Greibach's Hardest Context-Free Language

Tobias Nipkow

September 17, 2026

Abstract

Formalization of Greibach's [1] *hardest context-free language theorem*: There is a "hardest" context-free language L_0 such that every context-free language is an inverse homomorphic image of L_0.

1 Greibach's Hardest Context-Free Language

theory *Greibach_Hardest*
imports
Context_Free_Grammar.Context_Free_Language
Greibach_Normal_Form.Greibach_Normal_Form
Dyck_Language.Dyck_Language
begin

Formalization of Theorem 2.1 by Sheila Greibach [1]:

Theorem For every context-free language L there is a homomorphism h with $L - \{\varepsilon\} = h^{-1}(L_0 - \{\varepsilon\})$, where L_0 is one fixed "hardest" context-free language.

The construction encodes the leftmost derivations of a Greibach Normal Form (GNF) grammar as a nondeterministic Dyck word (the encoding homomorphism *enc_h*); the general case is reduced to GNF using the AFP entry `Greibach_Normal_Form` (the function *gnf_of*).

abbreviation *Cons_power* :: *'a* ⇒ *nat* ⇒ *'a list* (**infixl** #^ *70*) **where**
a #^ *n* ≡ *replicate n a*

lemma *bal_stk_append_split*:
assumes *bal_stk s* (*xs* @ *ys*) = (*t*, [])
obtains *s'* **where** *bal_stk s xs* = (*s'*, []) **and** *bal_stk s' ys* = (*t*, [])
using *assms* **by** (*auto simp*: *bal_stk_append split*: *prod.splits if_splits*)
lemma *bal_stk_replicate_Open*: *bal_stk s* (*Open a* #^*i*) = (*a*#^*i* @ *s*, [])
by (*induction i arbitrary*: *s*) (*auto simp*: *replicate_append_same*)
lemma *bal_stk_replicate_Close*: *bal_stk* (*a*#^*i* @ *t*) (*Close a* #^ *i*) = (*t*, [])
by (*induction i arbitrary*: *t*) *auto*
lemma *bal_stk_replicate_Close_inv*:

bal_stk t (replicate i (Close a) @ rest) = (s, []) ⟹
∃ t'. t = replicate i a @ t' ∧ bal_stk t' rest = (s, [])
proof (*induction i arbitrary*: *t*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc i*)
from *Suc.prems* **obtain** *b t1* **where** *t*: *t = b # t1* **by** (*cases t*) *auto*
with *Suc.prems* **have** *a = b bal_stk t1 (replicate i (Close a) @ rest) = (s, [])*
by (*auto split*: *if_splits*)
with *Suc.IH* **obtain** *t'* **where** *t1 = replicate i a @ t' bal_stk t' rest = (s, [])* **by** *blast*
with *t ‹a = b›* **show** *?case* **by** *auto*
qed
lemmas *derives_Nt_map_TmD = derives_start1*
lemma *Lang_lfp_unfold*:
Lang_lfp P A = (⋃α ∈ Rhss P A. inst_syms (Lang_lfp P) α)
unfolding *Lang_lfp_def*
using *fun_cong*[*OF subst_lang_def*[*of P lfp*(*subst_lang P*)], *of A*,*symmetric*]
by (*metis lfp_unfold*[*OF mono_if_omega_cont*[*OF omega_cont_Lang_lfp*]])
corollary *Lang_unfold*: *Lang P A = (⋃α ∈ Rhss P A. inst_syms (Lang P) α)*
by(*fact Lang_lfp_unfold*[*unfolded Lang_lfp_eq_Lang*])
lemma *concats_simps*[*simp*]:
concats [] = {[]}
concats (L#Ls) = L @@ concats Ls
by(*auto simp*: *concats_def*)
lemma *concats_append*[*simp*]: *concats (Ls1 @ Ls2) = concats Ls1 @@ concats Ls2*
by (*simp add*: *concats_def foldr_conc_conc*)
lemma *inst_sym_simps*[*simp*]:
inst_sym L (Tm a) = {[a]}
inst_sym L (Nt A) = L A
by(*auto simp*: *inst_sym_def*)
lemma *inst_syms_Nil*[*simp*]: *inst_syms L [] = {[]}*
by(*simp add*: *inst_syms_def*)
lemma *inst_syms_Cons*[*simp*]: *inst_syms L (s # β) = inst_sym L s @@ inst_syms L β*
by(*simp add*: *inst_syms_def*)
lemma *inst_syms_append*[*simp*]: *inst_syms L (α @ β) = inst_syms L α @@ inst_syms L β*
by(*simp add*: *inst_syms_def*)
lemma *Lang_I*: *(A, α) ∈ P ⟹ w ∈ inst_syms (Lang P) α ⟹ w ∈ Lang P A*
by (*subst Lang_unfold*) (*auto simp*: *Rhss_def*)
lemma *Lang_subset_if*:
assumes *⋀A α. (A,α) ∈ P ⟹ inst_syms R α ⊆ R A*
shows *Lang P A ⊆ R A*
proof −
have *subst_lang P R ≤ R*
using *assms* **by** (*fastforce simp*: *subst_lang_def le_fun_def Rhss_def*)
hence *lfp (subst_lang P) ≤ R* **by** (*rule lfp_lowerbound*)

hence *Lang_lfp P* $\leq$ *R* **by** (*simp add*: *Lang_lfp_def*)
thus *?thesis* **by** (*simp add*: *Lang_lfp_eq_Lang le_fun_def*)
qed

1.1 The hardest language L_0

1.1.1 The terminal alphabet of L_0

Greibach's alphabet is $T = \{a_1,\ a_2,\ |a_1,\ |a_2,\ c,\ ¢\}$ together with a fresh separator *d*. There are two bracket *kinds* a_1, a_2, modelled by the type *t0_A*. A bracket letter is *Aa* applied to an opening or closing bracket (of type *'a bracket*) over a kind: thus *Aa* (*Open A1*) = a_1, *Aa* (*Close A1*) = $|a_1$, and analogously for a_2. The remaining letters are *Cc* for *c*, *Ce* for ¢, and *Dd* for *d*.

datatype *t0_A* = *A1* | *A2*

datatype *t0* = *Aa t0_A bracket* | *Cc* | *Ce* | *Dd*

1.1.2 The Dyck set D on two letters

D is the Dyck set generated by $S \to SS \mid a_1\ S\ |a_1 \mid a_2\ S\ |a_2 \mid \varepsilon$, i.e. the set of balanced words over the two bracket pairs (a_1, $|a_1$) and (a_2, $|a_2$). As each bracket letter *Aa b* already carries a *t0_A bracket*, we reuse *bal* directly via the projection *brk*; no separate integer tagging is needed.

