simp: items-def)
  next
    assume ∗: ¬ item e = item b
    hence bin-upd e (b # bs) = b # bin-upd e bs
      using False by (auto split: pointer.splits entry.splits)
    moreover have distinct (items (bin-upd e bs))
      using Cons by (auto simp: items-def)
    ultimately show ?thesis
      using ∗ Cons.prems set-items-bin-upd
      by (metis Un-insert-right distinct.simps(2) insertE items-def list.simps(9)
sup-bot-right)
  qed
 qed
qed (auto simp: items-def)

lemma wf-bins-kth-bin:
 wf-bins G ω bs ⟹ k < length bs ⟹ x ∈ set (items (bs ! k)) ⟹ wf-item G ω x
∧ item-end x = k
 using wf-bin-def wf-bins-def wf-bin-items-def by blast

lemma wf-bin-bin-upd:
 assumes wf-bin G ω k b wf-item G ω (item e) ∧ item-end (item e) = k
 shows wf-bin G ω k (bin-upd e b)
 using assms
proof (induction b arbitrary: e)
 case (Cons b bs)
 show ?case
 proof (cases ∃ x xp xs y yp ys. e = Entry x (PreRed xp xs) ∧ b = Entry y (PreRed
yp ys))
   case True
   then obtain x xp xs y yp ys where e = Entry x (PreRed xp xs) b = Entry y
(PreRed yp ys)
     by blast
   thus ?thesis
     using Cons distinct-bin-upd wf-bin-def wf-bin-items-def set-items-bin-upd
     by (smt (verit, best) Un-insert-right insertE sup-bot.right-neutral)
 next
   case False
   then show ?thesis
   proof cases
     assume ∗: item e = item b
     hence bin-upd e (b # bs) = b # bs
       using False by (auto split: pointer.splits entry.splits)
     thus ?thesis
       using ∗ Cons.prems by (auto simp: items-def)
   next
     assume ∗: ¬ item e = item b
```

```
    hence bin-upd e (b # bs) = b # bin-upd e bs
      using False by (auto split: pointer.splits entry.splits)
    thus ?thesis
    using ∗ Cons.prems set-items-bin-upd distinct-bin-upd wf-bin-def wf-bin-items-def
      by (smt (verit, best) Un-insert-right insertE sup-bot-right)
  qed
 qed
qed (auto simp: items-def wf-bin-def wf-bin-items-def)

lemma wf-bin-bin-upds:
 assumes wf-bin G ω k b distinct (items es)
 assumes ∀ x ∈ set (items es). wf-item G ω x ∧ item-end x = k
 shows wf-bin G ω k (bin-upds es b)
 using assms by (induction es arbitrary: b) (auto simp: wf-bin-bin-upd items-def)

lemma wf-bins-bins-upd:
 assumes wf-bins G ω bs distinct (items es)
 assumes ∀ x ∈ set (items es). wf-item G ω x ∧ item-end x = k
 shows wf-bins G ω (bins-upd bs k es)
 unfolding bins-upd-def using assms wf-bin-bin-upds wf-bins-def
 by (metis length-list-update nth-list-update-eq nth-list-update-neq)

lemma wf-bins-impl-wf-items:
 wf-bins G ω bs ⟹ ∀ x ∈ (bins bs). wf-item G ω x
 unfolding wf-bins-def wf-bin-def wf-bin-items-def bins-def by auto

lemma bin-upds-eq-items:
 set (items es) ⊆ set (items b) ⟹ set (items (bin-upds es b)) = set (items b)
 apply (induction es arbitrary: b)
  apply (auto simp: set-items-bin-upd set-items-bin-upds)
  apply (simp add: items-def)
 by (metis Un-iff Un-subset-iff items-def list.simps(9) set-subset-Cons)

lemma bin-eq-items-bin-upd:
 item e ∈ set (items b) ⟹ items (bin-upd e b) = items b
proof (induction b arbitrary: e)
 case (Cons b bs)
 show ?case
 proof (cases ∃ x xp xs y yp ys. e = Entry x (PreRed xp xs) ∧ b = Entry y (PreRed
yp ys))
   case True
   then obtain x xp xs y yp ys where e = Entry x (PreRed xp xs) b = Entry y
(PreRed yp ys)
     by blast
   thus ?thesis
     using Cons by (auto simp: items-def)
 next
   case False
   then show ?thesis
```

```
    proof cases
      assume *: item e = item b
      hence bin-upd e (b # bs) = b # bs
        using False by (auto split: pointer.splits entry.splits)
      thus ?thesis
        using * Cons.prems by (auto simp: items-def)
    next
      assume *: ¬ item e = item b
      hence bin-upd e (b # bs) = b # bin-upd e bs
        using False by (auto split: pointer.splits entry.splits)
      thus ?thesis
        using * Cons by (auto simp: items-def)
    qed
  qed
qed (auto simp: items-def)

lemma bin-eq-items-bin-upds:
  assumes set (items es) ⊆ set (items b)
  shows items (bin-upds es b) = items b
  using assms
proof (induction es arbitrary: b)
  case (Cons e es)
  have items (bin-upds es (bin-upd e b)) = items (bin-upd e b)
    using Cons bin-upds-eq-items set-items-bin-upd set-items-bin-upds
    by (metis Un-upper2 bin-upds.simps(2) sup.coboundedI1)
  moreover have items (bin-upd e b) = items b
    using bin-eq-items-bin-upd Cons.prems by (auto simp: items-def)
  ultimately show ?case
    by simp
qed (auto simp: items-def)

lemma bins-eq-items-bins-upd:
  assumes set (items es) ⊆ set (items (bs!k))
  shows bins-eq-items (bins-upd bs k es) bs
  unfolding bins-upd-def using assms bin-eq-items-bin-upds bins-eq-items-def
  by (metis list-update-id map-update)

lemma bins-eq-items-imp-eq-bins:
  bins-eq-items bs bs' ⟹ bins bs = bins bs'
  unfolding bins-eq-items-def bins-def items-def
  by (metis (no-types, lifting) length-map nth-map)

lemma bin-eq-items-dist-bin-upd-bin:
  assumes items a = items b
  shows items (bin-upd e a) = items (bin-upd e b)
  using assms
proof (induction a arbitrary: e b)
  case (Cons a as)
  obtain b' bs where bs: b = b' # bs item a = item b' items as = items bs
```

```
    using Cons.prems by (auto simp: items-def)
  show ?case
  proof (cases ∃ x xp xs y yp ys. e = Entry x (PreRed xp xs) ∧ a = Entry y (PreRed
yp ys))
    case True
    then obtain x xp xs y yp ys where #: e = Entry x (PreRed xp xs) a = Entry
y (PreRed yp ys)
      by blast
    show ?thesis
    proof cases
      assume ∗: x = y
      hence items (bin-upd e (a # as)) = x # items as
        using # by (auto simp: items-def)
      moreover have items (bin-upd e (b′ # bs)) = x # items bs
        using bs # ∗ by (auto simp: items-def split: pointer.splits entry.splits)
      ultimately show ?thesis
        using bs by simp
    next
      assume ∗: ¬ x = y
      hence items (bin-upd e (a # as)) = y # items (bin-upd e as)
        using # by (auto simp: items-def)
      moreover have items (bin-upd e (b′ # bs)) = y # items (bin-upd e bs)
        using bs # ∗ by (auto simp: items-def split: pointer.splits entry.splits)
      ultimately show ?thesis
        using bs Cons.IH by simp
    qed
  next
    case False
    then show ?thesis
    proof cases
      assume ∗: item e = item a
      hence items (bin-upd e (a # as)) = item a # items as
        using False by (auto simp: items-def split: pointer.splits entry.splits)
      moreover have items (bin-upd e (b′ # bs)) = item b′ # items bs
        using bs False ∗ by (auto simp: items-def split: pointer.splits entry.splits)
      ultimately show ?thesis
        using bs by simp
    next
      assume ∗: ¬ item e = item a
      hence items (bin-upd e (a # as)) = item a # items (bin-upd e as)
        using False by (auto simp: items-def split: pointer.splits entry.splits)
     moreover have items (bin-upd e (b′ # bs)) = item b′ # items (bin-upd e bs)
        using bs False ∗ by (auto simp: items-def split: pointer.splits entry.splits)
      ultimately show ?thesis
        using bs Cons by simp
    qed
  qed
qed (auto simp: items-def)
```

lemma *bin-eq-items-dist-bin-upds-bin*:
assumes *items a* = *items b*
shows *items* (*bin-upds es a*) = *items* (*bin-upds es b*)
using *assms*
proof (*induction es arbitrary*: *a b*)
case (*Cons e es*)
hence *items* (*bin-upds es* (*bin-upd e a*)) = *items* (*bin-upds es* (*bin-upd e b*))
using *bin-eq-items-dist-bin-upd-bin* **by** *blast*
thus *?case*
by *simp*
qed *simp*

lemma *bin-eq-items-dist-bin-upd-entry*:
assumes *item e* = *item e′*
shows *items* (*bin-upd e b*) = *items* (*bin-upd e′ b*)
using *assms*
proof (*induction b arbitrary*: *e e′*)
case (*Cons a as*)
show *?case*
proof (*cases* ∃ *x xp xs y yp ys*. *e* = *Entry x* (*PreRed xp xs*) ∧ *a* = *Entry y* (*PreRed yp ys*))
case *True*
then obtain *x xp xs y yp ys* **where** #: *e* = *Entry x* (*PreRed xp xs*) *a* = *Entry y* (*PreRed yp ys*)
by *blast*
show *?thesis*
proof *cases*
assume ∗: *x* = *y*
thus *?thesis*
using # *Cons.prems* **by** (*auto simp*: *items-def split*: *pointer.splits entry.splits*)
next
assume ∗: ¬ *x* = *y*
thus *?thesis*
using # *Cons.prems*
by (*auto simp*: *items-def split*!: *pointer.splits entry.splits*, *metis Cons.IH Cons.prems items-def*)+
qed
next
case *False*
then show *?thesis*
proof *cases*
assume ∗: *item e* = *item a*
thus *?thesis*
using *Cons.prems* **by** (*auto simp*: *items-def split*: *pointer.splits entry.splits*)
next
assume ∗: ¬ *item e* = *item a*
thus *?thesis*
using *Cons.prems*
by (*auto simp*: *items-def split*!: *pointer.splits entry.splits*, *metis Cons.IH*

Cons.prems items-def)+
  qed
 qed
qed (*auto simp*: *items-def*)

lemma *bin-eq-items-dist-bin-upds-entries*:
 assumes *items es* = *items es'*
 shows *items* (*bin-upds es b*) = *items* (*bin-upds es' b*)
 using *assms*
proof (*induction es arbitrary*: *es' b*)
 case (*Cons e es*)
 then obtain *e' es''* **where** *item e* = *item e' items es* = *items es'' es'* = *e'* # *es''*
  by (*auto simp*: *items-def*)
 hence *items* (*bin-upds es* (*bin-upd e b*)) = *items* (*bin-upds es''* (*bin-upd e' b*))
  using *Cons.IH*
  by (*metis bin-eq-items-dist-bin-upd-entry bin-eq-items-dist-bin-upds-bin*)
 thus *?case*
  by (*simp add*: ‹*es'* = *e'* # *es''*›)
qed (*auto simp*: *items-def*)

lemma *bins-eq-items-dist-bins-upd*:
 assumes *bins-eq-items as bs items aes* = *items bes k* < *length as*
 shows *bins-eq-items* (*bins-upd as k aes*) (*bins-upd bs k bes*)
proof −
 have *k* < *length bs*
  using *assms*(*1*,*3*) *bins-eq-items-def map-eq-imp-length-eq* **by** *metis*
 hence *items* (*bin-upds* (*as*!*k*) *aes*) = *items* (*bin-upds* (*bs*!*k*) *bes*)
  using *bin-eq-items-dist-bin-upds-entries bin-eq-items-dist-bin-upds-bin bins-eq-items-def assms*
  by (*metis* (*no-types*, *lifting*) *nth-map*)
 thus *?thesis*
  using ‹*k* < *length bs*› *assms bin-eq-items-dist-bin-upds-bin bin-eq-items-dist-bin-upds-entries bins-eq-items-def bins-upd-def* **by** (*smt* (*verit*) *map-update nth-map*)
qed

8.4 Well-formed bins

lemma *distinct-Scan*$_L$:
 distinct (*items* (*Scan*$_L$ *k ω a x pre*))
 unfolding *Scan*$_L$*-def* **by** (*auto simp*: *items-def*)

lemma *distinct-Predict*$_L$:
 wf-𝒢 𝒢 ⟹ *distinct* (*items* (*Predict*$_L$ *k 𝒢 X*))
 unfolding *Predict*$_L$*-def wf-𝒢-defs* **by** (*auto simp*: *init-item-def rule-head-def distinct-map inj-on-def items-def*)

lemma *inj-on-inc-item*:
 ∀ *x* ∈ *A*. *item-end x* = *l* ⟹ *inj-on* (λ*x*. *inc-item x k*) *A*

unfolding *inj-on-def inc-item-def* **by** (*simp add*: *item.expand*)

lemma *distinct-Complete$_L$*:
assumes *wf-bins* $\mathcal{G}$ ω *bs item-origin y* < *length bs*
shows *distinct* (*items* (*Complete$_L$ k y bs red*))
proof −
let *?orig* = *bs* ! (*item-origin y*)
let *?is* = *filter-with-index* (λ*x*. *next-symbol x* = *Some* (*item-rule-head y*)) (*items ?orig*)
let *?is′* = *map* (λ(*x*, *pre*). (*Entry* (*inc-item x k*) (*PreRed* (*item-origin y*, *pre*, *red*) [])))) *?is*
have *wf*: *wf-bin* $\mathcal{G}$ ω (*item-origin y*) *?orig*
using *assms wf-bins-def* **by** *blast*
have *0*: ∀ *x* ∈ *set* (*map fst ?is*). *item-end x* = (*item-origin y*)
using *wf wf-bin-def wf-bin-items-def filter-is-subset filter-with-index-cong-filter* **by** (*metis in-mono*)
hence *distinct* (*items ?orig*)
using *wf* **unfolding** *wf-bin-def* **by** *blast*
hence *distinct* (*map fst ?is*)
using *filter-with-index-cong-filter distinct-filter* **by** *metis*
moreover have *items ?is′* = *map* (λ*x*. *inc-item x k*) (*map fst ?is*)
by (*induction ?is*) (*auto simp*: *items-def*)
moreover have *inj-on* (λ*x*. *inc-item x k*) (*set* (*map fst ?is*))
using *inj-on-inc-item 0* **by** *blast*
ultimately have *distinct* (*items ?is′*)
using *distinct-map* **by** *metis*
thus *?thesis*
unfolding *Complete$_L$-def* **by** *simp*
qed

lemma *wf-bins-Scan$_L$′*:
assumes *wf-bins* $\mathcal{G}$ ω *bs k* < *length bs x* ∈ *set* (*items* (*bs* ! *k*))
assumes *k* < *length* ω *next-symbol x* ≠ *None y* = *inc-item x* (*k*+*1*)
shows *wf-item* $\mathcal{G}$ ω *y* ∧ *item-end y* = *k*+*1*
using *assms wf-bins-kth-bin*[*OF assms*(*1*−*3*)]
unfolding *wf-item-def inc-item-def next-symbol-def is-complete-def item-rule-body-def*
by (*auto split*: *if-splits*)

lemma *wf-bins-Scan$_L$*:
assumes *wf-bins* $\mathcal{G}$ ω *bs k* < *length bs x* ∈ *set* (*items* (*bs* ! *k*)) *k* < *length* ω *next-symbol x* ≠ *None*
shows ∀ *y* ∈ *set* (*items* (*Scan$_L$ k* ω *a x pre*)). *wf-item* $\mathcal{G}$ ω *y* ∧ *item-end y* = (*k*+*1*)
using *wf-bins-Scan$_L$*′[*OF assms*] **by** (*simp add*: *Scan$_L$-def items-def*)

lemma *wf-bins-Predict$_L$*:
assumes *wf-bins* $\mathcal{G}$ ω *bs k* < *length bs k* ≤ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows ∀ *y* ∈ *set* (*items* (*Predict$_L$ k* $\mathcal{G}$ *X*)). *wf-item* $\mathcal{G}$ ω *y* ∧ *item-end y* = *k*
using *assms* **by** (*auto simp*: *Predict$_L$-def wf-item-def wf-bins-def wf-bin-def init-item-def*

wf-$\mathcal{G}$-defs items-def)

lemma *wf-item-inc-item*:
assumes *wf-item* $\mathcal{G}$ ω *x next-symbol x* = *Some a item-origin x* $\leq$ *k k* $\leq$ *length* ω
shows *wf-item* $\mathcal{G}$ ω (*inc-item x k*) $\wedge$ *item-end* (*inc-item x k*) = *k*
using *assms* **by** (*auto simp*: *wf-item-def inc-item-def item-rule-body-def next-symbol-def is-complete-def split*: *if-splits*)

lemma *wf-bins-Complete*$_L$:
assumes *wf-bins* $\mathcal{G}$ ω *bs k* < *length bs y* $\in$ *set* (*items* (*bs* ! *k*))
shows $\forall$ *x* $\in$ *set* (*items* (*Complete*$_L$ *k y bs red*)). *wf-item* $\mathcal{G}$ ω *x* $\wedge$ *item-end x* = *k*
proof −
let *?orig* = *bs* ! (*item-origin y*)
let *?is* = *filter-with-index* (λ*x*. *next-symbol x* = *Some* (*item-rule-head y*)) (*items ?orig*)
let *?is*′ = *map* (λ(*x*, *pre*). (*Entry* (*inc-item x k*) (*PreRed* (*item-origin y*, *pre*, *red*) []))) *?is*
{
fix *x*
assume ∗: *x* $\in$ *set* (*map fst ?is*)
have *item-end x* = *item-origin y*
using ∗ *assms wf-bins-kth-bin wf-item-def filter-with-index-cong-filter*
by (*metis dual-order.strict-trans2 filter-is-subset subsetD*)
have *wf-item* $\mathcal{G}$ ω *x*
using ∗ *assms wf-bins-kth-bin wf-item-def filter-with-index-cong-filter*
by (*metis dual-order.strict-trans2 filter-is-subset subsetD*)
moreover have *next-symbol x* = *Some* (*item-rule-head y*)
using ∗ *filter-set filter-with-index-cong-filter member-filter* **by** *metis*
moreover have *item-origin x* $\leq$ *k*
using ‹*item-end x* = *item-origin y*› ‹*wf-item* $\mathcal{G}$ ω *x*› *assms wf-bins-kth-bin wf-item-def*
by (*metis dual-order.order-iff-strict dual-order.strict-trans1*)
moreover have *k* $\leq$ *length* ω
using *assms wf-bins-kth-bin wf-item-def* **by** *blast*
ultimately have *wf-item* $\mathcal{G}$ ω (*inc-item x k*) *item-end* (*inc-item x k*) = *k*
by (*simp-all add*: *wf-item-inc-item*)
}
hence $\forall$ *x* $\in$ *set* (*items ?is*′). *wf-item* $\mathcal{G}$ ω *x* $\wedge$ *item-end x* = *k*
by (*auto simp*: *items-def rev-image-eqI*)
thus *?thesis*
unfolding *Complete*$_L$*-def* **by** *presburger*
qed

lemma *Ex-wf-bins*:
$\exists$ *n bs* ω $\mathcal{G}$. *n* $\leq$ *length* ω $\wedge$ *length bs* = *Suc* (*length* ω) $\wedge$ *wf-*$\mathcal{G}$ $\mathcal{G}$ $\wedge$ *wf-bins* $\mathcal{G}$ ω *bs*
apply (*rule exI*[**where** *x*=*0*])
apply (*rule exI*[**where** *x*=[[]]])

apply (*rule exI*[**where** *x*=[]])
apply (*auto simp*: *wf-bins-def wf-bin-def wf-*$\mathcal{G}$*-defs wf-bin-items-def items-def split*: *prod.splits*)
by (*metis cfg.sel distinct.simps*(*1*) *empty-iff empty-set inf-bot-right list.set-intros*(*1*))

definition *wf-earley-input* :: (*nat* × *'a cfg* × *'a sentence* × *'a bins*) *set* **where**
wf-earley-input = {
(*k*, $\mathcal{G}$, ω, *bs*) | *k* $\mathcal{G}$ ω *bs*.
k ≤ *length* ω ∧
length bs = *length* ω + *1* ∧
wf-$\mathcal{G}$ $\mathcal{G}$ ∧
wf-bins $\mathcal{G}$ ω *bs*
}

typedef *'a wf-bins* = *wf-earley-input*::(*nat* × *'a cfg* × *'a sentence* × *'a bins*) *set*
morphisms *from-wf-bins to-wf-bins*
using *Ex-wf-bins* **by** (*auto simp*: *wf-earley-input-def*)

lemma *wf-earley-input-elim*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
shows *k* ≤ *length* ω ∧ *k* < *length bs* ∧ *length bs* = *length* ω + *1* ∧ *wf-*$\mathcal{G}$ $\mathcal{G}$ ∧ *wf-bins* $\mathcal{G}$ ω *bs*
using *assms*(*1*) *from-wf-bins wf-earley-input-def* **by** (*smt* (*verit*) *Suc-eq-plus1 less-Suc-eq-le mem-Collect-eq prod.sel*(*1*) *snd-conv*)

lemma *wf-earley-input-intro*:
assumes *k* ≤ *length* ω *length bs* = *length* ω + *1 wf-*$\mathcal{G}$ $\mathcal{G}$ *wf-bins* $\mathcal{G}$ ω *bs*
shows (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
by (*simp add*: *assms wf-earley-input-def*)

lemma *wf-earley-input-Complete*$_L$:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input* ¬ *length* (*items* (*bs* ! *k*)) ≤ *i*
assumes *x* = *items* (*bs* ! *k*) ! *i next-symbol x* = *None*
shows (*k*, $\mathcal{G}$, ω, *bins-upd bs k* (*Complete*$_L$ *k x bs red*)) ∈ *wf-earley-input*
proof −
have ∗: *k* ≤ *length* ω *length bs* = *length* ω + *1 wf-*$\mathcal{G}$ $\mathcal{G}$ *wf-bins* $\mathcal{G}$ ω *bs*
using *wf-earley-input-elim assms*(*1*) **by** *metis*+
have *x*: *x* ∈ *set* (*items* (*bs* ! *k*))
using *assms*(*2*,*3*) **by** *simp*
have *item-origin x* < *length bs*
using *x wf-bins-kth-bin* ∗(*1*,*2*,*4*) *wf-item-def*
by (*metis One-nat-def add.right-neutral add-Suc-right dual-order.trans le-imp-less-Suc*)
hence *wf-bins* $\mathcal{G}$ ω (*bins-upd bs k* (*Complete*$_L$ *k x bs red*))
using ∗(*1*,*2*,*4*) *Suc-eq-plus1 distinct-Complete*$_L$ *le-imp-less-Suc wf-bins-Complete*$_L$ *wf-bins-bins-upd x* **by** *metis*
thus *?thesis*
by (*simp add*: ∗(*1*−*3*) *wf-earley-input-def*)
qed

lemma *wf-earley-input-Scan$_L$*:
assumes $(k, \mathcal{G}, \omega, bs) \in$ *wf-earley-input* $\neg$ *length* (*items* (*bs* ! *k*)) $\leq i$
assumes $x =$ *items* (*bs* ! *k*) ! *i next-symbol x* = *Some a*
assumes *is-terminal* $\mathcal{G}$ *a k* < *length* ω
shows $(k, \mathcal{G}, \omega,$ *bins-upd bs* (*k*+*1*) (*Scan$_L$ k* ω *a x pre*)) $\in$ *wf-earley-input*
proof −
have ∗: *k* ≤ *length* ω *length bs* = *length* ω + *1 wf-*$\mathcal{G}$ $\mathcal{G}$ *wf-bins* $\mathcal{G}$ ω *bs*
using *wf-earley-input-elim assms*(*1*) **by** *metis*+
have *x*: *x* ∈ *set* (*items*(*bs* ! *k*))
using *assms*(*2*,*3*) **by** *simp*
have *wf-bins* $\mathcal{G}$ ω (*bins-upd bs* (*k*+*1*) (*Scan$_L$ k* ω *a x pre*))
using ∗ *x assms*(*1*,*4*,*6*) *distinct-Scan$_L$ wf-bins-Scan$_L$ wf-bins-bins-upd wf-earley-input-elim*
by (*metis option.discI*)
thus *?thesis*
by (*simp add*: ∗(*1*−*3*) *wf-earley-input-def*)
qed

lemma *wf-earley-input-Predict$_L$*:
assumes $(k, \mathcal{G}, \omega, bs) \in$ *wf-earley-input* $\neg$ *length* (*items* (*bs* ! *k*)) ≤ *i*
assumes *x* = *items* (*bs* ! *k*) ! *i next-symbol x* = *Some a* $\neg$ *is-terminal* $\mathcal{G}$ *a*
shows $(k, \mathcal{G}, \omega,$ *bins-upd bs k* (*Predict$_L$ k* $\mathcal{G}$ *a*)) $\in$ *wf-earley-input*
proof −
have ∗: *k* ≤ *length* ω *length bs* = *length* ω + *1 wf-*$\mathcal{G}$ $\mathcal{G}$ *wf-bins* $\mathcal{G}$ ω *bs*
using *wf-earley-input-elim assms*(*1*) **by** *metis*+
have *x*: *x* ∈ *set* (*items* (*bs* ! *k*))
using *assms*(*2*,*3*) **by** *simp*
hence *wf-bins* $\mathcal{G}$ ω (*bins-upd bs k* (*Predict$_L$ k* $\mathcal{G}$ *a*))
using ∗ *x assms*(*1*,*4*) *distinct-Predict$_L$ wf-bins-Predict$_L$ wf-bins-bins-upd wf-earley-input-elim*
by *metis*
thus *?thesis*
by (*simp add*: ∗(*1*−*3*) *wf-earley-input-def*)
qed

fun *earley-measure* :: *nat* × *'a cfg* × *'a sentence* × *'a bins* ⇒ *nat* ⇒ *nat* **where**
earley-measure (*k*, $\mathcal{G}$, ω, *bs*) *i* = *card* { *x* | *x*. *wf-item* $\mathcal{G}$ ω *x* ∧ *item-end x* = *k* }
− *i*

lemma *Earley$_L$-bin'-simps*[*simp*]:
i ≥ *length* (*items* (*bs* ! *k*)) ⟹ *Earley$_L$-bin' k* $\mathcal{G}$ ω *bs i* = *bs*
¬ *i* ≥ *length* (*items* (*bs* ! *k*)) ⟹ *x* = *items* (*bs*!*k*) ! *i* ⟹ *next-symbol x* = *None*
⟹
Earley$_L$-bin' k $\mathcal{G}$ ω *bs i* = *Earley$_L$-bin' k* $\mathcal{G}$ ω (*bins-upd bs k* (*Complete$_L$ k x bs*
i)) (*i*+*1*)
¬ *i* ≥ *length* (*items* (*bs* ! *k*)) ⟹ *x* = *items* (*bs*!*k*) ! *i* ⟹ *next-symbol x* = *Some*
a ⟹
is-terminal $\mathcal{G}$ *a* ⟹ *k* < *length* ω ⟹ *Earley$_L$-bin' k* $\mathcal{G}$ ω *bs i* = *Earley$_L$-bin'*
k $\mathcal{G}$ ω (*bins-upd bs* (*k*+*1*) (*Scan$_L$ k* ω *a x i*)) (*i*+*1*)
¬ *i* ≥ *length* (*items* (*bs* ! *k*)) ⟹ *x* = *items* (*bs*!*k*) ! *i* ⟹ *next-symbol x* = *Some*
a ⟹

is-terminal $\mathcal{G}$ $a \Longrightarrow \neg\ k < \mathit{length}\ \omega \Longrightarrow \mathit{Earley}_L\text{-}\mathit{bin}'\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i = \mathit{Earley}_L\text{-}\mathit{bin}'\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ (i{+}1)$
$\neg\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k)) \Longrightarrow x = \mathit{items}\ (\mathit{bs}!k)\ !\ i \Longrightarrow \mathit{next\text{-}symbol}\ x = \mathit{Some}\ a \Longrightarrow$
$\neg$ *is-terminal* $\mathcal{G}$ $a \Longrightarrow \mathit{Earley}_L\text{-}\mathit{bin}'\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i = \mathit{Earley}_L\text{-}\mathit{bin}'\ k\ \mathcal{G}\ \omega\ (\mathit{bins\text{-}upd}\ \mathit{bs}\ k\ (\mathit{Predict}_L\ k\ \mathcal{G}\ a))\ (i{+}1)$
by (*subst* $\mathit{Earley}_L\text{-}\mathit{bin}'.\mathit{simps}$, *simp*)+

lemma $\mathit{Earley}_L\text{-}\mathit{bin}'\text{-}\mathit{induct}$[*case-names Base* $\mathit{Complete}_F$ Scan_F *Pass* $\mathit{Predict}_F$]:
assumes (k, $\mathcal{G}$, ω, *bs*) $\in$ *wf-earley-input*
assumes *base*: $\bigwedge k\ \mathcal{G}\ \omega\ \mathit{bs}\ i.\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k)) \Longrightarrow P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i$
assumes *complete*: $\bigwedge k\ \mathcal{G}\ \omega\ \mathit{bs}\ i\ x.\ \neg\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k)) \Longrightarrow x = \mathit{items}\ (\mathit{bs}\ !\ k)\ !\ i \Longrightarrow$
$\mathit{next\text{-}symbol}\ x = \mathit{None} \Longrightarrow P\ k\ \mathcal{G}\ \omega\ (\mathit{bins\text{-}upd}\ \mathit{bs}\ k\ (\mathit{Complete}_L\ k\ x\ \mathit{bs}\ i))\ (i{+}1) \Longrightarrow P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i$
assumes *scan*: $\bigwedge k\ \mathcal{G}\ \omega\ \mathit{bs}\ i\ x\ a.\ \neg\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k)) \Longrightarrow x = \mathit{items}\ (\mathit{bs}\ !\ k)\ !\ i \Longrightarrow$
$\mathit{next\text{-}symbol}\ x = \mathit{Some}\ a \Longrightarrow$ *is-terminal* $\mathcal{G}$ $a \Longrightarrow k < \mathit{length}\ \omega \Longrightarrow$
$P\ k\ \mathcal{G}\ \omega\ (\mathit{bins\text{-}upd}\ \mathit{bs}\ (k{+}1)\ (\mathit{Scan}_L\ k\ \omega\ a\ x\ i))\ (i{+}1) \Longrightarrow P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i$
assumes *pass*: $\bigwedge k\ \mathcal{G}\ \omega\ \mathit{bs}\ i\ x\ a.\ \neg\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k)) \Longrightarrow x = \mathit{items}\ (\mathit{bs}\ !\ k)\ !\ i \Longrightarrow$
$\mathit{next\text{-}symbol}\ x = \mathit{Some}\ a \Longrightarrow$ *is-terminal* $\mathcal{G}$ $a \Longrightarrow \neg\ k < \mathit{length}\ \omega \Longrightarrow$
$P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ (i{+}1) \Longrightarrow P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i$
assumes *predict*: $\bigwedge k\ \mathcal{G}\ \omega\ \mathit{bs}\ i\ x\ a.\ \neg\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k)) \Longrightarrow x = \mathit{items}\ (\mathit{bs}\ !\ k)\ !\ i \Longrightarrow$
$\mathit{next\text{-}symbol}\ x = \mathit{Some}\ a \Longrightarrow \neg$ *is-terminal* $\mathcal{G}$ $a \Longrightarrow$
$P\ k\ \mathcal{G}\ \omega\ (\mathit{bins\text{-}upd}\ \mathit{bs}\ k\ (\mathit{Predict}_L\ k\ \mathcal{G}\ a))\ (i{+}1) \Longrightarrow P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i$
shows $P\ k\ \mathcal{G}\ \omega\ \mathit{bs}\ i$
using *assms*(*1*)
proof (*induction n*≡*earley-measure* (k, $\mathcal{G}$, ω, *bs*) *i arbitrary*: *bs i rule*: *nat-less-induct*)
case *1*
have *wf*: $k \leq \mathit{length}\ \omega\ \mathit{length}\ \mathit{bs} = \mathit{length}\ \omega + 1$ *wf-*$\mathcal{G}$ $\mathcal{G}$ *wf-bins* $\mathcal{G}$ ω *bs*
using *1.prems wf-earley-input-elim* **by** *metis*+
hence *k*: $k < \mathit{length}\ \mathit{bs}$
by *simp*
have *fin*: *finite* $\{\ x \mid x.$ *wf-item* $\mathcal{G}\ \omega\ x \wedge$ *item-end* $x = k\ \}$
using *finiteness-UNIV-wf-item* **by** *fastforce*
show *?case*
proof *cases*
assume $i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k))$
then show *?thesis*
by (*simp add*: *base*)
next
assume *a1*: $\neg\ i \geq \mathit{length}\ (\mathit{items}\ (\mathit{bs}\ !\ k))$
let *?x* = $\mathit{items}\ (\mathit{bs}\ !\ k)\ !\ i$
have *x*: *?x* $\in$ *set* $(\mathit{items}\ (\mathit{bs}\ !\ k))$
using *a1* **by** *fastforce*
show *?thesis*
proof *cases*

assume *a2*: *next-symbol ?x = None*
let *?bs' = bins-upd bs k* (*Complete*$_L$ *k ?x bs i*)
have *item-origin ?x < length bs*
using *wf*(*4*) *k wf-bins-kth-bin wf-item-def x* **by** (*metis order-le-less-trans*)
hence *wf-bins'*: *wf-bins* $\mathcal{G}$ ω *?bs'*
using *wf-bins-Complete*$_L$ *distinct-Complete*$_L$ *wf*(*4*) *wf-bins-bins-upd k x* **by** *metis*
hence *wf'*: (*k*, $\mathcal{G}$, ω, *?bs'*) ∈ *wf-earley-input*
using *wf*(*1*,*2*,*3*) *wf-earley-input-intro* **by** *fastforce*
have *sub*: *set* (*items* (*?bs'* ! *k*)) ⊆ { *x* | *x*. *wf-item* $\mathcal{G}$ ω *x* ∧ *item-end x* = *k* }
using *wf*(*1*,*2*) *wf-bins'* **unfolding** *wf-bin-def wf-bins-def wf-bin-items-def* **using** *order-le-less-trans* **by** *auto*
have *i < length* (*items* (*?bs'* ! *k*))
using *a1* **by** (*metis dual-order.strict-trans1 items-def leI length-map length-nth-bin-bins-upd*)
also have *... = card* (*set* (*items* (*?bs'* ! *k*)))
using *wf*(*1*,*2*) *wf-bins' distinct-card wf-bins-def wf-bin-def* **by** (*metis k length-bins-upd*)
also have *... ≤ card* {*x* |*x*. *wf-item* $\mathcal{G}$ ω *x* ∧ *item-end x* = *k*}
using *card-mono fin sub* **by** *blast*
finally have *card* {*x* |*x*. *wf-item* $\mathcal{G}$ ω *x* ∧ *item-end x* = *k*} > *i*
by *blast*
hence *earley-measure* (*k*, $\mathcal{G}$, ω, *?bs'*) (*Suc i*) < *earley-measure* (*k*, $\mathcal{G}$, ω, *bs*) *i*
by *simp*
thus *?thesis*
using *1 a1 a2 complete wf'* **by** *simp*
next
assume *a2*: ¬ *next-symbol ?x = None*
then obtain *a* **where** *a-def*: *next-symbol ?x = Some a*
by *blast*
show *?thesis*
proof *cases*
assume *a3*: *is-terminal* $\mathcal{G}$ *a*
show *?thesis*
proof *cases*
assume *a4*: *k < length* ω
let *?bs' = bins-upd bs* (*k*+*1*) (*Scan*$_L$ *k* ω *a ?x i*)
have *wf-bins'*: *wf-bins* $\mathcal{G}$ ω *?bs'*
using *wf-bins-Scan*$_L$ *distinct-Scan*$_L$ *wf*(*1*,*4*) *wf-bins-bins-upd a2 a4 k x* **by** *metis*
hence *wf'*: (*k*, $\mathcal{G}$, ω, *?bs'*) ∈ *wf-earley-input*
using *wf*(*1*,*2*,*3*) *wf-earley-input-intro* **by** *fastforce*
have *sub*: *set* (*items* (*?bs'* ! *k*)) ⊆ { *x* | *x*. *wf-item* $\mathcal{G}$ ω *x* ∧ *item-end x* = *k* }
using *wf*(*1*,*2*) *wf-bins'* **unfolding** *wf-bin-def wf-bins-def wf-bin-items-def* **using** *order-le-less-trans* **by** *auto*
have *i < length* (*items* (*?bs'* ! *k*))
using *a1* **by** (*metis dual-order.strict-trans1 items-def leI length-map length-nth-bin-bins-upd*)
also have *... = card* (*set* (*items* (*?bs'* ! *k*)))

```
          using wf(1,2) wf-bins' distinct-card wf-bins-def wf-bin-def
          by (metis Suc-eq-plus1 le-imp-less-Suc length-bins-upd)
        also have ... ≤ card {x |x. wf-item G ω x ∧ item-end x = k}
          using card-mono fin sub by blast
        finally have card {x |x. wf-item G ω x ∧ item-end x = k} > i
          by blast
        hence earley-measure (k, G, ω, ?bs') (Suc i) < earley-measure (k, G, ω,
bs) i
          by simp
        thus ?thesis
          using 1 a1 a-def a3 a4 scan wf' by simp
      next
        assume a4: ¬ k < length ω
       have sub: set (items (bs ! k)) ⊆ { x | x. wf-item G ω x ∧ item-end x = k }
        using wf(1,2,4) unfolding wf-bin-def wf-bins-def wf-bin-items-def using
order-le-less-trans by auto
        have i < length (items (bs ! k))
          using a1 by simp
        also have ... = card (set (items (bs ! k)))
       using wf(1,2,4) distinct-card wf-bins-def wf-bin-def by (metis Suc-eq-plus1
le-imp-less-Suc)
        also have ... ≤ card {x |x. wf-item G ω x ∧ item-end x = k}
          using card-mono fin sub by blast
        finally have card {x |x. wf-item G ω x ∧ item-end x = k} > i
          by blast
       hence earley-measure (k, G, ω, bs) (Suc i) < earley-measure (k, G, ω, bs) i
          by simp
        thus ?thesis
          using 1 a1 a3 a4 a-def pass by simp
      qed
    next
      assume a3: ¬ is-terminal G a
      let ?bs' = bins-upd bs k (Predict_L k G a)
      have wf-bins': wf-bins G ω ?bs'
        using wf-bins-Predict_L distinct-Predict_L wf(1,3,4) wf-bins-bins-upd k x
by metis
      hence wf': (k, G, ω, ?bs') ∈ wf-earley-input
        using wf(1,2,3) wf-earley-input-intro by fastforce
      have sub: set (items (?bs' ! k)) ⊆ { x | x. wf-item G ω x ∧ item-end x = k }
        using wf(1,2) wf-bins' unfolding wf-bin-def wf-bins-def wf-bin-items-def
using order-le-less-trans by auto
      have i < length (items (?bs' ! k))
          using a1 by (metis dual-order.strict-trans1 items-def leI length-map
length-nth-bin-bins-upd)
      also have ... = card (set (items (?bs' ! k)))
        using wf(1,2) wf-bins' distinct-card wf-bins-def wf-bin-def
        by (metis Suc-eq-plus1 le-imp-less-Suc length-bins-upd)
      also have ... ≤ card {x |x. wf-item G ω x ∧ item-end x = k}
        using card-mono fin sub by blast
```