fun *brk* :: *t0* $\Rightarrow$ *t0_A bracket* **where**
brk (*Aa b*) = *b*

definition *bracks* :: *t0 set* **where**
bracks = *Aa* ' *UNIV*

definition *D* :: *t0 list set* **where**
D = {*w*. *set w* $\subseteq$ *bracks* $\wedge$ *bal* (*map brk w*)}

1.1.3 The hardest language L_0

The alphabet T (note: $d \notin T$, since *d* separates the blocks):

definition *Talph* :: *t0 set* **where**
Talph = *insert Cc* (*insert Ce bracks*)

A single block *x c y c z d*:

definition *blk* :: *t0 list* $\times$ *t0 list* $\times$ *t0 list* $\Rightarrow$ *t0 list* **where**
blk = (λ(*x*,*y*,*z*). *x* @ *Cc* # *y* @ *Cc* # *z* @ [*Dd*])

$L_0 = \{\varepsilon\} \cup \{x_1\ c\ y_1\ c\ z_1\ d\ \ldots\ x_n\ c\ y_n\ c\ z_n\ d \mid n \geq 1,\ y_1 \ldots y_n \in ¢D,\ x_i\ z_i \in T^*,\ y_i \in \{a_1,a_2,|a_1,|a_2\}^*$ *for* $i \geq 2\}$. The n blocks are given by a non-empty list of triples $bs = [(x_1,y_1,z_1), \ldots]$; the constraint $y_i \in$ *brackets*

for $i \geq 2$ is *tl bs*; and $y_1 \ldots y_n \in \mathfrak{c}D$ means the concatenation of the ys is $\mathfrak{c}$ followed by a word of D.

definition *L0* :: *t0 list set* **where**
L0 = {[]} ∪
{ *concat* (*map blk bs*) | *bs*.
bs ≠ [] ∧
(∀ (*x*,*y*,*z*) ∈ *set bs*. *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph*) ∧
(∀ (*x*,*y*,*z*) ∈ *set* (*tl bs*). *set y* ⊆ *bracks*) ∧
(∃ *v* ∈ *D*. *concat* (*map* (λ(*x*,*y*,*z*). *y*) *bs*) = *Ce* # *v*) }

1.2 Homomorphisms

A (string) homomorphism is determined by its action h on single letters and lifts to words by $\lambda w.\ concat\ (map\ h\ w)$. The inverse image of a language under it:

definition *inv_hom* :: ('*a* ⇒ '*b list*) ⇒ '*b list set* ⇒ '*a list set* **where**
inv_hom h L = {*w*. *concat* (*map h w*) ∈ *L*}

The nonterminals Y_1, …, Y_n (with $Y_1 = S$) are identified with natural-number indices via an injective map *idx* (only injectivity matters; the specific values, in particular *idx S*, are immaterial). The nonterminal Y_i is encoded in unary: it is *pushed* as $a_1\ {a_2}^i\ a_1$ and *popped* as $|a_1\ |{a_2}^i\ |a_1$.

definition *pushcode* :: *nat* ⇒ *t0 list* **where**
pushcode i = *Aa* (*Open A1*) # (*Aa* (*Open A2*) #^ *i*) @ [*Aa* (*Open A1*)]

definition *popcode* :: *nat* ⇒ *t0 list* **where**
popcode i = *Aa* (*Close A1*) # (*Aa* (*Close A2*) #^ *i*) @ [*Aa* (*Close A1*)]

$\xi(p)$ for a standard-form production $p = (Y_i \to a\ Y_{j1}\ \ldots\ Y_{jm})$: pop Y_i, then push the right-hand-side nonterminals. The pushes are in *reverse* order so that, with the standard Dyck convention (a closing bracket matches the nearest *left* opening, i.e. LIFO), the stack is encoded with its top (the leftmost / next-to-expand nonterminal) at the *right*: expanding the top Y_i pops it and the new nonterminals $Y_{j1}\ \ldots\ Y_{jm}$ become the new top, Y_{j1} rightmost.

definition *xi* :: ('*n* ⇒ *nat*) ⇒ ('*n*,'*t*) *prod* ⇒ *t0 list* **where**
xi idx p =
popcode (*idx* (*fst p*)) @
concat (*map* (λ*s*. *case s of Nt B* ⇒ *pushcode* (*idx B*) | *Tm* _ ⇒ []) (*rev* (*tl* (*snd p*))))

$\xi\hat{}(p)$: for productions of the start symbol S, prepend $\mathfrak{c}$ and the push code of S (which initializes the stack with S and the marker $\mathfrak{c}$).

definition *xihat* :: ('*n* ⇒ *nat*) ⇒ '*n* ⇒ ('*n*,'*t*) *prod* ⇒ *t0 list* **where**
xihat idx S p = (*if fst p* = *S then Ce* # *pushcode* (*idx S*) @ *xi idx p else xi idx p*)

$h(a) = c\ \xi\widehat{}(p_1)\ c\ \ldots\ c\ \xi\widehat{}(p_m)\ c\ d$, where $p_1, \ldots, p_m$ are all productions whose right-hand side starts with the terminal a.

definition *enc_h* :: ('*n* ⇒ *nat*) ⇒ '*n* ⇒ ('*n*,'*t*) *prods* ⇒ '*t* ⇒ *t0 list* **where**
enc_h idx S ps a =
concat (*map* (λ*p*. *Cc* # *xihat idx S p*)
(*filter* (λ*p*. ∃ *Bs*. *snd p* = *Tm a* # *map Nt Bs*) *ps*))
@ [*Cc*, *Dd*]

1.3 Basic machinery

1.3.1 Stack machinery: running the encoded brackets through *bal_stk*

The bracket image of a word over *t0*:

abbreviation *brks* :: *t0 list* ⇒ *t0_A bracket list* **where**
brks w ≡ *map brk w*

A single nonterminal Y_i occupies the stack fragment *frag i*; a whole nonterminal stack (top first) is encoded by *stkenc*. With the cons-stack of *bal_stk* (top = head), reading *pushcode i* pushes *frag i* and reading *popcode i* pops it.

definition *frag* :: *nat* ⇒ *t0_A list* **where**
frag i = *A1* # *A2*#^*i* @ [*A1*]

definition *stkenc* :: *nat list* ⇒ *t0_A list* **where**
stkenc cs = *concat* (*map frag cs*)

lemma *bal_stk_pushcode*: *bal_stk s* (*brks* (*pushcode i*)) = (*frag i* @ *s*, [])
by (*simp add*: *bal_stk_append bal_stk_replicate_Open frag_def pushcode_def*)

lemma *bal_stk_popcode*: *bal_stk* (*frag i* @ *s*) (*brks* (*popcode i*)) = (*s*, [])
by (*simp add*: *popcode_def frag_def bal_stk_append bal_stk_replicate_Close*)

Reading the concatenated push codes of a list of nonterminal indices pushes the whole encoded stack (note the reversal: the first index ends up deepest).