```
      finally have card {x |x. wf-item G ω x ∧ item-end x = k} > i
        by blast
      hence earley-measure (k, G, ω, ?bs') (Suc i) < earley-measure (k, G, ω, bs)
i
        by simp
      thus ?thesis
        using 1 a1 a-def a3 a-def predict wf' by simp
    qed
  qed
 qed
qed

lemma wf-earley-input-Earley_L-bin':
 assumes (k, G, ω, bs) ∈ wf-earley-input
 shows (k, G, ω, Earley_L-bin' k G ω bs i) ∈ wf-earley-input
 using assms
proof (induction i rule: Earley_L-bin'-induct[OF assms(1), case-names Base Com-
plete_F Scan_F Pass Predict_F])
 case (Complete_F k G ω bs i x)
 let ?bs' = bins-upd bs k (Complete_L k x bs i)
 have (k, G, ω, ?bs') ∈ wf-earley-input
  using Complete_F.hyps Complete_F.prems wf-earley-input-Complete_L by blast
 thus ?case
  using Complete_F.IH Complete_F.hyps by simp
next
 case (Scan_F k G ω bs i x a)
 let ?bs' = bins-upd bs (k+1) (Scan_L k ω a x i)
 have (k, G, ω, ?bs') ∈ wf-earley-input
  using Scan_F.hyps Scan_F.prems wf-earley-input-Scan_L by metis
 thus ?case
  using Scan_F.IH Scan_F.hyps by simp
next
 case (Predict_F k G ω bs i x a)
 let ?bs' = bins-upd bs k (Predict_L k G a)
 have (k, G, ω, ?bs') ∈ wf-earley-input
  using Predict_F.hyps Predict_F.prems wf-earley-input-Predict_L by metis
 thus ?case
  using Predict_F.IH Predict_F.hyps by simp
qed simp-all

lemma wf-earley-input-Earley_L-bin:
 assumes (k, G, ω, bs) ∈ wf-earley-input
 shows (k, G, ω, Earley_L-bin k G ω bs) ∈ wf-earley-input
 using assms by (simp add: Earley_L-bin-def wf-earley-input-Earley_L-bin')

lemma length-bins-Earley_L-bin':
 assumes (k, G, ω, bs) ∈ wf-earley-input
 shows length (Earley_L-bin' k G ω bs i) = length bs
 by (metis assms wf-earley-input-Earley_L-bin' wf-earley-input-elim)
```

lemma *length-nth-bin-Earley$_L$-bin′*:
assumes $(k, \mathcal{G}, \omega, bs) \in$ *wf-earley-input*
shows *length* (*items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*)) $\geq$ *length* (*items* (*bs* ! *l*))
using *length-nth-bin-bins-upd order-trans*
by (*induction i rule*: *Earley$_L$-bin′-induct*[*OF assms*]) (*auto simp*: *items-def*, *blast*+)

lemma *wf-bins-Earley$_L$-bin′*:
assumes $(k, \mathcal{G}, \omega, bs) \in$ *wf-earley-input*
shows *wf-bins* $\mathcal{G}$ ω (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i*)
using *assms wf-earley-input-Earley$_L$-bin′ wf-earley-input-elim* **by** *blast*

lemma *wf-bins-Earley$_L$-bin*:
assumes $(k, \mathcal{G}, \omega, bs) \in$ *wf-earley-input*
shows *wf-bins* $\mathcal{G}$ ω (*Earley$_L$-bin k* $\mathcal{G}$ ω *bs*)
using *assms Earley$_L$-bin-def wf-bins-Earley$_L$-bin′* **by** *metis*

lemma *kth-Earley$_L$-bin′-bins*:
assumes $(k, \mathcal{G}, \omega, bs) \in$ *wf-earley-input*
assumes $j <$ *length* (*items* (*bs* ! *l*))
shows *items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*) ! *j* = *items* (*bs* ! *l*) ! *j*
using *assms*(*2*)
proof (*induction i rule*: *Earley$_L$-bin′-induct*[*OF assms*(*1*), *case-names Base Complete$_F$ Scan$_F$ Pass Predict$_F$*])
case (*Complete$_F$ k* $\mathcal{G}$ ω *bs i x*)
let *?bs′* = *bins-upd bs k* (*Complete$_L$ k x bs i*)
have *items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*) ! *l*) ! *j* = *items* (*?bs′* ! *l*) ! *j*
using *Complete$_F$.IH Complete$_F$.prems length-nth-bin-bins-upd items-def order.strict-trans2* **by** (*metis length-map*)
also have ... = *items* (*bs* ! *l*) ! *j*
using *Complete$_F$.prems items-nth-idem-bins-upd nth-idem-bins-upd length-map items-def* **by** *metis*
finally show *?case*
using *Complete$_F$.hyps* **by** *simp*
next
case (*Scan$_F$ k* $\mathcal{G}$ ω *bs i x a*)
let *?bs′* = *bins-upd bs* (*k*+*1*) (*Scan$_L$ k* ω *a x i*)
have *items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*) ! *l*) ! *j* = *items* (*?bs′* ! *l*) ! *j*
using *Scan$_F$.IH Scan$_F$.prems length-nth-bin-bins-upd order.strict-trans2 items-def* **by** (*metis length-map*)
also have ... = *items* (*bs* ! *l*) ! *j*
using *Scan$_F$.prems items-nth-idem-bins-upd nth-idem-bins-upd length-map items-def* **by** *metis*
finally show *?case*
using *Scan$_F$.hyps* **by** *simp*
next
case (*Predict$_F$ k* $\mathcal{G}$ ω *bs i x a*)
let *?bs′* = *bins-upd bs k* (*Predict$_L$ k* $\mathcal{G}$ *a*)

have *items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*) ! *l*) ! *j* = *items* (*?bs′* ! *l*) ! *j*
using *Predict$_F$.IH Predict$_F$.prems length-nth-bin-bins-upd order.strict-trans2 items-def* **by** (*metis length-map*)
also have ... = *items* (*bs* ! *l*) ! *j*
using *Predict$_F$.prems items-nth-idem-bins-upd nth-idem-bins-upd length-map items-def* **by** *metis*
finally show *?case*
using *Predict$_F$.hyps* **by** *simp*
qed *simp-all*

lemma *nth-bin-sub-Earley$_L$-bin′*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
shows *set* (*items* (*bs* ! *l*)) ⊆ *set* (*items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*))
proof *standard*
fix *x*
assume *x* ∈ *set* (*items* (*bs* ! *l*))
then obtain *j* **where** ∗: *j* < *length* (*items* (*bs* ! *l*)) *items* (*bs* ! *l*) ! *j* = *x*
using *in-set-conv-nth* **by** *metis*
have *x* = *items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*) ! *j*
using *kth-Earley$_L$-bin′-bins assms* ∗ **by** *metis*
moreover have *j* < *length* (*items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*))
using *assms* ∗(*1*) *length-nth-bin-Earley$_L$-bin′ less-le-trans* **by** *blast*
ultimately show *x* ∈ *set* (*items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*))
by *simp*
qed

lemma *nth-Earley$_L$-bin′-eq*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
shows *l* < *k* ⟹ *Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l* = *bs* ! *l*
by (*induction i rule*: *Earley$_L$-bin′-induct*[*OF assms*]) (*auto simp*: *bins-upd-def*)

lemma *set-items-Earley$_L$-bin′-eq*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
shows *l* < *k* ⟹ *set* (*items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* ! *l*)) = *set* (*items* (*bs* ! *l*))
by (*simp add*: *assms nth-Earley$_L$-bin′-eq*)

lemma *bins-upto-k0-Earley$_L$-bin′-eq*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
shows *bins-upto* (*Earley$_L$-bin k* $\mathcal{G}$ ω *bs*) *k 0* = *bins-upto bs k 0*
unfolding *bins-upto-def bin-upto-def Earley$_L$-bin-def* **using** *set-items-Earley$_L$-bin′-eq assms nth-Earley$_L$-bin′-eq* **by** *fastforce*

lemma *wf-earley-input-Init$_L$*:
assumes *k* ≤ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows (*k*, $\mathcal{G}$, ω, *Init$_L$* $\mathcal{G}$ ω) ∈ *wf-earley-input*
proof −
let *?rs* = *filter* (λ*r*. *rule-head r* = $\mathfrak{S}$ $\mathcal{G}$) ($\mathfrak{R}$ $\mathcal{G}$)
let *?b0* = *map* (λ*r*. (*Entry* (*init-item r 0*) *Null*)) *?rs*
let *?bs* = *replicate* (*length* ω + *1*) ([])

have *distinct* (*items ?b0*)
using *assms* **unfolding** *wf-bin-def wf-item-def wf-*$\mathcal{G}$*-def distinct-rules-def items-def*
by (*auto simp*: *init-item-def distinct-map inj-on-def*)
moreover have $\forall\, x \in$ *set* (*items ?b0*). *wf-item* $\mathcal{G}\ \omega\ x \wedge$ *item-end* $x = 0$
using *assms* **unfolding** *wf-bin-def wf-item-def* **by** (*auto simp*: *init-item-def items-def*)
moreover have *wf-bins* $\mathcal{G}\ \omega$ *?bs*
unfolding *wf-bins-def wf-bin-def wf-bin-items-def items-def* **using** *less-Suc-eq-0-disj*
by *force*
ultimately show *?thesis*
using *assms length-replicate wf-earley-input-intro*
unfolding *wf-bin-def Init*$_L$*-def wf-bin-def wf-bin-items-def wf-bins-def*
by (*metis* (*no-types*, *lifting*) *length-list-update nth-list-update-eq nth-list-update-neq*)
qed

lemma *length-bins-Init*$_L$[*simp*]:
length (*Init*$_L$ $\mathcal{G}\ \omega$) = *length* ω + *1*
by (*simp add*: *Init*$_L$*-def*)

lemma *wf-earley-input-Earley*$_L$*-bins*[*simp*]:
assumes $k \leq$ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows (k, $\mathcal{G}$, ω, *Earley*$_L$*-bins* $k\ \mathcal{G}\ \omega$) $\in$ *wf-earley-input*
using *assms*
proof (*induction k*)
case *0*
have (k, $\mathcal{G}$, ω, *Init*$_L$ $\mathcal{G}\ \omega$) $\in$ *wf-earley-input*
using *assms wf-earley-input-Init*$_L$ **by** *blast*
thus *?case*
by (*simp add*: *assms*(*2*) *wf-earley-input-Init*$_L$ *wf-earley-input-Earley*$_L$*-bin*)
next
case (*Suc k*)
have (*Suc k*, $\mathcal{G}$, ω, *Earley*$_L$*-bins* $k\ \mathcal{G}\ \omega$) $\in$ *wf-earley-input*
using *Suc.IH Suc.prems*(*1*) *Suc-leD assms*(*2*) *wf-earley-input-elim wf-earley-input-intro*
by *metis*
thus *?case*
by (*simp add*: *wf-earley-input-Earley*$_L$*-bin*)
qed

lemma *length-Earley*$_L$*-bins*[*simp*]:
assumes $k \leq$ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *length* (*Earley*$_L$*-bins* $k\ \mathcal{G}\ \omega$) = *length* (*Init*$_L$ $\mathcal{G}\ \omega$)
using *assms wf-earley-input-Earley*$_L$*-bins wf-earley-input-elim* **by** *fastforce*

lemma *wf-bins-Earley*$_L$*-bins*[*simp*]:
assumes $k \leq$ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *wf-bins* $\mathcal{G}\ \omega$ (*Earley*$_L$*-bins* $k\ \mathcal{G}\ \omega$)
using *assms wf-earley-input-Earley*$_L$*-bins wf-earley-input-elim* **by** *fastforce*

lemma *wf-bins-Earley*$_L$:

wf-𝒢 𝒢 ⟹ *wf-bins 𝒢 ω* (*Earley*$_L$ *𝒢 ω*)
by (*simp add*: *Earley*$_L$*-def*)

8.5 Soundness

lemma *Init*$_L$*-eq-Init*$_F$:
bins (*Init*$_L$ *𝒢 ω*) = *Init*$_F$ *𝒢*
proof −
let *?rs* = *filter* (λ*r*. *rule-head r* = 𝔖 *𝒢*) (ℜ *𝒢*)
let *?b0* = *map* (λ*r*. (*Entry* (*init-item r 0*) *Null*)) *?rs*
let *?bs* = *replicate* (*length ω* + *1*) ([])
have *bins* (*?bs*[*0* := *?b0*]) = *set* (*items ?b0*)
proof −
have *bins* (*?bs*[*0* := *?b0*]) = ⋃ {*set* (*items* ((*?bs*[*0* := *?b0*]) ! *k*)) |*k*. *k* < *length* (*?bs*[*0* := *?b0*])}
unfolding *bins-def* **by** *blast*
also have ... = *set* (*items* ((*?bs*[*0* := *?b0*]) ! *0*)) ∪ ⋃ {*set* (*items* ((*?bs*[*0* := *?b0*]) ! *k*)) |*k*. *k* < *length* (*?bs*[*0* := *?b0*]) ∧ *k* ≠ *0*}
by *fastforce*
also have ... = *set* (*items* (*?b0*))
by (*auto simp*: *items-def*)
finally show *?thesis* **.**
qed
also have ... = *Init*$_F$ *𝒢*
by (*auto simp*: *Init*$_F$*-def items-def rule-head-def*)
finally show *?thesis*
by (*auto simp*: *Init*$_L$*-def*)
qed

lemma *Scan*$_L$*-sub-Scan*$_F$:
assumes *wf-bins 𝒢 ω bs bins bs* ⊆ *I x* ∈ *set* (*items* (*bs* ! *k*)) *k* < *length bs k* < *length ω*
assumes *next-symbol x* = *Some a*
shows *set* (*items* (*Scan*$_L$ *k ω a x pre*)) ⊆ *Scan*$_F$ *k ω I*
proof *standard*
fix *y*
assume ∗: *y* ∈ *set* (*items* (*Scan*$_L$ *k ω a x pre*))
have *x* ∈ *bin I k*
using *kth-bin-sub-bins assms*(*1*−*4*) *items-def wf-bin-def wf-bins-def wf-bin-items-def bin-def* **by** *fastforce*
{
assume #: *k* < *length ω ω*!*k* = *a*
hence *y* = *inc-item x* (*k*+*1*)
using ∗ **unfolding** *Scan*$_L$*-def* **by** (*simp add*: *items-def*)
hence *y* ∈ *Scan*$_F$ *k ω I*
using ‹*x* ∈ *bin I k*› # *assms*(*6*) **unfolding** *Scan*$_F$*-def* **by** *blast*
}
thus *y* ∈ *Scan*$_F$ *k ω I*
using ∗ *assms*(*5*) **unfolding** *Scan*$_L$*-def* **by** (*auto simp*: *items-def*)

qed

lemma *Predict$_L$-sub-Predict$_F$*:
assumes *wf-bins* $\mathcal{G}$ ω *bs bins bs* $\subseteq$ *I x* $\in$ *set* (*items* (*bs* ! *k*)) *k* < *length bs*
assumes *next-symbol x* = *Some X*
shows *set* (*items* (*Predict$_L$ k* $\mathcal{G}$ *X*)) $\subseteq$ *Predict$_F$ k* $\mathcal{G}$ *I*
proof *standard*
fix *y*
assume ∗: *y* $\in$ *set* (*items* (*Predict$_L$ k* $\mathcal{G}$ *X*))
have *x* $\in$ *bin I k*
using *kth-bin-sub-bins assms*(1−4) *items-def wf-bin-def wf-bins-def bin-def wf-bin-items-def* **by** *fast*
let *?rs* = *filter* (λ*r*. *rule-head r* = *X*) (ℜ $\mathcal{G}$)
let *?xs* = *map* (λ*r*. *init-item r k*) *?rs*
have *y* $\in$ *set ?xs*
using ∗ **unfolding** *Predict$_L$-def items-def* **by** *simp*
then obtain *r* **where** *y* = *init-item r k rule-head r* = *X r* $\in$ *set* (ℜ $\mathcal{G}$) *next-symbol x* = *Some* (*rule-head r*)
using *assms*(5) **by** *auto*
thus *y* $\in$ *Predict$_F$ k* $\mathcal{G}$ *I*
unfolding *Predict$_F$-def* **using** ‹*x* $\in$ *bin I k*› **by** *blast*
qed

lemma *Complete$_L$-sub-Complete$_F$*:
assumes *wf-bins* $\mathcal{G}$ ω *bs bins bs* $\subseteq$ *I y* $\in$ *set* (*items* (*bs* ! *k*)) *k* < *length bs*
assumes *next-symbol y* = *None*
shows *set* (*items* (*Complete$_L$ k y bs red*)) $\subseteq$ *Complete$_F$ k I*
proof *standard*
fix *x*
assume ∗: *x* $\in$ *set* (*items* (*Complete$_L$ k y bs red*))
have *y* $\in$ *bin I k*
using *kth-bin-sub-bins assms items-def wf-bin-def wf-bins-def bin-def wf-bin-items-def* **by** *fast*
let *?orig* = *bs* ! *item-origin y*
let *?xs* = *filter-with-index* (λ*x*. *next-symbol x* = *Some* (*item-rule-head y*)) (*items ?orig*)
let *?xs′* = *map* (λ(*x*, *pre*). (*Entry* (*inc-item x k*) (*PreRed* (*item-origin y*, *pre*, *red*) []))) *?xs*
have *0*: *item-origin y* < *length bs*
using *wf-bins-def wf-bin-def wf-item-def wf-bin-items-def assms*(1,3,4)
by (*metis Orderings.preorder-class.dual-order.strict-trans1 leD not-le-imp-less*)
{
fix *z*
assume ∗: *z* $\in$ *set* (*map fst ?xs*)
have *next-symbol z* = *Some* (*item-rule-head y*)
using ∗ **by** (*simp add*: *filter-with-index-cong-filter*)
moreover have *z* $\in$ *bin I* (*item-origin y*)
using *0* ∗ *assms*(1,2) *bin-def kth-bin-sub-bins wf-bins-kth-bin filter-with-index-cong-filter*
by (*metis* (*mono-tags*, *lifting*) *filter-is-subset in-mono mem-Collect-eq*)

ultimately have *next-symbol z = Some (item-rule-head y) z ∈ bin I (item-origin y)*
by *simp-all*
}
hence *1*: *∀ z ∈ set (map fst ?xs). next-symbol z = Some (item-rule-head y) ∧ z ∈ bin I (item-origin y)*
by *blast*
obtain *z* **where** *z*: *x = inc-item z k z ∈ set (map fst ?xs)*
using *∗* **unfolding** *Complete$_L$-def* **by** (*auto simp*: *rev-image-eqI items-def*)
moreover have *next-symbol z = Some (item-rule-head y) z ∈ bin I (item-origin y)*
using *1 z* **by** *blast+*
ultimately show *x ∈ Complete$_F$ k I*
using ‹*y ∈ bin I k*› *assms*(*5*) **unfolding** *Complete$_F$-def next-symbol-def* **by** (*auto split*: *if-splits*)
qed

lemma *sound-Scan$_L$*:
assumes *wf-bins 𝒢 ω bs bins bs ⊆ I x ∈ set (items (bs!k)) k < length bs k < length ω*
assumes *next-symbol x = Some a ∀ x ∈ I. wf-item 𝒢 ω x ∀ x ∈ I. sound-item 𝒢 ω x*
shows *∀ x ∈ set (items (Scan$_L$ k ω a x i)). sound-item 𝒢 ω x*
proof *standard*
fix *y*
assume *y ∈ set (items (Scan$_L$ k ω a x i))*
hence *y ∈ Scan$_F$ k ω I*
by (*meson Scan$_L$-sub-Scan$_F$ assms*(*1−6*) *in-mono*)
thus *sound-item 𝒢 ω y*
using *sound-Scan assms*(*7*,*8*) **unfolding** *Scan$_F$-def inc-item-def bin-def*
by (*smt* (*verit*, *best*) *item.exhaust-sel mem-Collect-eq*)
qed

lemma *sound-Predict$_L$*:
assumes *wf-bins 𝒢 ω bs bins bs ⊆ I x ∈ set (items (bs!k)) k < length bs*
assumes *next-symbol x = Some X ∀ x ∈ I. wf-item 𝒢 ω x ∀ x ∈ I. sound-item 𝒢 ω x*
shows *∀ x ∈ set (items (Predict$_L$ k 𝒢 X)). sound-item 𝒢 ω x*
proof *standard*
fix *y*
assume *y ∈ set (items (Predict$_L$ k 𝒢 X))*
hence *y ∈ Predict$_F$ k 𝒢 I*
by (*meson Predict$_L$-sub-Predict$_F$ assms*(*1−5*) *subsetD*)
thus *sound-item 𝒢 ω y*
using *sound-Predict assms*(*6*,*7*) **unfolding** *Predict$_F$-def init-item-def bin-def*
by (*smt* (*verit*, *del-insts*) *item.exhaust-sel mem-Collect-eq*)
qed

lemma *sound-Complete$_L$*:

assumes *wf-bins* $\mathcal{G}$ ω *bs bins bs* $\subseteq$ I $y \in$ *set* (*items* (*bs*!*k*)) $k <$ *length bs*
assumes *next-symbol* $y =$ *None* $\forall x \in I$. *wf-item* $\mathcal{G}$ ω x $\forall x \in I$. *sound-item* $\mathcal{G}$ ω x
shows $\forall x \in$ *set* (*items* (*Complete*$_L$ k y *bs* i)). *sound-item* $\mathcal{G}$ ω x
proof *standard*
fix x
assume $x \in$ *set* (*items* (*Complete*$_L$ k y *bs* i))
hence $x \in$ *Complete*$_F$ k I
using *Complete*$_L$*-sub-Complete*$_F$ *assms*(1−5) **by** *blast*
thus *sound-item* $\mathcal{G}$ ω x
using *sound-Complete assms*(6,7) **unfolding** *Complete*$_F$*-def inc-item-def bin-def*
by (*smt* (*verit*, *del-insts*) *item.exhaust-sel mem-Collect-eq*)
qed

lemma *sound-Earley*$_L$*-bin*′:
assumes (k, $\mathcal{G}$, ω, *bs*) $\in$ *wf-earley-input*
assumes $\forall x \in$ *bins bs*. *sound-item* $\mathcal{G}$ ω x
shows $\forall x \in$ *bins* (*Earley*$_L$*-bin*′ k $\mathcal{G}$ ω *bs* i). *sound-item* $\mathcal{G}$ ω x
using *assms*
proof (*induction i rule*: *Earley*$_L$*-bin*′*-induct*[*OF assms*(1), *case-names Base Complete*$_F$ *Scan*$_F$ *Pass Predict*$_F$])
case (*Complete*$_F$ k $\mathcal{G}$ ω *bs* i x)
let *?bs*′ = *bins-upd bs k* (*Complete*$_L$ k x *bs* i)
have $x \in$ *set* (*items* (*bs* ! *k*))
using *Complete*$_F$*.hyps*(1,2) **by** *force*
hence $\forall x \in$ *set* (*items* (*Complete*$_L$ k x *bs* i)). *sound-item* $\mathcal{G}$ ω x
using *sound-Complete*$_L$ *Complete*$_F$*.hyps*(3) *Complete*$_F$*.prems wf-earley-input-elim wf-bins-impl-wf-items* **by** *fastforce*
moreover have (k, $\mathcal{G}$, ω, *?bs*′) $\in$ *wf-earley-input*
using *Complete*$_F$*.hyps Complete*$_F$*.prems*(1) *wf-earley-input-Complete*$_L$ **by** *blast*
ultimately have $\forall x \in$ *bins* (*Earley*$_L$*-bin*′ k $\mathcal{G}$ ω *?bs*′ (i + 1)). *sound-item* $\mathcal{G}$ ω x
using *Complete*$_F$*.IH Complete*$_F$*.prems*(2) *length-bins-upd bins-bins-upd wf-earley-input-elim Suc-eq-plus1 Un-iff le-imp-less-Suc* **by** *metis*
thus *?case*
using *Complete*$_F$*.hyps* **by** *simp*
next
case (*Scan*$_F$ k $\mathcal{G}$ ω *bs* i x a)
let *?bs*′ = *bins-upd bs* (*k*+1) (*Scan*$_L$ k ω a x i)
have $x \in$ *set* (*items* (*bs* ! *k*))
using *Scan*$_F$*.hyps*(1,2) **by** *force*
hence $\forall x \in$ *set* (*items* (*Scan*$_L$ k ω a x i)). *sound-item* $\mathcal{G}$ ω x
using *sound-Scan*$_L$ *Scan*$_F$*.hyps*(3,5) *Scan*$_F$*.prems*(1,2) *wf-earley-input-elim wf-bins-impl-wf-items* **by** *fast*
moreover have (k, $\mathcal{G}$, ω, *?bs*′) $\in$ *wf-earley-input*
using *Scan*$_F$*.hyps Scan*$_F$*.prems*(1) *wf-earley-input-Scan*$_L$ **by** *metis*
ultimately have $\forall x \in$ *bins* (*Earley*$_L$*-bin*′ k $\mathcal{G}$ ω *?bs*′ (i + 1)). *sound-item* $\mathcal{G}$ ω x
using *Scan*$_F$*.IH Scan*$_F$*.hyps*(5) *Scan*$_F$*.prems*(2) *length-bins-upd bins-bins-upd*

wf-earley-input-elim
by (*metis UnE add-less-cancel-right*)
thus *?case*
using $Scan_F$*.hyps* **by** *simp*
next
case ($Predict_F$ *k* $\mathcal{G}$ ω *bs i x a*)
let *?bs′* = *bins-upd bs k* ($Predict_L$ *k* $\mathcal{G}$ *a*)
have *x* ∈ *set* (*items* (*bs* ! *k*))
using $Predict_F$*.hyps*(*1*,*2*) **by** *force*
hence ∀ *x* ∈ *set* (*items*($Predict_L$ *k* $\mathcal{G}$ *a*)). *sound-item* $\mathcal{G}$ ω *x*
using *sound-*$Predict_L$ $Predict_F$*.hyps*(*3*) $Predict_F$*.prems wf-earley-input-elim wf-bins-impl-wf-items* **by** *fast*
moreover have (*k*, $\mathcal{G}$, ω, *?bs′*) ∈ *wf-earley-input*
using $Predict_F$*.hyps* $Predict_F$*.prems*(*1*) *wf-earley-input-*$Predict_L$ **by** *metis*
ultimately have ∀ *x* ∈ *bins* ($Earley_L$*-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*)). *sound-item* $\mathcal{G}$ ω *x*
using $Predict_F$*.IH* $Predict_F$*.prems*(*2*) *length-bins-upd bins-bins-upd wf-earley-input-elim*
by (*metis Suc-eq-plus1 UnE*)
thus *?case*
using $Predict_F$*.hyps* **by** *simp*
qed *simp-all*

lemma *sound-*$Earley_L$*-bin*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes ∀ *x* ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω *x*
shows ∀ *x* ∈ *bins* ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*). *sound-item* $\mathcal{G}$ ω *x*
using *sound-*$Earley_L$*-bin′ assms* $Earley_L$*-bin-def* **by** *metis*

lemma $Earley_L$*-bin′-sub-*$Earley_F$*-bin*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes *bins bs* ⊆ *I*
shows *bins* ($Earley_L$*-bin′ k* $\mathcal{G}$ ω *bs i*) ⊆ $Earley_F$*-bin k* $\mathcal{G}$ ω *I*
using *assms*
proof (*induction i arbitrary*: *I rule*: $Earley_L$*-bin′-induct*[*OF assms*(*1*), *case-names Base* $Complete_F$ $Scan_F$ *Pass* $Predict_F$])
case (*Base k* $\mathcal{G}$ ω *bs i*)
thus *?case*
using $Earley_F$*-bin-mono* **by** *fastforce*
next
case ($Complete_F$ *k* $\mathcal{G}$ ω *bs i x*)
let *?bs′* = *bins-upd bs k* ($Complete_L$ *k x bs i*)
have *x* ∈ *set* (*items* (*bs* ! *k*))
using $Complete_F$*.hyps*(*1*,*2*) **by** *force*
hence *bins ?bs′* ⊆ *I* ∪ $Complete_F$ *k I*
using $Complete_L$*-sub-*$Complete_F$ $Complete_F$*.hyps*(*3*) $Complete_F$*.prems*(*1*,*2*) *bins-bins-upd wf-earley-input-elim* **by** *blast*
moreover have (*k*, $\mathcal{G}$, ω, *?bs′*) ∈ *wf-earley-input*
using $Complete_F$*.hyps* $Complete_F$*.prems*(*1*) *wf-earley-input-*$Complete_L$ **by** *blast*
ultimately have *bins* ($Earley_L$*-bin′ k* $\mathcal{G}$ ω *bs i*) ⊆ $Earley_F$*-bin k* $\mathcal{G}$ ω (*I* ∪

Complete$_F$ k I)
using *Complete$_F$.IH Complete$_F$.hyps* **by** *simp*
also have ... ⊆ *Earley$_F$-bin k G ω* (*Earley$_F$-bin k G ω I*)
using *Complete$_F$-Earley$_F$-bin-mono Earley$_F$-bin-mono Earley$_F$-bin-sub-mono*
by (*metis Un-subset-iff*)
finally show *?case*
using *Earley$_F$-bin-idem* **by** *blast*
next
case (*Scan$_F$ k G ω bs i x a*)
let *?bs′* = *bins-upd bs* (*k+1*) (*Scan$_L$ k ω a x i*)
have *x* ∈ *set* (*items* (*bs* ! *k*))
using *Scan$_F$.hyps*(*1*,*2*) **by** *force*
hence *bins ?bs′* ⊆ *I* ∪ *Scan$_F$ k ω I*
using *Scan$_L$-sub-Scan$_F$ Scan$_F$.hyps*(*3*,*5*) *Scan$_F$.prems bins-bins-upd wf-earley-input-elim*
by (*metis add-mono1 sup-mono*)
moreover have (*k*, *G*, *ω*, *?bs′*) ∈ *wf-earley-input*
using *Scan$_F$.hyps Scan$_F$.prems*(*1*) *wf-earley-input-Scan$_L$* **by** *metis*
ultimately have *bins* (*Earley$_L$-bin′ k G ω bs i*) ⊆ *Earley$_F$-bin k G ω* (*I* ∪ *Scan$_F$*
k ω I)
using *Scan$_F$.IH Scan$_F$.hyps* **by** *simp*
thus *?case*
using *Scan$_F$-Earley$_F$-bin-mono Earley$_F$-bin-mono Earley$_F$-bin-sub-mono Ear-*
ley$_F$-bin-idem **by** (*metis le-supI order-trans*)
next
case (*Pass k G ω bs i x a*)
thus *?case*
by *simp*
next
case (*Predict$_F$ k G ω bs i x a*)
let *?bs′* = *bins-upd bs k* (*Predict$_L$ k G a*)
have *x* ∈ *set* (*items* (*bs* ! *k*))
using *Predict$_F$.hyps*(*1*,*2*) **by** *force*
hence *bins ?bs′* ⊆ *I* ∪ *Predict$_F$ k G I*
using *Predict$_L$-sub-Predict$_F$ Predict$_F$.hyps*(*3*) *Predict$_F$.prems bins-bins-upd*
wf-earley-input-elim
by (*metis sup-mono*)
moreover have (*k*, *G*, *ω*, *?bs′*) ∈ *wf-earley-input*
using *Predict$_F$.hyps Predict$_F$.prems*(*1*) *wf-earley-input-Predict$_L$* **by** *metis*
ultimately have *bins* (*Earley$_L$-bin′ k G ω bs i*) ⊆ *Earley$_F$-bin k G ω* (*I* ∪
Predict$_F$ k G I)
using *Predict$_F$.IH Predict$_F$.hyps* **by** *simp*
thus *?case*
using *Predict$_F$-Earley$_F$-bin-mono Earley$_F$-bin-mono Earley$_F$-bin-sub-mono*
Earley$_F$-bin-idem **by** (*metis le-supI order-trans*)
qed

lemma *Earley$_L$-bin-sub-Earley$_F$-bin*:
assumes (*k*, *G*, *ω*, *bs*) ∈ *wf-earley-input*
assumes *bins bs* ⊆ *I*

shows *bins* (*Earley$_L$-bin k* $\mathcal{G}$ ω *bs*) $\subseteq$ *Earley$_F$-bin k* $\mathcal{G}$ ω *I*
using *assms Earley$_L$-bin'-sub-Earley$_F$-bin Earley$_L$-bin-def* **by** *metis*

lemma *Earley$_L$-bins-sub-Earley$_F$-bins*:
assumes $k \leq$ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *bins* (*Earley$_L$-bins k* $\mathcal{G}$ ω) $\subseteq$ *Earley$_F$-bins k* $\mathcal{G}$ ω
using *assms*
proof (*induction k*)
case *0*
have (*k*, $\mathcal{G}$, ω, *Init$_L$* $\mathcal{G}$ ω) $\in$ *wf-earley-input*
using *assms*(*1*) *assms*(*2*) *wf-earley-input-Init$_L$* **by** *blast*
thus *?case*
by (*simp add*: *Init$_L$-eq-Init$_F$ Earley$_L$-bin-sub-Earley$_F$-bin assms*(*2*) *wf-earley-input-Init$_L$*)
next
case (*Suc k*)
have (*Suc k*, $\mathcal{G}$, ω, *Earley$_L$-bins k* $\mathcal{G}$ ω) $\in$ *wf-earley-input*
by (*simp add*: *Suc.prems*(*1*) *Suc-leD assms*(*2*) *wf-earley-input-intro*)
thus *?case*
by (*simp add*: *Suc.IH Suc.prems*(*1*) *Suc-leD Earley$_L$-bin-sub-Earley$_F$-bin assms*(*2*))
qed

lemma *Earley$_L$-sub-Earley$_F$*:
wf-$\mathcal{G}$ $\mathcal{G}$ $\Longrightarrow$ *bins* (*Earley$_L$* $\mathcal{G}$ ω) $\subseteq$ *Earley$_F$* $\mathcal{G}$ ω
using *Earley$_L$-bins-sub-Earley$_F$-bins Earley$_F$-def Earley$_L$-def* **by** (*metis dual-order.refl*)

theorem *soundness-Earley$_L$*:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *recognizing* (*bins* (*Earley$_L$* $\mathcal{G}$ ω)) $\mathcal{G}$ ω
shows *derives* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω
using *assms Earley$_L$-sub-Earley$_F$ recognizing-def soundness-Earley$_F$* **by** (*meson subsetD*)

8.6 Completeness

lemma *bin-bins-upto-bins-eq*:
assumes *wf-bins* $\mathcal{G}$ ω *bs k* < *length bs i* $\geq$ *length* (*items* (*bs* ! *k*)) *l* $\leq$ *k*
shows *bin* (*bins-upto bs k i*) *l* = *bin* (*bins bs*) *l*
unfolding *bins-upto-def bins-def bin-def* **using** *assms nat-less-le*
apply (*auto simp*: *nth-list-update bin-upto-eq-set-items wf-bins-kth-bin items-def*)
apply (*metis imageI nle-le order-trans*, *fast*)
done

lemma *impossible-complete-item*:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *wf-item* $\mathcal{G}$ ω *x sound-item* $\mathcal{G}$ ω *x*
assumes *is-complete x item-origin x* = *k item-end x* = *k nonempty-derives* $\mathcal{G}$
shows *False*
proof −
have *derives* $\mathcal{G}$ [*item-rule-head x*] []
using *assms*(*3*−*6*) **by** (*simp add*: *slice-empty is-complete-def sound-item-def item-*β*-def*)

moreover have *is-nonterminal* $\mathcal{G}$ (*item-rule-head* x)
using *assms*(*1*,*2*) **unfolding** *wf-item-def item-rule-head-def rule-head-def*
by (*metis prod.collapse rule-nonterminal-type*)
ultimately show *?thesis*
using *assms*(*7*) *nonempty-derives-def is-nonterminal-def* **by** *metis*
qed

lemma *Complete$_F$-Un-eq-terminal*:
assumes *next-symbol* z = *Some* a *is-terminal* $\mathcal{G}$ a $\forall x \in I$. *wf-item* $\mathcal{G}$ ω x *wf-item* $\mathcal{G}$ ω z *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *Complete$_F$* k ($I \cup \{z\}$) = *Complete$_F$* k I
proof (*rule ccontr*)
assume *Complete$_F$* k ($I \cup \{z\}$) $\neq$ *Complete$_F$* k I
hence *Complete$_F$* k $I \subset$ *Complete$_F$* k ($I \cup \{z\}$)
using *Complete$_F$-sub-mono* **by** *blast*
then obtain w x y **where** $*$:
$w \in$ *Complete$_F$* k ($I \cup \{z\}$) $w \notin$ *Complete$_F$* k I w = *inc-item* x k
$x \in$ *bin* ($I \cup \{z\}$) (*item-origin* y) $y \in$ *bin* ($I \cup \{z\}$) k
is-complete y *next-symbol* x = *Some* (*item-rule-head* y)
unfolding *Complete$_F$-def* **by** *fast*
show *False*
proof (*cases* $x = z$)
case *True*
have *is-nonterminal* $\mathcal{G}$ (*item-rule-head* y)
using $*$(*5*,*6*) *assms*(*1*,*3*−*5*)
apply (*clarsimp simp*: *wf-item-def bin-def item-rule-head-def rule-head-def next-symbol-def*)
by (*metis prod.exhaust-sel rule-nonterminal-type*)
thus *?thesis*
using *True* $*$(*7*) *assms*(*1*,*2*,*5*) *is-terminal-nonterminal* **by** *fastforce*
next
case *False*
thus *?thesis*
using $*$ *assms*(*1*) **by** (*auto simp*: *next-symbol-def Complete$_F$-def bin-def*)
qed
qed

lemma *Complete$_F$-Un-eq-nonterminal*:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ $\forall x \in I$. *wf-item* $\mathcal{G}$ ω x $\forall x \in I$. *sound-item* $\mathcal{G}$ ω x
assumes *nonempty-derives* $\mathcal{G}$ *wf-item* $\mathcal{G}$ ω z
assumes *item-end* z = k *next-symbol* $z \neq$ *None*
shows *Complete$_F$* k ($I \cup \{z\}$) = *Complete$_F$* k I
proof (*rule ccontr*)
assume *Complete$_F$* k ($I \cup \{z\}$) $\neq$ *Complete$_F$* k I
hence *Complete$_F$* k $I \subset$ *Complete$_F$* k ($I \cup \{z\}$)
using *Complete$_F$-sub-mono* **by** *blast*
then obtain x x' y **where** $*$:
$x \in$ *Complete$_F$* k ($I \cup \{z\}$) $x \notin$ *Complete$_F$* k I x = *inc-item* x' k
$x' \in$ *bin* ($I \cup \{z\}$) (*item-origin* y) $y \in$ *bin* ($I \cup \{z\}$) k

```
    is-complete y next-symbol x' = Some (item-rule-head y)
    unfolding Complete_F-def by fast
  consider (A) x' = z | (B) y = z
    using *(2-7) Complete_F-def by (auto simp: bin-def; blast)
  thus False
  proof cases
    case A
    have item-origin y = k
     using *(4) A bin-def assms(6) by (metis (mono-tags, lifting) mem-Collect-eq)
    moreover have item-end y = k
      using *(5) bin-def by blast
    moreover have sound-item G ω y
     using *(5,6) assms(3,7) by (auto simp: bin-def next-symbol-def sound-item-def)
    moreover have wf-item G ω y
      using *(5) assms(2,5) wf-item-def by (auto simp: bin-def)
    ultimately show ?thesis
      using impossible-complete-item *(6) assms(1,4) by blast
  next
    case B
    thus ?thesis
      using *(6) assms(7) by (auto simp: next-symbol-def)
  qed
qed

lemma wf-item-in-kth-bin:
  wf-bins G ω bs ⟹ x ∈ bins bs ⟹ item-end x = k ⟹ x ∈ set (items (bs ! k))
  using bins-bin-exists wf-bins-kth-bin wf-bins-def by blast

lemma Complete_F-bins-upto-eq-bins:
  assumes wf-bins G ω bs k < length bs i ≥ length (items (bs ! k))
  shows Complete_F k (bins-upto bs k i) = Complete_F k (bins bs)
proof −
  have ⋀l. l ≤ k ⟹ bin (bins-upto bs k i) l = bin (bins bs) l
    using bin-bins-upto-bins-eq[OF assms] by blast
  moreover have ∀ x ∈ bins bs. wf-item G ω x
    using assms(1) wf-bins-impl-wf-items by metis
  ultimately show ?thesis
    unfolding Complete_F-def bin-def wf-item-def wf-item-def by auto
qed

lemma Complete_F-sub-bins-Un-Complete_L:
  assumes Complete_F k I ⊆ bins bs I ⊆ bins bs is-complete z wf-bins G ω bs
wf-item G ω z
  shows Complete_F k (I ∪ {z}) ⊆ bins bs ∪ set (items (Complete_L k z bs red))
proof standard
  fix w
  assume w ∈ Complete_F k (I ∪ {z})
  then obtain x y where *:
    w = inc-item x k x ∈ bin (I ∪ {z}) (item-origin y) y ∈ bin (I ∪ {z}) k
```

```
    is-complete y next-symbol x = Some (item-rule-head y)
    unfolding Complete_F-def by blast
  consider (A) x = z | (B) y = z | ¬ (x = z ∨ y = z)
    by blast
  thus w ∈ bins bs ∪ set (items (Complete_L k z bs red))
  proof cases
    case A
    thus ?thesis
      using ∗(5) assms(3) by (auto simp: next-symbol-def)
  next
    case B
    let ?orig = bs ! item-origin z
   let ?is = filter-with-index (λx. next-symbol x = Some (item-rule-head z)) (items
?orig)
    have x ∈ bin I (item-origin y)
      using B ∗(2) ∗(5) assms(3) by (auto simp: next-symbol-def bin-def)
    moreover have bin I (item-origin z) ⊆ set (items (bs ! item-origin z))
      using wf-item-in-kth-bin assms(2,4) bin-def by blast
    ultimately have x ∈ set (map fst ?is)
      using ∗(5) B by (simp add: filter-with-index-cong-filter in-mono)
    thus ?thesis
      unfolding Complete_L-def ∗(1) by (auto simp: rev-image-eqI items-def)
  next
    case 3
    thus ?thesis
      using ∗ assms(1) Complete_F-def by (auto simp: bin-def; blast)
  qed
qed

lemma Complete_L-eq-item-origin:
  bs ! item-origin y = bs′ ! item-origin y ⟹ Complete_L k y bs red = Complete_L
k y bs′ red
  by (auto simp: Complete_L-def)

lemma kth-bin-bins-upto-empty:
  assumes wf-bins G ω bs k < length bs
  shows bin (bins-upto bs k 0) k = {}
proof −
  {
    fix x
    assume x ∈ bins-upto bs k 0
    then obtain l where x ∈ set (items (bs ! l)) l < k
      unfolding bins-upto-def bin-upto-def by blast
    hence item-end x = l
      using wf-bins-kth-bin assms by fastforce
    hence item-end x < k
      using ‹l < k› by blast
  }
  thus ?thesis
```

by (*auto simp*: *bin-def*)
qed

lemma *Earley$_L$-bin′-mono*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
shows *bins bs* ⊆ *bins* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i*)
using *assms*
proof (*induction i rule*: *Earley$_L$-bin′-induct*[*OF assms*(*1*), *case-names Base Complete$_F$ Scan$_F$ Pass Predict$_F$*])
case (*Complete$_F$ k* $\mathcal{G}$ ω *bs i x*)
let *?bs′* = *bins-upd bs k* (*Complete$_L$ k x bs i*)
have *wf*: (*k*, $\mathcal{G}$, ω, *?bs′*) ∈ *wf-earley-input*
using *Complete$_F$.hyps Complete$_F$.prems*(*1*) *wf-earley-input-Complete$_L$* **by** *blast*
hence *bins bs* ⊆ *bins ?bs′*
using *length-bins-upd bins-bins-upd wf-earley-input-elim* **by** (*metis Un-upper1*)
also have ... ⊆ *bins* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*))
using *wf Complete$_F$.IH* **by** *blast*
finally show *?case*
using *Complete$_F$.hyps* **by** *simp*
next
case (*Scan$_F$ k* $\mathcal{G}$ ω *bs i x a*)
let *?bs′* = *bins-upd bs* (*k*+*1*) (*Scan$_L$ k* ω *a x i*)
have *wf*: (*k*, $\mathcal{G}$, ω, *?bs′*) ∈ *wf-earley-input*
using *Scan$_F$.hyps Scan$_F$.prems*(*1*) *wf-earley-input-Scan$_L$* **by** *metis*
hence *bins bs* ⊆ *bins ?bs′*
using *Scan$_F$.hyps*(*5*) *length-bins-upd bins-bins-upd wf-earley-input-elim*
by (*metis add-mono1 sup-ge1*)
also have ... ⊆ *bins* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*))
using *wf Scan$_F$.IH* **by** *blast*
finally show *?case*
using *Scan$_F$.hyps* **by** *simp*
next
case (*Predict$_F$ k* $\mathcal{G}$ ω *bs i x a*)
let *?bs′* = *bins-upd bs k* (*Predict$_L$ k* $\mathcal{G}$ *a*)
have *wf*: (*k*, $\mathcal{G}$, ω, *?bs′*) ∈ *wf-earley-input*
using *Predict$_F$.hyps Predict$_F$.prems*(*1*) *wf-earley-input-Predict$_L$* **by** *metis*
hence *bins bs* ⊆ *bins ?bs′*
using *length-bins-upd bins-bins-upd wf-earley-input-elim* **by** (*metis sup-ge1*)
also have ... ⊆ *bins* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*))
using *wf Predict$_F$.IH* **by** *blast*
finally show *?case*
using *Predict$_F$.hyps* **by** *simp*
qed *simp-all*

lemma *Earley$_F$-bin-step-sub-Earley$_L$-bin′*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes *Earley$_F$-bin-step k* $\mathcal{G}$ ω (*bins-upto bs k i*) ⊆ *bins bs*
assumes ∀ *x* ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω *x is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$
shows *Earley$_F$-bin-step k* $\mathcal{G}$ ω (*bins bs*) ⊆ *bins* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i*)

using *assms*
proof (*induction i rule*: $Earley_L$*-bin'-induct*[*OF assms*(*1*), *case-names Base Complete*$_F$ *Scan*$_F$ *Pass Predict*$_F$])
case (*Base k* $\mathcal{G}$ ω *bs i*)
have *bin* (*bins bs*) *k* = *bin* (*bins-upto bs k i*) *k*
using *Base.hyps Base.prems*(*1*) *bin-bins-upto-bins-eq wf-earley-input-elim* **by** *blast*
thus *?case*
using $Scan_F$*-bin-absorb* $Predict_F$*-bin-absorb* $Complete_F$*-bins-upto-eq-bins wf-earley-input-elim Base.hyps Base.prems*(*1*,*2*,*3*,*5*) $Earley_F$*-bin-step-def* $Complete_F$*-*$Earley_F$*-bin-step-mono* $Predict_F$*-*$Earley_F$*-bin-step-mono* $Scan_F$*-*$Earley_F$*-bin-step-mono* $Earley_L$*-bin'-mono*
by (*metis* (*no-types*, *lifting*) *Un-assoc sup.orderE*)
next
case ($Complete_F$ *k* $\mathcal{G}$ ω *bs i x*)
let *?bs'* = *bins-upd bs k* ($Complete_L$ *k x bs i*)
have *x*: *x* ∈ *set* (*items* (*bs* ! *k*))
using $Complete_F$*.hyps*(*1*,*2*) **by** *auto*
have *wf*: (*k*, $\mathcal{G}$, ω, *?bs'*) ∈ *wf-earley-input*
using $Complete_F$*.hyps* $Complete_F$*.prems*(*1*) *wf-earley-input-*$Complete_L$ **by** *blast*
hence *sound*: ∀ *x* ∈ *set* (*items* ($Complete_L$ *k x bs i*)). *sound-item* $\mathcal{G}$ ω *x*
using *sound-*$Complete_L$ $Complete_F$*.hyps*(*3*) $Complete_F$*.prems wf-earley-input-elim wf-bins-impl-wf-items x*
by (*metis dual-order.refl*)
have $Scan_F$ *k* ω (*bins-upto ?bs' k* (*i* + *1*)) ⊆ *bins ?bs'*
proof −
have $Scan_F$ *k* ω (*bins-upto ?bs' k* (*i* + *1*)) = $Scan_F$ *k* ω (*bins-upto ?bs' k i* ∪ {*items* (*?bs'* ! *k*) ! *i*})
using $Complete_F$*.hyps*(*1*) *bins-upto-Suc-Un length-nth-bin-bins-upd items-def*
by (*metis length-map linorder-not-less sup.boundedE sup.order-iff*)
also have ... = $Scan_F$ *k* ω (*bins-upto bs k i* ∪ {*x*})
using $Complete_F$*.hyps*(*1*,*2*) $Complete_F$*.prems*(*1*) *items-nth-idem-bins-upd bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis dual-order.refl items-def length-map not-le-imp-less*)
also have ... ⊆ *bins bs* ∪ $Scan_F$ *k* ω {*x*}
using $Complete_F$*.prems*(*2*,*3*) $Scan_F$*-Un* $Scan_F$*-*$Earley_F$*-bin-step-mono* **by** *fastforce*
also have ... = *bins bs*
using $Complete_F$*.hyps*(*3*) **by** (*auto simp*: $Scan_F$*-def bin-def*)
finally show *?thesis*
using $Complete_F$*.prems*(*1*) *wf-earley-input-elim bins-bins-upd* **by** *blast*
qed
moreover have $Predict_F$ *k* $\mathcal{G}$ (*bins-upto ?bs' k* (*i* + *1*)) ⊆ *bins ?bs'*
proof −
have $Predict_F$ *k* $\mathcal{G}$ (*bins-upto ?bs' k* (*i* + *1*)) = $Predict_F$ *k* $\mathcal{G}$ (*bins-upto ?bs' k i* ∪ {*items* (*?bs'* ! *k*) ! *i*})
using $Complete_F$*.hyps*(*1*) *bins-upto-Suc-Un length-nth-bin-bins-upd*
by (*metis dual-order.strict-trans1 items-def length-map not-le-imp-less*)
also have ... = $Predict_F$ *k* $\mathcal{G}$ (*bins-upto bs k i* ∪ {*x*})
using $Complete_F$*.hyps*(*1*,*2*) $Complete_F$*.prems*(*1*) *items-nth-idem-bins-upd*

bins-upto-kth-nth-idem wf-earley-input-elim
by (*metis dual-order.refl items-def length-map not-le-imp-less*)
also have ... $\subseteq$ *bins bs* $\cup$ *Predict*$_F$ *k* $\mathcal{G}$ $\{x\}$
using *Complete*$_F$*.prems*(*2*,*3*) *Predict*$_F$*-Un Predict*$_F$*-Earley*$_F$*-bin-step-mono*
by *blast*
also have ... = *bins bs*
using *Complete*$_F$*.hyps*(*3*) **by** (*auto simp*: *Predict*$_F$*-def bin-def*)
finally show *?thesis*
using *Complete*$_F$*.prems*(*1*) *wf-earley-input-elim bins-bins-upd* **by** *blast*
qed
moreover have *Complete*$_F$ *k* (*bins-upto ?bs*$'$ *k* (*i* + *1*)) $\subseteq$ *bins ?bs*$'$
proof −
have *Complete*$_F$ *k* (*bins-upto ?bs*$'$ *k* (*i* + *1*)) = *Complete*$_F$ *k* (*bins-upto ?bs*$'$ *k*
i $\cup$ {*items* (*?bs*$'$! *k*) ! *i*})
using *bins-upto-Suc-Un length-nth-bin-bins-upd Complete*$_F$*.hyps*(*1*)
by (*metis* (*no-types*, *opaque-lifting*) *dual-order.trans items-def length-map*
not-le-imp-less)
also have ... = *Complete*$_F$ *k* (*bins-upto bs k i* $\cup$ $\{x\}$)
using *items-nth-idem-bins-upd Complete*$_F$*.hyps*(*1*,*2*) *bins-upto-kth-nth-idem*
Complete$_F$*.prems*(*1*) *wf-earley-input-elim*
by (*metis dual-order.refl items-def length-map not-le-imp-less*)
also have ... $\subseteq$ *bins bs* $\cup$ *set* (*items* (*Complete*$_L$ *k x bs i*))
using *Complete*$_F$*-sub-bins-Un-Complete*$_L$ *Complete*$_F$*.hyps*(*3*) *Complete*$_F$*.prems*(*1*,*2*,*3*)
next-symbol-def
bins-upto-sub-bins wf-bins-kth-bin x Complete$_F$*-Earley*$_F$*-bin-step-mono wf-earley-input-elim*
by (*smt* (*verit*, *best*) *option.distinct*(*1*) *subset-trans*)
finally show *?thesis*
using *Complete*$_F$*.prems*(*1*) *wf-earley-input-elim bins-bins-upd* **by** *blast*
qed
ultimately have *Earley*$_F$*-bin-step k* $\mathcal{G}$ ω (*bins ?bs*$'$) $\subseteq$ *bins* (*Earley*$_L$*-bin*$'$ *k* $\mathcal{G}$ ω
?bs$'$ (*i*+*1*))
using *Complete*$_F$*.IH Complete*$_F$*.prems sound wf Earley*$_F$*-bin-step-def bins-upto-sub-bins*
wf-earley-input-elim bins-bins-upd
by (*metis UnE sup.boundedI*)
thus *?case*
using *Complete*$_F$*.hyps Complete*$_F$*.prems*(*1*) *Earley*$_L$*-bin*$'$*-simps*(*2*) *Earley*$_F$*-bin-step-sub-mono*
bins-bins-upd wf-earley-input-elim
by (*smt* (*verit*, *best*) *sup.coboundedI2 sup.orderE sup-ge1*)
next
case (*Scan*$_F$ *k* $\mathcal{G}$ ω *bs i x a*)
let *?bs*$'$ = *bins-upd bs* (*k*+*1*) (*Scan*$_L$ *k* ω *a x i*)
have *x*: *x* $\in$ *set* (*items* (*bs* ! *k*))
using *Scan*$_F$*.hyps*(*1*,*2*) **by** *auto*
hence *sound*: $\forall$ *x* $\in$ *set* (*items* (*Scan*$_L$ *k* ω *a x i*)). *sound-item* $\mathcal{G}$ ω *x*
using *sound-Scan*$_L$ *Scan*$_F$*.hyps*(*3*,*5*) *Scan*$_F$*.prems*(*1*,*2*,*3*) *wf-earley-input-elim*
wf-bins-impl-wf-items x
by (*metis dual-order.refl*)
have *wf*: (*k*, $\mathcal{G}$, ω, *?bs*$'$) $\in$ *wf-earley-input*
using *Scan*$_F$*.hyps Scan*$_F$*.prems*(*1*) *wf-earley-input-Scan*$_L$ **by** *metis*

have $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) \subseteq bins\ ?bs'$
proof −
have $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) = Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ i \cup \{items\ (?bs'\ !\ k)\ !\ i\})$
using *bins-upto-Suc-Un* $Scan_F$*.hyps*(1) *nth-idem-bins-upd*
by (*metis Suc-eq-plus1 items-def length-map lessI less-not-refl not-le-imp-less*)
also have ... = $Scan_F\ k\ \omega\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$
using $Scan_F$*.hyps*(1,2,5) $Scan_F$*.prems*(1,2) *nth-idem-bins-upd bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis add-mono-thms-linordered-field*(1) *items-def length-map less-add-one linorder-le-less-linear not-add-less1*)
also have ... ⊆ $bins\ bs \cup Scan_F\ k\ \omega\ \{x\}$
using $Scan_F$*.prems*(2,3) $Scan_F$*-Un* $Scan_F$*-Earley*$_F$*-bin-step-mono* **by** *fastforce*
finally have ∗: $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) \subseteq bins\ bs \cup Scan_F\ k\ \omega\ \{x\}$ **.**
show *?thesis*
proof *cases*
assume *a1*: $\omega!k = a$
hence $Scan_F\ k\ \omega\ \{x\} = \{inc\text{-}item\ x\ (k+1)\}$
using $Scan_F$*.hyps*(1−3,5) $Scan_F$*.prems*(1,2) *wf-earley-input-elim* **apply** (*auto simp*: $Scan_F$*-def bin-def*)
using *wf-bins-kth-bin x* **by** *blast*
hence $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) \subseteq bins\ bs \cup \{inc\text{-}item\ x\ (k+1)\}$
using ∗ **by** *blast*
also have ... = $bins\ bs \cup set\ (items\ (Scan_L\ k\ \omega\ a\ x\ i))$
using *a1* $Scan_F$*.hyps*(5) **by** (*auto simp*: $Scan_L$*-def items-def*)
also have ... = $bins\ ?bs'$
using $Scan_F$*.hyps*(5) $Scan_F$*.prems*(1) *wf-earley-input-elim bins-bins-upd* **by** (*metis add-mono1*)
finally show *?thesis* **.**
next
assume *a1*: ¬ $\omega!k = a$
hence $Scan_F\ k\ \omega\ \{x\} = \{\}$
using $Scan_F$*.hyps*(3) **by** (*auto simp*: $Scan_F$*-def bin-def*)
hence $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) \subseteq bins\ bs$
using ∗ **by** *blast*
also have ... ⊆ $bins\ ?bs'$
using $Scan_F$*.hyps*(5) $Scan_F$*.prems*(1) *wf-earley-input-elim bins-bins-upd*
by (*metis Un-left-absorb add-strict-right-mono subset-Un-eq*)
finally show *?thesis* **.**
qed
qed
moreover have $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) \subseteq bins\ ?bs'$
proof −
have $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ ?bs'\ k\ (i\ +\ 1)) = Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ ?bs'\ k\ i \cup \{items\ (?bs'\ !\ k)\ !\ i\})$
using *bins-upto-Suc-Un* $Scan_F$*.hyps*(1) *nth-idem-bins-upd*
by (*metis Suc-eq-plus1 dual-order.refl items-def length-map lessI linorder-not-less*)
also have ... = $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$

using $Scan_F.hyps(1,2,5)$ $Scan_F.prems(1,2)$ *nth-idem-bins-upd bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis add-strict-right-mono items-def le-add1 length-map less-add-one linorder-not-le*)
also have ... $\subseteq$ *bins bs* $\cup$ $Predict_F$ *k* $\mathcal{G}$ $\{x\}$
using $Scan_F.prems(2,3)$ $Predict_F$*-Un* $Predict_F$*-*$Earley_F$*-bin-step-mono* **by** *fastforce*
also have ... = *bins bs*
using $Scan_F.hyps(3,4)$ $Scan_F.prems(1)$ *is-terminal-nonterminal wf-earley-input-elim*
by (*auto simp*: $Predict_F$*-def bin-def rule-head-def*, *fastforce*)
finally show *?thesis*
using $Scan_F.hyps(5)$ $Scan_F.prems(1)$ **by** (*simp add*: *bins-bins-upd sup.coboundedI1 wf-earley-input-elim*)
qed
moreover have $Complete_F$ *k* (*bins-upto ?bs′ k* (*i* + 1)) $\subseteq$ *bins ?bs′*
proof −
have $Complete_F$ *k* (*bins-upto ?bs′ k* (*i* + 1)) = $Complete_F$ *k* (*bins-upto ?bs′ k i* $\cup$ {*items* (*?bs′* ! *k*) ! *i*})
using *bins-upto-Suc-Un* $Scan_F.hyps(1)$ *nth-idem-bins-upd*
by (*metis Suc-eq-plus1 items-def length-map lessI less-not-refl not-le-imp-less*)
also have ... = $Complete_F$ *k* (*bins-upto bs k i* $\cup$ $\{x\}$)
using $Scan_F.hyps(1,2,5)$ $Scan_F.prems(1,2)$ *nth-idem-bins-upd bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis add-mono1 items-def length-map less-add-one linorder-not-le not-add-less1*)
also have ... = $Complete_F$ *k* (*bins-upto bs k i*)
using $Complete_F$*-Un-eq-terminal* $Scan_F.hyps(3,4)$ $Scan_F.prems$ *bins-upto-sub-bins subset-iff*
wf-bins-impl-wf-items wf-bins-kth-bin wf-item-def x wf-earley-input-elim
by (*smt* (*verit*, *ccfv-threshold*))
finally show *?thesis*
using $Scan_F.hyps(5)$ $Scan_F.prems(1,2,3)$ $Complete_F$*-*$Earley_F$*-bin-step-mono* **by** (*auto simp*: *bins-bins-upd wf-earley-input-elim*, *blast*)
qed
ultimately have $Earley_F$*-bin-step k* $\mathcal{G}$ ω (*bins ?bs′*) $\subseteq$ *bins* ($Earley_L$*-bin′ k* $\mathcal{G}$ ω *?bs′* (*i*+1))
using $Scan_F.IH$ $Scan_F.prems$ $Scan_F.hyps(5)$ *sound wf* $Earley_F$*-bin-step-def bins-upto-sub-bins wf-earley-input-elim*
bins-bins-upd **by** (*metis UnE add-mono1 le-supI*)
thus *?case*
using $Earley_F$*-bin-step-sub-mono* $Earley_L$*-bin′-simps*(3) $Scan_F.hyps$ $Scan_F.prems(1)$ *wf-earley-input-elim bins-bins-upd*
by (*smt* (*verit*, *ccfv-SIG*) *add-mono1 sup.cobounded1 sup.coboundedI2 sup.orderE*)
next
case (*Pass k* $\mathcal{G}$ ω *bs i x a*)
have *x*: *x* $\in$ *set* (*items* (*bs* ! *k*))
using *Pass.hyps*(1,2) **by** *auto*
have $Scan_F$ *k* ω (*bins-upto bs k* (*i* + 1)) $\subseteq$ *bins bs*
using $Scan_F$*-def Pass.hyps*(5) **by** *auto*
moreover have $Predict_F$ *k* $\mathcal{G}$ (*bins-upto bs k* (*i* + 1)) $\subseteq$ *bins bs*

proof −
have $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ bs\ k\ (i\ +\ 1)) = Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ bs\ k\ i \cup \{items\ (bs\ !\ k)\ !\ i\})$
using *bins-upto-Suc-Un Pass.hyps*(*1*) **by** (*metis items-def length-map not-le-imp-less*)
also have ... = $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$
using *Pass.hyps*(*1*,*2*,*5*) *nth-idem-bins-upd bins-upto-kth-nth-idem* **by** *simp*
also have ... ⊆ *bins bs* ∪ $Predict_F\ k\ \mathcal{G}\ \{x\}$
using *Pass.prems*(*2*) $Predict_F$*-Un* $Predict_F$*-Earley*$_F$*-bin-step-mono* **by** *blast*
also have ... = *bins bs*
using *Pass.hyps*(*3*,*4*) *Pass.prems*(*1*) *is-terminal-nonterminal wf-earley-input-elim*
by (*auto simp*: $Predict_F$*-def bin-def rule-head-def*, *fastforce*)
finally show *?thesis*
using *bins-bins-upd Pass.hyps*(*5*) *Pass.prems*(*3*) **by** *auto*
qed
moreover have $Complete_F\ k\ (bins\text{-}upto\ bs\ k\ (i\ +\ 1)) \subseteq bins\ bs$
proof −
have $Complete_F\ k\ (bins\text{-}upto\ bs\ k\ (i\ +\ 1)) = Complete_F\ k\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$
using *bins-upto-Suc-Un Pass.hyps*(*1*,*2*)
by (*metis items-def length-map not-le-imp-less*)
also have ... = $Complete_F\ k\ (bins\text{-}upto\ bs\ k\ i)$
using $Complete_F$*-Un-eq-terminal Pass.hyps Pass.prems bins-upto-sub-bins subset-iff wf-bins-impl-wf-items wf-item-def wf-bins-kth-bin x wf-earley-input-elim* **by** (*smt* (*verit*, *best*))
finally show *?thesis*
using *Pass.prems*(*1*,*2*) $Complete_F$*-Earley*$_F$*-bin-step-mono wf-earley-input-elim* **by** *blast*
qed
ultimately have $Earley_F$*-bin-step k* $\mathcal{G}$ *ω* (*bins bs*) ⊆ *bins* ($Earley_L$*-bin′ k* $\mathcal{G}$ *ω bs* (*i*+*1*))
using *Pass.IH Pass.prems* $Earley_F$*-bin-step-def bins-upto-sub-bins wf-earley-input-elim*
by (*metis le-sup-iff*)
thus *?case*
using *bins-bins-upd Pass.hyps Pass.prems* **by** *simp*
next
case ($Predict_F\ k\ \mathcal{G}\ \omega\ bs\ i\ x\ a$)
let *?bs′* = *bins-upd bs k* ($Predict_L\ k\ \mathcal{G}\ a$)
have $k \geq length\ \omega \vee \neg\ \omega!k = a$
using $Predict_F$*.hyps*(*4*) $Predict_F$*.prems*(*4*) *is-word-is-terminal leI* **by** *blast*
have *x*: $x \in set\ (items\ (bs\ !\ k))$
using $Predict_F$*.hyps*(*1*,*2*) **by** *auto*
hence *sound*: ∀ $x \in set\ (items(Predict_L\ k\ \mathcal{G}\ a))$. *sound-item* $\mathcal{G}$ *ω x*
using *sound-*$Predict_L$ $Predict_F$*.hyps*(*3*) $Predict_F$*.prems wf-earley-input-elim wf-bins-impl-wf-items* **by** *fast*
have *wf*: (k, $\mathcal{G}$, *ω*, *?bs′*) ∈ *wf-earley-input*
using $Predict_F$*.hyps* $Predict_F$*.prems*(*1*) *wf-earley-input-*$Predict_L$ **by** *metis*
have *len*: $i < length\ (items\ (?bs'\ !\ k))$
using *length-nth-bin-bins-upd* $Predict_F$*.hyps*(*1*)

by (*metis dual-order.strict-trans1 items-def length-map linorder-not-less*)
have $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i + 1)) \subseteq bins\ ?bs'$
proof −
have $Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ (i + 1)) = Scan_F\ k\ \omega\ (bins\text{-}upto\ ?bs'\ k\ i \cup \{items\ (?bs'\ !\ k)\ !\ i\})$
using $Predict_F.hyps(1)$ *bins-upto-Suc-Un* **by** (*metis items-def len length-map*)
also have $... = Scan_F\ k\ \omega\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$
using $Predict_F.hyps(1,2)$ $Predict_F.prems(1)$ *items-nth-idem-bins-upd bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis dual-order.refl items-def length-map not-le-imp-less*)
also have $... \subseteq bins\ bs \cup Scan_F\ k\ \omega\ \{x\}$
using $Predict_F.prems(2,3)$ $Scan_F$*-Un* $Scan_F$-$Earley_F$*-bin-step-mono* **by** *fastforce*
also have $... = bins\ bs$
using $Predict_F.hyps(3)$ ‹*length* $\omega \leq k \vee \omega\ !\ k \neq a$› **by** (*auto simp*: $Scan_F$*-def bin-def*)
finally show *?thesis*
using $Predict_F.prems(1)$ *wf-earley-input-elim bins-bins-upd* **by** *blast*
qed
moreover have $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ ?bs'\ k\ (i + 1)) \subseteq bins\ ?bs'$
proof −
have $Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ ?bs'\ k\ (i + 1)) = Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ ?bs'\ k\ i \cup \{items\ (?bs'\ !\ k)\ !\ i\})$
using $Predict_F.hyps(1)$ *bins-upto-Suc-Un* **by** (*metis items-def len length-map*)
also have $... = Predict_F\ k\ \mathcal{G}\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$
using $Predict_F.hyps(1,2)$ $Predict_F.prems(1)$ *items-nth-idem-bins-upd bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis dual-order.refl items-def length-map not-le-imp-less*)
also have $... \subseteq bins\ bs \cup Predict_F\ k\ \mathcal{G}\ \{x\}$
using $Predict_F.prems(2,3)$ $Predict_F$*-Un* $Predict_F$-$Earley_F$*-bin-step-mono* **by** *fastforce*
also have $... = bins\ bs \cup set\ (items\ (Predict_L\ k\ \mathcal{G}\ a))$
using $Predict_F.hyps$ $Predict_F.prems(1-3)$ *wf-earley-input-elim*
apply (*auto simp*: $Predict_F$*-def* $Predict_L$*-def bin-def items-def*)
using *wf-bins-kth-bin x* **by** *blast*
finally show *?thesis*
using $Predict_F.prems(1)$ *wf-earley-input-elim bins-bins-upd* **by** *blast*
qed
moreover have $Complete_F\ k\ (bins\text{-}upto\ ?bs'\ k\ (i + 1)) \subseteq bins\ ?bs'$
proof −
have $Complete_F\ k\ (bins\text{-}upto\ ?bs'\ k\ (i + 1)) = Complete_F\ k\ (bins\text{-}upto\ ?bs'\ k\ i \cup \{items\ (?bs'\ !\ k)\ !\ i\})$
using *bins-upto-Suc-Un len* **by** (*metis items-def length-map*)
also have $... = Complete_F\ k\ (bins\text{-}upto\ bs\ k\ i \cup \{x\})$
using *items-nth-idem-bins-upd* $Predict_F.hyps(1,2)$ $Predict_F.prems(1)$ *bins-upto-kth-nth-idem wf-earley-input-elim*
by (*metis dual-order.refl items-def length-map not-le-imp-less*)
also have $... = Complete_F\ k\ (bins\text{-}upto\ bs\ k\ i)$
using $Complete_F$*-Un-eq-nonterminal* $Predict_F.prems$ *bins-upto-sub-bins Pre-*

dict$_F$.hyps(3)
subset-eq wf-bins-kth-bin x wf-bins-impl-wf-items wf-item-def wf-earley-input-elim
by (*smt* (*verit*, *ccfv-SIG*) *option.simps(3)*)
also have ... ⊆ *bins bs*
using *Complete$_F$-Earley$_F$-bin-step-mono Predict$_F$.prems(2)* **by** *blast*
finally show *?thesis*
using *bins-bins-upd Predict$_F$.prems(1,2,3) wf-earley-input-elim* **by** (*metis Un-upper1 dual-order.trans*)
qed
ultimately have *Earley$_F$-bin-step k 𝒢 ω* (*bins ?bs'*) ⊆ *bins* (*Earley$_L$-bin' k 𝒢 ω ?bs'* (*i+1*))
using *Predict$_F$.IH Predict$_F$.prems sound wf Earley$_F$-bin-step-def bins-upto-sub-bins bins-bins-upd wf-earley-input-elim* **by** (*metis UnE le-supI*)
hence *Earley$_F$-bin-step k 𝒢 ω* (*bins ?bs'*) ⊆ *bins* (*Earley$_L$-bin' k 𝒢 ω bs i*)
using *Predict$_F$.hyps Earley$_L$-bin'-simps(5)* **by** *simp*
moreover have *Earley$_F$-bin-step k 𝒢 ω* (*bins bs*) ⊆ *Earley$_F$-bin-step k 𝒢 ω* (*bins ?bs'*)
using *Earley$_F$-bin-step-sub-mono Predict$_F$.prems(1) wf-earley-input-elim bins-bins-upd* **by** (*metis Un-upper1*)
ultimately show *?case*
by *blast*
qed

lemma *Earley$_F$-bin-step-sub-Earley$_L$-bin*:
assumes (*k*, 𝒢, ω, *bs*) ∈ *wf-earley-input*
assumes *Earley$_F$-bin-step k 𝒢 ω* (*bins-upto bs k 0*) ⊆ *bins bs*
assumes ∀ *x* ∈ *bins bs*. *sound-item 𝒢 ω x is-word 𝒢 ω nonempty-derives 𝒢*
shows *Earley$_F$-bin-step k 𝒢 ω* (*bins bs*) ⊆ *bins* (*Earley$_L$-bin k 𝒢 ω bs*)
using *assms Earley$_F$-bin-step-sub-Earley$_L$-bin' Earley$_L$-bin-def* **by** *metis*

lemma *bins-eq-items-Complete$_L$*:
assumes *bins-eq-items as bs item-origin x* < *length as*
shows *items* (*Complete$_L$ k x as i*) = *items* (*Complete$_L$ k x bs i*)
proof −
let *?orig-a* = *as* ! *item-origin x*
let *?orig-b* = *bs* ! *item-origin x*
have *items ?orig-a* = *items ?orig-b*
using *assms* **by** (*metis* (*no-types*, *opaque-lifting*) *bins-eq-items-def length-map nth-map*)
thus *?thesis*
unfolding *Complete$_L$-def* **by** *simp*
qed

lemma *Earley$_L$-bin'-bins-eq*:
assumes (*k*, 𝒢, ω, *as*) ∈ *wf-earley-input*
assumes *bins-eq-items as bs wf-bins 𝒢 ω as*
shows *bins-eq-items* (*Earley$_L$-bin' k 𝒢 ω as i*) (*Earley$_L$-bin' k 𝒢 ω bs i*)
using *assms*
proof (*induction i arbitrary*: *bs rule*: *Earley$_L$-bin'-induct*[*OF assms(1)*, *case-names*

Base Complete$_F$ Scan$_F$ Pass Predict$_F$])
case (*Base k* $\mathcal{G}$ ω *as i*)
have *Earley$_L$-bin′ k* $\mathcal{G}$ ω *as i* = *as*
by (*simp add*: *Base.hyps*)
moreover have *Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* = *bs*
using *Base.hyps Base.prems*(*1*,*2*) **unfolding** *bins-eq-items-def*
by (*metis Earley$_L$-bin′-simps*(*1*) *length-map nth-map wf-earley-input-elim*)
ultimately show *?case*
using *Base.prems*(*2*) **by** *presburger*
next
case (*Complete$_F$ k* $\mathcal{G}$ ω *as i x*)
let *?as′* = *bins-upd as k* (*Complete$_L$ k x as i*)
let *?bs′* = *bins-upd bs k* (*Complete$_L$ k x bs i*)
have *k*: *k* < *length as*
using *Complete$_F$.prems*(*1*) *wf-earley-input-elim* **by** *blast*
hence *wf-x*: *wf-item* $\mathcal{G}$ ω *x*
using *Complete$_F$.hyps*(*1*,*2*) *Complete$_F$.prems*(*3*) *wf-bins-kth-bin* **by** *fastforce*
have (*k*, $\mathcal{G}$, ω, *?as′*) ∈ *wf-earley-input*
using *Complete$_F$.hyps Complete$_F$.prems*(*1*) *wf-earley-input-Complete$_L$* **by** *blast*
moreover have *bins-eq-items ?as′ ?bs′*
using *Complete$_F$.hyps*(*1*,*2*) *Complete$_F$.prems*(*2*,*3*) *bins-eq-items-dist-bins-upd bins-eq-items-Complete$_L$*
k wf-x wf-bins-kth-bin wf-item-def **by** (*metis dual-order.strict-trans2 leI nth-mem*)
ultimately have *bins-eq-items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?as′* (*i* + *1*)) (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*))
using *Complete$_F$.IH wf-earley-input-elim* **by** *blast*
moreover have *Earley$_L$-bin′ k* $\mathcal{G}$ ω *as i* = *Earley$_L$-bin′ k* $\mathcal{G}$ ω *?as′* (*i*+*1*)
using *Complete$_F$.hyps* **by** *simp*
moreover have *Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* = *Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i*+*1*)
using *Complete$_F$.hyps Complete$_F$.prems* **unfolding** *bins-eq-items-def*
by (*metis Earley$_L$-bin′-simps*(*2*) *map-eq-imp-length-eq nth-map wf-earley-input-elim*)
ultimately show *?case*
by *argo*
next
case (*Scan$_F$ k* $\mathcal{G}$ ω *as i x a*)
let *?as′* = *bins-upd as* (*k*+*1*) (*Scan$_L$ k* ω *a x i*)
let *?bs′* = *bins-upd bs* (*k*+*1*) (*Scan$_L$ k* ω *a x i*)
have (*k*, $\mathcal{G}$, ω, *?as′*) ∈ *wf-earley-input*
using *Scan$_F$.hyps Scan$_F$.prems*(*1*) *wf-earley-input-Scan$_L$* **by** *fast*
moreover have *bins-eq-items ?as′ ?bs′*
using *Scan$_F$.hyps*(*5*) *Scan$_F$.prems*(*1*,*2*) *bins-eq-items-dist-bins-upd add-mono1 wf-earley-input-elim* **by** *metis*
ultimately have *bins-eq-items* (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?as′* (*i* + *1*)) (*Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i* + *1*))
using *Scan$_F$.IH wf-earley-input-elim* **by** *blast*
moreover have *Earley$_L$-bin′ k* $\mathcal{G}$ ω *as i* = *Earley$_L$-bin′ k* $\mathcal{G}$ ω *?as′* (*i*+*1*)
using *Scan$_F$.hyps* **by** *simp*
moreover have *Earley$_L$-bin′ k* $\mathcal{G}$ ω *bs i* = *Earley$_L$-bin′ k* $\mathcal{G}$ ω *?bs′* (*i*+*1*)

using $Scan_F.hyps$ $Scan_F.prems$ **unfolding** *bins-eq-items-def*
by (*smt* (*verit*, *ccfv-threshold*) $Earley_L$*-bin′-simps*(*3*) *length-map nth-map wf-earley-input-elim*)
ultimately show *?case*
by *argo*
next
case ($Pass$ k $\mathcal{G}$ ω *as* i x a)
have *bins-eq-items* ($Earley_L$*-bin′* k $\mathcal{G}$ ω *as* (i + *1*)) ($Earley_L$*-bin′* k $\mathcal{G}$ ω *bs* (i + *1*))
using *Pass.prems Pass.IH* **by** *blast*
moreover have $Earley_L$*-bin′* k $\mathcal{G}$ ω *as* i = $Earley_L$*-bin′* k $\mathcal{G}$ ω *as* (*i+1*)
using *Pass.hyps* **by** *simp*
moreover have $Earley_L$*-bin′* k $\mathcal{G}$ ω *bs* i = $Earley_L$*-bin′* k $\mathcal{G}$ ω *bs* (*i+1*)
using *Pass.hyps Pass.prems* **unfolding** *bins-eq-items-def*
by (*metis* $Earley_L$*-bin′-simps*(*4*) *map-eq-imp-length-eq nth-map wf-earley-input-elim*)
ultimately show *?case*
by *argo*
next
case ($Predict_F$ k $\mathcal{G}$ ω *as* i x a)
let *?as′* = *bins-upd as k* ($Predict_L$ k $\mathcal{G}$ a)
let *?bs′* = *bins-upd bs k* ($Predict_L$ k $\mathcal{G}$ a)
have (k, $\mathcal{G}$, ω, *?as′*) ∈ *wf-earley-input*
using $Predict_F$*.hyps* $Predict_F$*.prems*(*1*) *wf-earley-input-*$Predict_L$ **by** *fast*
moreover have *bins-eq-items ?as′ ?bs′*
using $Predict_F$*.prems*(*1*,*2*) *bins-eq-items-dist-bins-upd wf-earley-input-elim* **by** *blast*
ultimately have *bins-eq-items* ($Earley_L$*-bin′* k $\mathcal{G}$ ω *?as′* (i + *1*)) ($Earley_L$*-bin′* k $\mathcal{G}$ ω *?bs′* (i + *1*))
using $Predict_F$*.IH wf-earley-input-elim* **by** *blast*
moreover have $Earley_L$*-bin′* k $\mathcal{G}$ ω *as* i = $Earley_L$*-bin′* k $\mathcal{G}$ ω *?as′* (*i+1*)
using $Predict_F$*.hyps* **by** *simp*
moreover have $Earley_L$*-bin′* k $\mathcal{G}$ ω *bs* i = $Earley_L$*-bin′* k $\mathcal{G}$ ω *?bs′* (*i+1*)
using $Predict_F$*.hyps* $Predict_F$*.prems* **unfolding** *bins-eq-items-def*
by (*metis* $Earley_L$*-bin′-simps*(*5*) *length-map nth-map wf-earley-input-elim*)
ultimately show *?case*
by *argo*
qed

lemma $Earley_L$*-bin′-idem*:
assumes (k, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes $i \leq j$ ∀ x ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω x *nonempty-derives* $\mathcal{G}$
shows *bins* ($Earley_L$*-bin′* k $\mathcal{G}$ ω ($Earley_L$*-bin′* k $\mathcal{G}$ ω *bs i*) *j*) = *bins* ($Earley_L$*-bin′* k $\mathcal{G}$ ω *bs i*)
using *assms*
proof (*induction i arbitrary*: *j rule*: $Earley_L$*-bin′-induct*[*OF assms*(*1*), *case-names Base* $Complete_F$ $Scan_F$ $Pass$ $Predict_F$])
case ($Complete_F$ k $\mathcal{G}$ ω *bs* i x)
let *?bs′* = *bins-upd bs k* ($Complete_L$ k x *bs* i)
have x: x ∈ *set* (*items* (*bs* ! k))
using $Complete_F$*.hyps*(*1*,*2*) **by** *auto*

```
  have wf: (k, 𝒢, ω, ?bs') ∈ wf-earley-input
   using Complete_F.hyps Complete_F.prems(1) wf-earley-input-Complete_L by blast
  hence ∀ x ∈ set (items (Complete_L k x bs i)). sound-item 𝒢 ω x
   using sound-Complete_L Complete_F.hyps(3) Complete_F.prems wf-earley-input-elim
wf-bins-impl-wf-items x
    by (metis dual-order.refl)
  hence sound: ∀ x ∈ bins ?bs'. sound-item 𝒢 ω x
    by (metis Complete_F.prems(1,3) UnE bins-bins-upd wf-earley-input-elim)
  show ?case
  proof cases
    assume i+1 ≤ j
    thus ?thesis
      using wf sound Complete_F Earley_L-bin'-simps(2) by metis
  next
    assume ¬ i+1 ≤ j
    hence i = j
      using Complete_F.prems(2) by simp
   have bins (Earley_L-bin' k 𝒢 ω (Earley_L-bin' k 𝒢 ω bs i) j) = bins (Earley_L-bin'
k 𝒢 ω (Earley_L-bin' k 𝒢 ω ?bs' (i+1)) j)
      using Earley_L-bin'-simps(2) Complete_F.hyps(1−3) by simp
     also have ... = bins (Earley_L-bin' k 𝒢 ω (Earley_L-bin' k 𝒢 ω ?bs' (i+1))
(j+1))
    proof −
      let ?bs'' = Earley_L-bin' k 𝒢 ω ?bs' (i+1)
      have length (items (?bs'' ! k)) ≥ length (items (bs ! k))
          using length-nth-bin-Earley_L-bin' length-nth-bin-bins-upd order-trans wf
Complete_F.hyps Complete_F.prems(1)
        by (smt (verit, ccfv-threshold) Earley_L-bin'-simps(2))
      hence 0: ¬ length (items (?bs'' ! k)) ≤ j
        using ‹i = j› Complete_F.hyps(1) by linarith
      have x = items (?bs' ! k) ! j
        using ‹i = j› items-nth-idem-bins-upd Complete_F.hyps(1,2)
        by (metis items-def length-map not-le-imp-less)
      hence 1: x = items (?bs'' ! k) ! j
        using ‹i = j› kth-Earley_L-bin'-bins Complete_F.hyps Complete_F.prems(1)
Earley_L-bin'-simps(2) leI by metis
      have bins (Earley_L-bin' k 𝒢 ω ?bs'' j) = bins (Earley_L-bin' k 𝒢 ω (bins-upd
?bs'' k (Complete_L k x ?bs'' i)) (j+1))
       using Earley_L-bin'-simps(2) 0 1 Complete_F.hyps(1,3) Complete_F.prems(2)
‹i = j› by auto
      moreover have bins-eq-items (bins-upd ?bs'' k (Complete_L k x ?bs'' i)) ?bs''
      proof −
        have k < length bs
          using Complete_F.prems(1) wf-earley-input-elim by blast
        have 0: set (Complete_L k x bs i) = set (Complete_L k x ?bs'' i)
        proof (cases item-origin x = k)
          case True
          thus ?thesis
                using impossible-complete-item kth-bin-sub-bins Complete_F.hyps(3)
```

Complete$_F$.prems wf-earley-input-elim
wf-bins-kth-bin x next-symbol-def **by** (*metis option.distinct(1) subsetD*)
next
case *False*
hence *item-origin x < k*
using *x Complete$_F$.prems(1) wf-bins-kth-bin wf-item-def nat-less-le* **by** (*metis wf-earley-input-elim*)
hence *bs ! item-origin x = ?bs'' ! item-origin x*
using *False nth-idem-bins-upd nth-Earley$_L$-bin'-eq wf* **by** *metis*
thus *?thesis*
using *Complete$_L$-eq-item-origin* **by** *metis*
qed
have *set (items (Complete$_L$ k x bs i)) ⊆ set (items (?bs' ! k))*
by (*simp add*: ‹*k < length bs*› *bins-upd-def set-items-bin-upds*)
hence *set (items (Complete$_L$ k x ?bs'' i)) ⊆ set (items (?bs' ! k))*
using *0* **by** (*simp add*: *items-def*)
also have *... ⊆ set (items (?bs'' ! k))*
by (*simp add*: *wf nth-bin-sub-Earley$_L$-bin'*)
finally show *?thesis*
using *bins-eq-items-bins-upd* **by** *blast*
qed
moreover have *(k, 𝒢, ω, bins-upd ?bs'' k (Complete$_L$ k x ?bs'' i)) ∈ wf-earley-input*
using *wf-earley-input-Earley$_L$-bin' wf-earley-input-Complete$_L$ Complete$_F$.hyps Complete$_F$.prems(1)*
‹*length (items (bs ! k)) ≤ length (items (?bs'' ! k))*› *kth-Earley$_L$-bin'-bins 0 1* **by** *blast*
ultimately show *?thesis*
using *Earley$_L$-bin'-bins-eq bins-eq-items-imp-eq-bins wf-earley-input-elim* **by** *blast*
qed
also have *... = bins (Earley$_L$-bin' k 𝒢 ω ?bs' (i + 1))*
using *Complete$_F$.IH*[*OF wf - sound Complete$_F$.prems(4)*] ‹*i = j*› **by** *blast*
finally show *?thesis*
using *Complete$_F$.hyps* **by** *simp*
qed
next
case (*Scan$_F$ k 𝒢 ω bs i x a*)
let *?bs' = bins-upd bs (k+1) (Scan$_L$ k ω a x i)*
have *x*: *x ∈ set (items (bs ! k))*
using *Scan$_F$.hyps(1,2)* **by** *auto*
hence *∀ x ∈ set (items (Scan$_L$ k ω a x i)). sound-item 𝒢 ω x*
using *sound-Scan$_L$ Scan$_F$.hyps(3,5) Scan$_F$.prems(1,2,3) wf-earley-input-elim wf-bins-impl-wf-items x*
by (*metis dual-order.refl*)
hence *sound*: *∀ x ∈ bins ?bs'. sound-item 𝒢 ω x*
using *Scan$_F$.hyps(5) Scan$_F$.prems(1,3) bins-bins-upd wf-earley-input-elim*
by (*metis UnE add-less-cancel-right*)
have *wf*: *(k, 𝒢, ω, ?bs') ∈ wf-earley-input*