lemma *bal_stk_pushes*:
bal_stk s (*brks* (*concat* (*map pushcode is*))) = (*stkenc* (*rev is*) @ *s*, [])
proof (*induction is arbitrary*: *s*)
case *Nil* **show** *?case* **by** (*simp add*: *stkenc_def*)
next
case (*Cons i is*)
thus *?case* **by** (*simp add*: *bal_stk_append bal_stk_pushcode stkenc_def*)
qed

The right-hand side of *xi* as an explicit pop followed by pushes.

lemma *xi_eq*:
snd p = *Tm a* # *map Nt Bs* ⟹

xi idx p = popcode (idx (fst p)) @ concat (map pushcode (rev (map idx Bs)))
by (*simp add*: *xi_def rev_map o_def*)

Running $\xi(p)$ for a standard-form production $p = (A \to a\ B_1\ \ldots\ B_m)$ on a stack whose top is A: it pops A and pushes $B_1\ \ldots\ B_m$ (with B_1 becoming the new top), exactly modelling one leftmost derivation step.

lemma *bal_stk_xi*:
assumes *snd p = Tm a # map Nt Bs*
shows *bal_stk (frag (idx (fst p)) @ s) (brks (xi idx p)) = (stkenc (map idx Bs) @ s, [])*
by (*simp add*: *xi_eq*[*OF assms*] *bal_stk_append bal_stk_popcode bal_stk_pushes*)

1.3.2 Letters occurring in the encoded words

lemma *set_pushcode*: *set (pushcode i) ⊆ bracks* **by** (*auto simp*: *pushcode_def bracks_def*)
lemma *set_popcode*: *set (popcode i) ⊆ bracks* **by** (*auto simp*: *popcode_def bracks_def*)

A ξ-code uses only the four bracket letters; a $\xi\hat{}$-code may additionally use ¢.

lemma *set_xi*: *set (xi idx p) ⊆ bracks*
unfolding *xi_def* **using** *set_popcode set_pushcode* **by** (*auto*)

lemma *set_xihat*: *set (xihat idx S p) ⊆ insert Ce bracks*
using *set_xi*[*of idx p*] *set_pushcode*[*of idx S*] **by** (*auto simp*: *xihat_def*)

For productions not expanding the start symbol, $\xi\hat{}$ coincides with ξ.

lemma *map_xihat_no_S*:
∀ p ∈ set ps. fst p ≠ S ⟹ map (xihat idx S) ps = map (xi idx) ps
by (*induction ps*) (*auto simp*: *xihat_def*)

1.3.3 Block decomposition of the encoded word

lemma *set_xihat_Talph*: *set (xihat idx S q) ⊆ Talph*
using *set_xihat*[*of idx S q*] **by** (*auto simp*: *Talph_def bracks_def*)

lemma *set_Cc_xihat*: *set (Cc # xihat idx S q) ⊆ Talph*
using *set_xihat_Talph*[*of idx S q*] **by** (*auto simp*: *Talph_def*)

lemma *set_concat_Cc_xihat*: *set (concat (map (λq. Cc # xihat idx S q) qs)) ⊆ Talph*
using *set_Cc_xihat*[**where** *idx=idx*] **by** *auto*

lemma *set_concat_xihat_Cc*: *set (concat (map (λq. xihat idx S q @ [Cc]) qs)) ⊆ Talph*
using *set_Cc_xihat*[**where** *idx=idx*] **by** *auto*

Single-c separators: peeling a leading c off the c d-terminated block body turns the c-prefixed fragments into c-suffixed ones. This is the key rewriting

for splitting a block at a chosen production.

lemma *cc_shift2*:
  concat (*map* (λ*p*. *Cc* # *xihat idx S p*) *qs*) @ [*Cc*, *Dd*]
  = *Cc* # *concat* (*map* (λ*q*. *xihat idx S q* @ [*Cc*]) *qs*) @ [*Dd*]
  by (*induction qs*) *auto*

One block of *enc_h idx S P a*: any production p of P whose right-hand side starts with *Tm a* can be selected as the middle y, the surrounding x z (the other c-delimited fragments) being words over T.

lemma *enc_h_block*:
  assumes *pin*: *p* ∈ *set P* **and** *snd_p*: *snd p* = *Tm a* # *map Nt Bs*
  shows ∃ *x z*. *enc_h idx S P a* = *blk* (*x*, *xihat idx S p*, *z*) ∧ *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph*
proof −
  have *p* ∈ *set* (*filter* (λ*p*. ∃ *Bs*. *snd p* = *Tm a* # *map Nt Bs*) *P*) **using** *pin snd_p* **by** *auto*
  then obtain *qs1 qs2* **where** *split*: *filter* (λ*p*. ∃ *Bs*. *snd p* = *Tm a* # *map Nt Bs*) *P* = *qs1* @ *p* # *qs2*
    by (*meson split_list*)
  define *x* **where** *x* = *concat* (*map* (λ*p*. *Cc* # *xihat idx S p*) *qs1*)
  define *z* **where** *z* = *concat* (*map* (λ*q*. *xihat idx S q* @ [*Cc*]) *qs2*)
  have *enc_h idx S P a* = *blk* (*x*, *xihat idx S p*, *z*) **by** (*simp add*: *blk_def x_def z_def cc_shift2 enc_h_def split*)
  moreover have *set x* ⊆ *Talph* **unfolding** *x_def* **by** (*rule set_concat_Cc_xihat*)
  moreover have *set z* ⊆ *Talph* **unfolding** *z_def* **by** (*rule set_concat_xihat_Cc*)
  ultimately show *?thesis* **by** *blast*
qed

Lifting the block decomposition over a whole word: given a list of productions matching the letters of w, the encoded word $h(w)$ is a concatenation of blocks whose middles are exactly the $\xi\widehat{}$-codes of the productions.