```
    using Scan_F.hyps Scan_F.prems(1) wf-earley-input-Scan_L by metis
  show ?case
  proof cases
    assume i+1 ≤ j
    thus ?thesis
      using sound Scan_F by (metis Earley_L-bin'-simps(3) wf-earley-input-Scan_L)
  next
    assume ¬ i+1 ≤ j
    hence i = j
      using Scan_F.prems(2) by auto
   have bins (Earley_L-bin' k G ω (Earley_L-bin' k G ω bs i) j) = bins (Earley_L-bin'
k G ω (Earley_L-bin' k G ω ?bs' (i+1)) j)
      using Scan_F.hyps by simp
     also have ... = bins (Earley_L-bin' k G ω (Earley_L-bin' k G ω ?bs' (i+1))
(j+1))
    proof −
      let ?bs'' = Earley_L-bin' k G ω ?bs' (i+1)
      have length (items (?bs'' ! k)) ≥ length (items (bs ! k))
      using length-nth-bin-Earley_L-bin' length-nth-bin-bins-upd order-trans Scan_F.hyps
Scan_F.prems(1) Earley_L-bin'-simps(3)
        by (smt (verit, ccfv-SIG))
     hence bins (Earley_L-bin' k G ω ?bs'' j) = bins (Earley_L-bin' k G ω (bins-upd
?bs'' (k+1) (Scan_L k ω a x i)) (j+1))
       using ‹i = j› kth-Earley_L-bin'-bins nth-idem-bins-upd Earley_L-bin'-simps(3)
Scan_F.hyps Scan_F.prems(1) by (smt (verit, best) leI le-trans)
      moreover have bins-eq-items (bins-upd ?bs'' (k+1) (Scan_L k ω a x i)) ?bs''
      proof −
        have k+1 < length bs
          using Scan_F.hyps(5) Scan_F.prems wf-earley-input-elim by fastforce+
        hence set (items (Scan_L k ω a x i)) ⊆ set (items (?bs' ! (k+1)))
          by (simp add: bins-upd-def set-items-bin-upds)
        also have ... ⊆ set (items (?bs'' ! (k+1)))
          using wf nth-bin-sub-Earley_L-bin' by blast
        finally show ?thesis
          using bins-eq-items-bins-upd by blast
      qed
        moreover have (k, G, ω, bins-upd ?bs'' (k+1) (Scan_L k ω a x i)) ∈
wf-earley-input
      using wf-earley-input-Earley_L-bin' wf-earley-input-Scan_L Scan_F.hyps Scan_F.prems(1)
          ‹length (items (bs ! k)) ≤ length (items (?bs'' ! k))› kth-Earley_L-bin'-bins
       by (smt (verit, ccfv-SIG) Earley_L-bin'-simps(3) linorder-not-le order.trans)
      ultimately show ?thesis
       using Earley_L-bin'-bins-eq bins-eq-items-imp-eq-bins wf-earley-input-elim by
blast
    qed
    also have ... = bins (Earley_L-bin' k G ω ?bs' (i + 1))
     using ‹i = j› Scan_F.IH Scan_F.prems Scan_F.hyps sound wf-earley-input-Scan_L
by fast
    finally show ?thesis
```

```
    using Scan_F.hyps by simp
  qed
next
  case (Pass k 𝒢 ω bs i x a)
  show ?case
  proof cases
    assume i+1 ≤ j
    thus ?thesis
      using Pass by (metis Earley_L-bin'-simps(4))
  next
    assume ¬ i+1 ≤ j
    show ?thesis
     using Pass Earley_L-bin'-simps(1,4) kth-Earley_L-bin'-bins by (metis Suc-eq-plus1
Suc-leI antisym-conv2 not-le-imp-less)
  qed
next
  case (Predict_F k 𝒢 ω bs i x a)
  let ?bs' = bins-upd bs k (Predict_L k 𝒢 a)
  have x: x ∈ set (items (bs ! k))
    using Predict_F.hyps(1,2) by auto
  hence ∀ x ∈ set (items(Predict_L k 𝒢 a)). sound-item 𝒢 ω x
     using sound-Predict_L Predict_F.hyps(3) Predict_F.prems wf-earley-input-elim
wf-bins-impl-wf-items by fast
  hence sound: ∀ x ∈ bins ?bs'. sound-item 𝒢 ω x
    using Predict_F.prems(1,3) UnE bins-bins-upd wf-earley-input-elim by metis
  have wf: (k, 𝒢, ω, ?bs') ∈ wf-earley-input
    using Predict_F.hyps Predict_F.prems(1) wf-earley-input-Predict_L by metis
  have len: i < length (items (?bs' ! k))
   using length-nth-bin-bins-upd Predict_F.hyps(1) Orderings.preorder-class.dual-order.strict-trans1
linorder-not-less
    by (metis items-def length-map)
  show ?case
  proof cases
    assume i+1 ≤ j
    thus ?thesis
      using sound wf Predict_F by (metis Earley_L-bin'-simps(5))
  next
    assume ¬ i+1 ≤ j
    hence i = j
      using Predict_F.prems(2) by auto
   have bins (Earley_L-bin' k 𝒢 ω (Earley_L-bin' k 𝒢 ω bs i) j) = bins (Earley_L-bin'
k 𝒢 ω (Earley_L-bin' k 𝒢 ω ?bs' (i+1)) j)
      using Predict_F.hyps by simp
     also have ... = bins (Earley_L-bin' k 𝒢 ω (Earley_L-bin' k 𝒢 ω ?bs' (i+1))
(j+1))
    proof −
      let ?bs'' = Earley_L-bin' k 𝒢 ω ?bs' (i+1)
      have length (items (?bs'' ! k)) ≥ length (items (bs ! k))
        using length-nth-bin-Earley_L-bin' length-nth-bin-bins-upd order-trans wf
```

by (*metis* (*no-types*, *lifting*) *items-def length-map*)
hence *bins* (*Earley*$_L$*-bin'* *k* $\mathcal{G}$ ω *?bs''* *j*) = *bins* (*Earley*$_L$*-bin'* *k* $\mathcal{G}$ ω (*bins-upd* *?bs''* *k* (*Predict*$_L$ *k* $\mathcal{G}$ *a*)) (*j*+*1*))
using ‹*i* = *j*› *kth-Earley*$_L$*-bin'-bins nth-idem-bins-upd Earley*$_L$*-bin'-simps*(*5*) *Predict*$_F$*.hyps Predict*$_F$*.prems*(*1*) *length-bins-Earley*$_L$*-bin'*
wf-bins-Earley$_L$*-bin'* *wf-bins-kth-bin wf-item-def x* **by** (*smt* (*verit*, *ccfv-SIG*) *linorder-not-le order.trans*)
moreover have *bins-eq-items* (*bins-upd* *?bs''* *k* (*Predict*$_L$ *k* $\mathcal{G}$ *a*)) *?bs''*
proof −
have *k* < *length bs*
using *wf-earley-input-elim*[*OF Predict*$_F$*.prems*(*1*)] **by** *blast*
hence *set* (*items* (*Predict*$_L$ *k* $\mathcal{G}$ *a*)) ⊆ *set* (*items* (*?bs'* ! *k*))
by (*simp add*: *bins-upd-def set-items-bin-upds*)
also have ... ⊆ *set* (*items* (*?bs''* ! *k*))
using *wf nth-bin-sub-Earley*$_L$*-bin'* **by** *blast*
finally show *?thesis*
using *bins-eq-items-bins-upd* **by** *blast*
qed
moreover have (*k*, $\mathcal{G}$, ω, *bins-upd* *?bs''* *k* (*Predict*$_L$ *k* $\mathcal{G}$ *a*)) ∈ *wf-earley-input*
using *wf-earley-input-Earley*$_L$*-bin'* *wf-earley-input-Predict*$_L$ *Predict*$_F$*.hyps Predict*$_F$*.prems*(*1*)
‹*length* (*items* (*bs* ! *k*)) ≤ *length* (*items* (*?bs''* ! *k*))› *kth-Earley*$_L$*-bin'-bins*
by (*smt* (*verit*, *best*) *Earley*$_L$*-bin'-simps*(*5*) *dual-order.trans not-le-imp-less*)
ultimately show *?thesis*
using *Earley*$_L$*-bin'-bins-eq bins-eq-items-imp-eq-bins wf-earley-input-elim* **by** *blast*
qed
also have ... = *bins* (*Earley*$_L$*-bin'* *k* $\mathcal{G}$ ω *?bs'* (*i* + *1*))
using ‹*i* = *j*› *Predict*$_F$*.IH Predict*$_F$*.prems sound wf* **by** (*metis order-refl*)
finally show *?thesis*
using *Predict*$_F$*.hyps* **by** *simp*
qed
qed *simp*

lemma *Earley*$_L$*-bin-idem*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes ∀ *x* ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω *x nonempty-derives* $\mathcal{G}$
shows *bins* (*Earley*$_L$*-bin* *k* $\mathcal{G}$ ω (*Earley*$_L$*-bin* *k* $\mathcal{G}$ ω *bs*)) = *bins* (*Earley*$_L$*-bin* *k* $\mathcal{G}$ ω *bs*)
using *assms Earley*$_L$*-bin'-idem Earley*$_L$*-bin-def le0* **by** *metis*

lemma *funpower-Earley*$_F$*-bin-step-sub-Earley*$_L$*-bin*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes *Earley*$_F$*-bin-step* *k* $\mathcal{G}$ ω (*bins-upto bs k 0*) ⊆ *bins bs* ∀ *x* ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω *x*
assumes *is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$
shows *funpower* (*Earley*$_F$*-bin-step* *k* $\mathcal{G}$ ω) *n* (*bins bs*) ⊆ *bins* (*Earley*$_L$*-bin* *k* $\mathcal{G}$ ω *bs*)
using *assms*

proof (*induction n*)
case *0*
thus *?case*
using $Earley_L$*-bin′-mono* $Earley_L$*-bin-def* **by** (*simp add*: $Earley_L$*-bin′-mono* $Earley_L$*-bin-def*)
next
case (*Suc n*)
have *0*: $Earley_F$*-bin-step k* $\mathcal{G}$ ω (*bins-upto* ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*) *k 0*) ⊆ *bins* ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*)
using $Earley_L$*-bin′-mono bins-upto-k0-*$Earley_L$*-bin′-eq assms*(*1*,*2*) $Earley_L$*-bin-def order-trans*
by (*metis* (*no-types*, *lifting*))
have *funpower* ($Earley_F$*-bin-step k* $\mathcal{G}$ ω) (*Suc n*) (*bins bs*) ⊆ $Earley_F$*-bin-step k* $\mathcal{G}$ ω (*bins* ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*))
using $Earley_F$*-bin-step-sub-mono Suc* **by** (*metis funpower.simps*(*2*))
also have ... ⊆ *bins* ($Earley_L$*-bin k* $\mathcal{G}$ ω ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*))
using $Earley_F$*-bin-step-sub-*$Earley_L$*-bin Suc.prems wf-bins-*$Earley_L$*-bin sound-*$Earley_L$*-bin 0 wf-earley-input-*$Earley_L$*-bin* **by** *blast*
also have ... ⊆ *bins* ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*)
using $Earley_L$*-bin-idem Suc.prems* **by** *blast*
finally show *?case* **.**
qed

lemma $Earley_F$*-bin-sub-*$Earley_L$*-bin*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes $Earley_F$*-bin-step k* $\mathcal{G}$ ω (*bins-upto bs k 0*) ⊆ *bins bs* ∀ *x* ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω *x*
assumes *is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$
shows $Earley_F$*-bin k* $\mathcal{G}$ ω (*bins bs*) ⊆ *bins* ($Earley_L$*-bin k* $\mathcal{G}$ ω *bs*)
using *assms funpower-*$Earley_F$*-bin-step-sub-*$Earley_L$*-bin* $Earley_F$*-bin-def elem-limit-simp* **by** *fastforce*

lemma $Earley_F$*-bins-sub-*$Earley_L$*-bins*:
assumes *k* ≤ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$
assumes *is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$
shows $Earley_F$*-bins k* $\mathcal{G}$ ω ⊆ *bins* ($Earley_L$*-bins k* $\mathcal{G}$ ω)
using *assms*
proof (*induction k*)
case *0*
hence $Earley_F$*-bin 0* $\mathcal{G}$ ω ($Init_F$ $\mathcal{G}$) ⊆ *bins* ($Earley_L$*-bin 0* $\mathcal{G}$ ω ($Init_L$ $\mathcal{G}$ ω))
using $Earley_F$*-bin-sub-*$Earley_L$*-bin* $Init_L$*-eq-*$Init_F$ *length-bins-*$Init_L$ $Init_L$*-eq-*$Init_F$ *sound-Init bins-upto-empty*
$Earley_F$*-bin-step-empty bins-upto-sub-bins wf-earley-input-*$Init_L$ *wf-earley-input-elim*
by (*smt* (*verit*, *ccfv-threshold*) $Init_F$*-sub-Earley basic-trans-rules*(*31*) *sound-Earley wf-bins-impl-wf-items*)
thus *?case*
by *simp*
next
case (*Suc k*)

have *wf*: (*Suc k*, $\mathcal{G}$, ω, *Earley*$_L$*-bins k* $\mathcal{G}$ ω) $\in$ *wf-earley-input*
by (*simp add*: *Suc.prems*(*1*) *Suc-leD assms*(*2*) *wf-earley-input-intro*)
have *sub*: *Earley*$_F$*-bin-step* (*Suc k*) $\mathcal{G}$ ω (*bins-upto* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω) (*Suc k*) *0*) $\subseteq$ *bins* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω)
proof −
have *bin* (*bins-upto* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω) (*Suc k*) *0*) (*Suc k*) = {}
using *kth-bin-bins-upto-empty wf Suc.prems wf-earley-input-elim* **by** *blast*
hence *Earley*$_F$*-bin-step* (*Suc k*) $\mathcal{G}$ ω (*bins-upto* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω) (*Suc k*) *0*) = *bins-upto* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω) (*Suc k*) *0*
unfolding *Earley*$_F$*-bin-step-def Scan*$_F$*-def Complete*$_F$*-def Predict*$_F$*-def bin-def* **by** *blast*
also have ... $\subseteq$ *bins* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω)
using *wf Suc.prems bins-upto-sub-bins wf-earley-input-elim* **by** *blast*
finally show *?thesis* **.**
qed
have *sound*: $\forall$ *x* $\in$ *bins* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω). *sound-item* $\mathcal{G}$ ω *x*
using *Suc Earley*$_L$*-bins-sub-Earley*$_F$*-bins* **by** (*metis Suc-leD Earley*$_F$*-bins-sub-Earley in-mono sound-Earley wf-Earley*)
have *Earley*$_F$*-bins* (*Suc k*) $\mathcal{G}$ ω $\subseteq$ *Earley*$_F$*-bin* (*Suc k*) $\mathcal{G}$ ω (*bins* (*Earley*$_L$*-bins k* $\mathcal{G}$ ω))
using *Suc Earley*$_F$*-bin-sub-mono* **by** *simp*
also have ... $\subseteq$ *bins* (*Earley*$_L$*-bin* (*Suc k*) $\mathcal{G}$ ω (*Earley*$_L$*-bins k* $\mathcal{G}$ ω))
using *Earley*$_F$*-bin-sub-Earley*$_L$*-bin wf sub sound Suc.prems* **by** *fastforce*
finally show *?case*
by *simp*
qed

lemma *Earley*$_F$*-sub-Earley*$_L$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$
shows *Earley*$_F$ $\mathcal{G}$ ω $\subseteq$ *bins* (*Earley*$_L$ $\mathcal{G}$ ω)
using *assms Earley*$_F$*-bins-sub-Earley*$_L$*-bins Earley*$_F$*-def Earley*$_L$*-def* **by** (*metis le-refl*)

theorem *completeness-Earley*$_L$:
assumes *derives* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω *is-word* $\mathcal{G}$ ω *wf-*$\mathcal{G}$ $\mathcal{G}$ *nonempty-derives* $\mathcal{G}$
shows *recognizing* (*bins* (*Earley*$_L$ $\mathcal{G}$ ω)) $\mathcal{G}$ ω
using *assms Earley*$_F$*-sub-Earley*$_L$ *Earley*$_L$*-sub-Earley*$_F$ *completeness-Earley*$_F$ **by** (*metis subset-antisym*)

8.7 Correctness

theorem *Earley-eq-Earley*$_L$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$
shows *Earley* $\mathcal{G}$ ω = *bins* (*Earley*$_L$ $\mathcal{G}$ ω)
using *assms Earley*$_F$*-sub-Earley*$_L$ *Earley*$_L$*-sub-Earley*$_F$ *Earley-eq-Earley*$_F$ **by** *blast*

theorem *correctness-Earley*$_L$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *is-word* $\mathcal{G}$ ω *nonempty-derives* $\mathcal{G}$

shows *recognizing* (*bins* (*Earley*$_L$ $\mathcal{G}$ ω)) $\mathcal{G}$ ω $\longleftrightarrow$ *derives* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω
using *assms Earley-eq-Earley*$_L$ *correctness-Earley* **by** *fastforce*

end
theory *Earley-Parser*
imports
Earley-Recognizer
HOL−Library.Monad-Syntax
begin

9 Earley parser

9.1 Pointer lemmas

definition *predicts* :: *'a item* ⇒ *bool* **where**
predicts x ≡ *item-origin x* = *item-end x* ∧ *item-dot x* = *0*

definition *scans* :: *'a sentence* ⇒ *nat* ⇒ *'a item* ⇒ *'a item* ⇒ *bool* **where**
scans ω *k x y* ≡ *y* = *inc-item x k* ∧ (∃ *a*. *next-symbol x* = *Some a* ∧ ω!(*k*−*1*) = *a*)

definition *completes* :: *nat* ⇒ *'a item* ⇒ *'a item* ⇒ *'a item* ⇒ *bool* **where**
completes k x y z ≡ *y* = *inc-item x k* ∧ *is-complete z* ∧ *item-origin z* = *item-end x* ∧
(∃ *N*. *next-symbol x* = *Some N* ∧ *N* = *item-rule-head z*)

definition *sound-null-ptr* :: *'a entry* ⇒ *bool* **where**
sound-null-ptr e ≡ (*pointer e* = *Null* ⟶ *predicts* (*item e*))

definition *sound-pre-ptr* :: *'a sentence* ⇒ *'a bins* ⇒ *nat* ⇒ *'a entry* ⇒ *bool* **where**
sound-pre-ptr ω *bs k e* ≡ ∀ *pre*. *pointer e* = *Pre pre* ⟶
k > *0* ∧ *pre* < *length* (*bs*!(*k*−*1*)) ∧ *scans* ω *k* (*item* (*bs*!(*k*−*1*)!*pre*)) (*item e*)

definition *sound-prered-ptr* :: *'a bins* ⇒ *nat* ⇒ *'a entry* ⇒ *bool* **where**
sound-prered-ptr bs k e ≡ ∀ *p ps k' pre red*. *pointer e* = *PreRed p ps* ∧ (*k'*, *pre*, *red*) ∈ *set* (*p*#*ps*) ⟶
k' < *k* ∧ *pre* < *length* (*bs*!*k'*) ∧ *red* < *length* (*bs*!*k*) ∧ *completes k* (*item* (*bs*!*k'*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))

definition *sound-ptrs* :: *'a sentence* ⇒ *'a bins* ⇒ *bool* **where**
sound-ptrs ω *bs* ≡ ∀ *k* < *length bs*. ∀ *e* ∈ *set* (*bs*!*k*).
sound-null-ptr e ∧ *sound-pre-ptr* ω *bs k e* ∧ *sound-prered-ptr bs k e*

definition *mono-red-ptr* :: *'a bins* ⇒ *bool* **where**
mono-red-ptr bs ≡ ∀ *k* < *length bs*. ∀ *i* < *length* (*bs*!*k*).
∀ *k' pre red ps*. *pointer* (*bs*!*k*!*i*) = *PreRed* (*k'*, *pre*, *red*) *ps* ⟶ *red* < *i*

lemma *nth-item-bin-upd*:
n < *length es* ⟹ *item* (*bin-upd e es* ! *n*) = *item* (*es*!*n*)

by (*induction es arbitrary*: *e n*) (*auto simp*: *less-Suc-eq-0-disj split*: *entry.splits pointer.splits*)

lemma *bin-upd-append*:
item e ∉ *set* (*items es*) ⟹ *bin-upd e es* = *es* @ [*e*]
by (*induction es arbitrary*: *e*) (*auto simp*: *items-def split*: *entry.splits pointer.splits*)

lemma *bin-upd-null-pre*:
item e ∈ *set* (*items es*) ⟹ *pointer e* = *Null* ∨ *pointer e* = *Pre pre* ⟹ *bin-upd e es* = *es*
by (*induction es arbitrary*: *e*) (*auto simp*: *items-def split*: *entry.splits*)

lemma *bin-upd-prered-nop*:
assumes *distinct* (*items es*) *i* < *length es*
assumes *item e* = *item* (*es*!*i*) *pointer e* = *PreRed p ps* ∄*p ps*. *pointer* (*es*!*i*) = *PreRed p ps*
shows *bin-upd e es* = *es*
using *assms*
by (*induction es arbitrary*: *e i*) (*auto simp*: *less-Suc-eq-0-disj items-def split*: *entry.splits pointer.splits*)

lemma *bin-upd-prered-upd*:
assumes *distinct* (*items es*) *i* < *length es*
assumes *item e* = *item* (*es*!*i*) *pointer e* = *PreRed p rs pointer* (*es*!*i*) = *PreRed p′ rs′ bin-upd e es* = *es′*
shows *pointer* (*es′*!*i*) = *PreRed p′* (*p*#*rs*@*rs′*) ∧ (∀ *j* < *length es′*. *i*≠*j* ⟶ *es′*!*j* = *es*!*j*) ∧ *length* (*bin-upd e es*) = *length es*
using *assms*
proof (*induction es arbitrary*: *e i es′*)
case (*Cons e′ es*)
show *?case*
proof *cases*
assume ∗: *item e* = *item e′*
show *?thesis*
proof (*cases* ∃ *x xp xs y yp ys*. *e* = *Entry x* (*PreRed xp xs*) ∧ *e′* = *Entry y* (*PreRed yp ys*))
case *True*
then obtain *x xp xs y yp ys* **where** *ee′*: *e* = *Entry x* (*PreRed xp xs*) *e′* = *Entry y* (*PreRed yp ys*) *x* = *y*
using ∗ **by** *auto*
have *simp*: *bin-upd e* (*e′* # *es′*) = *Entry x* (*PreRed yp* (*xp* # *xs* @ *ys*)) # *es′*
using *True ee′* **by** *simp*
show *?thesis*
using *Cons simp ee′* **apply** (*auto simp*: *items-def*)
using *less-Suc-eq-0-disj* **by** *fastforce*+
next
case *False*
hence *bin-upd e* (*e′* # *es′*) = *e′* # *es′*
using ∗ **by** (*auto split*: *pointer.splits entry.splits*)

```
    thus ?thesis
      using False ∗ Cons.prems(1,2,3,4,5) by (auto simp: less-Suc-eq-0-disj
items-def split: entry.splits)
  qed
 next
  assume ∗: item e ≠ item e'
  have simp: bin-upd e (e' # es) = e' # bin-upd e es
   using ∗ by (auto split: pointer.splits entry.splits)
  have 0: distinct (items es)
   using Cons.prems(1) unfolding items-def by simp
  have 1: i−1 < length es
   using Cons.prems(2,3) ∗ by (metis One-nat-def leI less-diff-conv2 less-one
list.size(4) nth-Cons-0)
  have 2: item e = item (es!(i−1))
   using Cons.prems(3) ∗ by (metis nth-Cons')
  have 3: pointer e = PreRed p rs
   using Cons.prems(4) by simp
  have 4: pointer (es!(i−1)) = PreRed p' rs'
   using Cons.prems(3,5) ∗ by (metis nth-Cons')
  have pointer (bin-upd e es!(i−1)) = PreRed p' (p # rs @ rs') ∧
   (∀ j < length (bin-upd e es). i−1 ≠ j ⟶ (bin-upd e es) ! j = es ! j)
   using Cons.IH[OF 0 1 2 3 4] by blast
  hence pointer ((e' # bin-upd e es) ! i) = PreRed p' (p # rs @ rs') ∧
   (∀ j < length (e' # bin-upd e es). i ≠ j ⟶ (e' # bin-upd e es) ! j = (e' #
es) ! j)
   using ∗ Cons.prems(2,3) less-Suc-eq-0-disj by auto
  moreover have e' # bin-upd e es = es'
   using Cons.prems(6) simp by auto
  ultimately show ?thesis
   by (metis 0 1 2 3 4 Cons.IH Cons.prems(6) length-Cons)
 qed
qed simp

lemma sound-ptrs-bin-upd:
 assumes sound-ptrs ω bs k < length bs es = bs!k distinct (items es)
 assumes sound-null-ptr e sound-pre-ptr ω bs k e sound-prered-ptr bs k e
 shows sound-ptrs ω (bs[k := bin-upd e es])
 unfolding sound-ptrs-def
proof (standard, standard, standard)
 fix idx elem
 let ?bs = bs[k := bin-upd e es]
 assume a0: idx < length ?bs
 assume a1: elem ∈ set (?bs ! idx)
 show sound-null-ptr elem ∧ sound-pre-ptr ω ?bs idx elem ∧ sound-prered-ptr ?bs
idx elem
 proof cases
  assume a2: idx = k
  have elem ∈ set es ⟹ sound-pre-ptr ω bs idx elem
   using a0 a2 assms(1−3) sound-ptrs-def by blast
```

hence *pre-es*: *elem* ∈ *set es* ⟹ *sound-pre-ptr* ω *?bs idx elem*
using *a2* **unfolding** *sound-pre-ptr-def* **by** *force*
have *elem* = *e* ⟹ *sound-pre-ptr* ω *bs idx elem*
using *a2 assms*(*6*) **by** *auto*
hence *pre-e*: *elem* = *e* ⟹ *sound-pre-ptr* ω *?bs idx elem*
using *a2* **unfolding** *sound-pre-ptr-def* **by** *force*
have *elem* ∈ *set es* ⟹ *sound-prered-ptr bs idx elem*
using *a0 a2 assms*(*1*−*3*) *sound-ptrs-def* **by** *blast*
hence *prered-es*: *elem* ∈ *set es* ⟹ *sound-prered-ptr* (*bs*[*k* := *bin-upd e es*]) *idx elem*
using *a2 assms*(*2*,*3*) *length-bin-upd nth-item-bin-upd* **unfolding** *sound-prered-ptr-def*
by (*smt* (*verit*, *ccfv-SIG*) *dual-order.strict-trans1 nth-list-update*)
have *elem* = *e* ⟹ *sound-prered-ptr bs idx elem*
using *a2 assms*(*7*) **by** *auto*
hence *prered-e*: *elem* = *e* ⟹ *sound-prered-ptr ?bs idx elem*
using *a2 assms*(*2*,*3*) *length-bin-upd nth-item-bin-upd* **unfolding** *sound-prered-ptr-def*
by (*smt* (*verit*, *best*) *dual-order.strict-trans1 nth-list-update*)
consider (*A*) *item e* ∉ *set* (*items es*) |
(*B*) *item e* ∈ *set* (*items es*) ∧ (∃ *pre*. *pointer e* = *Null* ∨ *pointer e* = *Pre pre*)
|
(*C*) *item e* ∈ *set* (*items es*) ∧ ¬ (∃ *pre*. *pointer e* = *Null* ∨ *pointer e* = *Pre pre*)
by *blast*
thus *?thesis*
proof *cases*
case *A*
hence *elem* ∈ *set* (*es* @ [*e*])
using *a1 a2 bin-upd-append assms*(*2*) **by** *force*
thus *?thesis*
using *assms*(*1*−*3*,*5*) *pre-e pre-es prered-e prered-es sound-ptrs-def* **by** *auto*
next
case *B*
hence *elem* ∈ *set es*
using *a1 a2 bin-upd-null-pre assms*(*2*) **by** *force*
thus *?thesis*
using *assms*(*1*−*3*) *pre-es prered-es sound-ptrs-def* **by** *blast*
next
case *C*
then obtain *i p ps* **where** *C*: *i* < *length es* ∧ *item e* = *item* (*es*!*i*) ∧ *pointer e* = *PreRed p ps*
by (*metis assms*(*4*) *distinct-Ex1 items-def length-map nth-map pointer.exhaust*)
show *?thesis*
proof *cases*
assume ∄ *p' ps'*. *pointer* (*es*!*i*) = *PreRed p' ps'*
hence *C*: *elem* ∈ *set es*
using *a1 a2 C bin-upd-prered-nop assms*(*2*,*4*) **by** (*metis nth-list-update-eq*)
thus *?thesis*
using *assms*(*1*−*3*) *sound-ptrs-def pre-es prered-es* **by** *blast*
next

assume ¬ (∄ *p′ ps′*. *pointer* (*es*!*i*) = *PreRed p′ ps′*)
then obtain *p′ ps′* **where** *D*: *pointer* (*es*!*i*) = *PreRed p′ ps′*
by *blast*
hence *0*: *pointer* (*bin-upd e es*!*i*) = *PreRed p′* (*p*#*ps*@*ps′*) ∧ (∀ *j* < *length* (*bin-upd e es*). *i*≠*j* ⟶ *bin-upd e es*!*j* = *es*!*j*)
using *C assms*(*4*) *bin-upd-prered-upd* **by** *blast*
obtain *j* **where** *1*: *j* < *length es* ∧ *elem* = *bin-upd e es*!*j*
using *a1 a2 assms*(*2*) *C items-def bin-eq-items-bin-upd* **by** (*metis in-set-conv-nth length-map nth-list-update-eq nth-map*)
show *?thesis*
proof *cases*
assume *a3*: *i*=*j*
hence *a3*: *pointer elem* = *PreRed p′* (*p*#*ps*@*ps′*)
using *0 1* **by** *blast*
have *sound-null-ptr elem*
using *a3* **unfolding** *sound-null-ptr-def* **by** *simp*
moreover have *sound-pre-ptr* ω *?bs idx elem*
using *a3* **unfolding** *sound-pre-ptr-def* **by** *simp*
moreover have *sound-prered-ptr ?bs idx elem*
unfolding *sound-prered-ptr-def*
proof (*standard*, *standard*, *standard*, *standard*, *standard*, *standard*)
fix *P PS k′ pre red*
assume *a4*: *pointer elem* = *PreRed P PS* ∧ (*k′*, *pre*, *red*) ∈ *set* (*P*#*PS*)
show *k′* < *idx* ∧ *pre* < *length* (*bs*[*k* := *bin-upd e es*]!*k′*) ∧ *red* < *length* (*bs*[*k* := *bin-upd e es*]!*idx*) ∧
completes idx (*item* (*bs*[*k* := *bin-upd e es*]!*k′*!*pre*)) (*item elem*) (*item* (*bs*[*k* := *bin-upd e es*]!*idx*!*red*))
proof *cases*
assume *a5*: (*k′*, *pre*, *red*) ∈ *set* (*p*#*ps*)
show *?thesis*
using *0 1 C a2 a4 a5 prered-es assms*(*2*,*3*,*7*) *sound-prered-ptr-def length-bin-upd nth-item-bin-upd*
by (*smt* (*verit*) *dual-order.strict-trans1 nth-list-update-eq nth-list-update-neq nth-mem*)
next
assume *a5*: (*k′*, *pre*, *red*) ∉ *set* (*p*#*ps*)
hence *a5*: (*k′*, *pre*, *red*) ∈ *set* (*p′*#*ps′*)
using *a3 a4* **by** *auto*
have *k′* < *idx* ∧ *pre* < *length* (*bs*!*k′*) ∧ *red* < *length* (*bs*!*idx*) ∧
completes idx (*item* (*bs*!*k′*!*pre*)) (*item e*) (*item* (*bs*!*idx*!*red*))
using *assms*(*1*−*3*) *C D a2 a5* **unfolding** *sound-ptrs-def sound-prered-ptr-def* **by** (*metis nth-mem*)
thus *?thesis*
using *0 1 C a4 assms*(*2*,*3*) *length-bin-upd nth-item-bin-upd prered-es sound-prered-ptr-def*
by (*smt* (*verit*, *best*) *dual-order.strict-trans1 nth-list-update-eq nth-list-update-neq nth-mem*)
qed
qed

```
        ultimately show ?thesis
          by blast
      next
        assume a3: i≠j
        hence elem ∈ set es
          using 0 1 by (metis length-bin-upd nth-mem order-less-le-trans)
        thus ?thesis
          using assms(1−3) pre-es prered-es sound-ptrs-def by blast
      qed
    qed
  qed
 next
  assume a2: idx ≠ k
  have null: sound-null-ptr elem
    using a0 a1 a2 assms(1) sound-ptrs-def by auto
  have sound-pre-ptr ω bs idx elem
    using a0 a1 a2 assms(1,2) unfolding sound-ptrs-def by simp
  hence pre: sound-pre-ptr ω ?bs idx elem
   using assms(2,3) length-bin-upd nth-item-bin-upd unfolding sound-pre-ptr-def
    using dual-order.strict-trans1 nth-list-update by fastforce
  have sound-prered-ptr bs idx elem
    using a0 a1 a2 assms(1,2) unfolding sound-ptrs-def by simp
  hence prered: sound-prered-ptr ?bs idx elem
   using assms(2,3) length-bin-upd nth-item-bin-upd unfolding sound-prered-ptr-def
    by (smt (verit, best) dual-order.strict-trans1 nth-list-update)
  show ?thesis
    using null pre prered by blast
 qed
qed

lemma mono-red-ptr-bin-upd:
 assumes mono-red-ptr bs k < length bs es = bs!k distinct (items es)
 assumes ∀ k' pre red ps. pointer e = PreRed (k', pre, red) ps ⟶ red < length
es
 shows mono-red-ptr (bs[k := bin-upd e es])
 unfolding mono-red-ptr-def
proof (standard, standard)
 fix idx
 let ?bs = bs[k := bin-upd e es]
 assume a0: idx < length ?bs
 show ∀ i < length (?bs!idx). ∀ k' pre red ps. pointer (?bs!idx!i) = PreRed (k',
pre, red) ps ⟶ red < i
 proof cases
  assume a1: idx=k
  consider (A) item e ∉ set (items es) |
    (B) item e ∈ set (items es) ∧ (∃ pre. pointer e = Null ∨ pointer e = Pre pre)
|
    (C) item e ∈ set (items es) ∧ ¬ (∃ pre. pointer e = Null ∨ pointer e = Pre
pre)
```

```
    by blast
  thus ?thesis
  proof cases
    case A
    hence bin-upd e es = es @ [e]
      using bin-upd-append by blast
    thus ?thesis
      using a1 assms(1−3,5) mono-red-ptr-def
     by (metis length-append-singleton less-antisym nth-append nth-append-length
nth-list-update-eq)
  next
    case B
    hence bin-upd e es = es
      using bin-upd-null-pre by blast
    thus ?thesis
      using a1 assms(1−3) mono-red-ptr-def by force
  next
    case C
   then obtain i p ps where C: i < length es item e = item (es!i) pointer e =
PreRed p ps
      by (metis in-set-conv-nth items-def length-map nth-map pointer.exhaust)
    show ?thesis
    proof cases
      assume ∄ p' ps'. pointer (es!i) = PreRed p' ps'
      hence bin-upd e es = es
        using bin-upd-prered-nop C assms(4) by blast
      thus ?thesis
        using a1 assms(1−3) mono-red-ptr-def by (metis nth-list-update-eq)
    next
      assume ¬ (∄ p' ps'. pointer (es!i) = PreRed p' ps')
      then obtain p' ps' where D: pointer (es!i) = PreRed p' ps'
        by blast
      have 0: pointer (bin-upd e es!i) = PreRed p' (p#ps@ps') ∧
        (∀ j < length (bin-upd e es). i ≠ j ⟶ bin-upd e es!j = es!j) ∧
        length (bin-upd e es) = length es
        using C D assms(4) bin-upd-prered-upd by blast
      show ?thesis
    proof (standard, standard, standard, standard, standard, standard, standard)
        fix j k' pre red PS
        assume a2: j < length (?bs!idx)
        assume a3: pointer (?bs!idx!j) = PreRed (k', pre, red) PS
        have 1: ?bs!idx = bin-upd e es
          by (simp add: a1 assms(2))
        show red < j
        proof cases
          assume a4: i=j
          show ?thesis
            using 0 1 C(1) D a3 a4 assms(1−3) unfolding mono-red-ptr-def by
(metis pointer.inject(2))
```

```
        next
          assume a4: i≠j
          thus ?thesis
              using 0 1 a2 a3 assms(1) assms(2) assms(3) mono-red-ptr-def by
force
        qed
      qed
    qed
  qed
 next
  assume a1: idx≠k
  show ?thesis
   using a0 a1 assms(1) mono-red-ptr-def by fastforce
 qed
qed

lemma sound-mono-ptrs-bin-upds:
 assumes sound-ptrs ω bs mono-red-ptr bs k < length bs b = bs!k distinct (items
b) distinct (items es)
 assumes ∀ e ∈ set es. sound-null-ptr e ∧ sound-pre-ptr ω bs k e ∧ sound-prered-ptr
bs k e
 assumes ∀ e ∈ set es. ∀ k' pre red ps. pointer e = PreRed (k', pre, red) ps ⟶
red < length b
 shows sound-ptrs ω (bs[k := bin-upds es b]) ∧ mono-red-ptr (bs[k := bin-upds es
b])
 using assms
proof (induction es arbitrary: b bs)
 case (Cons e es)
 let ?bs = bs[k := bin-upd e b]
 have 0: sound-ptrs ω ?bs
  using sound-ptrs-bin-upd Cons.prems(1,3−5,7) by (metis list.set-intros(1))
 have 1: mono-red-ptr ?bs
  using mono-red-ptr-bin-upd Cons.prems(2−5,8) by auto
 have 2: k < length ?bs
  using Cons.prems(3) by simp
 have 3: bin-upd e b = ?bs!k
  using Cons.prems(3) by simp
 have 4: ∀ e' ∈ set es. sound-null-ptr e' ∧ sound-pre-ptr ω ?bs k e' ∧ sound-prered-ptr
?bs k e'
   using Cons.prems(3,4,7) length-bin-upd nth-item-bin-upd sound-pre-ptr-def
sound-prered-ptr-def
  by (smt (verit, ccfv-threshold) list.set-intros(2) nth-list-update order-less-le-trans)
 have 5: ∀ e' ∈ set es. ∀ k' pre red ps. pointer e' = PreRed (k', pre, red) ps ⟶
red < length (bin-upd e b)
   by (meson Cons.prems(8) length-bin-upd order-less-le-trans set-subset-Cons
subsetD)
 have sound-ptrs ω ((bs[k := bin-upd e b])[k := bin-upds es (bin-upd e b)]) ∧
   mono-red-ptr (bs[k := bin-upd e b, k := bin-upds es (bin-upd e b)])
  using Cons.IH[OF 0 1 2 3 - - 4 5] distinct-bin-upd Cons.prems(4,5,6) items-def
```

by (*metis distinct.simps*(*2*) *list.simps*(*9*))
thus *?case*
by *simp*
qed *simp*

lemma *sound-mono-ptrs-Earley$_L$-bin'*:
assumes (*k*, $\mathcal{G}$, ω, *bs*) ∈ *wf-earley-input*
assumes *sound-ptrs* ω *bs* ∀ *x* ∈ *bins bs*. *sound-item* $\mathcal{G}$ ω *x*
assumes *mono-red-ptr bs*
assumes *nonempty-derives* $\mathcal{G}$ *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *sound-ptrs* ω (*Earley$_L$-bin'* *k* $\mathcal{G}$ ω *bs i*) ∧ *mono-red-ptr* (*Earley$_L$-bin'* *k* $\mathcal{G}$ ω *bs i*)
using *assms*
proof (*induction i rule*: *Earley$_L$-bin'-induct*[*OF assms*(*1*), *case-names Base Complete$_F$ Scan$_F$ Pass Predict$_F$*])
case (*Complete$_F$ k* $\mathcal{G}$ ω *bs i x*)
let *?bs'* = *bins-upd bs k* (*Complete$_L$ k x bs i*)
have *x*: *x* ∈ *set* (*items* (*bs* ! *k*))
using *Complete$_F$.hyps*(*1*,*2*) **by** *force*
hence ∀ *x* ∈ *set* (*items* (*Complete$_L$ k x bs i*)). *sound-item* $\mathcal{G}$ ω *x*
using *sound-Complete$_L$ Complete$_F$.hyps*(*3*) *Complete$_F$.prems wf-earley-input-elim wf-bins-impl-wf-items x*
by (*metis dual-order.refl*)
hence *sound*: ∀ *x* ∈ *bins ?bs'*. *sound-item* $\mathcal{G}$ ω *x*
by (*metis Complete$_F$.prems*(*1*,*3*) *UnE bins-bins-upd wf-earley-input-elim*)
have *0*: *k* < *length bs*
using *Complete$_F$.prems*(*1*) *wf-earley-input-elim* **by** *auto*
have *1*: ∀ *e* ∈ *set* (*Complete$_L$ k x bs i*). *sound-null-ptr e*
unfolding *Complete$_L$-def sound-null-ptr-def* **by** *auto*
have *2*: ∀ *e* ∈ *set* (*Complete$_L$ k x bs i*). *sound-pre-ptr* ω *bs k e*
unfolding *Complete$_L$-def sound-pre-ptr-def* **by** *auto*
{
fix *e*
assume *a0*: *e* ∈ *set* (*Complete$_L$ k x bs i*)
fix *p ps k' pre red*
assume *a1*: *pointer e* = *PreRed p ps* (*k'*, *pre*, *red*) ∈ *set* (*p*#*ps*)
have *k'* = *item-origin x*
using *a0 a1* **unfolding** *Complete$_L$-def* **by** *auto*
moreover have *wf-item* $\mathcal{G}$ ω *x item-end x* = *k*
using *Complete$_F$.prems*(*1*) *x wf-earley-input-elim wf-bins-kth-bin* **by** *blast+*
ultimately have *0*: *k'* ≤ *k*
using *wf-item-def* **by** *blast*
have *1*: *k'* ≠ *k*
proof (*rule ccontr*)
assume ¬ *k'* ≠ *k*
have *sound-item* $\mathcal{G}$ ω *x*
using *Complete$_F$.prems*(*1*,*3*) *x kth-bin-sub-bins wf-earley-input-elim* **by** (*metis subset-eq*)
moreover have *is-complete x*

using $Complete_F.hyps(3)$ **by** (*auto simp*: *next-symbol-def split*: *if-splits*)
moreover have *item-origin* $x = k$
using $\langle \neg\ k' \neq k \rangle$ $\langle k' = item\text{-}origin\ x \rangle$ **by** *auto*
ultimately show *False*
using *impossible-complete-item* $Complete_F.prems(1,5)$ *wf-earley-input-elim* $\langle item\text{-}end\ x = k \rangle$ $\langle wf\text{-}item\ \mathcal{G}\ \omega\ x \rangle$ **by** *blast*
qed
have *2*: *pre* $<$ *length* (*bs*!*k'*)
using *a0 a1 index-filter-with-index-lt-length* **unfolding** $Complete_L$*-def* **by** (*auto simp*: *items-def*; *fastforce*)
have *3*: *red* $<$ *i*+*1*
using *a0 a1* **unfolding** $Complete_L$*-def* **by** *auto*

have *item e* = *inc-item* (*item* (*bs*!*k'*!*pre*)) *k*
using *a0 a1 0 2* $Complete_F.hyps(1,2,3)$ $Complete_F.prems(1)$ $\langle k' = item\text{-}origin\ x \rangle$ **unfolding** $Complete_L$*-def*
by (*auto simp*: *items-def*, *metis filter-with-index-nth nth-map*)
moreover have *is-complete* (*item* (*bs*!*k*!*red*))
using *a0 a1 0 2* $Complete_F.hyps(1,2,3)$ $Complete_F.prems(1)$ $\langle k' = item\text{-}origin\ x \rangle$ **unfolding** $Complete_L$*-def*
by (*auto simp*: *next-symbol-def items-def split*: *if-splits*)
moreover have *item-origin* (*item* (*bs*!*k*!*red*)) = *item-end* (*item* (*bs*!*k'*!*pre*))
using *a0 a1 0 2* $Complete_F.hyps(1,2,3)$ $Complete_F.prems(1)$ $\langle k' = item\text{-}origin\ x \rangle$ **unfolding** $Complete_L$*-def*
apply (*clarsimp simp*: *items-def*)
by (*metis dual-order.strict-trans index-filter-with-index-lt-length items-def le-neq-implies-less nth-map nth-mem wf-bins-kth-bin wf-earley-input-elim*)
moreover have ($\exists N.$ *next-symbol* (*item* (*bs* ! *k'* ! *pre*)) = *Some N* $\wedge$ *N* = *item-rule-head* (*item* (*bs* ! *k* ! *red*)))
using *a0 a1 0 2* $Complete_F.hyps(1,2,3)$ $Complete_F.prems(1)$ $\langle k' = item\text{-}origin\ x \rangle$ **unfolding** $Complete_L$*-def*
by (*auto simp*: *items-def*, *metis* (*mono-tags*, *lifting*) *filter-with-index-P filter-with-index-nth nth-map*)
ultimately have *4*: *completes k* (*item* (*bs*!*k'*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))
unfolding *completes-def* **by** *blast*
have *k'* $<$ *k pre* $<$ *length* (*bs*!*k'*) *red* $<$ *i*+*1 completes k* (*item* (*bs*!*k'*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))
using *0 1 2 3 4* **by** *simp-all*
}
hence $\forall\ e \in set\ (Complete_L\ k\ x\ bs\ i).\ \forall\ p\ ps\ k'\ pre\ red.\ pointer\ e = PreRed\ p\ ps \wedge (k', pre, red) \in set\ (p\#ps) \longrightarrow$
k' $<$ *k* $\wedge$ *pre* $<$ *length* (*bs*!*k'*) $\wedge$ *red* $<$ *i*+*1* $\wedge$ *completes k* (*item* (*bs*!*k'*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))
by *force*
hence *3*: $\forall\ e \in set\ (Complete_L\ k\ x\ bs\ i).$ *sound-prered-ptr bs k e*
using $Complete_F.hyps(1)$
by (*simp add*: *not-le length-items-eq sound-prered-ptr-def*) *fastforce*
have *4*: *distinct* (*items* ($Complete_L$ *k x bs i*))
using *distinct-Complete*$_L$ *x* $Complete_F.prems(1)$ *wf-earley-input-elim wf-bin-def*

wf-bin-items-def wf-bins-def wf-item-def
by (*metis order-le-less-trans*)
have *sound-ptrs* ω *?bs'* $\wedge$ *mono-red-ptr ?bs'*
using *sound-mono-ptrs-bin-upds*[*OF Complete*$_F$*.prems*(*2*) *Complete*$_F$*.prems*(*4*) *0*] *1 2 3 4 sound-prered-ptr-def*
Complete$_F$*.prems*(*1*) *bins-upd-def wf-earley-input-elim wf-bin-def wf-bins-def*
by (*smt* (*verit*, *ccfv-SIG*) *list.set-intros*(*1*))
moreover have (k, $\mathcal{G}$, ω, *?bs'*) $\in$ *wf-earley-input*
using *Complete*$_F$*.hyps Complete*$_F$*.prems*(*1*) *wf-earley-input-Complete*$_L$ **by** *blast*
ultimately have *sound-ptrs* ω (*Earley*$_L$*-bin'* k $\mathcal{G}$ ω *?bs'* (*i+1*)) $\wedge$ *mono-red-ptr* (*Earley*$_L$*-bin'* k $\mathcal{G}$ ω *?bs'* (*i+1*))
using *Complete*$_F$*.IH Complete*$_F$*.prems*(*4−6*) *sound* **by** *blast*
thus *?case*
using *Complete*$_F$*.hyps* **by** *simp*
next
case (*Scan*$_F$ k $\mathcal{G}$ ω *bs i x a*)
let *?bs'* = *bins-upd bs* (*k+1*) (*Scan*$_L$ k ω *a x i*)
have $x \in$ *set* (*items* (*bs* ! k))
using *Scan*$_F$*.hyps*(*1*,*2*) **by** *force*
hence $\forall x \in$ *set* (*items* (*Scan*$_L$ k ω *a x i*)). *sound-item* $\mathcal{G}$ ω x
using *sound-Scan*$_L$ *Scan*$_F$*.hyps*(*3*,*5*) *Scan*$_F$*.prems*(*1*,*2*,*3*) *wf-earley-input-elim wf-bins-impl-wf-items wf-bins-impl-wf-items* **by** *fast*
hence *sound*: $\forall x \in$ *bins ?bs'*. *sound-item* $\mathcal{G}$ ω x
using *Scan*$_F$*.hyps*(*5*) *Scan*$_F$*.prems*(*1*,*3*) *bins-bins-upd wf-earley-input-elim*
by (*metis UnE add-less-cancel-right*)
have *0*: *k+1* < *length bs*
using *Scan*$_F$*.hyps*(*5*) *Scan*$_F$*.prems*(*1*) *wf-earley-input-elim* **by** *force*
have *1*: $\forall e \in$ *set* (*Scan*$_L$ k ω *a x i*). *sound-null-ptr e*
unfolding *Scan*$_L$*-def sound-null-ptr-def* **by** *auto*
have *2*: $\forall e \in$ *set* (*Scan*$_L$ k ω *a x i*). *sound-pre-ptr* ω *bs* (*k+1*) *e*
using *Scan*$_F$*.hyps*(*1*,*2*,*3*) **unfolding** *sound-pre-ptr-def Scan*$_L$*-def scans-def items-def* **by** *auto*
have *3*: $\forall e \in$ *set* (*Scan*$_L$ k ω *a x i*). *sound-prered-ptr bs* (*k+1*) *e*
unfolding *Scan*$_L$*-def sound-prered-ptr-def* **by** *simp*
have *4*: *distinct* (*items* (*Scan*$_L$ k ω *a x i*))
using *distinct-Scan*$_L$ **by** *fast*
have *sound-ptrs* ω *?bs'* $\wedge$ *mono-red-ptr ?bs'*
using *sound-mono-ptrs-bin-upds*[*OF Scan*$_F$*.prems*(*2*) *Scan*$_F$*.prems*(*4*) *0*] *0 1 2 3 4 sound-prered-ptr-def*
Scan$_F$*.prems*(*1*) *bins-upd-def wf-earley-input-elim wf-bin-def wf-bins-def*
by (*smt* (*verit*, *ccfv-threshold*) *list.set-intros*(*1*))
moreover have (k, $\mathcal{G}$, ω, *?bs'*) $\in$ *wf-earley-input*
using *Scan*$_F$*.hyps Scan*$_F$*.prems*(*1*) *wf-earley-input-Scan*$_L$ **by** *metis*
ultimately have *sound-ptrs* ω (*Earley*$_L$*-bin'* k $\mathcal{G}$ ω *?bs'* (*i+1*)) $\wedge$ *mono-red-ptr* (*Earley*$_L$*-bin'* k $\mathcal{G}$ ω *?bs'* (*i+1*))
using *Scan*$_F$*.IH Scan*$_F$*.prems*(*4−6*) *sound* **by** *blast*
thus *?case*
using *Scan*$_F$*.hyps* **by** *simp*
next

case ($Predict_F$ k $\mathcal{G}$ ω bs i x a)
let $?bs' = bins\text{-}upd\ bs\ k\ (Predict_L\ k\ \mathcal{G}\ a)$
have $x \in set\ (items\ (bs\ !\ k))$
using $Predict_F.hyps(1,2)$ **by** *force*
hence $\forall x \in set\ (items(Predict_L\ k\ \mathcal{G}\ a)).\ sound\text{-}item\ \mathcal{G}\ \omega\ x$
using *sound-Predict*$_L$ $Predict_F.hyps(3)$ $Predict_F.prems$ *wf-earley-input-elim wf-bins-impl-wf-items* **by** *fast*
hence *sound*: $\forall x \in bins\ ?bs'.\ sound\text{-}item\ \mathcal{G}\ \omega\ x$
using $Predict_F.prems(1,3)$ *UnE bins-bins-upd wf-earley-input-elim* **by** *metis*
have *0*: $k < length\ bs$
using $Predict_F.prems(1)$ *wf-earley-input-elim* **by** *force*
have *1*: $\forall e \in set\ (Predict_L\ k\ \mathcal{G}\ a).\ sound\text{-}null\text{-}ptr\ e$
unfolding *sound-null-ptr-def* $Predict_L$*-def predicts-def* **by** (*auto simp*: *init-item-def*)
have *2*: $\forall e \in set\ (Predict_L\ k\ \mathcal{G}\ a).\ sound\text{-}pre\text{-}ptr\ \omega\ bs\ k\ e$
unfolding *sound-pre-ptr-def* $Predict_L$*-def* **by** *simp*
have *3*: $\forall e \in set\ (Predict_L\ k\ \mathcal{G}\ a).\ sound\text{-}prered\text{-}ptr\ bs\ k\ e$
unfolding *sound-prered-ptr-def* $Predict_L$*-def* **by** *simp*
have *4*: *distinct* (*items* ($Predict_L$ k $\mathcal{G}$ a))
using $Predict_F.prems(6)$ *distinct-*$Predict_L$ **by** *fast*
have *sound-ptrs* ω $?bs' \wedge$ *mono-red-ptr* $?bs'$
using *sound-mono-ptrs-bin-upds*[*OF* $Predict_F.prems(2)$ $Predict_F.prems(4)$ *0*] *0 1 2 3 4 sound-prered-ptr-def*
$Predict_F.prems(1)$ *bins-upd-def wf-earley-input-elim wf-bin-def wf-bins-def*
by (*smt* (*verit*, *ccfv-threshold*) *list.set-intros*(*1*))
moreover have (k, $\mathcal{G}$, ω, $?bs'$) $\in$ *wf-earley-input*
using $Predict_F.hyps$ $Predict_F.prems(1)$ *wf-earley-input-*$Predict_L$ **by** *metis*
ultimately have *sound-ptrs* ω ($Earley_L$*-bin'* k $\mathcal{G}$ ω $?bs'$ (i+*1*)) $\wedge$ *mono-red-ptr* ($Earley_L$*-bin'* k $\mathcal{G}$ ω $?bs'$ (i+*1*))
using $Predict_F.IH$ $Predict_F.prems(4-6)$ *sound* **by** *blast*
thus *?case*
using $Predict_F.hyps$ **by** *simp*
qed *simp-all*

lemma *sound-mono-ptrs-*$Earley_L$*-bin*:
assumes (k, $\mathcal{G}$, ω, bs) $\in$ *wf-earley-input*
assumes *sound-ptrs* ω bs $\forall x \in bins\ bs.\ sound\text{-}item\ \mathcal{G}\ \omega\ x$
assumes *mono-red-ptr bs*
assumes *nonempty-derives* $\mathcal{G}$ *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *sound-ptrs* ω ($Earley_L$*-bin* k $\mathcal{G}$ ω bs) $\wedge$ *mono-red-ptr* ($Earley_L$*-bin* k $\mathcal{G}$ ω bs)
using *assms sound-mono-ptrs-*$Earley_L$*-bin' Earley*$_L$*-bin-def* **by** *metis*

lemma *sound-ptrs-*$Init_L$:
sound-ptrs ω ($Init_L$ $\mathcal{G}$ ω)
unfolding *sound-ptrs-def sound-null-ptr-def sound-pre-ptr-def sound-prered-ptr-def predicts-def scans-def completes-def* $Init_L$*-def*
by (*auto simp*: *init-item-def less-Suc-eq-0-disj*)

lemma *mono-red-ptr-*$Init_L$:

mono-red-ptr (*Init*$_L$ $\mathcal{G}$ ω)
unfolding *mono-red-ptr-def Init*$_L$*-def*
by (*auto simp*: *init-item-def less-Suc-eq-0-disj*)

lemma *sound-mono-ptrs-Earley*$_L$*-bins*:
assumes $k \leq$ *length* ω *wf-*$\mathcal{G}$ $\mathcal{G}$ *nonempty-derives* $\mathcal{G}$ *wf-*$\mathcal{G}$ $\mathcal{G}$
shows *sound-ptrs* ω (*Earley*$_L$*-bins* k $\mathcal{G}$ ω) $\wedge$ *mono-red-ptr* (*Earley*$_L$*-bins* k $\mathcal{G}$ ω)
using *assms*
proof (*induction* k)
case *0*
have (*0*, $\mathcal{G}$, ω, (*Init*$_L$ $\mathcal{G}$ ω)) $\in$ *wf-earley-input*
using *assms*(*2*) *wf-earley-input-Init*$_L$ **by** *blast*
moreover have $\forall\, x \in$ *bins* (*Init*$_L$ $\mathcal{G}$ ω). *sound-item* $\mathcal{G}$ ω x
by (*metis Init*$_L$*-eq-Init*$_F$ *Init*$_F$*-sub-Earley sound-Earley subsetD wf-Earley*)
ultimately show *?case*
using *sound-mono-ptrs-Earley*$_L$*-bin sound-ptrs-Init*$_L$ *mono-red-ptr-Init*$_L$ *0.prems*(*2*,*3*)
by *fastforce*
next
case (*Suc* k)
have (*Suc* k, $\mathcal{G}$, ω, *Earley*$_L$*-bins* k $\mathcal{G}$ ω) $\in$ *wf-earley-input*
by (*simp add*: *Suc.prems*(*1*) *Suc-leD assms*(*2*) *wf-earley-input-intro*)
moreover have *sound-ptrs* ω (*Earley*$_L$*-bins* k $\mathcal{G}$ ω)
using *Suc* **by** *simp*
moreover have $\forall\, x \in$ *bins* (*Earley*$_L$*-bins* k $\mathcal{G}$ ω). *sound-item* $\mathcal{G}$ ω x
by (*meson Suc.prems*(*1*) *Suc-leD Earley*$_L$*-bins-sub-Earley*$_F$*-bins Earley*$_F$*-bins-sub-Earley assms*(*2*)
sound-Earley subsetD wf-bins-Earley$_L$*-bins wf-bins-impl-wf-items*)
ultimately show *?case*
using *Suc.prems*(*1*,*3*,*4*) *sound-mono-ptrs-Earley*$_L$*-bin Suc.IH* **by** *fastforce*
qed

lemma *sound-mono-ptrs-Earley*$_L$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *nonempty-derives* $\mathcal{G}$
shows *sound-ptrs* ω (*Earley*$_L$ $\mathcal{G}$ ω) $\wedge$ *mono-red-ptr* (*Earley*$_L$ $\mathcal{G}$ ω)
using *assms sound-mono-ptrs-Earley*$_L$*-bins Earley*$_L$*-def* **by** (*metis dual-order.refl*)

9.2 Common Definitions

datatype *'a tree* =
Leaf 'a
| *Branch 'a 'a tree list*

fun *yield-tree* :: *'a tree* $\Rightarrow$ *'a sentence* **where**
yield-tree (*Leaf a*) = [*a*]
| *yield-tree* (*Branch - ts*) = *concat* (*map yield-tree ts*)

fun *root-tree* :: *'a tree* $\Rightarrow$ *'a* **where**
root-tree (*Leaf a*) = *a*
| *root-tree* (*Branch N -*) = *N*

fun *wf-rule-tree* :: *'a cfg* ⇒ *'a tree* ⇒ *bool* **where**
  wf-rule-tree - (Leaf a) ⟷ *True*
| *wf-rule-tree G (Branch N ts)* ⟷ (
    (∃ *r* ∈ *set* (ℜ *G*). *N* = *rule-head r* ∧ *map root-tree ts* = *rule-body r*) ∧
    (∀ *t* ∈ *set ts*. *wf-rule-tree G t*))

fun *wf-item-tree* :: *'a cfg* ⇒ *'a item* ⇒ *'a tree* ⇒ *bool* **where**
  wf-item-tree G - (Leaf a) ⟷ *True*
| *wf-item-tree G x (Branch N ts)* ⟷ (
    N = *item-rule-head x* ∧ *map root-tree ts* = *take* (*item-dot x*) (*item-rule-body x*)
∧
    (∀ *t* ∈ *set ts*. *wf-rule-tree G t*))

definition *wf-yield-tree* :: *'a sentence* ⇒ *'a item* ⇒ *'a tree* ⇒ *bool* **where**
  wf-yield-tree ω x t ⟷ *yield-tree t* = *slice* (*item-origin x*) (*item-end x*) *ω*

datatype *'a forest* =
  FLeaf 'a
  | *FBranch 'a 'a forest list list*

fun *combinations* :: *'a list list* ⇒ *'a list list* **where**
  combinations [] = [[]]
| *combinations* (*xs*#*xss*) = [*x*#*cs* . *x* <− *xs*, *cs* <− *combinations xss*]

fun *trees* :: *'a forest* ⇒ *'a tree list* **where**
  trees (*FLeaf a*) = [*Leaf a*]
| *trees* (*FBranch N fss*) = (
    let tss = (*map* (λ*fs*. *concat* (*map* (λ*f*. *trees f*) *fs*)) *fss*) *in*
    map (λ*ts*. *Branch N ts*) (*combinations tss*)
)

lemma *list-comp-flatten*:
  [*f xs* . *xs* <− [*g xs ys* . *xs* <− *as*, *ys* <− *bs*]] = [*f* (*g xs ys*) . *xs* <− *as*, *ys*
<− *bs*]
  by (*induction as*) *auto*

lemma *list-comp-flatten-Cons*:
  [*x*#*xs* . *x* <− *as*, *xs* <− [*xs* @ *ys*. *xs* <− *bs*, *ys* <− *cs*]] = [*x*#*xs*@*ys*. *x* <−
as, *xs* <− *bs*, *ys* <− *cs*]
  by (*induction as*) (*auto simp*: *list-comp-flatten*)

lemma *list-comp-flatten-append*:
  [*xs*@*ys* . *xs* <− [*x*#*xs* . *x* <− *as*, *xs* <− *bs*], *ys* <− *cs*] = [*x*#*xs*@*ys* . *x* <−
as, *xs* <− *bs*, *ys* <− *cs*]
  by (*induction as*) (*auto simp*: *o-def*, *meson append-Cons map-eq-conv*)

lemma *combinations-append*:
  combinations (*xss* @ *yss*) = [*xs* @ *ys* . *xs* <− *combinations xss*, *ys* <− *combi-*

nations yss]
by (*induction xss*) (*auto simp*: *list-comp-flatten-Cons list-comp-flatten-append map-idI*)

lemma *trees-append*:
trees (*FBranch N* (*xss* @ *yss*)) = (
let xtss = (*map* (λ*xs*. *concat* (*map* (λ*f*. *trees f*) *xs*)) *xss*) *in*
let ytss = (*map* (λ*ys*. *concat* (*map* (λ*f*. *trees f*) *ys*)) *yss*) *in*
map (λ*ts*. *Branch N ts*) [*xs* @ *ys* . *xs* <− *combinations xtss*, *ys* <− *combinations ytss*])
using *combinations-append* **by** (*metis map-append trees.simps*(*2*))

lemma *trees-append-singleton*:
trees (*FBranch N* (*xss* @ [*ys*])) = (
let xtss = (*map* (λ*xs*. *concat* (*map* (λ*f*. *trees f*) *xs*)) *xss*) *in*
let ytss = [*concat* (*map trees ys*)] *in*
map (λ*ts*. *Branch N ts*) [*xs* @ *ys* . *xs* <− *combinations xtss*, *ys* <− *combinations ytss*])
by (*subst trees-append*, *simp*)

lemma *trees-append-single-singleton*:
trees (*FBranch N* (*xss* @ [[*y*]])) = (
let xtss = (*map* (λ*xs*. *concat* (*map* (λ*f*. *trees f*) *xs*)) *xss*) *in*
map (λ*ts*. *Branch N ts*) [*xs* @ *ys* . *xs* <− *combinations xtss*, *ys* <− [[*t*] . *t* <− *trees y*]])
by (*subst trees-append-singleton*, *auto*)

9.3 foldl lemmas

lemma *foldl-add-nth*:
k < *length xs* ⟹ *foldl* (+) *z* (*map length* (*take k xs*)) + *length* (*xs*!*k*) = *foldl* (+) *z* (*map length* (*take* (*k*+*1*) *xs*))
proof (*induction xs arbitrary*: *k z*)
case (*Cons x xs*)
then show *?case*
proof (*cases k* = *0*)
case *False*
thus *?thesis*
using *Cons* **by** (*auto simp add*: *take-Cons'*)
qed *simp*
qed *simp*

lemma *foldl-acc-mono*:
a ≤ *b* ⟹ *foldl* (+) *a xs* ≤ *foldl* (+) *b xs* **for** *a* :: *nat*
by (*induction xs arbitrary*: *a b*) *auto*

lemma *foldl-ge-z-nth*:
j < *length xs* ⟹ *z* + *length* (*xs*!*j*) ≤ *foldl* (+) *z* (*map length* (*take* (*j*+*1*) *xs*))
proof (*induction xs arbitrary*: *j z*)

```
  case (Cons x xs)
  show ?case
  proof (cases j = 0)
    case False
    have z + length ((x # xs) ! j) = z + length (xs!(j−1))
      using False by simp
    also have ... ≤ foldl (+) z (map length (take (j−1+1) xs))
     using Cons False by (metis add-diff-inverse-nat length-Cons less-one nat-add-left-cancel-less
plus-1-eq-Suc)
    also have ... = foldl (+) z (map length (take j xs))
      using False by simp
    also have ... ≤ foldl (+) (z + length x) (map length (take j xs))
      using foldl-acc-mono by force
    also have ... = foldl (+) z (map length (take (j+1) (x#xs)))
      by simp
    finally show ?thesis
      by blast
  qed simp
qed simp

lemma foldl-add-nth-ge:
  i ≤ j ⟹ j < length xs ⟹ foldl (+) z (map length (take i xs)) + length (xs!j)
≤ foldl (+) z (map length (take (j+1) xs))
proof (induction xs arbitrary: i j z)
  case (Cons x xs)
  show ?case
  proof (cases i = 0)
    case True
    have foldl (+) z (map length (take i (x # xs))) + length ((x # xs) ! j) = z +
length ((x # xs) ! j)
      using True by simp
    also have ... ≤ foldl (+) z (map length (take (j+1) (x#xs)))
      using foldl-ge-z-nth Cons.prems(2) by blast
    finally show ?thesis
      by blast
  next
    case False
    have i−1 ≤ j−1
      by (simp add: Cons.prems(1) diff-le-mono)
    have j−1 < length xs
      using Cons.prems(1,2) False by fastforce
    have foldl (+) z (map length (take i (x # xs))) + length ((x # xs) ! j) =
      foldl (+) (z + length x) (map length (take (i−1) xs)) + length ((x#xs)!j)
      using False by (simp add: take-Cons')
   also have ... = foldl (+) (z + length x) (map length (take (i−1) xs)) + length
(xs!(j−1))
      using Cons.prems(1) False by auto
    also have ... ≤ foldl (+) (z + length x) (map length (take (j−1+1) xs))
      using Cons.IH ‹i − 1 ≤ j − 1› ‹j − 1 < length xs› by blast
```

```
    also have ... = foldl (+) (z + length x) (map length (take j xs))
      using Cons.prems(1) False by fastforce
    also have ... = foldl (+) z (map length (take (j+1) (x#xs)))
      by fastforce
    finally show ?thesis
      by blast
  qed
qed simp

lemma foldl-ge-acc:
  foldl (+) z (map length xs) ≥ z
  by (induction xs arbitrary: z) (auto elim: add-leE)

lemma foldl-take-mono:
  i ≤ j ⟹ foldl (+) z (map length (take i xs)) ≤ foldl (+) z (map length (take j
xs))
proof (induction xs arbitrary: z i j)
  case (Cons x xs)
  show ?case
  proof (cases i = 0)
    case True
    have foldl (+) z (map length (take i (x # xs))) = z
      using True by simp
    also have ... ≤ foldl (+) z (map length (take j (x # xs)))
      by (simp add: foldl-ge-acc)
    ultimately show ?thesis
      by simp
  next
    case False
    then show ?thesis
      using Cons by (simp add: take-Cons')
  qed
qed simp
```

9.4 Parse tree

```
partial-function (option) build-tree' :: 'a bins ⇒ 'a sentence ⇒ nat ⇒ nat ⇒ 'a
tree option where
  build-tree' bs ω k i = (
    let e = bs!k!i in (
    case pointer e of
      Null ⇒ Some (Branch (item-rule-head (item e)) []) — start building sub-tree
    | Pre pre ⇒ ( — add sub-tree starting from terminal
        do {
          t ← build-tree' bs ω (k−1) pre;
          case t of
            Branch N ts ⇒ Some (Branch N (ts @ [Leaf (ω!(k−1))]))
          | - ⇒ undefined — impossible case
        })
```

```
    | PreRed (k', pre, red) - ⇒ ( — add sub-tree starting from non-terminal
        do {
          t ← build-tree' bs ω k' pre;
          case t of
            Branch N ts ⇒
              do {
                t ← build-tree' bs ω k red;
                Some (Branch N (ts @ [t]))
              }
          | - ⇒ undefined — impossible case
        })
  ))

declare build-tree'.simps [code]

definition build-tree :: 'a cfg ⇒ 'a sentence ⇒ 'a bins ⇒ 'a tree option where
  build-tree G ω bs = (
    let k = length bs − 1 in (
    case filter-with-index (λx. is-finished G ω x) (items (bs!k)) of
      [] ⇒ None
    | (-, i)#- ⇒ build-tree' bs ω k i
  ))

lemma build-tree'-simps[simp]:
    e = bs!k!i ⟹ pointer e = Null ⟹ build-tree' bs ω k i = Some (Branch
(item-rule-head (item e)) [])
  e = bs!k!i ⟹ pointer e = Pre pre ⟹ build-tree' bs ω (k−1) pre = None ⟹
    build-tree' bs ω k i = None
  e = bs!k!i ⟹ pointer e = Pre pre ⟹ build-tree' bs ω (k−1) pre = Some (Branch
N ts) ⟹
    build-tree' bs ω k i = Some (Branch N (ts @ [Leaf (ω!(k−1))]))
  e = bs!k!i ⟹ pointer e = Pre pre ⟹ build-tree' bs ω (k−1) pre = Some (Leaf
a) ⟹
    build-tree' bs ω k i = undefined
  e = bs!k!i ⟹ pointer e = PreRed (k', pre, red) reds ⟹ build-tree' bs ω k' pre
= None ⟹
    build-tree' bs ω k i = None
  e = bs!k!i ⟹ pointer e = PreRed (k', pre, red) reds ⟹ build-tree' bs ω k' pre
= Some (Branch N ts) ⟹
    build-tree' bs ω k red = None ⟹ build-tree' bs ω k i = None
  e = bs!k!i ⟹ pointer e = PreRed (k', pre, red) reds ⟹ build-tree' bs ω k' pre
= Some (Leaf a) ⟹
    build-tree' bs ω k i = undefined
  e = bs!k!i ⟹ pointer e = PreRed (k', pre, red) reds ⟹ build-tree' bs ω k' pre
= Some (Branch N ts) ⟹
    build-tree' bs ω k red = Some t ⟹
    build-tree' bs ω k i = Some (Branch N (ts @ [t]))
  by (subst build-tree'.simps, simp)+
```

definition *wf-tree-input* :: (*'a bins* × *'a sentence* × *nat* × *nat*) *set* **where**
wf-tree-input = {
(*bs*, *ω*, *k*, *i*) | *bs ω k i*.
sound-ptrs ω bs ∧
mono-red-ptr bs ∧
k < *length bs* ∧
i < *length* (*bs*!*k*)
}

fun *build-tree'-measure* :: (*'a bins* × *'a sentence* × *nat* × *nat*) ⇒ *nat* **where**
build-tree'-measure (*bs*, *ω*, *k*, *i*) = *foldl* (+) *0* (*map length* (*take k bs*)) + *i*

lemma *wf-tree-input-pre*:
assumes (*bs*, *ω*, *k*, *i*) ∈ *wf-tree-input*
assumes *e* = *bs*!*k*!*i pointer e* = *Pre pre*
shows (*bs*, *ω*, (*k*−*1*), *pre*) ∈ *wf-tree-input*
using *assms* **unfolding** *wf-tree-input-def*
using *less-imp-diff-less nth-mem* **by** (*fastforce simp*: *sound-ptrs-def sound-pre-ptr-def*)

lemma *wf-tree-input-prered-pre*:
assumes (*bs*, *ω*, *k*, *i*) ∈ *wf-tree-input*
assumes *e* = *bs*!*k*!*i pointer e* = *PreRed* (*k'*, *pre*, *red*) *ps*
shows (*bs*, *ω*, *k'*, *pre*) ∈ *wf-tree-input*
using *assms* **unfolding** *wf-tree-input-def*
apply (*auto simp*: *sound-ptrs-def sound-prered-ptr-def*)
apply *metis*+
apply (*metis dual-order.strict-trans nth-mem*)
by (*metis nth-mem*)

lemma *wf-tree-input-prered-red*:
assumes (*bs*, *ω*, *k*, *i*) ∈ *wf-tree-input*
assumes *e* = *bs*!*k*!*i pointer e* = *PreRed* (*k'*, *pre*, *red*) *ps*
shows (*bs*, *ω*, *k*, *red*) ∈ *wf-tree-input*
using *assms* **unfolding** *wf-tree-input-def*
apply (*auto simp add*: *sound-ptrs-def sound-prered-ptr-def*)
apply (*metis nth-mem*)+
done

lemma *build-tree'-induct*:
assumes (*bs*, *ω*, *k*, *i*) ∈ *wf-tree-input*
assumes ⋀*bs ω k i*.
(⋀*e pre*. *e* = *bs*!*k*!*i* ⟹ *pointer e* = *Pre pre* ⟹ *P bs ω* (*k*−*1*) *pre*) ⟹
(⋀*e k' pre red ps*. *e* = *bs*!*k*!*i* ⟹ *pointer e* = *PreRed* (*k'*, *pre*, *red*) *ps* ⟹ *P bs ω k' pre*) ⟹
(⋀*e k' pre red ps*. *e* = *bs*!*k*!*i* ⟹ *pointer e* = *PreRed* (*k'*, *pre*, *red*) *ps* ⟹ *P bs ω k red*) ⟹
P bs ω k i
shows *P bs ω k i*
using *assms*(*1*)

```
proof (induction n≡build-tree′-measure (bs, ω, k, i) arbitrary: k i rule: nat-less-induct)
  case 1
  obtain e where entry: e = bs!k!i
    by simp
  consider (Null) pointer e = Null
    | (Pre) ∃ pre. pointer e = Pre pre
    | (PreRed) ∃ k′ pre red reds. pointer e = PreRed (k′, pre, red) reds
    by (metis pointer.exhaust surj-pair)
  thus ?case
  proof cases
    case Null
    thus ?thesis
      using assms(2) entry by fastforce
  next
    case Pre
    then obtain pre where pre: pointer e = Pre pre
      by blast
    define n where n: n = build-tree′-measure (bs, ω, (k−1), pre)
    have 0 < k pre < length (bs!(k−1))
     using 1(2) entry pre unfolding wf-tree-input-def sound-ptrs-def sound-pre-ptr-def
      by (smt (verit) mem-Collect-eq nth-mem prod.inject)+
    have k < length bs
      using 1(2) unfolding wf-tree-input-def by blast+
   have foldl (+) 0 (map length (take k bs)) + i − (foldl (+) 0 (map length (take
(k−1) bs)) + pre) =
       foldl (+) 0 (map length (take (k−1) bs)) + length (bs!(k−1)) + i − (foldl
(+) 0 (map length (take (k−1) bs)) + pre)
      using foldl-add-nth[of ‹k−1› bs 0] by (simp add: ‹0 < k› ‹k < length bs›
less-imp-diff-less)
    also have ... = length (bs!(k−1)) + i − pre
      by simp
    also have ... > 0
      using ‹pre < length (bs!(k−1))› by auto
     finally have build-tree′-measure (bs, ω, k, i) − build-tree′-measure (bs, ω,
(k−1), pre) > 0
      by simp
    hence P bs ω (k−1) pre
      using 1 n wf-tree-input-pre entry pre zero-less-diff by blast
    thus ?thesis
      using assms(2) entry pre pointer.distinct(5) pointer.inject(1) by presburger
  next
    case PreRed
    then obtain k′ pre red ps where prered: pointer e = PreRed (k′, pre, red) ps
      by blast
    have k′ < k pre < length (bs!k′)
    using 1(2) entry prered unfolding wf-tree-input-def sound-ptrs-def sound-prered-ptr-def
      apply simp-all
      apply (metis nth-mem)+
      done
```

```
    have red < i
      using 1(2) entry prered unfolding wf-tree-input-def mono-red-ptr-def by
blast
    have k < length bs i < length (bs!k)
      using 1(2) unfolding wf-tree-input-def by blast+
    define n-pre where n-pre: n-pre = build-tree'-measure (bs, ω, k', pre)
    have 0 < length (bs!k') + i − pre
      by (simp add: ‹pre < length (bs!k')› add.commute trans-less-add2)
     also have ... = foldl (+) 0 (map length (take k' bs)) + length (bs!k') + i −
(foldl (+) 0 (map length (take k' bs)) + pre)
      by simp
    also have ... ≤ foldl (+) 0 (map length (take (k'+1) bs)) + i − (foldl (+) 0
(map length (take k' bs)) + pre)
      using foldl-add-nth-ge[of k' k' bs 0] ‹k < length bs› ‹k' < k› by simp
    also have ... ≤ foldl (+) 0 (map length (take k bs)) + i − (foldl (+) 0 (map
length (take k' bs)) + pre)
      using foldl-take-mono by (metis Suc-eq-plus1 Suc-leI ‹k' < k› add.commute
add-le-cancel-left diff-le-mono)
    finally have build-tree'-measure (bs, ω, k, i) − build-tree'-measure (bs, ω, k',
pre) > 0
      by simp
    hence x: P bs ω k' pre
     using 1(1) zero-less-diff by (metis 1.prems entry prered wf-tree-input-prered-pre)
    define n-red where n-red: n-red = build-tree'-measure (bs, ω, k, red)
    have build-tree'-measure (bs, ω, k, i) − build-tree'-measure (bs, ω, k, red) > 0
      using ‹red < i› by simp
    hence y: P bs ω k red
      using 1.hyps 1.prems entry prered wf-tree-input-prered-red zero-less-diff by
blast
    show ?thesis
      using assms(2) x y entry prered
     by (smt (verit, best) Pair-inject filter-cong pointer.distinct(5) pointer.inject(2))
  qed
qed

lemma build-tree'-termination:
  assumes (bs, ω, k, i) ∈ wf-tree-input
  shows ∃ N ts. build-tree' bs ω k i = Some (Branch N ts)
proof −
  have ∃ N ts. build-tree' bs ω k i = Some (Branch N ts)
    apply (induction rule: build-tree'-induct[OF assms(1)])
    subgoal premises IH for bs ω k i
    proof −
      define e where entry: e = bs!k!i
      consider (Null) pointer e = Null
        | (Pre) ∃ pre. pointer e = Pre pre
        | (PreRed) ∃ k' pre red ps. pointer e = PreRed (k', pre, red) ps
        by (metis pointer.exhaust surj-pair)
      thus ?thesis
```

```
    proof cases
      case Null
      thus ?thesis
        using build-tree'-simps(1) entry by simp
    next
      case Pre
      then obtain pre where pre: pointer e = Pre pre
        by blast
      obtain N ts where Nts: build-tree' bs ω (k−1) pre = Some (Branch N ts)
        using IH(1) entry pre by blast
      have build-tree' bs ω k i = Some (Branch N (ts @ [Leaf (ω!(k−1))]))
        using build-tree'-simps(3) entry pre Nts by simp
      thus ?thesis
        by simp
    next
      case PreRed
      then obtain k' pre red ps where prered: pointer e = PreRed (k', pre, red)
ps
        by blast
     then obtain N ts where Nts: build-tree' bs ω k' pre = Some (Branch N ts)
        using IH(2) entry prered by blast
      obtain t-red where t-red: build-tree' bs ω k red = Some t-red
        using IH(3) entry prered Nts by (metis option.exhaust)
      have build-tree' bs ω k i = Some (Branch N (ts @ [t-red]))
        using build-tree'-simps(8) entry prered Nts t-red by auto
      thus ?thesis
        by blast
    qed
  qed
  done
 thus ?thesis
  by blast
qed

lemma wf-item-tree-build-tree':
 assumes (bs, ω, k, i) ∈ wf-tree-input
 assumes wf-bins G ω bs
 assumes k < length bs i < length (bs!k)
 assumes build-tree' bs ω k i = Some t
 shows wf-item-tree G (item (bs!k!i)) t
proof −
 have wf-item-tree G (item (bs!k!i)) t
  using assms
  apply (induction arbitrary: t rule: build-tree'-induct[OF assms(1)])
  subgoal premises prems for bs ω k i t
  proof −
    define e where entry: e = bs!k!i
    consider (Null) pointer e = Null
      | (Pre) ∃ pre. pointer e = Pre pre
```