lemma *block_decomp*:
  list_all2 (λ*b p*. *p* ∈ *set P* ∧ (∃ *Bs*. *snd p* = *Tm b* # *map Nt Bs*)) *w ps* ⟹
   ∃ *bs*. *concat* (*map* (*enc_h idx S P*) *w*) = *concat* (*map blk bs*)
     ∧ *map* (λ(*x*,*y*,*z*). *y*) *bs* = *map* (*xihat idx S*) *ps*
     ∧ (∀ (*x*,*y*,*z*) ∈ *set bs*. *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph*)
proof (*induction w ps rule*: *list_all2_induct*)
  case *Nil*
  show *?case* **by** (*intro exI*[*of* _ []]) *simp*
next
  case (*Cons b w′ p ps′*)
  from *Cons.hyps*(*1*) **obtain** *Bs* **where** *pin*: *p* ∈ *set P* **and** *snd_p*: *snd p* = *Tm b* # *map Nt Bs* **by** *blast*
  from *enc_h_block*[*OF pin snd_p*] **obtain** *x z* **where**
    blk_eq: *enc_h idx S P b* = *blk* (*x*, *xihat idx S p*, *z*) **and**
    xT: *set x* ⊆ *Talph* **and** *zT*: *set z* ⊆ *Talph* **by** *blast*
  from *Cons.IH* **obtain** *bs* **where**
    bs_eq: *concat* (*map* (*enc_h idx S P*) *w′*) = *concat* (*map blk bs*) **and**

bs_y: *map* (λ(*x*,*y*,*z*). *y*) *bs* = *map* (*xihat idx S*) *ps′* **and**
bs_T: ∀(*x*,*y*,*z*) ∈ *set bs*. *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph* **by** *blast*
let *?bs* = (*x*, *xihat idx S p*, *z*) # *bs*
have *concat* (*map* (*enc_h idx S P*) (*b* # *w′*)) = *concat* (*map blk ?bs*)
by (*simp add*: *blk_eq bs_eq*)
moreover have *map* (λ(*x*,*y*,*z*). *y*) *?bs* = *map* (*xihat idx S*) (*p* # *ps′*) **by** (*simp add*: *bs_y*)
moreover have ∀(*x*,*y*,*z*) ∈ *set ?bs*. *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph* **using** *xT zT bs_T* **by** *auto*
ultimately show *?case* **by** *blast*
qed

1.3.4 Inverting the stack machinery (for the backward direction)

The fragment *frag i = A1 A2*^{i} *A1* determines i and the rest uniquely as a prefix.

lemma *frag_append_inj*:
frag i @ *xs* = *frag j* @ *ys* ⟹ *i* = *j* ∧ *xs* = *ys*
unfolding *frag_def* **by** (*auto simp add*: *append_Cons_eq_append_Cons*)

lemma *stkenc_eq_Nil*: *stkenc cs* = [] ⟹ *cs* = []
by (*cases cs*) (*auto simp*: *stkenc_def frag_def*)

Inverse of *bal_stk_popcode*: a successful pop forces the stack to start with the matching fragment.

lemma *bal_stk_popcode_inv*:
assumes *bal_stk t* (*brks* (*popcode i*)) = (*s*, []) **shows** *t* = *frag i* @ *s*
proof −
from *assms* **have** *A*: *bal_stk t* (*replicate 1* (*Close A1*) @ (*replicate i* (*Close A2*) @ [*Close A1*])) = (*s*, [])
by (*simp add*: *popcode_def*)
from *bal_stk_replicate_Close_inv*[*OF A*] **obtain** *t1* **where**
t1: *t* = *replicate 1 A1* @ *t1* **and**
B: *bal_stk t1* (*replicate i* (*Close A2*) @ [*Close A1*]) = (*s*, []) **by** *blast*
from *bal_stk_replicate_Close_inv*[*OF B*] **obtain** *t2* **where**
t2: *t1* = *replicate i A2* @ *t2* **and** *C*: *bal_stk t2* [*Close A1*] = (*s*, []) **by** *blast*
have *t2* = *A1* # *s*
proof (*cases t2*)
case *Nil* **thus** *?thesis* **using** *C* **by** *simp*
next
case (*Cons c t3*) **thus** *?thesis* **using** *C* **by** (*auto split*: *if_splits*)
qed
with *t1 t2* **show** *t* = *frag i* @ *s* **by** (*simp add*: *frag_def*)
qed

Inverse of *bal_stk_xi*: if running $\xi(p)$ on the encoded stack α consumes all input, then α's top is *fst p* (this uses injectivity of *idx*) and the resulting stack is the encoding of *Bs* on top of the remaining stack.

lemma *bal_stk_xi_inv*:
assumes *inj*: *inj_on idx N* **and** *snd_p*: *snd p* = *Tm a* # *map Nt Bs*
and *fpN*: *fst p* ∈ *N* **and** *αN*: *set α* ⊆ *N*
and *run*: *bal_stk* (*stkenc* (*map idx α*)) (*brks* (*xi idx p*)) = (*s*, [])
shows ∃*α′*. *α* = *fst p* # *α′* ∧ *s* = *stkenc* (*map idx* (*Bs* @ *α′*))
proof −
from *run snd_p* **have**
bal_stk (*stkenc* (*map idx α*))
(*brks* (*popcode* (*idx* (*fst p*))) @ *brks* (*concat* (*map pushcode* (*rev* (*map idx Bs*))))) = (*s*, [])
by (*simp add*: *xi_eq*)
then obtain *s′* **where**
pop: *bal_stk* (*stkenc* (*map idx α*)) (*brks* (*popcode* (*idx* (*fst p*)))) = (*s′*, []) **and**
push: *bal_stk s′* (*brks* (*concat* (*map pushcode* (*rev* (*map idx Bs*))))) = (*s*, [])
by (*rule bal_stk_append_split*)
from *bal_stk_popcode_inv*[*OF pop*] **have** *stk_eq*: *stkenc* (*map idx α*) = *frag* (*idx* (*fst p*)) @ *s′* **.**
have *s_eq*: *s* = *stkenc* (*map idx Bs*) @ *s′*
using *bal_stk_pushes*[*of s′ rev* (*map idx Bs*)] *push* **by** *simp*
have *α* ≠ []
proof
assume *α* = [] **with** *stk_eq* **show** *False* **by** (*simp add*: *frag_def stkenc_def*)
qed
then obtain *A′ α′* **where** *αc*: *α* = *A′* # *α′* **by** (*cases α*) *auto*
with *stk_eq* **have** *frag* (*idx A′*) @ *stkenc* (*map idx α′*) = *frag* (*idx* (*fst p*)) @ *s′*
by (*simp add*: *stkenc_def*)
from *frag_append_inj*[*OF this*] **have** *iA*: *idx A′* = *idx* (*fst p*) **and** *seq′*: *stkenc* (*map idx α′*) = *s′*
by *auto*
from *αc αN* **have** *A′* ∈ *N* **by** *simp*
with *iA inj fpN* **have** *A′* = *fst p* **by** (*auto dest*: *inj_onD*)
with *αc* **have** *α* = *fst p* # *α′* **by** *simp*
moreover have *s* = *stkenc* (*map idx* (*Bs* @ *α′*))
using *s_eq seq′*[*symmetric*] **by** (*simp add*: *stkenc_def*)
ultimately show *?thesis* **by** *blast*
qed

1.3.5 Parsing a block word back into productions (for the backward direction)

Two lists of *s*-terminated, *s*-internal-free blocks with equal concatenation are equal.

lemma *concat_block_align*:
concat xss = *concat yss* ⟹
(∀ *xs*∈*set xss*. ∃ *p*. *xs* = *p* @ [*s*] ∧ *s* ∉ *set p*) ⟹
(∀ *ys*∈*set yss*. ∃ *p*. *ys* = *p* @ [*s*] ∧ *s* ∉ *set p*) ⟹ *xss* = *yss*
proof (*induction xss arbitrary*: *yss*)
case *Nil*
thus *?case* **by** (*cases yss*) *auto*

```
next
  case (Cons xs xss')
  from Cons.prems(2) obtain p1 where xs: xs = p1 @ [s] and p1: s ∉ set p1
by auto
  show ?case
  proof (cases yss)
    case Nil
    with Cons.prems(1) xs show ?thesis by simp
  next
    case (Cons ys yss')
    from Cons.prems(3) ‹yss = ys # yss'› obtain p2 where ys: ys = p2 @ [s]
and p2: s ∉ set p2
      by auto
    from Cons.prems(1) xs ys ‹yss = ys # yss'›
    have p1 @ s # concat xss' = p2 @ s # concat yss' by simp
    from this p1 p2
   have p1 = p2 and concat xss' = concat yss' by (auto simp add: append_Cons_eq_append_Cons)
    have xss' = yss'
     using Cons.IH[OF ‹concat xss' = concat yss'›] Cons.prems(2) Cons.prems(3)
‹yss = ys # yss'›
      by auto
    with ‹p1 = p2› xs ys ‹yss = ys # yss'› show ?thesis by simp
  qed
qed
```

A c-free middle delimited by two cs inside a single-c-separated block body must be one of the fragments.

```
lemma cfrag_parse_gen:
  assumes c ∉ set y and fc: ⋀q. c ∉ set (f q)
  shows x @ c # y @ c # z = concat (map (λq. c # f q) qs) @ [c] ⟹ y ∈ set
(map f qs)
proof (induction qs arbitrary: x z)
  case Nil thus ?case by (cases x) auto
next
  case (Cons q qs')
  have rhs: concat (map (λq. c # f q) (q # qs')) @ [c]
          = (c # f q) @ (concat (map (λq. c # f q) qs') @ [c]) by simp
  let ?REST = concat (map (λq. c # f q) qs') @ [c]
  have restCc: ?REST = c # tl ?REST by (cases qs') auto
  from Cons.prems rhs have eq: x @ c # y @ c # z = (c # f q) @ ?REST by
simp
  show ?case
  proof (cases x)
    case Nil
    with eq have y @ c # z = f q @ c # tl ?REST using restCc by simp
   from this assms(1) fc have y = f q by (simp add: append_Cons_eq_append_Cons)
    thus ?thesis by simp
  next
    case (Cons xa x')
```

```
    with eq obtain t where
      x' = f q @ t ∧ t @ (c # y @ c # z) = ?REST ∨ x' @ t = f q ∧ c # y @ c
# z = t @ ?REST
      using append_eq_append_conv2[of x' c # y @ c # z f q ?REST] by auto
    thus ?thesis
    proof
      assume x' = f q @ t ∧ t @ (c # y @ c # z) = ?REST
      with Cons.IH show ?thesis by auto
    next
      assume x' @ t = f q ∧ c # y @ c # z = t @ ?REST
      hence fqt: f q = x' @ t and ceq: c # y @ c # z = t @ ?REST by auto
      have ct: c ∉ set t using fqt fc[of q] by auto
      have t = [] using ceq ct by (cases t) auto
      with ceq have [] @ c # y @ c # z = ?REST by simp
      from Cons.IH[OF this] show ?thesis by simp
    qed
  qed
qed
```

Each block *enc_h idx S P a* ends in exactly one *d*.

```
lemma enc_h_Dd: ∃ p. enc_h idx S P a = p @ [Dd] ∧ Dd ∉ set p
using set_xihat unfolding enc_h_def Talph_def bracks_def by fastforce
```

Inverting one block: if *blk* (x,y,z) equals an encoded block *enc_h idx S P a* and the middle y is free of c (which holds for every block of an L_0-witness), then y is the $\hat{\xi}$-code of some production of P whose right-hand side starts with a.

```
lemma block_parse:
  assumes eq: blk (x,y,z) = enc_h idx S P a
      and yset: set y ⊆ insert Ce bracks
  shows ∃ p. p ∈ set P ∧ (∃ Bs. snd p = Tm a # map Nt Bs) ∧ y = xihat idx S p
proof −
  let ?qs = filter (λp. ∃ Bs. snd p = Tm a # map Nt Bs) P
  have eq': x @ Cc # y @ Cc # z = concat (map (λp. Cc # xihat idx S p) ?qs)
@ [Cc]
    using eq by (simp add: blk_def enc_h_def)
  have ccy: Cc ∉ set y using yset by (auto simp: bracks_def)
  have ccf: Cc ∉ set (xihat idx S q) for q
    using set_xihat[of idx S q] by (auto simp: bracks_def)
  from cfrag_parse_gen[OF ccy ccf eq']
  show ?thesis by fastforce
qed
```

For a complete derivation read off from an L_0-witness, the concatenation of the encoded middles is $\mathfrak{c}$ followed by the push of S and the ξ-codes. This is justified directly from the $\mathfrak{c}/D$-conditions (the first production expands S, the later ones do not).

```
lemma concat_xihat_eq_fst:
```

assumes *ne*: $ps \neq []$ **and** *fp0*: $fst\ (hd\ ps) = S$ **and** *tlS*: $\forall\, p{\in}set\ (tl\ ps).\ fst\ p \neq S$
shows $concat\ (map\ (xihat\ idx\ S)\ ps) = Ce\ \#\ pushcode\ (idx\ S)\ @\ concat\ (map\ (xi\ idx)\ ps)$
using *assms fp0 tlS* **by**(*auto simp*: *neq_Nil_conv xihat_def map_xihat_no_S*)

Inverting a whole block word: a list of blocks matching the letters of w (with c-free middles) yields a production list ps matching w, whose $\hat{\xi}$-codes are the middles.

lemma *blocks_to_ps*:
$list\_all2\ (\lambda b\ a.\ blk\ b = enc\_h\ idx\ S\ P\ a)\ bs\ w \Longrightarrow$
$(\forall\,(x,y,z){\in}set\ bs.\ set\ y \subseteq insert\ Ce\ bracks) \Longrightarrow$
$\exists\, ps.\ list\_all2\ (\lambda a\ p.\ p \in set\ P \wedge (\exists\, Bs.\ snd\ p = Tm\ a\ \#\ map\ Nt\ Bs))\ w\ ps$
$\wedge\ map\ (\lambda(x,y,z).\ y)\ bs = map\ (xihat\ idx\ S)\ ps$
proof (*induction bs w rule*: *list_all2_induct*)
case *Nil*
show *?case* **by** (*intro exI*[*of* _ []]) *auto*
next
case (*Cons b bs′ a w′*)
obtain *x y z* **where** $b = (x,y,z)$ **by** (*cases b*)
with *Cons block_parse* **show** *?case* **by** *fastforce*
qed