```
    | (PreRed) ∃ k' pre red ps. pointer e = PreRed (k', pre, red) ps
     by (metis pointer.exhaust surj-pair)
   thus ?thesis
   proof cases
    case Null
    hence build-tree' bs ω k i = Some (Branch (item-rule-head (item e)) [])
     using entry by simp
    have simp: t = Branch (item-rule-head (item e)) []
     using build-tree'-simps(1) Null prems(8) entry by simp
    have sound-ptrs ω bs
     using prems(4) unfolding wf-tree-input-def by blast
    hence predicts (item e)
   using Null prems(6,7) nth-mem entry unfolding sound-ptrs-def sound-null-ptr-def
by blast
    hence item-dot (item e) = 0
     unfolding predicts-def by blast
    thus ?thesis
     using simp entry by simp
   next
    case Pre
    then obtain pre where pre: pointer e = Pre pre
     by blast
    obtain N ts where Nts: build-tree' bs ω (k−1) pre = Some (Branch N ts)
     using build-tree'-termination entry pre prems(4) wf-tree-input-pre by blast
    have simp: build-tree' bs ω k i = Some (Branch N (ts @ [Leaf (ω!(k−1))]))
     using build-tree'-simps(3) entry pre Nts by simp
    have sound-ptrs ω bs
     using prems(4) unfolding wf-tree-input-def by blast
    hence pre < length (bs!(k−1))
      using entry pre prems(6,7) unfolding sound-ptrs-def sound-pre-ptr-def
by (metis nth-mem)
    moreover have k−1 < length bs
     by (simp add: prems(6) less-imp-diff-less)
    ultimately have IH: wf-item-tree G (item (bs!(k−1)!pre)) (Branch N ts)
     using prems(1,2,4,5) entry pre Nts by (meson wf-tree-input-pre)
    have scans: scans ω k (item (bs!(k−1)!pre)) (item e)
     using entry pre prems(6−7) ‹sound-ptrs ω bs› unfolding sound-ptrs-def
sound-pre-ptr-def by simp
    hence ∗:
     item-rule-head (item (bs!(k−1)!pre)) = item-rule-head (item e)
     item-rule-body (item (bs!(k−1)!pre)) = item-rule-body (item e)
     item-dot (item (bs!(k−1)!pre)) + 1 = item-dot (item e)
     next-symbol (item (bs!(k−1)!pre)) = Some (ω!(k−1))
      unfolding scans-def inc-item-def by (simp-all add: item-rule-head-def
item-rule-body-def)
    have map root-tree (ts @ [Leaf (ω!(k−1))]) = map root-tree ts @ [ω!(k−1)]
     by simp
    also have ... = take (item-dot (item (bs!(k−1)!pre))) (item-rule-body (item
(bs!(k−1)!pre))) @ [ω!(k−1)]
```

using *IH* **by** *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!(*k*−1)!*pre*))) (*item-rule-body* (*item* *e*)) @ [*ω*!(*k*−1)]
using ∗(2) **by** *simp*
also have ... = *take* (*item-dot* (*item* *e*)) (*item-rule-body* (*item* *e*))
using ∗(2−4) **by** (*auto simp*: *next-symbol-def is-complete-def split*: *if-splits*; *metis leI take-Suc-conv-app-nth*)
finally have *map root-tree* (*ts* @ [*Leaf* (*ω*!(*k*−1))]) = *take* (*item-dot* (*item* *e*)) (*item-rule-body* (*item* *e*)) **.**
hence *wf-item-tree* *G* (*item* *e*) (*Branch* *N* (*ts* @ [*Leaf* (*ω*!(*k*−1))]))
using *IH* ∗(1) **by** *simp*
thus *?thesis*
using *entry prems*(8) *simp* **by** *auto*
next
case *PreRed*
then obtain *k′ pre red ps* **where** *prered*: *pointer* *e* = *PreRed* (*k′*, *pre*, *red*) *ps*
by *blast*
obtain *N ts* **where** *Nts*: *build-tree′ bs ω k′ pre* = *Some* (*Branch* *N ts*)
using *build-tree′-termination entry prems*(4) *prered wf-tree-input-prered-pre* **by** *blast*
obtain *N-red ts-red* **where** *Nts-red*: *build-tree′ bs ω k red* = *Some* (*Branch* *N-red ts-red*)
using *build-tree′-termination entry prems*(4) *prered wf-tree-input-prered-red* **by** *blast*
have *simp*: *build-tree′ bs ω k i* = *Some* (*Branch* *N* (*ts* @ [*Branch* *N-red ts-red*]))
using *build-tree′-simps*(8) *entry prered Nts Nts-red* **by** *auto*
have *sound-ptrs ω bs*
using *prems*(4) *wf-tree-input-def* **by** *fastforce*
have *bounds*: *k′* < *k pre* < *length* (*bs*!*k′*) *red* < *length* (*bs*!*k*)
using *prered entry prems*(6,7) ‹*sound-ptrs ω bs*›
unfolding *sound-prered-ptr-def sound-ptrs-def* **by** *fastforce*+
have *completes*: *completes k* (*item* (*bs*!*k′*!*pre*)) (*item* *e*) (*item* (*bs*!*k*!*red*))
using *prered entry prems*(6,7) ‹*sound-ptrs ω bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** *fastforce*
have ∗:
item-rule-head (*item* (*bs*!*k′*!*pre*)) = *item-rule-head* (*item* *e*)
item-rule-body (*item* (*bs*!*k′*!*pre*)) = *item-rule-body* (*item* *e*)
item-dot (*item* (*bs*!*k′*!*pre*)) + 1 = *item-dot* (*item* *e*)
next-symbol (*item* (*bs*!*k′*!*pre*)) = *Some* (*item-rule-head* (*item* (*bs*!*k*!*red*)))
is-complete (*item* (*bs*!*k*!*red*))
using *completes* **unfolding** *completes-def inc-item-def*
by (*auto simp*: *item-rule-head-def item-rule-body-def is-complete-def*)
have (*bs*, *ω*, *k′*, *pre*) ∈ *wf-tree-input*
using *wf-tree-input-prered-pre*[*OF prems*(4) *entry prered*] **by** *blast*
hence *IH-pre*: *wf-item-tree* *G* (*item* (*bs*!*k′*!*pre*)) (*Branch* *N ts*)
using *prems*(2)[*OF entry prered* - *prems*(5)] *Nts bounds*(1,2) *order-less-trans prems*(6) **by** *blast*

have $(bs, \omega, k, red) \in$ *wf-tree-input*
using *wf-tree-input-prered-red*[*OF prems*(*4*) *entry prered*] **by** *blast*
hence *IH-r*: *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*red*)) (*Branch N-red ts-red*)
using *bounds*(*3*) *entry prems*(*3*,*5*,*6*) *prered Nts-red* **by** *blast*
have *map root-tree* (*ts* @ [*Branch N-red ts-red*]) = *map root-tree ts* @ [*root-tree* (*Branch N-red ts*)]
by *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!*k*'!*pre*))) (*item-rule-body* (*item* (*bs*!*k*'!*pre*))) @ [*root-tree* (*Branch N-red ts*)]
using *IH-pre* **by** *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!*k*'!*pre*))) (*item-rule-body* (*item* (*bs*!*k*'!*pre*))) @ [*item-rule-head* (*item* (*bs*!*k*!*red*))]
using *IH-r* **by** *simp*
also have ... = *take* (*item-dot* (*item e*)) (*item-rule-body* (*item e*))
using ∗ **by** (*auto simp*: *next-symbol-def is-complete-def split*: *if-splits*; *metis leI take-Suc-conv-app-nth*)
finally have *roots*: *map root-tree* (*ts* @ [*Branch N-red ts*]) = *take* (*item-dot* (*item e*)) (*item-rule-body* (*item e*)) **by** *simp*
have *wf-item* $\mathcal{G}$ ω (*item* (*bs*!*k*!*red*))
using *prems*(*5*,*6*) *bounds*(*3*) **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def* **by** (*auto simp*: *items-def*)
moreover have *N-red* = *item-rule-head* (*item* (*bs*!*k*!*red*))
using *IH-r* **by** *fastforce*
moreover have *map root-tree ts-red* = *item-rule-body* (*item* (*bs*!*k*!*red*))
using *IH-r* ∗(*5*) **by** (*auto simp*: *is-complete-def*)
ultimately have ∃ *r* ∈ *set* (ℜ $\mathcal{G}$). *N-red* = *rule-head r* ∧ *map root-tree ts-red* = *rule-body r*
unfolding *wf-item-def item-rule-body-def item-rule-head-def* **by** *blast*
hence *wf-rule-tree* $\mathcal{G}$ (*Branch N-red ts-red*)
using *IH-r* **by** *simp*
hence *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) (*Branch N* (*ts* @ [*Branch N-red ts-red*]))
using ∗(*1*) *roots IH-pre entry* **by** *simp*
thus *?thesis*
using *Nts-red prems*(*8*) *simp* **by** *auto*
qed
qed
done
thus *?thesis*
using *assms*(*2*) **by** *blast*
qed

lemma *wf-yield-tree-build-tree'*:
assumes $(bs, \omega, k, i) \in$ *wf-tree-input*
assumes *wf-bins* $\mathcal{G}$ ω *bs*
assumes *k* < *length bs i* < *length* (*bs*!*k*) *k* ≤ *length* ω
assumes *build-tree'* *bs* ω *k i* = *Some t*
shows *wf-yield-tree* ω (*item* (*bs*!*k*!*i*)) *t*
proof −
have *wf-yield-tree* ω (*item* (*bs*!*k*!*i*)) *t*

```
    using assms
    apply (induction arbitrary: t rule: build-tree'-induct[OF assms(1)])
    subgoal premises prems for bs ω k i t
    proof −
      define e where entry: e = bs!k!i
      consider (Null) pointer e = Null
        | (Pre) ∃ pre. pointer e = Pre pre
        | (PreRed) ∃ k' pre red reds. pointer e = PreRed (k', pre, red) reds
        by (metis pointer.exhaust surj-pair)
      thus ?thesis
      proof cases
        case Null
        hence build-tree' bs ω k i = Some (Branch (item-rule-head (item e)) [])
          using entry by simp
        have simp: t = Branch (item-rule-head (item e)) []
          using build-tree'-simps(1) Null prems(9) entry by simp
        have sound-ptrs ω bs
          using prems(4) unfolding wf-tree-input-def by blast
        hence predicts (item e)
       using Null prems(6,7) nth-mem entry unfolding sound-ptrs-def sound-null-ptr-def
by blast
        thus ?thesis
          unfolding wf-yield-tree-def predicts-def using simp entry by (auto simp:
slice-empty)
      next
        case Pre
        then obtain pre where pre: pointer e = Pre pre
          by blast
        obtain N ts where Nts: build-tree' bs ω (k−1) pre = Some (Branch N ts)
         using build-tree'-termination entry pre prems(4) wf-tree-input-pre by blast
        have simp: build-tree' bs ω k i = Some (Branch N (ts @ [Leaf (ω!(k−1))]))
          using build-tree'-simps(3) entry pre Nts by simp
        have sound-ptrs ω bs
          using prems(4) unfolding wf-tree-input-def by blast
        hence bounds: k > 0 pre < length (bs!(k−1))
          using entry pre prems(6,7) unfolding sound-ptrs-def sound-pre-ptr-def
by (metis nth-mem)+
        moreover have k−1 < length bs
          by (simp add: prems(6) less-imp-diff-less)
        ultimately have IH: wf-yield-tree ω (item (bs!(k−1)!pre)) (Branch N ts)
          using prems(1) entry pre Nts wf-tree-input-pre prems(4,5,8) by fastforce
        have scans: scans ω k (item (bs!(k−1)!pre)) (item e)
          using entry pre prems(6−7) ‹sound-ptrs ω bs› unfolding sound-ptrs-def
sound-pre-ptr-def by simp
        have wf:
          item-origin (item (bs!(k−1)!pre)) ≤ item-end (item (bs!(k−1)!pre))
          item-end (item (bs!(k−1)!pre)) = k−1
          item-end (item e) = k
              using entry prems(5,6,7) bounds unfolding wf-bins-def wf-bin-def
```

wf-bin-items-def items-def wf-item-def
by (*auto*, *meson less-imp-diff-less nth-mem*)
have *yield-tree* (*Branch N* (*ts* @ [*Leaf* (ω!(*k*−*1*))])) = *concat* (*map yield-tree* (*ts* @ [*Leaf* (ω!(*k*−*1*))]))
by *simp*
also have ... = *concat* (*map yield-tree ts*) @ [ω!(*k*−*1*)]
by *simp*
also have ... = *slice* (*item-origin* (*item* (*bs*!(*k*−*1*)!*pre*))) (*item-end* (*item* (*bs*!(*k*−*1*)!*pre*))) ω @ [ω!(*k*−*1*)]
using *IH* **by** (*simp add*: *wf-yield-tree-def*)
also have ... = *slice* (*item-origin* (*item* (*bs*!(*k*−*1*)!*pre*))) (*item-end* (*item* (*bs*!(*k*−*1*)!*pre*)) + *1*) ω
using *slice-append-nth wf* ‹*k* > *0*› *prems*(*8*)
by (*metis diff-less le-eq-less-or-eq less-imp-diff-less less-numeral-extra*(*1*))
also have ... = *slice* (*item-origin* (*item e*)) (*item-end* (*item* (*bs*!(*k*−*1*)!*pre*)) + *1*) ω
using *scans* **unfolding** *scans-def inc-item-def* **by** *simp*
also have ... = *slice* (*item-origin* (*item e*)) *k* ω
using *scans wf* **unfolding** *scans-def* **by** (*metis Suc-diff-1 Suc-eq-plus1 bounds*(*1*))
also have ... = *slice* (*item-origin* (*item e*)) (*item-end* (*item e*)) ω
using *wf* **by** *auto*
finally show *?thesis*
using *wf-yield-tree-def entry prems*(*9*) *simp* **by** *force*
next
case *PreRed*
then obtain *k′ pre red ps* **where** *prered*: *pointer e* = *PreRed* (*k′*, *pre*, *red*) *ps*
by *blast*
obtain *N ts* **where** *Nts*: *build-tree′ bs* ω *k′ pre* = *Some* (*Branch N ts*)
using *build-tree′-termination entry prems*(*4*) *prered wf-tree-input-prered-pre* **by** *blast*
obtain *N-red ts-red* **where** *Nts-red*: *build-tree′ bs* ω *k red* = *Some* (*Branch N-red ts-red*)
using *build-tree′-termination entry prems*(*4*) *prered wf-tree-input-prered-red* **by** *blast*
have *simp*: *build-tree′ bs* ω *k i* = *Some* (*Branch N* (*ts* @ [*Branch N-red ts-red*]))
using *build-tree′-simps*(*8*) *entry prered Nts Nts-red* **by** *auto*
have *sound-ptrs* ω *bs*
using *prems*(*4*) *wf-tree-input-def* **by** *fastforce*
have *bounds*: *k′* < *k pre* < *length* (*bs*!*k′*) *red* < *length* (*bs*!*k*)
using *prered entry prems*(*6*,*7*) ‹*sound-ptrs* ω *bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** *fastforce*+
have *completes*: *completes k* (*item* (*bs*!*k*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))
using *prered entry prems*(*6*,*7*) ‹*sound-ptrs* ω *bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** *fastforce*
have (*bs*, ω, *k′*, *pre*) ∈ *wf-tree-input*
using *wf-tree-input-prered-pre*[*OF prems*(*4*) *entry prered*] **by** *blast*

```
      hence IH-pre: wf-yield-tree ω (item (bs!k'!pre)) (Branch N ts)
       using prems(2)[OF entry prered - prems(5)] Nts bounds(1,2) prems(6−8)
        by (meson dual-order.strict-trans1 nat-less-le)
      have (bs, ω, k, red) ∈ wf-tree-input
        using wf-tree-input-prered-red[OF prems(4) entry prered] by blast
      hence IH-r: wf-yield-tree ω (item (bs!k!red)) (Branch N-red ts-red)
        using bounds(3) entry prems(3,5,6,8) prered Nts-red by blast
      have wf1:
        item-origin (item (bs!k'!pre)) ≤ item-end (item (bs!k'!pre))
        item-origin (item (bs!k!red)) ≤ item-end (item (bs!k!red))
      using prems(5−7) bounds unfolding wf-bins-def wf-bin-def wf-bin-items-def
items-def wf-item-def
        by (metis length-map nth-map nth-mem order-less-trans)+
      have wf2:
        item-end (item (bs!k!red)) = k
        item-end (item (bs!k!i)) = k
      using prems(5−7) bounds unfolding wf-bins-def wf-bin-def wf-bin-items-def
items-def by simp-all
       have yield-tree (Branch N (ts @ [Branch N-red ts-red])) = concat (map
yield-tree (ts @ [Branch N-red ts-red]))
        by (simp add: Nts-red)
      also have ... = concat (map yield-tree ts) @ yield-tree (Branch N-red ts-red)
        by simp
         also have ... = slice (item-origin (item (bs!k'!pre))) (item-end (item
(bs!k'!pre))) ω @
        slice (item-origin (item (bs!k!red))) (item-end (item (bs!k!red))) ω
        using IH-pre IH-r by (simp add: wf-yield-tree-def)
         also have ... = slice (item-origin (item (bs!k'!pre))) (item-end (item
(bs!k!red))) ω
       using slice-concat wf1 completes-def completes by (metis (no-types, lifting))
      also have ... = slice (item-origin (item e)) (item-end (item (bs!k!red))) ω
        using completes unfolding completes-def inc-item-def by simp
      also have ... = slice (item-origin (item e)) (item-end (item e)) ω
        using wf2 entry by presburger
      finally show ?thesis
        using wf-yield-tree-def entry prems(9) simp by force
    qed
   qed
   done
  thus ?thesis
   using assms(2) by blast
qed

theorem wf-rule-root-yield-tree-build-forest:
  assumes wf-bins G ω bs sound-ptrs ω bs mono-red-ptr bs length bs = length ω +
1
  assumes build-tree G ω bs = Some t
  shows wf-rule-tree G t ∧ root-tree t = 𝔖 G ∧ yield-tree t = ω
proof −
```

let *?k = length bs − 1*
define *finished* **where** *finished-def*: *finished = filter-with-index* (*is-finished* $\mathcal{G}$ ω) (*items* (*bs*!*?k*))
then obtain *x i* **where** ∗: (*x*,*i*) ∈ *set finished Some t = build-tree′ bs* ω *?k i*
using *assms*(*5*) **unfolding** *finished-def build-tree-def* **by** (*auto simp*: *Let-def split*: *list.splits*, *presburger*)
have *k*: *?k < length bs ?k* ≤ *length* ω
using *assms*(*4*) **by** *simp-all*
have *i*: *i < length* (*bs*!*?k*)
using *index-filter-with-index-lt-length* ∗ *items-def finished-def* **by** (*metis length-map*)
have *x*: *x = item* (*bs*!*?k*!*i*)
using ∗ *i filter-with-index-nth items-def nth-map finished-def* **by** *metis*
have *finished*: *is-finished* $\mathcal{G}$ ω *x*
using ∗ *filter-with-index-P finished-def* **by** *metis*
have *wf-trees-input*: (*bs*, ω, *?k*, *i*) ∈ *wf-tree-input*
unfolding *wf-tree-input-def* **using** *assms*(*2*,*3*) *i k*(*1*) **by** *blast*
hence *wf-item-tree*: *wf-item-tree* $\mathcal{G}$ *x t*
using *wf-item-tree-build-tree′ assms*(*1*,*2*) *i k*(*1*) *x* ∗(*2*) **by** *metis*
have *wf-item*: *wf-item* $\mathcal{G}$ ω (*item* (*bs*!*?k*!*i*))
using *k*(*1*) *i assms*(*1*) **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def* **by** (*simp add*: *items-def*)
obtain *N ts* **where** *t*: *t = Branch N ts*
by (*metis* ∗(*2*) *build-tree′-termination option.inject wf-trees-input*)
hence *N = item-rule-head x*
map root-tree ts = item-rule-body x
using *finished wf-item-tree* **by** (*auto simp*: *is-finished-def is-complete-def*)
hence ∃ *r* ∈ *set* ($\mathfrak{R}$ $\mathcal{G}$). *N = rule-head r* ∧ *map root-tree ts = rule-body r*
using *wf-item x* **unfolding** *wf-item-def item-rule-body-def item-rule-head-def* **by** *blast*
hence *wf-rule*: *wf-rule-tree* $\mathcal{G}$ *t*
using *wf-item-tree t* **by** *simp*
have *root*: *root-tree t* = $\mathfrak{S}$ $\mathcal{G}$
using *finished t* ‹*N = item-rule-head x*› **by** (*auto simp*: *is-finished-def*)
have *yield-tree t = slice* (*item-origin* (*item* (*bs*!*?k*!*i*))) (*item-end* (*item* (*bs*!*?k*!*i*))) ω
using *k i assms*(*1*) *wf-trees-input wf-yield-tree-build-tree′ wf-yield-tree-def* ∗(*2*) **by** (*metis* (*no-types*, *lifting*))
hence *yield*: *yield-tree t* = ω
using *finished x* **unfolding** *is-finished-def* **by** *simp*
show *?thesis*
using ∗ *wf-rule root yield assms*(*4*) **unfolding** *build-tree-def* **by** *simp*
qed

corollary *wf-rule-root-yield-tree-build-tree-Earley*$_L$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *nonempty-derives* $\mathcal{G}$
assumes *build-tree* $\mathcal{G}$ ω (*Earley*$_L$ $\mathcal{G}$ ω) = *Some t*
shows *wf-rule-tree* $\mathcal{G}$ *t* ∧ *root-tree t* = $\mathfrak{S}$ $\mathcal{G}$ ∧ *yield-tree t* = ω
using *assms wf-rule-root-yield-tree-build-forest wf-bins-Earley*$_L$ *sound-mono-ptrs-Earley*$_L$ *Earley*$_L$*-def*

length-Earley$_L$-bins length-bins-Init$_L$ **by** (*metis nle-le*)

theorem *correctness-build-tree-Earley$_L$*:
assumes *wf-$\mathcal{G}$ $\mathcal{G}$ is-word $\mathcal{G}$ ω nonempty-derives $\mathcal{G}$*
shows ($\exists\, t.$ *build-tree $\mathcal{G}$ ω (Earley$_L$ $\mathcal{G}$ ω) = Some t*) $\longleftrightarrow$ *derives $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω* (**is** *?L $\longleftrightarrow$?R*)
proof *standard*
assume $*$: *?L*
let *?k = length (Earley$_L$ $\mathcal{G}$ ω) − 1*
define *finished* **where** *finished-def*: *finished = filter-with-index (is-finished $\mathcal{G}$ ω) (items ((Earley$_L$ $\mathcal{G}$ ω)!?k))*
then obtain *t x i* **where** $*$: *(x,i) ∈ set finished Some t = build-tree′ (Earley$_L$ $\mathcal{G}$ ω) ω ?k i*
using $*$ **unfolding** *finished-def build-tree-def* **by** (*auto simp*: *Let-def split*: *list.splits*, *presburger*)
have *k*: *?k < length (Earley$_L$ $\mathcal{G}$ ω) ?k ≤ length ω*
by (*simp-all add*: *Earley$_L$-def assms(1)*)
have *i*: *i < length ((Earley$_L$ $\mathcal{G}$ ω) ! ?k)*
using *index-filter-with-index-lt-length $*$ items-def finished-def* **by** (*metis length-map*)
have *x*: *x = item ((Earley$_L$ $\mathcal{G}$ ω)!?k!i)*
using $*$ *i filter-with-index-nth items-def nth-map finished-def* **by** *metis*
have *finished*: *is-finished $\mathcal{G}$ ω x*
using $*$ *filter-with-index-P finished-def* **by** *metis*
moreover have *x ∈ set (items ((Earley$_L$ $\mathcal{G}$ ω) ! ?k))*
using *x* **by** (*auto simp*: *items-def*; *metis One-nat-def i imageI nth-mem*)
ultimately have *recognizing (bins (Earley$_L$ $\mathcal{G}$ ω)) $\mathcal{G}$ ω*
by (*meson k(1) kth-bin-sub-bins recognizing-def subsetD*)
thus *?R*
using *correctness-Earley$_L$ assms* **by** *blast*
next
assume $*$: *?R*
let *?k = length (Earley$_L$ $\mathcal{G}$ ω) − 1*
define *finished* **where** *finished-def*: *finished = filter-with-index (is-finished $\mathcal{G}$ ω) (items ((Earley$_L$ $\mathcal{G}$ ω)!?k))*
have *recognizing (bins (Earley$_L$ $\mathcal{G}$ ω)) $\mathcal{G}$ ω*
using *assms $*$ correctness-Earley$_L$* **by** *blast*
moreover have *?k = length ω*
by (*simp add*: *Earley$_L$-def assms(1)*)
ultimately have $\exists\, x \in$ *set (items ((Earley$_L$ $\mathcal{G}$ ω)!?k)). is-finished $\mathcal{G}$ ω x*
unfolding *recognizing-def* **using** *assms(1) is-finished-def wf-bins-Earley$_L$ wf-item-in-kth-bin* **by** *metis*
then obtain *x i xs* **where** *xis*: *finished = (x,i)#xs*
using *filter-with-index-Ex-first* **by** (*metis finished-def*)
hence *simp*: *build-tree $\mathcal{G}$ ω (Earley$_L$ $\mathcal{G}$ ω) = build-tree′ (Earley$_L$ $\mathcal{G}$ ω) ω ?k i*
unfolding *build-tree-def finished-def* **by** *auto*
have *(x,i) ∈ set finished*
using *xis* **by** *simp*
hence *i < length ((Earley$_L$ $\mathcal{G}$ ω)!?k)*
using *index-filter-with-index-lt-length* **by** (*metis finished-def items-def length-map*)

moreover have *?k* < *length* (*Earley*$_L$ $\mathcal{G}$ ω)
by (*simp add*: *Earley*$_L$*-def assms*(*1*))
ultimately have (*Earley*$_L$ $\mathcal{G}$ ω, ω, *?k*, *i*) ∈ *wf-tree-input*
unfolding *wf-tree-input-def* **using** *sound-mono-ptrs-Earley*$_L$ *assms*(*1*,*3*) **by** *blast*
then obtain *N ts* **where** *build-tree*′ (*Earley*$_L$ $\mathcal{G}$ ω) ω *?k i* = *Some* (*Branch N ts*)
using *build-tree*′*-termination* **by** *blast*
thus *?L*
using *simp* **by** *simp*
qed

9.5 those, map, map option lemmas

lemma *those-map-exists*:
Some ys = *those* (*map f xs*) ⟹ *y* ∈ *set ys* ⟹ ∃ *x*. *x* ∈ *set xs* ∧ *Some y* ∈ *set* (*map f xs*)
proof (*induction xs arbitrary*: *ys*)
case (*Cons a xs*)
then show *?case*
apply (*clarsimp split*: *option.splits*)
by (*smt* (*verit*, *best*) *map-option-eq-Some set-ConsD*)
qed *auto*

lemma *those-Some*:
(∀ *x* ∈ *set xs*. ∃ *a*. *x* = *Some a*) ⟷ (∃ *ys*. *those xs* = *Some ys*)
by (*induction xs*) (*auto split*: *option.splits*)

lemma *those-Some-P*:
assumes ∀ *x* ∈ *set xs*. ∃ *ys*. *x* = *Some ys* ∧ (∀ *y* ∈ *set ys*. *P y*)
shows ∃ *yss*. *those xs* = *Some yss* ∧ (∀ *ys* ∈ *set yss*. ∀ *y* ∈ *set ys*. *P y*)
using *assms* **by** (*induction xs*) *auto*

lemma *map-Some-P*:
assumes *z* ∈ *set* (*map f xs*)
assumes ∀ *x* ∈ *set xs*. ∃ *ys*. *f x* = *Some ys* ∧ (∀ *y* ∈ *set ys*. *P y*)
shows ∃ *ys*. *z* = *Some ys* ∧ (∀ *y* ∈ *set ys*. *P y*)
using *assms* **by** (*induction xs*) *auto*

lemma *those-map-FBranch-only*:
assumes *g* = (λ*f*. *case f of FBranch N fss* ⇒ *Some* (*FBranch N* (*fss* @ [[*FLeaf* (ω!(*k*−*1*))]])) | - ⇒ *None*)
assumes *Some fs* = *those* (*map g pres*) *f* ∈ *set fs*
assumes ∀ *f* ∈ *set pres*. ∃ *N fss*. *f* = *FBranch N fss*
shows ∃ *f-pre N fss*. *f* = *FBranch N* (*fss* @ [[*FLeaf* (ω!(*k*−*1*))]]) ∧ *f-pre* = *FBranch N fss* ∧ *f-pre* ∈ *set pres*
using *assms*
apply (*induction pres arbitrary*: *fs f*)
apply (*auto*)

by (*smt* (*verit*, *best*) *list.inject list.set-cases map-option-eq-Some*)

lemma *those-map-Some-concat-exists*:
assumes $y \in set\ (concat\ ys)$
assumes $Some\ ys = those\ (map\ f\ xs)$
shows $\exists ys\ x.\ Some\ ys = f\ x \wedge y \in set\ ys \wedge x \in set\ xs$
using *assms*
apply (*induction xs arbitrary*: *ys y*)
apply (*auto split*: *option.splits*)
by (*smt* (*verit*, *ccfv-threshold*) *list.inject list.set-cases map-option-eq-Some*)

lemma *map-option-concat-those-map-exists*:
assumes $Some\ fs = map\text{-}option\ concat\ (those\ (map\ F\ xs))$
assumes $f \in set\ fs$
shows $\exists fss\ fs'.\ Some\ fss = those\ (map\ F\ xs) \wedge fs' \in set\ fss \wedge f \in set\ fs'$
using *assms*
apply (*induction xs arbitrary*: *fs f*)
apply (*auto split*: *option.splits*)
by (*smt* (*verit*, *best*) *UN-E map-option-eq-Some set-concat*)

lemma [*partial-function-mono*]:
monotone option.le-fun option-ord
(λ*f*. *map-option concat* (*those* (*map* (λ((*k'*, *pre*), *reds*).
f ((((*r*, *s*), *k'*), *pre*), {*pre*}) ⤜
(λ*pres*. *those* (*map* (λ*red*. *f* ((((*r*, *s*), *t*), *red*), *b* ∪ {*red*})) *reds*) ⤜
(λ*rss*. *those* (*map* (λ*f*. *case f of FBranch N fss* ⇒ *Some* (*FBranch N* (*fss* @ [*concat rss*])) | - ⇒ *None*) *pres*))))
xs)))
proof −
let *?f* =
(λ*f*. *map-option concat* (*those* (*map* (λ((*k'*, *pre*), *reds*).
f ((((*r*, *s*), *k'*), *pre*), {*pre*}) ⤜
(λ*pres*. *those* (*map* (λ*red*. *f* ((((*r*, *s*), *t*), *red*), *b* ∪ {*red*})) *reds*) ⤜
(λ*rss*. *those* (*map* (λ*f*. *case f of FBranch N fss* ⇒ *Some* (*FBranch N* (*fss* @ [*concat rss*])) | - ⇒ *None*) *pres*))))
xs)))
have *0*: $\bigwedge x\ y.\ option.le\text{-}fun\ x\ y \Longrightarrow option\text{-}ord\ (?f\ x)\ (?f\ y)$
apply (*auto simp*: *flat-ord-def fun-ord-def option.leq-refl split*: *option.splits forest.splits*)
subgoal premises *prems* **for** *x y*
proof −
let *?t* = *those* (*map* (λ((*k'*, *pre*), *reds*).
x ((((*r*, *s*), *k'*), *pre*), {*pre*}) ⤜
(λ*pres*. *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ⤜
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*))))
xs) = *None*
show *?t*
proof (*rule ccontr*)

assume *a*: ¬ *?t*
obtain *fss* **where** *fss*: *those* (*map* (λ((*k′*, *pre*), *reds*).
x ((((*r*, *s*), *k′*), *pre*), {*pre*}) ≫=
(λ*pres*. *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*))))
xs) = *Some fss*
using *a* **by** *blast*
{
fix *k′ pre reds*
assume ∗: ((*k′*, *pre*), *reds*) ∈ *set xs*
obtain *pres* **where** *pres*: *x* ((((*r*, *s*), *k′*), *pre*), {*pre*}) = *Some pres*
using *fss* ∗ *those-Some* **by** *force*
have ∃ *fs*. *Some fs* = *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*))
proof (*rule ccontr*)
assume ∄ *fs*. *Some fs* =
those (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*))
hence *None* =
those (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*))
by (*smt* (*verit*) *not-None-eq*)
hence *None* = *x* ((((*r*, *s*), *k′*), *pre*), {*pre*}) ≫=
(λ*pres*. *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*)))
by (*simp add*: *pres*)
hence ∃ ((*k′*, *pre*), *reds*) ∈ *set xs*. *None* = *x* ((((*r*, *s*), *k′*), *pre*), {*pre*}) ≫=
(λ*pres*. *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*)))
using ∗ **by** *blast*
thus *False*
using *fss those-Some* **by** *force*
qed
then obtain *fs* **where** *fs*: *Some fs* = *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) ≫=
(λ*rss*. *those* (*map* (*case-forest Map.empty* (λ*N fss*. *Some* (*FBranch N* (*fss* @ [*concat rss*]))))) *pres*))
by *blast*
obtain *rss* **where** *rss*: *those* (*map* (λ*red*. *x* ((((*r*, *s*), *t*), *red*), *insert red b*)) *reds*) = *Some rss*
using *fs* **by** *force*

```
      have x ((((r, s), k'), pre), {pre}) = y ((((r, s), k'), pre), {pre})
        using pres prems(1) by (metis option.distinct(1))
      moreover have those (map (λred. x ((((r, s), t), red), insert red b)) reds)
≫=
       (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss])))) pres))
       = those (map (λred. y ((((r, s), t), red), insert red b)) reds) ≫=
       (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss])))) pres))
       proof −
        have ∀ red ∈ set reds. x ((((r, s), t), red), insert red b) = y ((((r, s), t),
red), insert red b)
         proof standard
          fix red
          assume red ∈ set reds
          have ∀ x∈set (map (λred. x ((((r, s), t), red), insert red b)) reds) . ∃ a.
x = Some a
            using rss those-Some by blast
          then obtain f where x ((((r, s), t), red), insert red b) = Some f
            using ‹red ∈ set reds› by auto
          thus x ((((r, s), t), red), insert red b) = y ((((r, s), t), red), insert red
b)
            using prems(1) by (metis option.distinct(1))
         qed
         thus ?thesis
          by (smt (verit, best) map-eq-conv)
       qed
       ultimately have  x ((((r, s), k'), pre), {pre}) ≫=
       (λpres. those (map (λred. x ((((r, s), t), red), insert red b)) reds) ≫=
       (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss])))) pres)))
      = y ((((r, s), k'), pre), {pre}) ≫=
       (λpres. those (map (λred. y ((((r, s), t), red), insert red b)) reds) ≫=
       (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss])))) pres)))
        by (metis bind.bind-lunit pres)
      }
      hence ∀ ((k', pre), reds) ∈ set xs. x ((((r, s), k'), pre), {pre}) ≫=
       (λpres. those (map (λred. x ((((r, s), t), red), insert red b)) reds) ≫=
       (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss])))) pres)))
      = y ((((r, s), k'), pre), {pre}) ≫=
       (λpres. those (map (λred. y ((((r, s), t), red), insert red b)) reds) ≫=
       (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss])))) pres)))
       by blast
      hence those (map (λ((k', pre), reds).
      x ((((r, s), k'), pre), {pre}) ≫=
       (λpres. those (map (λred. x ((((r, s), t), red), insert red b)) reds) ≫=
```

```
        (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss]))))) pres))))
      xs) = those (map (λ((k', pre), reds).
      y ((((r, s), k'), pre), {pre}) ≫=
        (λpres. those (map (λred. y ((((r, s), t), red), insert red b)) reds) ≫=
          (λrss. those (map (case-forest Map.empty (λN fss. Some (FBranch N (fss
@ [concat rss]))))) pres))))
      xs)
      using prems(1) by (smt (verit, best) case-prod-conv map-eq-conv split-cong)
      thus False
        using prems(2) by simp
    qed
  qed
  done
 show ?thesis
  using monotoneI[of option.le-fun option-ord ?f, OF 0] by blast
qed
```

9.6 Parse trees

```
fun insert-group :: ('a ⇒ 'k) ⇒ ('a ⇒ 'v) ⇒ 'a ⇒ ('k × 'v list) list ⇒ ('k × 'v
list) list where
  insert-group K V a [] = [(K a, [V a])]
| insert-group K V a ((k, vs)#xs) = (
    if K a = k then (k, V a # vs) # xs
    else (k, vs) # insert-group K V a xs
  )

fun group-by :: ('a ⇒ 'k) ⇒ ('a ⇒ 'v) ⇒ 'a list ⇒ ('k × 'v list) list where
  group-by K V [] = []
| group-by K V (x#xs) = insert-group K V x (group-by K V xs)

lemma insert-group-cases:
  assumes (k, vs) ∈ set (insert-group K V a xs)
  shows (k = K a ∧ vs = [V a]) ∨ (k, vs) ∈ set xs ∨ (∃(k', vs') ∈ set xs. k' = k
∧ k = K a ∧ vs = V a # vs')
  using assms by (induction xs) (auto split: if-splits)

lemma group-by-exists-kv:
  (k, vs) ∈ set (group-by K V xs) ⟹ ∃x ∈ set xs. k = K x ∧ (∃v ∈ set vs. v =
V x)
  using insert-group-cases by (induction xs) (simp, force)

lemma group-by-forall-v-exists-k:
  (k, vs) ∈ set (group-by K V xs) ⟹ v ∈ set vs ⟹ ∃x ∈ set xs. k = K x ∧ v =
V x
proof (induction xs arbitrary: vs)
  case (Cons x xs)
  show ?case
```

```
  proof (cases (k, vs) ∈ set (group-by K V xs))
    case True
    thus ?thesis
      using Cons by simp
  next
    case False
    hence (k, vs) ∈ set (insert-group K V x (group-by K V xs))
      using Cons.prems(1) by force
    then consider (A) (k = K x ∧ vs = [V x])
      | (B) (k, vs) ∈ set (group-by K V xs)
      | (C) (∃(k', vs') ∈ set (group-by K V xs). k' = k ∧ k = K x ∧ vs = V x #
vs')
      using insert-group-cases by fastforce
    thus ?thesis
    proof cases
      case A
      thus ?thesis
        using Cons.prems(2) by auto
    next
      case B
      then show ?thesis
        using False by linarith
    next
      case C
      then show ?thesis
        using Cons.IH Cons.prems(2) by fastforce
    qed
  qed
qed simp

partial-function (option) build-trees' :: 'a bins ⇒ 'a sentence ⇒ nat ⇒ nat ⇒
nat set ⇒ 'a forest list option where
  build-trees' bs ω k i I = (
    let e = bs!k!i in (
    case pointer e of
      Null ⇒ Some ([FBranch (item-rule-head (item e)) []]) — start building sub-
trees
    | Pre pre ⇒ ( — add sub-trees starting from terminal
        do {
          pres ← build-trees' bs ω (k−1) pre {pre};
          those (map (λf.
            case f of
              FBranch N fss ⇒ Some (FBranch N (fss @ [[FLeaf (ω!(k−1))]]))
            | - ⇒ None  — impossible case
          ) pres)
        })
    | PreRed p ps ⇒ ( — add sub-trees starting from non-terminal
        let ps' = filter (λ(k', pre, red). red ∉ I) (p#ps) in
        let gs = group-by (λ(k', pre, red). (k', pre)) (λ(k', pre, red). red) ps' in
```

```
      map-option concat (those (map (λ((k', pre), reds).
        do {
          pres ← build-trees' bs ω k' pre {pre};
          rss ← those (map (λred. build-trees' bs ω k red (I ∪ {red})) reds);
          those (map (λf.
            case f of
              FBranch N fss ⇒ Some (FBranch N (fss @ [concat rss]))
            | - ⇒ None — impossible case
          ) pres)
        }
      ) gs))
    )
))
```

declare *build-trees'.simps* [*code*]

definition *build-trees* :: *'a cfg ⇒ 'a sentence ⇒ 'a bins ⇒ 'a forest list option*
where
build-trees G ω bs = (
  let k = length bs − 1 in
  let finished = filter-with-index (λx. is-finished G ω x) (items (bs!k)) in
  map-option concat (those (map (λ(-, i). build-trees' bs ω k i {i}) finished))
)

lemma *build-forest'-simps*[*simp*]:
e = bs!k!i ⟹ pointer e = Null ⟹ build-trees' bs ω k i I = Some ([FBranch (item-rule-head (item e)) []])
e = bs!k!i ⟹ pointer e = Pre pre ⟹ build-trees' bs ω (k−1) pre {pre} = None ⟹ build-trees' bs ω k i I = None
e = bs!k!i ⟹ pointer e = Pre pre ⟹ build-trees' bs ω (k−1) pre {pre} = Some pres ⟹
  build-trees' bs ω k i I = those (map (λf. case f of FBranch N fss ⇒ Some (FBranch N (fss @ [[FLeaf (ω!(k−1))]])) | - ⇒ None) pres)
by (*subst build-trees'.simps*, *simp*)+

definition *wf-trees-input* :: *('a bins × 'a sentence × nat × nat × nat set) set*
where
wf-trees-input = {
  (*bs*, *ω*, *k*, *i*, *I*) | *bs ω k i I*.
    sound-ptrs ω bs ∧
    k < *length bs* ∧
    i < *length* (*bs*!*k*) ∧
    I ⊆ {*0*..<*length* (*bs*!*k*)} ∧
    i ∈ *I*
}

fun *build-forest'-measure* :: *('a bins × 'a sentence × nat × nat × nat set) ⇒ nat*
where
build-forest'-measure (*bs*, *ω*, *k*, *i*, *I*) = *foldl* (+) *0* (*map length* (*take* (*k*+*1*) *bs*))

$-$ *card I*

lemma *wf-trees-input-pre*:
assumes $(bs, \omega, k, i, I) \in$ *wf-trees-input*
assumes $e = bs!k!i$ *pointer* $e =$ *Pre pre*
shows $(bs, \omega, (k-1), pre, \{pre\}) \in$ *wf-trees-input*
using *assms* **unfolding** *wf-trees-input-def*
apply (*auto simp*: *sound-ptrs-def sound-pre-ptr-def*)
apply (*metis nth-mem*)
done

lemma *wf-trees-input-prered-pre*:
assumes $(bs, \omega, k, i, I) \in$ *wf-trees-input*
assumes $e = bs!k!i$ *pointer* $e =$ *PreRed p ps*
assumes $ps' =$ *filter* $(\lambda(k', pre, red).\ red \notin I)\ (p\#ps)$
assumes $gs =$ *group-by* $(\lambda(k', pre, red).\ (k', pre))\ (\lambda(k', pre, red).\ red)\ ps'$
assumes $((k', pre), reds) \in$ *set gs*
shows $(bs, \omega, k', pre, \{pre\}) \in$ *wf-trees-input*
proof $-$
obtain *red* **where** $(k', pre, red) \in$ *set* ps'
using *assms*(*5*,*6*) *group-by-exists-kv* **by** *fast*
hence $*$: $(k', pre, red) \in$ *set* $(p\#ps)$
using *assms*(*4*) **by** (*meson filter-is-subset in-mono*)
have $k <$ *length bs* $e \in$ *set* $(bs!k)$
using *assms*(*1*,*2*) **unfolding** *wf-trees-input-def* **by** *auto*
hence $k' < k$ *pre* $<$ *length* $(bs!k')$
using *assms*(*1*,*3*) $*$ **unfolding** *wf-trees-input-def sound-ptrs-def sound-prered-ptr-def*
by *blast+*
thus *?thesis*
using *assms* **by** (*simp add*: *wf-trees-input-def*)
qed

lemma *wf-trees-input-prered-red*:
assumes $(bs, \omega, k, i, I) \in$ *wf-trees-input*
assumes $e = bs!k!i$ *pointer* $e =$ *PreRed p ps*
assumes $ps' =$ *filter* $(\lambda(k', pre, red).\ red \notin I)\ (p\#ps)$
assumes $gs =$ *group-by* $(\lambda(k', pre, red).\ (k', pre))\ (\lambda(k', pre, red).\ red)\ ps'$
assumes $((k', pre), reds) \in$ *set gs* $red \in$ *set reds*
shows $(bs, \omega, k, red, I \cup \{red\}) \in$ *wf-trees-input*
proof $-$
have $(k', pre, red) \in$ *set* ps'
using *assms*(*5*,*6*,*7*) *group-by-forall-v-exists-k* **by** *fastforce*
hence $*$: $(k', pre, red) \in$ *set* $(p\#ps)$
using *assms*(*4*) **by** (*meson filter-is-subset in-mono*)
have $k <$ *length bs* $e \in$ *set* $(bs!k)$
using *assms*(*1*,*2*) **unfolding** *wf-trees-input-def* **by** *auto*
hence *0*: *red* $<$ *length* $(bs!k)$
using *assms*(*1*,*3*) $*$ **unfolding** *wf-trees-input-def sound-ptrs-def sound-prered-ptr-def*
by *blast*

moreover have $I \subseteq \{0..<length\ (bs!k)\}$
using *assms(1)* **unfolding** *wf-trees-input-def* **by** *blast*
ultimately have *1*: $I \cup \{red\} \subseteq \{0..<length\ (bs!k)\}$
by *simp*
show *?thesis*
using *0 1 assms(1)* **unfolding** *wf-trees-input-def* **by** *blast*
qed

lemma *build-trees'-induct*:
assumes $(bs, \omega, k, i, I) \in$ *wf-trees-input*
assumes $\bigwedge bs\ \omega\ k\ i\ I.$
$(\bigwedge e\ pre.\ e = bs!k!i \Longrightarrow pointer\ e = Pre\ pre \Longrightarrow P\ bs\ \omega\ (k-1)\ pre\ \{pre\}) \Longrightarrow$
$(\bigwedge e\ p\ ps\ ps'\ gs\ k'\ pre\ reds.\ e = bs!k!i \Longrightarrow pointer\ e = PreRed\ p\ ps \Longrightarrow$
$ps' = filter\ (\lambda(k', pre, red).\ red \notin I)\ (p\#ps) \Longrightarrow$
$gs = group\text{-}by\ (\lambda(k', pre, red).\ (k', pre))\ (\lambda(k', pre, red).\ red)\ ps' \Longrightarrow$
$((k', pre), reds) \in set\ gs \Longrightarrow P\ bs\ \omega\ k'\ pre\ \{pre\}) \Longrightarrow$
$(\bigwedge e\ p\ ps\ ps'\ gs\ k'\ pre\ red\ reds\ reds'.\ e = bs!k!i \Longrightarrow pointer\ e = PreRed\ p\ ps$
$\Longrightarrow$
$ps' = filter\ (\lambda(k', pre, red).\ red \notin I)\ (p\#ps) \Longrightarrow$
$gs = group\text{-}by\ (\lambda(k', pre, red).\ (k', pre))\ (\lambda(k', pre, red).\ red)\ ps' \Longrightarrow$
$((k', pre), reds) \in set\ gs \Longrightarrow red \in set\ reds \Longrightarrow P\ bs\ \omega\ k\ red\ (I \cup \{red\})) \Longrightarrow$
$P\ bs\ \omega\ k\ i\ I$
shows $P\ bs\ \omega\ k\ i\ I$
using *assms(1)*
proof (*induction* $n \equiv$ *build-forest'-measure* (bs, ω, k, i, I) *arbitrary*: $k\ i\ I$ *rule*: *nat-less-induct*)
case *1*
obtain e **where** *entry*: $e = bs!k!i$
by *simp*
consider (*Null*) $pointer\ e = Null$
| (*Pre*) $\exists pre.\ pointer\ e = Pre\ pre$
| (*PreRed*) $\exists p\ ps.\ pointer\ e = PreRed\ p\ ps$
by (*metis pointer.exhaust*)
thus *?case*
proof *cases*
case *Null*
thus *?thesis*
using *assms(2) entry* **by** *fastforce*
next
case *Pre*
then obtain *pre* **where** *pre*: $pointer\ e = Pre\ pre$
by *blast*
define n **where** n: $n = build\text{-}forest'\text{-}measure\ (bs, \omega, (k-1), pre, \{pre\})$
have $0 < k\ pre < length\ (bs!(k-1))$
using *1(2) entry pre* **unfolding** *wf-trees-input-def sound-ptrs-def sound-pre-ptr-def*
by (*smt* (*verit*) *mem-Collect-eq nth-mem prod.inject*)+
have $k < length\ bs\ i < length\ (bs!k)\ I \subseteq \{0..<length\ (bs!k)\}\ i \in I$
using *1(2)* **unfolding** *wf-trees-input-def* **by** *blast*+
have $length\ (bs!(k-1)) > 0$

```
      using ‹pre < length (bs!(k−1))› by force
    hence foldl (+) 0 (map length (take k bs)) > 0
      by (smt (verit, del-insts) foldl-add-nth ‹0 < k› ‹k < length bs›
       add.commute add-diff-inverse-nat less-imp-diff-less less-one zero-eq-add-iff-both-eq-0)
    have card I ≤ length (bs!k)
      by (simp add: ‹I ⊆ {0..<length (bs ! k)}› subset-eq-atLeast0-lessThan-card)
    have card I + (foldl (+) 0 (map length (take (Suc (k − Suc 0)) bs)) − Suc 0)
=
      card I + (foldl (+) 0 (map length (take k bs)) − 1)
      using ‹0 < k› by simp
    also have ... = card I + foldl (+) 0 (map length (take k bs)) − 1
      using ‹0 < foldl (+) 0 (map length (take k bs))› by auto
    also have ... < card I + foldl (+) 0 (map length (take k bs))
      by (simp add: ‹0 < foldl (+) 0 (map length (take k bs))›)
    also have ... ≤ foldl (+) 0 (map length (take k bs)) + length (bs!k)
      by (simp add: ‹card I ≤ length (bs ! k)›)
    also have ... = foldl (+) 0 (map length (take (k+1) bs))
      using foldl-add-nth ‹k < length bs› by blast
    finally have build-forest'-measure (bs, ω, k, i, I) − build-forest'-measure (bs,
ω, (k−1), pre, {pre}) > 0
      by simp
    hence P bs ω (k−1) pre {pre}
      using 1 n wf-trees-input-pre entry pre zero-less-diff by blast
    thus ?thesis
      using assms(2) entry pre pointer.distinct(5) pointer.inject(1) by presburger
  next
    case PreRed
    then obtain p ps where pps: pointer e = PreRed p ps
      by blast
    define ps' where ps': ps' = filter (λ(k', pre, red). red ∉ I) (p#ps)
    define gs where gs: gs = group-by (λ(k', pre, red). (k', pre)) (λ(k', pre, red).
red) ps'
    have 0: ∀(k', pre, red) ∈ set ps'. k' < k ∧ pre < length (bs!k') ∧ red < length
(bs!k)
      using entry pps ps' 1(2) unfolding wf-trees-input-def sound-ptrs-def sound-prered-ptr-def
      apply (auto simp del: filter.simps)
      apply (metis nth-mem prod-cases3)+
      done
    hence sound-gs: ∀((k', pre), reds) ∈ set gs. k' < k ∧ pre < length (bs!k')
      using gs group-by-exists-kv by fast
    have sound-gs2: ∀((k', pre), reds) ∈ set gs. ∀red ∈ set reds. red < length (bs!k)
    proof (standard, standard, standard, standard)
      fix x a b k' pre red
      assume x ∈ set gs x = (a, b) (k', pre) = a red ∈ set b
      hence ∃x ∈ set ps'. red = (λ(k', pre, red). red) x
        using group-by-forall-v-exists-k gs ps' by meson
      thus red < length (bs!k)
        using 0 by fast
    qed
```

```
  {
    fix k' pre reds red
    assume a0: ((k', pre), reds) ∈ set gs
   define n-pre where n-pre: n-pre = build-forest'-measure (bs, ω, k', pre, {pre})
    have k < length bs i < length (bs!k) I ⊆ {0..<length (bs!k)} i ∈ I
      using 1(2) unfolding wf-trees-input-def by blast+
    have k' < k pre < length (bs!k')
      using sound-gs a0 by fastforce+
    have length (bs!k') > 0
      using ‹pre < length (bs!k')› by force
    hence gt0: foldl (+) 0 (map length (take (k'+1) bs)) > 0
      by (smt (verit, del-insts) foldl-add-nth ‹k < length bs› ‹k' < k› add-gr-0
order.strict-trans)
    have card-bound: card I ≤ length (bs!k)
     by (simp add: ‹I ⊆ {0..<length (bs ! k)}› subset-eq-atLeast0-lessThan-card)
    have card I + (foldl (+) 0 (map length (take (Suc k') bs)) - Suc 0) =
    card I + foldl (+) 0 (map length (take (Suc k') bs)) - 1
      by (metis Nat.add-diff-assoc One-nat-def Suc-eq-plus1 Suc-leI ‹0 < foldl
(+) 0 (map length (take (k' + 1) bs))›)
    also have ... < card I + foldl (+) 0 (map length (take (Suc k') bs))
      using gt0 by auto
    also have ... ≤ foldl (+) 0 (map length (take (Suc k') bs)) + length (bs!k)
      using card-bound by simp
    also have ... ≤ foldl (+) 0 (map length (take (k+1) bs))
      using foldl-add-nth-ge Suc-leI ‹k < length bs› ‹k' < k› by blast
   finally have build-forest'-measure (bs, ω, k, i, I) - build-forest'-measure (bs,
ω, k', pre, {pre}) > 0
      by simp
    hence P bs ω k' pre {pre}
      using 1(1) wf-trees-input-prered-pre[OF 1.prems(1) entry pps ps' gs a0]
zero-less-diff by blast
  }
  moreover {
    fix k' pre reds red
    assume a0: ((k', pre), reds) ∈ set gs
    assume a1: red ∈ set reds
    define n-red where n-red: n-red = build-forest'-measure (bs, ω, k, red, I ∪
{red})
    have k < length bs i < length (bs!k) I ⊆ {0..<length (bs!k)} i ∈ I
      using 1(2) unfolding wf-trees-input-def by blast+
    have k' < k pre < length (bs!k') red < length (bs!k)
      using a0 a1 sound-gs sound-gs2 by fastforce+
    have red ∉ I
      using a0 a1 unfolding gs ps'
       by (smt (verit, best) group-by-forall-v-exists-k case-prodE case-prod-conv
mem-Collect-eq set-filter)
    have card-bound: card I ≤ length (bs!k)
     by (simp add: ‹I ⊆ {0..<length (bs ! k)}› subset-eq-atLeast0-lessThan-card)
    have length (bs!k') > 0
```

```
    using ‹pre < length (bs!k')› by force
  hence gt0: foldl (+) 0 (map length (take (k'+1) bs)) > 0
    by (smt (verit, del-insts) foldl-add-nth ‹k < length bs› ‹k' < k› add-gr-0
order.strict-trans)
  have lt: foldl (+) 0 (map length (take (Suc k') bs)) + length (bs!k) ≤ foldl
(+) 0 (map length (take (k+1) bs))
    using foldl-add-nth-ge Suc-leI ‹k < length bs› ‹k' < k› by blast
  have card I + (foldl (+) 0 (map length (take (Suc k) bs)) − card (insert red
I)) =
    card I + (foldl (+) 0 (map length (take (Suc k) bs)) − card I − 1)
    using ‹I ⊆ {0..<length (bs ! k)}› ‹red ∉ I› finite-subset by fastforce
  also have ... = foldl (+) 0 (map length (take (Suc k) bs)) − 1
    using gt0 card-bound lt by force
  also have ... < foldl (+) 0 (map length (take (Suc k) bs))
    using gt0 lt by auto
 finally have build-forest'-measure (bs, ω, k, i, I) − build-forest'-measure (bs,
ω, k, red, I ∪ {red}) > 0
    by simp
  moreover have (bs, ω, k, red, I ∪ {red}) ∈ wf-trees-input
    using wf-trees-input-prered-red[OF 1(2) entry pps ps' gs] a0 a1 by blast
  ultimately have P bs ω k red (I ∪ {red})
    using 1(1) zero-less-diff by blast
 }
 moreover have (⋀e pre. e = bs!k!i ⟹ pointer e = Pre pre ⟹ P bs ω (k−1)
pre {pre})
  using entry pps by fastforce
 ultimately show ?thesis
  using assms(2) entry gs pointer.inject(2) pps ps' by presburger
qed
qed

lemma build-trees'-termination:
 assumes (bs, ω, k, i, I) ∈ wf-trees-input
 shows ∃fs. build-trees' bs ω k i I = Some fs ∧ (∀f ∈ set fs. ∃N fss. f = FBranch
N fss)
proof −
 have ∃fs. build-trees' bs ω k i I = Some fs ∧ (∀f ∈ set fs. ∃N fss. f = FBranch
N fss)
  apply (induction rule: build-trees'-induct[OF assms(1)])
  subgoal premises IH for bs ω k i I
  proof −
   define e where entry: e = bs!k!i
   consider (Null) pointer e = Null
     | (Pre) ∃pre. pointer e = Pre pre
     | (PreRed) ∃k' pre red reds. pointer e = PreRed (k', pre, red) reds
     by (metis pointer.exhaust surj-pair)
   thus ?thesis
   proof cases
     case Null
```

```
    have build-trees' bs ω k i I = Some ([FBranch (item-rule-head (item e)) []])
      using build-forest'-simps(1) Null entry by simp
    thus ?thesis
      by simp
  next
    case Pre
    then obtain pre where pre: pointer e = Pre pre
      by blast
    obtain fs where fs: build-trees' bs ω (k−1) pre {pre} = Some fs
      ∀f ∈ set fs. ∃N fss. f = FBranch N fss
      using IH(1) entry pre by blast
    let ?g = λf. case f of FLeaf a ⇒ None
      | FBranch N fss ⇒ Some (FBranch N (fss @ [[FLeaf (ω!(k−1))]]))
    have simp: build-trees' bs ω k i I = those (map ?g fs)
      using build-forest'-simps(3) entry pre fs by blast
    moreover have ∀f ∈ set (map ?g fs). ∃a. f = Some a
      using fs(2) by auto
    ultimately obtain fs' where fs': build-trees' bs ω k i I = Some fs'
      using those-Some by (smt (verit, best))
    moreover have ∀f ∈ set fs'. ∃N fss. f = FBranch N fss
    proof standard
      fix f
      assume f ∈ set fs'
      then obtain x where x ∈ set fs Some f ∈ set (map ?g fs)
        using those-map-exists by (metis (no-types, lifting) fs' simp)
      thus ∃N fss. f = FBranch N fss
        using fs(2) by auto
    qed
    ultimately show ?thesis
      by blast
  next
    case PreRed
    then obtain p ps where pps: pointer e = PreRed p ps
      by blast
    define ps' where ps': ps' = filter (λ(k', pre, red). red ∉ I) (p#ps)
     define gs where gs: gs = group-by (λ(k', pre, red). (k', pre)) (λ(k', pre,
red). red) ps'
    let ?g = λ((k', pre), reds).
        do {
          pres ← build-trees' bs ω k' pre {pre};
          rss ← those (map (λred. build-trees' bs ω k red (I ∪ {red})) reds);
          those (map (λf.
            case f of
              FBranch N fss ⇒ Some (FBranch N (fss @ [concat rss]))
            | - ⇒ None — impossible case
          ) pres)
        }
    have simp: build-trees' bs ω k i I = map-option concat (those (map ?g gs))
        using entry pps ps' gs by (subst build-trees'.simps) (auto simp del:
```

filter.simps)
have $\forall fso \in set\ (map\ ?g\ gs).\ \exists fs.\ fso = Some\ fs \wedge (\forall f \in set\ fs.\ \exists N\ fss.\ f = FBranch\ N\ fss)$
proof *standard*
fix *fso*
assume $fso \in set\ (map\ ?g\ gs)$
moreover have $\forall ps \in set\ gs.\ \exists fs.\ ?g\ ps = Some\ fs \wedge (\forall f \in set\ fs.\ \exists N\ fss.\ f = FBranch\ N\ fss)$
proof *standard*
fix *ps*
assume $ps \in set\ gs$
then obtain $k'\ pre\ reds$ **where** $*$: $((k', pre), reds) \in set\ gs\ ((k', pre), reds) = ps$
by (*metis surj-pair*)
then obtain *pres* **where** *pres*: *build-trees'* $bs\ \omega\ k'\ pre\ \{pre\} = Some\ pres$
$\forall f \in set\ pres.\ \exists N\ fss.\ f = FBranch\ N\ fss$
using *IH*(*2*) *entry pps ps' gs* **by** *blast*
have $\forall f \in set\ (map\ (\lambda red.\ build\text{-}trees'\ bs\ \omega\ k\ red\ (I \cup \{red\}))\ reds).\ \exists a.\ f = Some\ a$
using *IH*(*3*)[*OF entry pps ps' gs* $*$(*1*)] **by** *auto*
then obtain *rss* **where** *rss*: $Some\ rss = those\ (map\ (\lambda red.\ build\text{-}trees'\ bs\ \omega\ k\ red\ (I \cup \{red\}))\ reds)$
using *those-Some* **by** (*metis* (*full-types*))
let $?h = \lambda f.\ case\ f\ of\ FBranch\ N\ fss \Rightarrow Some\ (FBranch\ N\ (fss\ @\ [concat\ rss]))\ |\ \text{-} \Rightarrow None$
have $\forall x \in set\ (map\ ?h\ pres).\ \exists a.\ x = Some\ a$
using *pres*(*2*) **by** *auto*
then obtain *fs* **where** *fs*: $Some\ fs = those\ (map\ ?h\ pres)$
using *those-Some* **by** (*smt* (*verit*, *best*))
have $\forall f \in set\ fs.\ \exists N\ fss.\ f = FBranch\ N\ fss$
proof *standard*
fix *f*
assume $*$: $f \in set\ fs$
hence $\exists x.\ x \in set\ pres \wedge Some\ f \in set\ (map\ ?h\ pres)$
using *those-map-exists*[*OF fs* $*$] **by** *blast*
then obtain *x* **where** *x*: $x \in set\ pres \wedge Some\ f \in set\ (map\ ?h\ pres)$
by *blast*
thus $\exists N\ fss.\ f = FBranch\ N\ fss$
using *pres*(*2*) **by** *auto*
qed
moreover have $?g\ ps = Some\ fs$
using *fs pres rss* $*$ **by** (*auto*, *metis bind.bind-lunit*)
ultimately show $\exists fs.\ ?g\ ps = Some\ fs \wedge (\forall f \in set\ fs.\ \exists N\ fss.\ f = FBranch\ N\ fss)$
by *blast*
qed
ultimately show $\exists fs.\ fso = Some\ fs \wedge (\forall f \in set\ fs.\ \exists N\ fss.\ f = FBranch\ N\ fss)$
using *map-Some-P* **by** *auto*

```
      qed
      then obtain fss where those (map ?g gs) = Some fss ∀ fs ∈ set fss. ∀ f ∈
set fs. ∃ N fss. f = FBranch N fss
        using those-Some-P by blast
      hence build-trees' bs ω k i I = Some (concat fss) ∀ f ∈ set (concat fss). ∃ N
fss. f = FBranch N fss
        using simp by auto
      thus ?thesis
        by blast
    qed
  qed
  done
 thus ?thesis
  by blast
qed

lemma wf-item-tree-build-trees':
 assumes (bs, ω, k, i, I) ∈ wf-trees-input
 assumes wf-bins G ω bs
 assumes k < length bs i < length (bs!k)
 assumes build-trees' bs ω k i I = Some fs
 assumes f ∈ set fs
 assumes t ∈ set (trees f)
 shows wf-item-tree G (item (bs!k!i)) t
proof −
 have wf-item-tree G (item (bs!k!i)) t
  using assms
  apply (induction arbitrary: fs f t rule: build-trees'-induct[OF assms(1)])
  subgoal premises prems for bs ω k i I fs f t
  proof −
   define e where entry: e = bs!k!i
   consider (Null) pointer e = Null
    | (Pre) ∃ pre. pointer e = Pre pre
    | (PreRed) ∃ p ps. pointer e = PreRed p ps
    by (metis pointer.exhaust)
   thus ?thesis
   proof cases
    case Null
    hence simp: build-trees' bs ω k i I = Some ([FBranch (item-rule-head (item
e)) []])
      using entry by simp
     moreover have f = FBranch (item-rule-head (item e)) []
      using build-forest'-simps(1) Null prems(8,9) entry by auto
     ultimately have simp: t = Branch (item-rule-head (item e)) []
      using prems(10) by simp
     have sound-ptrs ω bs
      using prems(4) unfolding wf-trees-input-def by blast
     hence predicts (item e)
    using Null prems(6,7) nth-mem entry unfolding sound-ptrs-def sound-null-ptr-def
```

by *blast*
hence *item-dot (item e) = 0*
unfolding *predicts-def* **by** *blast*
thus *?thesis*
using *simp entry* **by** *simp*
next
case *Pre*
then obtain *pre* **where** *pre*: *pointer e = Pre pre*
by *blast*
have *sound*: *sound-ptrs ω bs*
using *prems(4)* **unfolding** *wf-trees-input-def* **by** *blast*
have *scans*: *scans ω k (item (bs!(k−1)!pre)) (item e)*
using *entry pre prems(6−7)* ‹*sound-ptrs ω bs*› **unfolding** *sound-ptrs-def sound-pre-ptr-def* **by** *simp*
hence ∗:
item-rule-head (item (bs!(k−1)!pre)) = item-rule-head (item e)
item-rule-body (item (bs!(k−1)!pre)) = item-rule-body (item e)
item-dot (item (bs!(k−1)!pre)) + 1 = item-dot (item e)
next-symbol (item (bs!(k−1)!pre)) = Some (ω!(k−1))
unfolding *scans-def inc-item-def* **by** (*simp-all add*: *item-rule-head-def item-rule-body-def*)
have *wf*: *(bs, ω, k−1, pre, {pre}) ∈ wf-trees-input*
using *entry pre prems(4) wf-trees-input-pre* **by** *blast*
then obtain *pres* **where** *pres*: *build-trees′ bs ω (k−1) pre {pre} = Some pres*
∀f ∈ set pres. ∃N fss. f = FBranch N fss
using *build-trees′-termination wf* **by** *blast*
let *?g = λf. case f of FBranch N fss ⇒ Some (FBranch N (fss @ [[FLeaf (ω!(k−1))]])) | - ⇒ None*
have *build-trees′ bs ω k i I = those (map ?g pres)*
using *entry pre pres* **by** *simp*
hence *fs*: *Some fs = those (map ?g pres)*
using *prems(8)* **by** *simp*
then obtain *f-pre N fss* **where** *Nfss*: *f = FBranch N (fss @ [[FLeaf (ω!(k−1))]])*
f-pre = FBranch N fss f-pre ∈ set pres
using *those-map-FBranch-only fs pres(2) prems(9)* **by** *blast*
define *tss* **where** *tss*: *tss = map (λfs. concat (map (λf. trees f) fs)) fss*
have *trees (FBranch N (fss @ [[FLeaf (ω!(k−1))]])) =*
map (λts. Branch N ts) [ts @ [Leaf (ω!(k−1))] . ts <− combinations tss]
by (*subst tss*, *subst trees-append-single-singleton*, *simp*)
moreover have *t ∈ set (trees (FBranch N (fss @ [[FLeaf (ω!(k−1))]])))*
using *Nfss(1) prems(10)* **by** *blast*
ultimately obtain *ts* **where** *ts*: *t = Branch N (ts @ [Leaf (ω!(k−1))]) ∧ ts ∈ set (combinations tss)*
by *auto*
have *sound-ptrs ω bs*
using *prems(4)* **unfolding** *wf-trees-input-def* **by** *blast*
hence *pre < length (bs!(k−1))*

using *entry pre prems*(6,7) **unfolding** *sound-ptrs-def sound-pre-ptr-def*
by (*metis nth-mem*)
moreover have $k - 1 <$ *length bs*
by (*simp add*: *prems*(6) *less-imp-diff-less*)
moreover have *Branch N ts* ∈ *set* (*trees* (*FBranch N fss*))
using *ts tss* **by** *simp*
ultimately have *IH*: *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!(*k*−1)!*pre*)) (*Branch N ts*)
using *prems*(1,2,4,5) *entry pre Nfss*(2,3) *wf pres*(1) **by** *blast*
have *map root-tree* (*ts* @ [*Leaf* (ω!(*k*−1))]) = *map root-tree ts* @ [ω!(*k*−1)]
by *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!(*k*−1)!*pre*))) (*item-rule-body* (*item* (*bs*!(*k*−1)!*pre*))) @ [ω!(*k*−1)]
using *IH* **by** *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!(*k*−1)!*pre*))) (*item-rule-body* (*item e*)) @ [ω!(*k*−1)]
using ∗(2) **by** *simp*
also have ... = *take* (*item-dot* (*item e*)) (*item-rule-body* (*item e*))
using ∗(2−4) **by** (*auto simp*: *next-symbol-def is-complete-def split*: *if-splits*; *metis leI take-Suc-conv-app-nth*)
finally have *map root-tree* (*ts* @ [*Leaf* (ω!(*k*−1))]) = *take* (*item-dot* (*item e*)) (*item-rule-body* (*item e*)) **.**
hence *wf-item-tree* $\mathcal{G}$ (*item e*) (*Branch N* (*ts* @ [*Leaf* (ω!(*k*−1))]))
using *IH* ∗(1) **by** *simp*
thus *?thesis*
using *ts entry* **by** *fastforce*
next
case *PreRed*
then obtain *p ps* **where** *prered*: *pointer e* = *PreRed p ps*
by *blast*
define *ps′* **where** *ps′*: *ps′* = *filter* (λ(*k′*, *pre*, *red*). *red* ∉ *I*) (*p*#*ps*)
define *gs* **where** *gs*: *gs* = *group-by* (λ(*k′*, *pre*, *red*). (*k′*, *pre*)) (λ(*k′*, *pre*, *red*). *red*) *ps′*
let *?g* = λ((*k′*, *pre*), *reds*).
do {
pres ← *build-trees′ bs* ω *k′ pre* {*pre*};
rss ← *those* (*map* (λ*red*. *build-trees′ bs* ω *k red* (*I* ∪ {*red*})) *reds*);
those (*map* (λ*f*.
case f of
FBranch N fss ⇒ *Some* (*FBranch N* (*fss* @ [*concat rss*]))
| - ⇒ *None* — impossible case
) *pres*)
}
have *simp*: *build-trees′ bs* ω *k i I* = *map-option concat* (*those* (*map ?g gs*))
using *entry prered ps′ gs* **by** (*subst build-trees′.simps*) (*auto simp del*: *filter.simps*)
have ∀ *fso* ∈ *set* (*map ?g gs*). ∃ *fs*. *fso* = *Some fs* ∧ (∀ *f* ∈ *set fs*. ∀ *t* ∈ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*)
proof *standard*
fix *fso*

assume *fso* ∈ *set* (*map ?g gs*)
moreover have ∀ *ps* ∈ *set gs*. ∃ *fs*. *?g ps* = *Some fs* ∧ (∀ *f* ∈ *set fs*. ∀ *t* ∈ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*)
proof *standard*
fix *g*
assume *g* ∈ *set gs*
then obtain *k′ pre reds* **where** *g*: ((*k′*, *pre*), *reds*) ∈ *set gs* ((*k′*, *pre*), *reds*) = *g*
by (*metis surj-pair*)
moreover have *wf-pre*: (*bs*, ω, *k′*, *pre*, {*pre*}) ∈ *wf-trees-input*
using *wf-trees-input-prered-pre*[*OF prems*(*4*) *entry prered ps′ gs g*(*1*)] **by** *blast*
ultimately obtain *pres* **where** *pres*: *build-trees′ bs* ω *k′ pre* {*pre*} = *Some pres*
∀ *f-pre* ∈ *set pres*. ∃ *N fss*. *f-pre* = *FBranch N fss*
using *build-trees′-termination* **by** *blast*
have *wf-reds*: ∀ *red* ∈ *set reds*. (*bs*, ω, *k*, *red*, *I* ∪ {*red*}) ∈ *wf-trees-input*
using *wf-trees-input-prered-red*[*OF prems*(*4*) *entry prered ps′ gs g*(*1*)] **by** *blast*
hence ∀ *f* ∈ *set* (*map* (λ*red*. *build-trees′ bs* ω *k red* (*I* ∪ {*red*})) *reds*). ∃ *a*. *f* = *Some a*
using *build-trees′-termination* **by** *fastforce*
then obtain *rss* **where** *rss*: *Some rss* = *those* (*map* (λ*red*. *build-trees′ bs* ω *k red* (*I* ∪ {*red*})) *reds*)
using *those-Some* **by** (*metis* (*full-types*))
let *?h* = λ*f*. *case f of FBranch N fss* ⇒ *Some* (*FBranch N* (*fss* @ [*concat rss*])) | - ⇒ *None*
have ∀ *x* ∈ *set* (*map ?h pres*). ∃ *a*. *x* = *Some a*
using *pres*(*2*) **by** *auto*
then obtain *fs* **where** *fs*: *Some fs* = *those* (*map ?h pres*)
using *those-Some* **by** (*smt* (*verit*, *best*))
have ∀ *f* ∈ *set fs*. ∀ *t* ∈ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*
proof (*standard*, *standard*)
fix *f t*
assume *ft*: *f* ∈ *set fs t* ∈ *set* (*trees f*)
hence ∃ *x*. *x* ∈ *set pres* ∧ *Some f* ∈ *set* (*map ?h pres*)
using *those-map-exists*[*OF fs ft*(*1*)] **by** *blast*
then obtain *f-pre N fss* **where** *f-pre*: *f-pre* ∈ *set pres f-pre* = *FBranch N fss*
f = *FBranch N* (*fss* @ [*concat rss*])
using *pres*(*2*) **by** *force*
define *tss* **where** *tss*: *tss* = *map* (λ*fs*. *concat* (*map* (λ*f*. *trees f*) *fs*)) *fss*
have *trees* (*FBranch N* (*fss* @ [*concat rss*])) =
map (λ*ts*. *Branch N ts*) [*ts0* @ *ts1* . *ts0* <− *combinations tss*,
ts1 <− *combinations* [*concat* (*map* (λ*f*. *trees f*) (*concat rss*))]]
by (*subst tss*, *subst trees-append-singleton*, *simp*)
moreover have *t* ∈ *set* (*trees* (*FBranch N* (*fss* @ [*concat rss*])))
using *ft*(*2*) *f-pre*(*3*) **by** *blast*
ultimately obtain *ts0 ts1* **where** *tsx*: *t* = *Branch N* (*ts0* @ [*ts1*]) *ts0*

$\in$ *set* (*combinations tss*)
ts1 $\in$ *set* (*concat* (*map* (λf. *trees f*) (*concat rss*)))
by *fastforce*
then obtain *f-red* **where** *f-red*: *f-red* $\in$ *set* (*concat rss*) *ts1* $\in$ *set* (*trees f-red*)
by *auto*
obtain *fs-red red* **where** *red*: *Some fs-red* = *build-trees′ bs* ω *k red* (*I* $\cup$ {*red*})
f-red $\in$ *set fs-red red* $\in$ *set reds*
using *f-red*(*1*) *rss those-map-Some-concat-exists* **by** *fast*
then obtain *N-red fss-red* **where** *f-red* = *FBranch N-red fss-red*
using *build-trees′-termination wf-reds* **by** (*metis option.inject*)
then obtain *ts* **where** *ts*: *Branch N-red ts* = *ts1*
using *tsx*(*3*) *f-red* **by** *auto*
have (*k′*, *pre*, *red*) $\in$ *set ps′*
using *group-by-forall-v-exists-k* ‹((*k′*, *pre*), *reds*) $\in$ *set gs*› *gs* ‹*red* $\in$ *set reds*› **by** *fast*
hence *mem*: (*k′*, *pre*, *red*) $\in$ *set* (*p*#*ps*)
using *ps′* **by** (*metis filter-set member-filter*)
have *sound-ptrs* ω *bs*
using *prems*(*4*) *wf-trees-input-def* **by** *fastforce*
have *bounds*: *k′* < *k pre* < *length* (*bs*!*k′*) *red* < *length* (*bs*!*k*)
using *prered entry prems*(*6*,*7*) ‹*sound-ptrs* ω *bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** (*meson mem nth-mem*)+
have *completes*: *completes k* (*item* (*bs*!*k′*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))
using *prered entry prems*(*6*,*7*) ‹*sound-ptrs* ω *bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** (*metis mem nth-mem*)
have *transform*:
item-rule-head (*item* (*bs*!*k′*!*pre*)) = *item-rule-head* (*item e*)
item-rule-body (*item* (*bs*!*k′*!*pre*)) = *item-rule-body* (*item e*)
item-dot (*item* (*bs*!*k′*!*pre*)) + *1* = *item-dot* (*item e*)
next-symbol (*item* (*bs*!*k′*!*pre*)) = *Some* (*item-rule-head* (*item* (*bs*!*k*!*red*)))
is-complete (*item* (*bs*!*k*!*red*))
using *completes* **unfolding** *completes-def inc-item-def*
by (*auto simp*: *item-rule-head-def item-rule-body-def is-complete-def*)
have *Branch N ts0* $\in$ *set* (*trees* (*FBranch N fss*))
using *tss tsx*(*2*) **by** *simp*
hence *IH-pre*: *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k′*!*pre*)) (*Branch N ts0*)
using *prems*(*2*)[*OF entry prered ps′ gs* ‹((*k′*, *pre*), *reds*) $\in$ *set gs*› *wf-pre prems*(*5*)]
pres(*1*) *f-pre f-pre*(*3*) *bounds*(*1*,*2*) *prems*(*6*) **by** *fastforce*
have *IH-r*: *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*red*)) (*Branch N-red ts*)
using *prems*(*3*)[*OF entry prered ps′ gs* ‹((*k′*, *pre*), *reds*) $\in$ *set gs*› ‹*red* $\in$ *set reds*› - *prems*(*5*)]
bounds(*3*) *f-red*(*2*) *red ts wf-reds prems*(*6*) **by** *metis*
have *map root-tree* (*ts0* @ [*Branch N-red ts*]) = *map root-tree ts0* @ [*root-tree* (*Branch N-red ts*)]

by *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!*k*'!*pre*))) (*item-rule-body* (*item* (*bs*!*k*'!*pre*))) @ [*root-tree* (*Branch N-red ts*)]
using *IH-pre* **by** *simp*
also have ... = *take* (*item-dot* (*item* (*bs*!*k*'!*pre*))) (*item-rule-body* (*item* (*bs*!*k*'!*pre*))) @ [*item-rule-head* (*item* (*bs*!*k*!*red*))]
using *IH-r* **by** *simp*
also have ... = *take* (*item-dot* (*item e*)) (*item-rule-body* (*item e*))
using *transform* **by** (*auto simp*: *next-symbol-def is-complete-def split*: *if-splits*; *metis leI take-Suc-conv-app-nth*)
finally have *roots*: *map root-tree* (*ts0* @ [*Branch N-red ts*]) = *take* (*item-dot* (*item e*)) (*item-rule-body* (*item e*)) **.**
have *wf-item* $\mathcal{G}$ ω (*item* (*bs*!*k*!*red*))
using *prems*(*5*,*6*) *bounds*(*3*) **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def* **by** (*auto simp*: *items-def*)
moreover have *N-red* = *item-rule-head* (*item* (*bs*!*k*!*red*))
using *IH-r* **by** *fastforce*
moreover have *map root-tree ts* = *item-rule-body* (*item* (*bs*!*k*!*red*))
using *IH-r transform*(*5*) **by** (*auto simp*: *is-complete-def*)
ultimately have $\exists\, r \in$ *set* ($\mathfrak{R}$ $\mathcal{G}$). *N-red* = *rule-head r* $\wedge$ *map root-tree ts* = *rule-body r*
unfolding *wf-item-def item-rule-body-def item-rule-head-def* **by** *blast*
hence *wf-rule-tree* $\mathcal{G}$ (*Branch N-red ts*)
using *IH-r* **by** *simp*
hence *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) (*Branch N* (*ts0* @ [*Branch N-red ts*]))
using *transform*(*1*) *roots IH-pre entry* **by** *simp*
thus *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*
using *tsx*(*1*) *red ts* **by** *blast*
qed
moreover have *?g g* = *Some fs*
using *fs pres rss g* **by** (*auto*, *metis bind.bind-lunit*)
ultimately show $\exists$*fs*. *?g g* = *Some fs* $\wedge$ ($\forall$ *f* $\in$ *set fs*. $\forall$ *t* $\in$ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*)
by *blast*
qed
ultimately show $\exists$*fs*. *fso* = *Some fs* $\wedge$ ($\forall$ *f* $\in$ *set fs*. $\forall$ *t* $\in$ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*)
using *map-Some-P* **by** *auto*
qed
then obtain *fss* **where** *those* (*map ?g gs*) = *Some fss* $\forall$ *fs* $\in$ *set fss*. $\forall$ *f* $\in$ *set fs*. $\forall$ *t* $\in$ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*
using *those-Some-P* **by** *blast*
hence *build-trees'* *bs* ω *k i I* = *Some* (*concat fss*) $\forall$ *f* $\in$ *set* (*concat fss*). $\forall$ *t* $\in$ *set* (*trees f*). *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*k*!*i*)) *t*
using *simp* **by** *auto*
thus *?thesis*
using *prems*(*8*−*10*) **by** *auto*
qed