1.4 Theorem 2.1

We fix a grammar in Greibach *standard form*: every production is $A \to a\ B_1\ \ldots\ B_m$ (a terminal followed by nonterminals) and the start symbol S occurs on no right-hand side. The nonterminals are indexed injectively by *idx* (only injectivity on $insert\ S\ (Nts\ (set\ P))$ matters; the specific index values, in particular $idx\ S$, are immaterial).

locale *greibach_std* =
fixes $P :: ('n,'t)\ prods$
and $S :: 'n$
and $idx :: 'n \Rightarrow nat$
assumes *std*: $\bigwedge A\ \alpha.\ (A,\alpha) \in set\ P \Longrightarrow \exists\, a\ Bs.\ \alpha = Tm\ a\ \#\ map\ Nt\ Bs$
and *S_notin_rhs*: $\bigwedge A\ a\ Bs.\ (A,\ Tm\ a\ \#\ map\ Nt\ Bs) \in set\ P \Longrightarrow S \notin set\ Bs$
and *idx_inj*: $inj\_on\ idx\ (insert\ S\ (Nts\ (set\ P)))$
begin

drives ps α *w* β: applying the production list ps as successive *leftmost* steps to the nonterminal stack α (top first) emits the terminal word w and reaches stack β. Each step expands the stack top by a standard-form production from P.

inductive $drives :: ('n,'t)\ prod\ list \Rightarrow 'n\ list \Rightarrow 't\ list \Rightarrow 'n\ list \Rightarrow bool$ **where**
drives_Nil: $drives\ []\ \alpha\ []\ \alpha$
| *drives_Cons*: $(A,\ Tm\ a\ \#\ map\ Nt\ Bs) \in set\ P \Longrightarrow drives\ ps\ (Bs\ @\ \alpha)\ w\ \beta \Longrightarrow$
$drives\ ((A,\ Tm\ a\ \#\ map\ Nt\ Bs)\ \#\ ps)\ (A\ \#\ \alpha)\ (a\ \#\ w)\ \beta$

Central invariant. Running the encoded ξ-brackets of *ps* through *bal_stk*, starting from the encoded stack α, ends at the encoded stack β. Each leftmost step is one application of *bal_stk_xi*: pop the top nonterminal, push the production's right-hand-side nonterminals.

lemma *drives_bal_stk*:
  drives ps α w β ⟹
   bal_stk (stkenc (map idx α)) (brks (concat (map (xi idx) ps))) = (stkenc (map idx β), [])
proof (*induction rule*: *drives.induct*)
  case (*drives_Nil α*)
  show *?case* **by** *simp*
next
  case (*drives_Cons A a Bs ps α w β*)
  let *?p = (A, Tm a # map Nt Bs)*
  show *?case* **using** *drives_Cons bal_stk_xi[of ?p a Bs idx stkenc (map idx α)]*
    by (*simp add*: *stkenc_def bal_stk_append step*)
qed

For a complete leftmost derivation (the stack $[S]$ is emptied), the bracket word after the leading ¢ (namely *pushcode* (*idx S*) followed by the ξ-codes) is balanced, i.e. in *D*.

lemma *drives_bal_complete*:
  assumes *drives ps [S] w []*
  shows *bal (brks (pushcode (idx S) @ concat (map (xi idx) ps)))*
using *drives_bal_stk[OF assms]*
by (*simp add*: *bal_stk_append bal_stk_pushcode stkenc_def bal_iff_bal_stk*)

1.4.1 Bridge between *drives* and the grammar language *Lang*

Easy direction: a *drives* sequence is a leftmost derivation.

lemma *drives_imp_derivels*:
  drives ps α w β ⟹ set P ⊢ map Nt α ⇒l∗ map Tm w @ map Nt β
proof (*induction rule*: *drives.induct*)
  case (*drives_Nil α*)
  show *?case* **by** *simp*
next
  case (*drives_Cons A a Bs ps α w β*)
  have *step*: *set P ⊢ Nt A # map Nt α ⇒l Tm a # map Nt (Bs @ α)*
    using *derivel.intros[OF drives_Cons.hyps(1), of [] map Nt α]* **by** *simp*
  have *set P ⊢ Tm a # map Nt (Bs @ α) ⇒l∗ Tm a # (map Tm w @ map Nt β)*
    using *drives_Cons.IH* **by** (*simp add*: *derivels_Tm_Cons*)
  with *step* **have** *set P ⊢ Nt A # map Nt α ⇒l∗ Tm a # (map Tm w @ map Nt β)*
    by (*rule converse_rtranclp_into_rtranclp*)
  thus *?case* **by** *simp*
qed

Hard direction: a leftmost derivation to a terminal word yields a *drives* se-

quence. Induction on the number of leftmost steps. (We suppress *relpowp.simps* (*2*) so that the leftmost-step lemmas for $\Rightarrow l(Suc\ n)$ fire before the relation power is unfolded.)

lemma *deriveln_imp_drives*:
  set P $\vdash$ *map Nt* α $\Rightarrow l(n)$ *map Tm w* $\Longrightarrow$ $\exists$ *ps. drives ps* α *w* []
proof (*induction n arbitrary*: α *w*)
  case *0*
  hence *map Nt* α = *map Tm w* **by** *simp*
  hence α = [] $\wedge$ *w* = [] **by** (*cases* α; *cases w*; *auto*)
  thus *?case* **using** *drives_Nil* **by** *auto*
next
  case (*Suc n*)
  show *?case*
  proof (*cases* α)
    case *Nil*
    with *Suc.prems* **have** *set P* $\vdash$ [] $\Rightarrow l(Suc\ n)$ *map Tm w* **by** *simp*
    hence *False* **by** (*simp del*: *relpowp.simps*(*2*))
    thus *?thesis* **..**
  next
    case (*Cons A* α')
    with *Suc.prems* **have** *set P* $\vdash$ *Nt A* # *map Nt* α' $\Rightarrow l(Suc\ n)$ *map Tm w* **by** *simp*
    then obtain γ **where** γ: (*A*, γ) $\in$ *set P* **and** *der*: *set P* $\vdash$ γ @ *map Nt* α' $\Rightarrow l(n)$ *map Tm w*
      by (*auto simp*: *deriveln_Nt_Cons simp del*: *relpowp.simps*(*2*))
    from *std*[*OF* γ] **obtain** *a Bs* **where** γ*eq*: γ = *Tm a* # *map Nt Bs* **by** *blast*
    from *der* γ*eq* **have** *set P* $\vdash$ *Tm a* # *map Nt* (*Bs* @ α') $\Rightarrow l(n)$ *map Tm w* **by** *simp*
    then obtain *w*$'$ **where** *weq*: *map Tm w* = *Tm a* # *w*$'$ **and** *der*$'$: *set P* $\vdash$ *map Nt* (*Bs* @ α') $\Rightarrow l(n)$ *w*$'$
      by (*auto simp*: *deriveln_Tm_Cons simp del*: *relpowp.simps*(*2*))
    from *weq* **obtain** *w*$''$ **where** *w_eq*: *w* = *a* # *w*$''$ **and** *w*$'$*_eq*: *w*$'$ = *map Tm w*$''$
      by (*cases w*) *auto*
    from *der*$'$ *w*$'$*_eq* **have** *der*$''$: *set P* $\vdash$ *map Nt* (*Bs* @ α') $\Rightarrow l(n)$ *map Tm w*$''$ **by** *simp*
    from *Suc.IH*[*OF der*$''$] **obtain** *ps* **where** *ps*: *drives ps* (*Bs* @ α') *w*$''$ [] **by** *blast*
    have *drives* ((*A*, *Tm a* # *map Nt Bs*) # *ps*) (*A* # α') (*a* # *w*$''$) []
      using γ γ*eq ps* **by** (*auto intro*: *drives_Cons*)
    thus *?thesis* **using** *Cons w_eq* **by** *auto*
  qed
qed