```
    qed
    done
  thus ?thesis
    by blast
qed

lemma wf-yield-tree-build-trees':
  assumes (bs, ω, k, i, I) ∈ wf-trees-input
  assumes wf-bins G ω bs
  assumes k < length bs i < length (bs!k) k ≤ length ω
  assumes build-trees' bs ω k i I = Some fs
  assumes f ∈ set fs
  assumes t ∈ set (trees f)
  shows wf-yield-tree ω (item (bs!k!i)) t
proof −
  have wf-yield-tree ω (item (bs!k!i)) t
    using assms
    apply (induction arbitrary: fs f t rule: build-trees'-induct[OF assms(1)])
    subgoal premises prems for bs ω k i I fs f t
    proof −
      define e where entry: e = bs!k!i
      consider (Null) pointer e = Null
        | (Pre) ∃ pre. pointer e = Pre pre
        | (PreRed) ∃ p ps. pointer e = PreRed p ps
        by (metis pointer.exhaust)
      thus ?thesis
      proof cases
        case Null
       hence simp: build-trees' bs ω k i I = Some ([FBranch (item-rule-head (item
e)) []])
          using entry by simp
        moreover have f = FBranch (item-rule-head (item e)) []
          using build-forest'-simps(1) Null prems(9,10) entry by auto
        ultimately have simp: t = Branch (item-rule-head (item e)) []
          using prems(11) by simp
        have sound-ptrs ω bs
          using prems(4) unfolding wf-trees-input-def by blast
        hence predicts (item e)
       using Null prems(6,7) nth-mem entry unfolding sound-ptrs-def sound-null-ptr-def
by blast
        thus ?thesis
          unfolding wf-yield-tree-def predicts-def using simp entry by (auto simp:
slice-empty)
      next
        case Pre
        then obtain pre where pre: pointer e = Pre pre
          by blast
        have sound: sound-ptrs ω bs
          using prems(4) unfolding wf-trees-input-def by blast
```

hence *bounds*: $k > 0$ *pre* $<$ *length* (*bs*!($k-1$))
using *entry pre prems*(6,7) **unfolding** *sound-ptrs-def sound-pre-ptr-def* **by** (*metis nth-mem*)+
have *scans*: *scans* ω *k* (*item* (*bs*!($k-1$)!*pre*)) (*item e*)
using *entry pre prems*($6-7$) ‹*sound-ptrs* ω *bs*› **unfolding** *sound-ptrs-def sound-pre-ptr-def* **by** *simp*
have *wf*: (*bs*, ω, $k-1$, *pre*, {*pre*}) $\in$ *wf-trees-input*
using *entry pre prems*(4) *wf-trees-input-pre* **by** *blast*
then obtain *pres* **where** *pres*: *build-trees'* *bs* ω ($k-1$) *pre* {*pre*} = *Some pres*
$\forall f \in$ *set pres*. $\exists$ *N fss*. *f* = *FBranch N fss*
using *build-trees'-termination wf* **by** *blast*
let *?g* = λf. *case f of FBranch N fss* $\Rightarrow$ *Some* (*FBranch N* (*fss* @ [[*FLeaf* (ω!($k-1$))]])) | - $\Rightarrow$ *None*
have *build-trees'* *bs* ω *k i I* = *those* (*map ?g pres*)
using *entry pre pres* **by** *simp*
hence *fs*: *Some fs* = *those* (*map ?g pres*)
using *prems*(9) **by** *simp*
then obtain *f-pre N fss* **where** *Nfss*: *f* = *FBranch N* (*fss* @ [[*FLeaf* (ω!($k-1$))]])
f-pre = *FBranch N fss f-pre* $\in$ *set pres*
using *those-map-FBranch-only fs pres*(2) *prems*(10) **by** *blast*
define *tss* **where** *tss*: *tss* = *map* (λ*fs*. *concat* (*map* (λf. *trees f*) *fs*)) *fss*
have *trees* (*FBranch N* (*fss* @ [[*FLeaf* (ω!($k-1$))]])) =
map (λ*ts*. *Branch N ts*) [*ts* @ [*Leaf* (ω!($k-1$))] . *ts* <− *combinations tss*]
by (*subst tss*, *subst trees-append-single-singleton*, *simp*)
moreover have *t* $\in$ *set* (*trees* (*FBranch N* (*fss* @ [[*FLeaf* (ω!($k-1$))]])))
using *Nfss*(1) *prems*(11) **by** *blast*
ultimately obtain *ts* **where** *ts*: *t* = *Branch N* (*ts* @ [*Leaf* (ω!($k-1$))]) $\wedge$ *ts* $\in$ *set* (*combinations tss*)
by *auto*
have *sound-ptrs* ω *bs*
using *prems*(4) **unfolding** *wf-trees-input-def* **by** *blast*
hence *pre* $<$ *length* (*bs*!($k-1$))
using *entry pre prems*(6,7) **unfolding** *sound-ptrs-def sound-pre-ptr-def* **by** (*metis nth-mem*)
moreover have $k-1$ $<$ *length bs*
by (*simp add*: *prems*(6) *less-imp-diff-less*)
moreover have *Branch N ts* $\in$ *set* (*trees* (*FBranch N fss*))
using *ts tss* **by** *simp*
ultimately have *IH*: *wf-yield-tree* ω (*item* (*bs*!($k-1$)!*pre*)) (*Branch N ts*)
using *prems*(1,2,4,5,8) *entry pre Nfss*(2,3) *wf pres*(1) **by** *simp*
have *transform*:
item-origin (*item* (*bs*!($k-1$)!*pre*)) $\leq$ *item-end* (*item* (*bs*!($k-1$)!*pre*))
item-end (*item* (*bs*!($k-1$)!*pre*)) = $k-1$
item-end (*item e*) = *k*
using *entry prems*(5,6,7) *bounds* **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def items-def wf-item-def*
by (*auto*, *meson less-imp-diff-less nth-mem*)

have *yield-tree t = concat (map yield-tree (ts @ [Leaf (ω!(k−1))]))*
by (*simp add*: *ts*)
also have *... = concat (map yield-tree ts) @ [ω!(k−1)]*
by *simp*
also have *... = slice (item-origin (item (bs!(k−1)!pre))) (item-end (item (bs!(k−1)!pre))) ω @ [ω!(k−1)]*
using *IH* **by** (*simp add*: *wf-yield-tree-def*)
also have *... = slice (item-origin (item (bs!(k−1)!pre))) (item-end (item (bs!(k−1)!pre)) + 1) ω*
using *slice-append-nth transform ‹k > 0› prems(8)*
by (*metis diff-less le-eq-less-or-eq less-imp-diff-less less-numeral-extra(1)*)
also have *... = slice (item-origin (item e)) (item-end (item (bs!(k−1)!pre)) + 1) ω*
using *scans* **unfolding** *scans-def inc-item-def* **by** *simp*
also have *... = slice (item-origin (item e)) k ω*
using *scans transform* **unfolding** *scans-def* **by** (*metis Suc-diff-1 Suc-eq-plus1 bounds(1)*)
also have *... = slice (item-origin (item e)) (item-end (item e)) ω*
using *transform* **by** *auto*
finally show *?thesis*
using *wf-yield-tree-def entry* **by** *blast*
next
case *PreRed*
then obtain *p ps* **where** *prered*: *pointer e = PreRed p ps*
by *blast*
define *ps′* **where** *ps′*: *ps′ = filter (λ(k′, pre, red). red ∉ I) (p#ps)*
define *gs* **where** *gs*: *gs = group-by (λ(k′, pre, red). (k′, pre)) (λ(k′, pre, red). red) ps′*
let *?g = λ((k′, pre), reds).*
do {
pres ← build-trees′ bs ω k′ pre {pre};
rss ← those (map (λred. build-trees′ bs ω k red (I ∪ {red})) reds);
those (map (λf.
case f of
FBranch N fss ⇒ Some (FBranch N (fss @ [concat rss]))
| - ⇒ None — impossible case
) pres)
}
have *simp*: *build-trees′ bs ω k i I = map-option concat (those (map ?g gs))*
using *entry prered ps′ gs* **by** (*subst build-trees′.simps*) (*auto simp del*: *filter.simps*)
have *∀ fso ∈ set (map ?g gs). ∃ fs. fso = Some fs ∧ (∀ f ∈ set fs. ∀ t ∈ set (trees f). wf-yield-tree ω (item (bs!k!i)) t)*
proof *standard*
fix *fso*
assume *fso ∈ set (map ?g gs)*
moreover have *∀ ps ∈ set gs. ∃ fs. ?g ps = Some fs ∧ (∀ f ∈ set fs. ∀ t ∈ set (trees f). wf-yield-tree ω (item (bs!k!i)) t)*
proof *standard*

fix g
assume $g \in$ *set gs*
then obtain k' *pre reds* **where** g: $((k', pre), reds) \in$ *set gs* $((k', pre), reds) = g$
by (*metis surj-pair*)
moreover have *wf-pre*: $(bs, \omega, k', pre, \{pre\}) \in$ *wf-trees-input*
using *wf-trees-input-prered-pre*[*OF prems*(4) *entry prered ps' gs g*(1)] **by** *blast*
ultimately obtain *pres* **where** *pres*: *build-trees' bs* ω k' *pre* $\{pre\}$ = *Some pres*
$\forall$ *f-pre* $\in$ *set pres*. $\exists$ *N fss*. *f-pre* = *FBranch N fss*
using *build-trees'-termination* **by** *blast*
have *wf-reds*: $\forall$ *red* $\in$ *set reds*. $(bs, \omega, k, red, I \cup \{red\}) \in$ *wf-trees-input*
using *wf-trees-input-prered-red*[*OF prems*(4) *entry prered ps' gs g*(1)] **by** *blast*
hence $\forall f \in$ *set* (*map* (λ*red*. *build-trees' bs* ω *k red* ($I \cup \{red\}$)) *reds*). $\exists$ *a*. f = *Some a*
using *build-trees'-termination* **by** *fastforce*
then obtain *rss* **where** *rss*: *Some rss* = *those* (*map* (λ*red*. *build-trees' bs* ω *k red* ($I \cup \{red\}$)) *reds*)
using *those-Some* **by** (*metis* (*full-types*))
let *?h* = λf. *case f of FBranch N fss* $\Rightarrow$ *Some* (*FBranch N* (*fss* @ [*concat rss*])) | - $\Rightarrow$ *None*
have $\forall x \in$ *set* (*map ?h pres*). $\exists$ *a*. x = *Some a*
using *pres*(2) **by** *auto*
then obtain *fs* **where** *fs*: *Some fs* = *those* (*map ?h pres*)
using *those-Some* **by** (*smt* (*verit*, *best*))
have $\forall f \in$ *set fs*. $\forall t \in$ *set* (*trees f*). *wf-yield-tree* ω (*item* (*bs*!*k*!*i*)) *t*
proof (*standard*, *standard*)
fix *f t*
assume *ft*: *f* $\in$ *set fs t* $\in$ *set* (*trees f*)
hence $\exists x$. $x \in$ *set pres* $\wedge$ *Some f* $\in$ *set* (*map ?h pres*)
using *those-map-exists*[*OF fs ft*(1)] **by** *blast*
then obtain *f-pre N fss* **where** *f-pre*: *f-pre* $\in$ *set pres f-pre* = *FBranch N fss*
f = *FBranch N* (*fss* @ [*concat rss*])
using *pres*(2) **by** *force*
define *tss* **where** *tss*: *tss* = *map* (λ*fs*. *concat* (*map* (λf. *trees f*) *fs*)) *fss*
have *trees* (*FBranch N* (*fss* @ [*concat rss*])) =
map (λ*ts*. *Branch N ts*) [*ts0* @ *ts1* . *ts0* <− *combinations tss*,
ts1 <− *combinations* [*concat* (*map* (λf. *trees f*) (*concat rss*))]]
by (*subst tss*, *subst trees-append-singleton*, *simp*)
moreover have $t \in$ *set* (*trees* (*FBranch N* (*fss* @ [*concat rss*])))
using *ft*(2) *f-pre*(3) **by** *blast*
ultimately obtain *ts0 ts1* **where** *tsx*: t = *Branch N* (*ts0* @ [*ts1*]) *ts0* $\in$ *set* (*combinations tss*)
ts1 $\in$ *set* (*concat* (*map* (λf. *trees f*) (*concat rss*)))
by *fastforce*
then obtain *f-red* **where** *f-red*: *f-red* $\in$ *set* (*concat rss*) *ts1* $\in$ *set* (*trees*

f-red)
by *auto*
obtain *fs-red red* **where** *red*: *Some fs-red* = *build-trees′ bs ω k red* (*I* ∪ {*red*})
f-red ∈ *set fs-red red* ∈ *set reds*
using *f-red*(*1*) *rss those-map-Some-concat-exists* **by** *fast*
then obtain *N-red fss-red* **where** *f-red* = *FBranch N-red fss-red*
using *build-trees′-termination wf-reds* **by** (*metis option.inject*)
then obtain *ts* **where** *ts*: *Branch N-red ts* = *ts1*
using *tsx*(*3*) *f-red* **by** *auto*
have (*k′*, *pre*, *red*) ∈ *set ps′*
using *group-by-forall-v-exists-k* ‹((*k′*, *pre*), *reds*) ∈ *set gs*› *gs* ‹*red* ∈ *set reds*› **by** *fast*
hence *mem*: (*k′*, *pre*, *red*) ∈ *set* (*p*#*ps*)
using *ps′* **by** (*metis filter-set member-filter*)
have *sound-ptrs ω bs*
using *prems*(*4*) *wf-trees-input-def* **by** *fastforce*
have *bounds*: *k′* < *k pre* < *length* (*bs*!*k′*) *red* < *length* (*bs*!*k*)
using *prered entry prems*(*6*,*7*) ‹*sound-ptrs ω bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** (*meson mem nth-mem*)+
have *completes*: *completes k* (*item* (*bs*!*k′*!*pre*)) (*item e*) (*item* (*bs*!*k*!*red*))
using *prered entry prems*(*6*,*7*) ‹*sound-ptrs ω bs*›
unfolding *sound-ptrs-def sound-prered-ptr-def* **by** (*metis mem nth-mem*)
have *transform*:
item-rule-head (*item* (*bs*!*k′*!*pre*)) = *item-rule-head* (*item e*)
item-rule-body (*item* (*bs*!*k′*!*pre*)) = *item-rule-body* (*item e*)
item-dot (*item* (*bs*!*k′*!*pre*)) + *1* = *item-dot* (*item e*)
next-symbol (*item* (*bs*!*k′*!*pre*)) = *Some* (*item-rule-head* (*item* (*bs*!*k*!*red*)))
is-complete (*item* (*bs*!*k*!*red*))
using *completes* **unfolding** *completes-def inc-item-def*
by (*auto simp*: *item-rule-head-def item-rule-body-def is-complete-def*)
have *Branch N ts0* ∈ *set* (*trees* (*FBranch N fss*))
using *tss tsx*(*2*) **by** *simp*
hence *IH-pre*: *wf-yield-tree ω* (*item* (*bs*!*k′*!*pre*)) (*Branch N ts0*)
using *prems*(*2*)[*OF entry prered ps′ gs* ‹((*k′*, *pre*), *reds*) ∈ *set gs*› *wf-pre prems*(*5*)]
pres(*1*) *f-pre f-pre*(*3*) *bounds*(*1*,*2*) *prems*(*6*,*8*) **by** *simp*
have *IH-r*: *wf-yield-tree ω* (*item* (*bs*!*k*!*red*)) (*Branch N-red ts*)
using *prems*(*3*)[*OF entry prered ps′ gs* ‹((*k′*, *pre*), *reds*) ∈ *set gs*› ‹*red* ∈ *set reds*› - *prems*(*5*)]
bounds(*3*) *f-red*(*2*) *red ts wf-reds prems*(*6*,*8*) **by** *metis*
have *wf1*:
item-origin (*item* (*bs*!*k′*!*pre*)) ≤ *item-end* (*item* (*bs*!*k′*!*pre*))
item-origin (*item* (*bs*!*k*!*red*)) ≤ *item-end* (*item* (*bs*!*k*!*red*))
using *prems*(*5*−*7*) *bounds* **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def items-def wf-item-def*
by (*metis length-map nth-map nth-mem order-less-trans*)+

have *wf2*:
item-end (*item* (*bs*!*k*!*red*)) = *k*
item-end (*item* (*bs*!*k*!*i*)) = *k*
using *prems*(5−7) *bounds* **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def items-def* **by** *simp-all*
have *yield-tree t* = *concat* (*map yield-tree* (*ts0* @ [*Branch N-red ts*]))
by (*simp add*: *ts tsx*(1))
also have ... = *concat* (*map yield-tree ts0*) @ *yield-tree* (*Branch N-red ts*)
by *simp*
also have ... = *slice* (*item-origin* (*item* (*bs*!*k*'!*pre*))) (*item-end* (*item* (*bs*!*k*'!*pre*))) *ω* @
slice (*item-origin* (*item* (*bs*!*k*!*red*))) (*item-end* (*item* (*bs*!*k*!*red*))) *ω*
using *IH-pre IH-r* **by** (*simp add*: *wf-yield-tree-def*)
also have ... = *slice* (*item-origin* (*item* (*bs*!*k*'!*pre*))) (*item-end* (*item* (*bs*!*k*!*red*))) *ω*
using *slice-concat wf1 completes-def completes* **by** (*metis* (*no-types*, *lifting*))
also have ... = *slice* (*item-origin* (*item e*)) (*item-end* (*item* (*bs*!*k*!*red*))) *ω*
using *completes* **unfolding** *completes-def inc-item-def* **by** *simp*
also have ... = *slice* (*item-origin* (*item e*)) (*item-end* (*item e*)) *ω*
using *wf2 entry* **by** *presburger*
finally show *wf-yield-tree ω* (*item* (*bs*!*k*!*i*)) *t*
using *wf-yield-tree-def entry* **by** *blast*
qed
moreover have *?g g* = *Some fs*
using *fs pres rss g* **by** (*auto*, *metis bind.bind-lunit*)
ultimately show ∃*fs*. *?g g* = *Some fs* ∧ (∀ *f* ∈ *set fs*. ∀ *t* ∈ *set* (*trees f*). *wf-yield-tree ω* (*item* (*bs*!*k*!*i*)) *t*)
by *blast*
qed
ultimately show ∃*fs*. *fso* = *Some fs* ∧ (∀ *f* ∈ *set fs*. ∀ *t* ∈ *set* (*trees f*). *wf-yield-tree ω* (*item* (*bs*!*k*!*i*)) *t*)
using *map-Some-P* **by** *auto*
qed
then obtain *fss* **where** *those* (*map ?g gs*) = *Some fss* ∀ *fs* ∈ *set fss*. ∀ *f* ∈ *set fs*. ∀ *t* ∈ *set* (*trees f*). *wf-yield-tree ω* (*item* (*bs*!*k*!*i*)) *t*
using *those-Some-P* **by** *blast*
hence *build-trees*′ *bs ω k i I* = *Some* (*concat fss*) ∀ *f* ∈ *set* (*concat fss*). ∀ *t* ∈ *set* (*trees f*). *wf-yield-tree ω* (*item* (*bs*!*k*!*i*)) *t*
using *simp* **by** *auto*
thus *?thesis*
using *prems*(9−11) **by** *auto*
qed
qed
done
thus *?thesis*
using *assms*(2) **by** *blast*

qed

theorem *wf-rule-root-yield-tree-build-trees*:
assumes *wf-bins* $\mathcal{G}$ ω *bs sound-ptrs* ω *bs length bs* = *length* ω + *1*
assumes *build-trees* $\mathcal{G}$ ω *bs* = *Some fs f* ∈ *set fs t* ∈ *set* (*trees f*)
shows *wf-rule-tree* $\mathcal{G}$ *t* ∧ *root-tree t* = $\mathfrak{S}$ $\mathcal{G}$ ∧ *yield-tree t* = ω
proof −
let *?k* = *length bs* − *1*
define *finished* **where** *finished-def*: *finished* = *filter-with-index* (*is-finished* $\mathcal{G}$ ω) (*items* (*bs*!*?k*))
have #: *Some fs* = *map-option concat* (*those* (*map* (λ(-, *i*). *build-trees*′ *bs* ω *?k i* {*i*}) *finished*))
using *assms*(*4*) *build-trees-def finished-def* **by** (*metis* (*full-types*))
then obtain *fss fs*′ **where** *fss*: *Some fss* = *those* (*map* (λ(-, *i*). *build-trees*′ *bs* ω *?k i* {*i*}) *finished*)
fs′ ∈ *set fss f* ∈ *set fs*′
using *map-option-concat-those-map-exists assms*(*5*) **by** *fastforce*
then obtain *x i* **where** ∗: (*x*,*i*) ∈ *set finished Some fs*′ = *build-trees*′ *bs* ω (*length bs* − *1*) *i* {*i*}
using *those-map-exists*[*OF fss*(*1*,*2*)] **by** *auto*
have *k*: *?k* < *length bs ?k* ≤ *length* ω
using *assms*(*3*) **by** *simp-all*
have *i*: *i* < *length* (*bs*!*?k*)
using *index-filter-with-index-lt-length* ∗ *items-def finished-def* **by** (*metis* (*no-types*, *opaque-lifting*) *length-map*)
have *x*: *x* = *item* (*bs*!*?k*!*i*)
using ∗ *i filter-with-index-nth items-def nth-map finished-def assms*(*3*) **by** *metis*
have *finished*: *is-finished* $\mathcal{G}$ ω *x*
using ∗ *filter-with-index-P finished-def* **by** *metis*
have {*i*} ⊆ {*0*..<*length* (*bs*!*?k*)}
using *atLeastLessThan-iff i* **by** *blast*
hence *wf*: (*bs*, ω, *?k*, *i*, {*i*}) ∈ *wf-trees-input*
unfolding *wf-trees-input-def* **using** *assms*(*2*) *i k*(*1*) **by** *simp*
hence *wf-item-tree*: *wf-item-tree* $\mathcal{G}$ (*item* (*bs*!*?k*!*i*)) *t*
using *wf-item-tree-build-trees*′ *assms*(*1*,*2*,*5*,*6*) *i k*(*1*) *x* ∗(*2*) *fss*(*3*) **by** *metis*
have *wf-item*: *wf-item* $\mathcal{G}$ ω (*item* (*bs*!*?k*!*i*))
using *k*(*1*) *i assms*(*1*) **unfolding** *wf-bins-def wf-bin-def wf-bin-items-def* **by** (*simp add*: *items-def*)
obtain *N fss* **where** *Nfss*: *f* = *FBranch N fss*
using *build-trees*′*-termination*[*OF wf*] **by** (*metis* ∗(*2*) *fss*(*3*) *option.inject*)
then obtain *ts* **where** *ts*: *t* = *Branch N ts*
using *assms*(*6*) **by** *auto*
hence *N* = *item-rule-head x*
map root-tree ts = *item-rule-body x*
using *finished wf-item-tree x* **by** (*auto simp*: *is-finished-def is-complete-def*)
hence ∃ *r* ∈ *set* (ℜ $\mathcal{G}$). *N* = *rule-head r* ∧ *map root-tree ts* = *rule-body r*
using *wf-item x* **unfolding** *wf-item-def item-rule-body-def item-rule-head-def* **by** *blast*
hence *wf-rule*: *wf-rule-tree* $\mathcal{G}$ *t*

using *wf-item-tree ts* **by** *simp*
have *root*: *root-tree t* = 𝔖 𝒢
using *finished ts* ‹*N* = *item-rule-head x*› **by** (*auto simp*: *is-finished-def*)
have *yield-tree t* = *slice* (*item-origin* (*item* (*bs*!*?k*!*i*))) (*item-end* (*item* (*bs*!*?k*!*i*))) *ω*
using *k i assms*(*1*,*6*) *wf wf-yield-tree-build-trees*′ *wf-yield-tree-def* ∗(*2*) *fss*(*3*)
by (*smt* (*verit*, *best*))
hence *yield*: *yield-tree t* = *ω*
using *finished x* **unfolding** *is-finished-def* **by** *simp*
show *?thesis*
using ∗ *wf-rule root yield assms*(*4*) **unfolding** *build-trees-def* **by** *simp*
qed

corollary *wf-rule-root-yield-tree-build-trees-Earley*$_L$:
assumes *wf-𝒢 𝒢 nonempty-derives 𝒢*
assumes *build-trees 𝒢 ω* (*Earley*$_L$ *𝒢 ω*) = *Some fs f* ∈ *set fs t* ∈ *set* (*trees f*)
shows *wf-rule-tree 𝒢 t* ∧ *root-tree t* = 𝔖 *𝒢* ∧ *yield-tree t* = *ω*
using *assms wf-rule-root-yield-tree-build-trees wf-bins-Earley*$_L$ *Earley*$_L$*-def*
length-Earley$_L$*-bins length-bins-Init*$_L$ *sound-mono-ptrs-Earley*$_L$
by (*metis dual-order.eq-iff*)

theorem *soundness-build-trees-Earley*$_L$:
assumes *wf-𝒢 𝒢 is-word 𝒢 ω nonempty-derives 𝒢*
assumes *build-trees 𝒢 ω* (*Earley*$_L$ *𝒢 ω*) = *Some fs f* ∈ *set fs t* ∈ *set* (*trees f*)
shows *derives 𝒢* [𝔖 *𝒢*] *ω*
proof −
let *?k* = *length* (*Earley*$_L$ *𝒢 ω*) − *1*
define *finished* **where** *finished-def*: *finished* = *filter-with-index* (*is-finished 𝒢 ω*) (*items* ((*Earley*$_L$ *𝒢 ω*)!*?k*))
have #: *Some fs* = *map-option concat* (*those* (*map* (λ(-, *i*). *build-trees*′ (*Earley*$_L$ *𝒢 ω*) *ω ?k i* {*i*}) *finished*))
using *assms*(*4*) *build-trees-def finished-def* **by** (*metis* (*full-types*))
then obtain *fss fs*′ **where** *fss*: *Some fss* = *those* (*map* (λ(-, *i*). *build-trees*′ (*Earley*$_L$ *𝒢 ω*) *ω ?k i* {*i*}) *finished*)
fs′ ∈ *set fss f* ∈ *set fs*′
using *map-option-concat-those-map-exists assms*(*5*) **by** *fastforce*
then obtain *x i* **where** ∗: (*x*,*i*) ∈ *set finished Some fs*′ = *build-trees*′ (*Earley*$_L$ *𝒢 ω*) *ω ?k i* {*i*}
using *those-map-exists*[*OF fss*(*1*,*2*)] **by** *auto*
have *k*: *?k* < *length* (*Earley*$_L$ *𝒢 ω*) *?k* ≤ *length ω*
by (*simp-all add*: *Earley*$_L$*-def assms*(*1*))
have *i*: *i* < *length* ((*Earley*$_L$ *𝒢 ω*) ! *?k*)
using *index-filter-with-index-lt-length* ∗ *items-def finished-def* **by** (*metis length-map*)
have *x*: *x* = *item* ((*Earley*$_L$ *𝒢 ω*)!*?k*!*i*)
using ∗ *i filter-with-index-nth items-def nth-map finished-def* **by** *metis*
have *finished*: *is-finished 𝒢 ω x*
using ∗ *filter-with-index-P finished-def* **by** *metis*
moreover have *x* ∈ *set* (*items* ((*Earley*$_L$ *𝒢 ω*) ! *?k*))
using *x* **by** (*auto simp*: *items-def*; *metis One-nat-def i imageI nth-mem*)

ultimately have *recognizing* (*bins* (*Earley*$_L$ $\mathcal{G}$ ω)) $\mathcal{G}$ ω
by (*meson k*(*1*) *kth-bin-sub-bins recognizing-def subsetD*)
thus *?thesis*
using *correctness-Earley*$_L$ *assms* **by** *blast*
qed

theorem *termination-build-tree-Earley*$_L$:
assumes *wf-*$\mathcal{G}$ $\mathcal{G}$ *nonempty-derives* $\mathcal{G}$ *derives* $\mathcal{G}$ [$\mathfrak{S}$ $\mathcal{G}$] ω
shows $\exists$*fs*. *build-trees* $\mathcal{G}$ ω (*Earley*$_L$ $\mathcal{G}$ ω) = *Some fs*
proof −
let *?k* = *length* (*Earley*$_L$ $\mathcal{G}$ ω) − *1*
define *finished* **where** *finished-def*: *finished* = *filter-with-index* (*is-finished* $\mathcal{G}$ ω) (*items* ((*Earley*$_L$ $\mathcal{G}$ ω)!*?k*))
have $\forall f \in$ *set finished*. (*Earley*$_L$ $\mathcal{G}$ ω, ω, *?k*, *snd f*, {*snd f*}) $\in$ *wf-trees-input*
proof *standard*
fix *f*
assume *a*: $f \in$ *set finished*
then obtain *x i* **where** ∗: (*x*,*i*) = *f*
by (*metis surj-pair*)
have *sound-ptrs* ω (*Earley*$_L$ $\mathcal{G}$ ω)
using *sound-mono-ptrs-Earley*$_L$ *assms* **by** *blast*
moreover have *?k* < *length* (*Earley*$_L$ $\mathcal{G}$ ω)
by (*simp add*: *Earley*$_L$*-def assms*(*1*))
moreover have *i* < *length* ((*Earley*$_L$ $\mathcal{G}$ ω)!*?k*)
using *index-filter-with-index-lt-length a* ∗ *items-def finished-def* **by** (*metis length-map*)
ultimately show (*Earley*$_L$ $\mathcal{G}$ ω, ω, *?k*, *snd f*, {*snd f*}) $\in$ *wf-trees-input*
using ∗ **unfolding** *wf-trees-input-def* **by** *auto*
qed
hence $\forall$*fso* $\in$ *set* (*map* (λ(-, *i*). *build-trees*′ (*Earley*$_L$ $\mathcal{G}$ ω) ω *?k i* {*i*}) *finished*). $\exists$*fs*. *fso* = *Some fs*
using *build-trees*′*-termination* **by** *fastforce*
then obtain *fss* **where** *fss*: *Some fss* = *those* (*map* (λ(-, *i*). *build-trees*′ (*Earley*$_L$ $\mathcal{G}$ ω) ω *?k i* {*i*}) *finished*)
by (*smt* (*verit*, *best*) *those-Some*)
then obtain *fs* **where** *fs*: *Some fs* = *map-option concat* (*those* (*map* (λ(-, *i*). *build-trees*′ (*Earley*$_L$ $\mathcal{G}$ ω) ω *?k i* {*i*}) *finished*))
by (*metis map-option-eq-Some*)
show *?thesis*
using *finished-def fss fs build-trees-def* **by** (*metis* (*full-types*))
qed

end
theory *Examples*
imports *Earley-Parser*
begin

10 Epsilon productions

definition *ε-free* :: *'a cfg* ⇒ *bool* **where**
ε-free $\mathcal{G}$ ⟷ (∀ *r* ∈ *set* (ℜ $\mathcal{G}$). *rule-body r* ≠ [])

lemma *ε-free-impl-non-empty-sentence-deriv*:
ε-free $\mathcal{G}$ ⟹ *a* ≠ [] ⟹ ¬ *Derivation* $\mathcal{G}$ *a D* []
proof (*induction length D arbitrary*: *a D rule*: *nat-less-induct*)
case *1*
show *?case*
proof (*rule ccontr*)
assume *assm*: ¬ ¬ *Derivation* $\mathcal{G}$ *a D* []
show *False*
proof (*cases D* = [])
case *True*
then show *?thesis*
using *1.prems*(*2*) *assm* **by** *auto*
next
case *False*
then obtain *d D'* α **where** ∗:
D = *d* # *D' Derives1* $\mathcal{G}$ *a* (*fst d*) (*snd d*) α *Derivation* $\mathcal{G}$ α *D'* [] *snd d* ∈ *set* (ℜ $\mathcal{G}$)
using *list.exhaust assm Derives1-def* **by** (*metis Derivation.simps*(*2*))
show *?thesis*
proof *cases*
assume α = []
thus *?thesis*
using ∗(*2*,*4*) *Derives1-split ε-free-def rule-body-def 1.prems*(*1*) **by** (*metis append-is-Nil-conv*)
next
assume ¬ α = []
thus *?thesis*
using ∗(*1*,*3*) *1.hyps 1.prems*(*1*) **by** *auto*
qed
qed
qed
qed

lemma *ε-free-impl-non-empty-deriv*:
ε-free $\mathcal{G}$ ⟹ ∀ *N* ∈ *set* (ℜ $\mathcal{G}$). ¬ *derives* $\mathcal{G}$ [*N*] []
using *ε-free-impl-non-empty-sentence-deriv derives-implies-Derivation* **by** (*metis not-Cons-self2*)

lemma *nonempty-deriv-impl-ε-free*:
assumes ∀ *N* ∈ *set* (ℜ $\mathcal{G}$). ¬ *derives* $\mathcal{G}$ [*N*] [] ∀ (*N*, α) ∈ *set* (ℜ $\mathcal{G}$). *N* ∈ *set* (ℜ $\mathcal{G}$)
shows *ε-free* $\mathcal{G}$
proof (*rule ccontr*)
assume ¬ *ε-free* $\mathcal{G}$

then obtain N α **where** $*$: $(N, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G})$ *rule-body* $(N, \alpha) = []$
unfolding *ε-free-def* **by** *auto*
hence *derives1* $\mathcal{G}$ $[N]$ []
unfolding *derives1-def rule-body-def* **by** *auto*
hence *derives* $\mathcal{G}$ $[N]$ []
by *auto*
moreover have $N \in set\ (\mathfrak{N}\ \mathcal{G})$
using $*(1)$ *assms*(2) **by** *blast*
ultimately show *False*
using *assms*(1) **by** *blast*
qed

lemma *nonempty-deriv-iff-ε-free*:
assumes $\forall (N, \alpha) \in set\ (\mathfrak{R}\ \mathcal{G}).\ N \in set\ (\mathfrak{N}\ \mathcal{G})$
shows $(\forall N \in set\ (\mathfrak{N}\ \mathcal{G}).\ \neg\ derives\ \mathcal{G}\ [N]\ []) \longleftrightarrow \varepsilon\text{-}free\ \mathcal{G}$
using *ε-free-impl-non-empty-deriv nonempty-deriv-impl-ε-free*[*OF - assms*] **by** *blast*

11 Example 1: Addition

datatype *t1* = *x* | *plus*
datatype *n1* = *S*
datatype *s1* = *Terminal t1* | *Nonterminal n1*

definition *nonterminals1* :: *s1 list* **where**
nonterminals1 = [*Nonterminal S*]

definition *terminals1* :: *s1 list* **where**
terminals1 = [*Terminal x*, *Terminal plus*]

definition *rules1* :: *s1 rule list* **where**
rules1 = [
(*Nonterminal S*, [*Terminal x*]),
(*Nonterminal S*, [*Nonterminal S*, *Terminal plus*, *Nonterminal S*])
]

definition *start-symbol1* :: *s1* **where**
start-symbol1 = *Nonterminal S*

definition *cfg1* :: *s1 cfg* **where**
cfg1 = *CFG nonterminals1 terminals1 rules1 start-symbol1*

definition *inp1* :: *s1 list* **where**
inp1 = [*Terminal x*, *Terminal plus*, *Terminal x*, *Terminal plus*, *Terminal x*]

lemmas *cfg1-defs* = *cfg1-def nonterminals1-def terminals1-def rules1-def start-symbol1-def*

lemma *wf-G1*:
wf-G cfg1

by (*auto simp*: *wf-$\mathcal{G}$-defs cfg1-defs*)

lemma *is-word-inp1*:
is-word cfg1 inp1
by (*auto simp*: *is-word-def is-terminal-def cfg1-defs inp1-def*)

lemma *nonempty-derives1*:
nonempty-derives cfg1
by (*auto simp*: *ε-free-def cfg1-defs rule-body-def nonempty-derives-def ε-free-impl-non-empty-deriv*)

lemma *correctness1*:
recognizing (*bins* (*Earley$_L$ cfg1 inp1*)) *cfg1 inp1* $\longleftrightarrow$ *derives cfg1* [$\mathfrak{S}$ *cfg1*] *inp1*
using *correctness-Earley$_L$ wf-$\mathcal{G}$1 is-word-inp1 nonempty-derives1* **by** *blast*

lemma *wf-tree1*:
assumes *build-tree cfg1 inp1* (*Earley$_L$ cfg1 inp1*) = *Some t*
shows *wf-rule-tree cfg1 t* $\wedge$ *root-tree t* = $\mathfrak{S}$ *cfg1* $\wedge$ *yield-tree t* = *inp1*
using *assms nonempty-derives1 wf-$\mathcal{G}$1 wf-rule-root-yield-tree-build-tree-Earley$_L$*
by *blast*

lemma *correctness-tree1*:
($\exists$ *t*. *build-tree cfg1 inp1* (*Earley$_L$ cfg1 inp1*) = *Some t*) $\longleftrightarrow$ *derives cfg1* [$\mathfrak{S}$ *cfg1*] *inp1*
using *correctness-build-tree-Earley$_L$ is-word-inp1 nonempty-derives1 wf-$\mathcal{G}$1* **by** *blast*

lemma *wf-trees1*:
assumes *build-trees cfg1 inp1* (*Earley$_L$ cfg1 inp1*) = *Some fs f* $\in$ *set fs t* $\in$ *set* (*trees f*)
shows *wf-rule-tree cfg1 t* $\wedge$ *root-tree t* = $\mathfrak{S}$ *cfg1* $\wedge$ *yield-tree t* = *inp1*
using *assms nonempty-derives1 wf-$\mathcal{G}$1 wf-rule-root-yield-tree-build-trees-Earley$_L$*
by *blast*

lemma *soundness-trees1*:
assumes *build-trees cfg1 inp1* (*Earley$_L$ cfg1 inp1*) = *Some fs f* $\in$ *set fs t* $\in$ *set* (*trees f*)
shows *derives cfg1* [$\mathfrak{S}$ *cfg1*] *inp1*
using *assms is-word-inp1 nonempty-derives1 soundness-build-trees-Earley$_L$ wf-$\mathcal{G}$1*
by *blast*

12 Example 2: Cyclic reduction pointers

datatype *t2* = *x*
datatype *n2* = *A* | *B*
datatype *s2* = *Terminal t2* | *Nonterminal n2*

definition *nonterminals2* :: *s2 list* **where**
nonterminals2 = [*Nonterminal A*, *Nonterminal B*]

definition *terminals2* :: *s2 list* **where**
terminals2 = [*Terminal x*]

definition *rules2* :: *s2 rule list* **where**
rules2 = [
(*Nonterminal B*, [*Nonterminal A*]),
(*Nonterminal A*, [*Nonterminal B*]),
(*Nonterminal A*, [*Terminal x*])
]

definition *start-symbol2* :: *s2* **where**
start-symbol2 = *Nonterminal A*

definition *cfg2* :: *s2 cfg* **where**
cfg2 = *CFG nonterminals2 terminals2 rules2 start-symbol2*

definition *inp2* :: *s2 list* **where**
inp2 = [*Terminal x*]

lemmas *cfg2-defs* = *cfg2-def nonterminals2-def terminals2-def rules2-def start-symbol2-def*

lemma *wf-𝒢2*:
wf-𝒢 cfg2
by (*auto simp*: *wf-𝒢-defs cfg2-defs*)

lemma *is-word-inp2*:
is-word cfg2 inp2
by (*auto simp*: *is-word-def is-terminal-def cfg2-defs inp2-def*)

lemma *nonempty-derives2*:
nonempty-derives cfg2
by (*auto simp*: *ε-free-def cfg2-defs rule-body-def nonempty-derives-def ε-free-impl-non-empty-deriv*)

lemma *correctness2*:
recognizing (*bins* (*Earley*$_L$ *cfg2 inp2*)) *cfg2 inp2* ⟷ *derives cfg2* [𝔖 *cfg2*] *inp2*
using *correctness-Earley*$_L$ *wf-𝒢2 is-word-inp2 nonempty-derives2* **by** *blast*

lemma *wf-tree2*:
assumes *build-tree cfg2 inp2* (*Earley*$_L$ *cfg2 inp2*) = *Some t*
shows *wf-rule-tree cfg2 t* ∧ *root-tree t* = 𝔖 *cfg2* ∧ *yield-tree t* = *inp2*
using *assms nonempty-derives2 wf-𝒢2 wf-rule-root-yield-tree-build-tree-Earley*$_L$
by *blast*

lemma *correctness-tree2*:
(∃ *t*. *build-tree cfg2 inp2* (*Earley*$_L$ *cfg2 inp2*) = *Some t*) ⟷ *derives cfg2* [𝔖 *cfg2*] *inp2*
using *correctness-build-tree-Earley*$_L$ *is-word-inp2 nonempty-derives2 wf-𝒢2* **by** *blast*

lemma *wf-trees2*:
assumes *build-trees cfg2 inp2* (*Earley*$_L$ *cfg2 inp2*) = *Some fs f* ∈ *set fs t* ∈ *set* (*trees f*)
shows *wf-rule-tree cfg2 t* ∧ *root-tree t* = 𝔖 *cfg2* ∧ *yield-tree t* = *inp2*
using *assms nonempty-derives2 wf-𝒢2 wf-rule-root-yield-tree-build-trees-Earley*$_L$
by *blast*

lemma *soundness-trees2*:
assumes *build-trees cfg2 inp2* (*Earley*$_L$ *cfg2 inp2*) = *Some fs f* ∈ *set fs t* ∈ *set* (*trees f*)
shows *derives cfg2* [𝔖 *cfg2*] *inp2*
using *assms is-word-inp2 nonempty-derives2 soundness-build-trees-Earley*$_L$ *wf-𝒢2*
by *blast*

end

References

[1] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94102, 1970.

[2] C. B. Jones. Formal development of correct algorithms: An example based on earley's recogniser. In *Proceedings of ACM Conference on Proving Assertions about Programs*, page 150169, New York, NY, USA, 1972. Association for Computing Machinery.

[3] S. Obua. Local lexing. *Archive of Formal Proofs*, 2017. https://isa-afp.org/entries/LocalLexing.html, Formal proof development.

[4] S. Obua, P. Scott, and J. Fleuriot. Local lexing, 2017.

[5] E. Scott. Sppf-style parsing from earley recognisers. *Electronic Notes in Theoretical Computer Science*, 203(2):53–67, 2008. Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).