Combining both directions with the leftmost/standard derivation equivalence *derivels_iff_derives*: membership in the grammar's language is exactly the existence of a *drives* sequence from $[S]$ to the empty stack.

lemma *drives_iff_Lang*:
  ($\exists$ *ps. drives ps* [*S*] *w* []) $\longleftrightarrow$ *w* $\in$ *Lang* (*set P*) *S*

proof
  assume $\exists\, ps.\ drives\ ps\ [S]\ w\ []$
  with *drives_imp_derivels*
  show $w \in Lang\ (set\ P)\ S$ **unfolding** *Lang_def*
    using *derivels_iff_derives* **by** *fastforce*
next
  assume $w \in Lang\ (set\ P)\ S$
 hence $set\ P \vdash [Nt\ S] \Rightarrow l*\ map\ Tm\ w$ **by** (*simp add*: *Lang_def derivels_iff_derives*)
  thus $\exists\, ps.\ drives\ ps\ [S]\ w\ []$
    using *deriveln_imp_drives rtranclp_power* **by** *fastforce*
qed

1.4.2 Block structure of the encoded word

Once S has left the stack it never returns (it is on no right-hand side), so no later production expands S.

lemma *drives_no_S*:
  $drives\ ps\ \alpha\ w\ \beta \Longrightarrow S \notin set\ \alpha \Longrightarrow (\forall\, p \in set\ ps.\ fst\ p \neq S)$
proof (*induction rule*: *drives.induct*)
  case (*drives_Nil* α) **show** *?case* **by** *simp*
next
  case (*drives_Cons A a Bs ps* α *w* β)
  show *?case* **using** *S_notin_rhs drives_Cons* **by** *auto*
qed

The concatenation of the encoded middles of a complete derivation: only the first production (which expands S) contributes the ¢ marker and the initial push of S; all later productions contribute pure ξ-codes.

lemma *concat_xihat_eq*:
  assumes *drives ps* [*S*] *w* []
  shows *concat* (*map* (*xihat idx S*) *ps*) = *Ce* # *pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*)
proof −
  from *assms* **obtain** *a Bs ps′ w′* **where**
      ps_eq: *ps* = (*S*, *Tm a* # *map Nt Bs*) # *ps′*
      and *mem*: (*S*, *Tm a* # *map Nt Bs*) ∈ *set P*
      and *rest*: *drives ps′ Bs w′* []
    by (*auto elim*: *drives.cases*)
  have *S* ∉ *set Bs* **using** *S_notin_rhs*[*OF mem*] **by** *simp*
   with *drives_no_S*[*OF rest*] **have** ∀ *p* ∈ *set* (*tl ps*). *fst p* ≠ *S* **by** (*simp add*: *ps_eq*)
  from *concat_xihat_eq_fst*[*OF _ _ this*] **show** *?thesis* **by** (*simp add*: *ps_eq*)
qed

1.4.3 Forward direction of Theorem 2.1

The terminal letters of w are matched, in order, by the productions of any *drives* sequence producing w.

lemma *drives_list_all2*:
drives ps α w β $\Longrightarrow$ *list_all2* (λ*b p*. *p* ∈ *set P* ∧ (∃ *Bs*. *snd p* = *Tm b* # *map Nt Bs*)) *w ps*
by (*induction rule*: *drives.induct*) *auto*

If $w \in L(G)$ (witnessed by a complete *drives* sequence) then the encoded word $h(w)$ lies in L_0: the blocks select the $\hat{\xi}$-codes, whose concatenation is ¢ followed by a balanced ($\in D$) word.

lemma *forward*:
assumes *drv*: *drives ps* [*S*] *w* []
shows *concat* (*map* (*enc_h idx S P*) *w*) ∈ *L0*
proof −
from *drv* **obtain** *a Bs ps′ w′* **where**
ps_eq: *ps* = (*S*, *Tm a* # *map Nt Bs*) # *ps′*
and *mem*: (*S*, *Tm a* # *map Nt Bs*) ∈ *set P*
and *rest*: *drives ps′ Bs w′* []
by (*auto elim*: *drives.cases*)
have *SnotBs*: *S* ∉ *set Bs* **using** *S_notin_rhs*[*OF mem*] **by** *simp*
have *noS*: ∀ *p* ∈ *set ps′*. *fst p* ≠ *S* **using** *drives_no_S*[*OF rest*] *SnotBs* **by** *simp*
from *block_decomp*[*OF drives_list_all2*[*OF drv*]] **obtain** *bs* **where**
bw: *concat* (*map* (*enc_h idx S P*) *w*) = *concat* (*map blk bs*) **and**
by_eq: *map* (λ(*x*,*y*,*z*). *y*) *bs* = *map* (*xihat idx S*) *ps* **and**
bT: ∀ (*x*,*y*,*z*) ∈ *set bs*. *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph* **by** *blast*
have *bs_ne*: *bs* ≠ [] **using** *by_eq ps_eq* **by** (*cases bs*) *auto*
let *?v* = *pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*)
have *ys*: *concat* (*map* (λ(*x*,*y*,*z*). *y*) *bs*) = *Ce* # *?v*
using *by_eq concat_xihat_eq*[*OF drv*] **by** *simp*
have *vD*: *?v* ∈ *D*
proof −
have *bal* (*brks ?v*) **by** (*rule drives_bal_complete*[*OF drv*])
moreover have *set ?v* ⊆ *bracks*
unfolding *set_append* **using** *Un_least*[*OF set_pushcode*] *set_xi* **by** *fastforce*
ultimately show *?thesis* **by** (*simp add*: *D_def*)
qed
have *tl_y*: *map* (λ(*x*,*y*,*z*). *y*) (*tl bs*) = *map* (*xihat idx S*) *ps′*
proof −
have *map* (λ(*x*,*y*,*z*). *y*) *bs* = *map* (*xihat idx S*) ((*S*, *Tm a* # *map Nt Bs*) # *ps′*)
using *by_eq ps_eq* **by** *simp*
thus *?thesis* **by** (*cases bs*) *auto*
qed
have *key*: *set y* ⊆ *bracks* **if** (*x*,*y*,*z*) ∈ *set* (*tl bs*) **for** *x y z*
proof −
from *that* **have** *y* ∈ *set* (*map* (λ(*x*,*y*,*z*). *y*) (*tl bs*)) **by** *force*
then obtain *p* **where** *p*: *p* ∈ *set ps′* **and** *yp*: *y* = *xihat idx S p* **using** *tl_y* **by** *auto*
have *fst p* ≠ *S* **using** *p noS* **by** *simp*
hence *y* = *xi idx p* **using** *yp* **by** (*simp add*: *xihat_def*)
thus *?thesis* **using** *set_xi* **by** *simp*

```
  qed
  have tl_bracks: ∀(x,y,z) ∈ set (tl bs). set y ⊆ bracks using key by fast
  have exv: ∃v∈D. concat (map (λ(x,y,z). y) bs) = Ce # v using ys vD by blast
  have concat (map blk bs) ∈ L0
   unfolding L0_def using bs_ne bT tl_bracks exv by (auto intro!: exI[of _ bs])
  thus ?thesis using bw by simp
qed
```

1.4.4 Backward direction of Theorem 2.1

Converse of the central invariant. If the encoded ξ-brackets of a production list *ps* (whose productions are standard-form and match the letters of *w*) run successfully on the encoded stack α (consuming everything, ending empty), then *ps* really is a leftmost derivation *drives ps* α *w* []. Each step is inverted by *bal_stk_xi_inv*, which (using injectivity of *idx*) forces the production's left-hand side to be the current stack top.

```
lemma drives_bal_stk_inv:
  list_all2 (λb p. p ∈ set P ∧ (∃Bs. snd p = Tm b # map Nt Bs)) w ps ⟹
   set α ⊆ insert S (Nts (set P)) ⟹
   bal_stk (stkenc (map idx α)) (brks (concat (map (xi idx) ps))) = ([], []) ⟹
   drives ps α w []
proof (induction w ps arbitrary: α rule: list_all2_induct)
  case (Nil α)
  from Nil.prems(2) have stkenc (map idx α) = [] by simp
  hence α = [] using stkenc_eq_Nil by auto
  thus ?case by (simp add: drives.drives_Nil)
next
  case (Cons b w' p ps' α)
  from Cons.hyps(1) obtain Bs where pin: p ∈ set P and snd_p: snd p = Tm
b # map Nt Bs by blast
 have fpN: fst p ∈ insert S (Nts (set P)) using pin by (force simp: Nts_Lhss_Rhs_Nts
Lhss_def)
  have BsN: set Bs ⊆ insert S (Nts (set P))
  proof −
    have (fst p, Tm b # map Nt Bs) = p using snd_p by (cases p) auto
    with pin have (fst p, Tm b # map Nt Bs) ∈ set P by simp
     hence Nts_syms (Tm b # map Nt Bs) ⊆ Rhs_Nts (set P) by (auto simp:
Rhs_Nts_def)
     moreover have set Bs ⊆ Nts_syms (Tm b # map Nt Bs) by (auto simp:
Nts_syms_def)
    ultimately have set Bs ⊆ Rhs_Nts (set P) by blast
   also have Rhs_Nts (set P) ⊆ insert S (Nts (set P)) by (auto simp: Nts_Lhss_Rhs_Nts)
    finally show ?thesis .
  qed
  from Cons.prems(1) have αN: set α ⊆ insert S (Nts (set P)) .
  from Cons.prems(2) have bal_stk (stkenc (map idx α))
        (brks (xi idx p) @ brks (concat (map (xi idx) ps'))) = ([], []) by simp
  then obtain s' where
```

```
    step: bal_stk (stkenc (map idx α)) (brks (xi idx p)) = (s', []) and
    rest: bal_stk s' (brks (concat (map (xi idx) ps'))) = ([], [])
    by (rule bal_stk_append_split)
  from bal_stk_xi_inv[OF idx_inj snd_p fpN αN step] obtain α' where
    αeq: α = fst p # α' and s'eq: s' = stkenc (map idx (Bs @ α')) by blast
  from αeq αN have set α' ⊆ insert S (Nts (set P)) by auto
  with BsN have BsαN: set (Bs @ α') ⊆ insert S (Nts (set P)) by auto
  from rest s'eq have
    bal_stk (stkenc (map idx (Bs @ α'))) (brks (concat (map (xi idx) ps'))) = ([],
[]) by simp
  from Cons.IH[OF BsαN this] have driv': drives ps' (Bs @ α') w' [] .
  have peq: p = (fst p, Tm b # map Nt Bs) using snd_p by (cases p) auto
  with pin have memP: (fst p, Tm b # map Nt Bs) ∈ set P by simp
  from drives_Cons[OF memP driv']
  have drives ((fst p, Tm b # map Nt Bs) # ps') (fst p # α') (b # w') [] .
  thus ?case using αeq peq by simp
qed
```

Backward direction. If the encoded word $h(w)$ is a non-empty member of L_0, parse its block structure back into a production list and run the converse central invariant to obtain a *drives* sequence (hence $w \in L(G)$). Block alignment uses that both *blk*- and *enc_h*-blocks end in exactly one d; each middle is c-free, so equals a $\hat{\xi}$-code (*block_parse*); the $\mathfrak{c}/D$-condition gives the balanced run feeding *drives_bal_stk_inv*.

```
lemma backward_drives:
  assumes inL0: concat (map (enc_h idx S P) w) ∈ L0
      and ne: concat (map (enc_h idx S P) w) ≠ []
  shows ∃ps. drives ps [S] w []
proof −
  from inL0 ne have ∃bs. concat (map (enc_h idx S P) w) = concat (map blk bs)
∧ bs ≠ [] ∧
       (∀(x,y,z)∈set bs. set x ⊆ Talph ∧ set z ⊆ Talph) ∧
       (∀(x,y,z)∈set (tl bs). set y ⊆ bracks) ∧
       (∃v∈D. concat (map (λ(x,y,z). y) bs) = Ce # v)
    by (auto simp: L0_def)
  then obtain bs where
    cc: concat (map (enc_h idx S P) w) = concat (map blk bs) and
    bs_ne0: bs ≠ [] and
    xzT: ∀(x,y,z)∈set bs. set x ⊆ Talph ∧ set z ⊆ Talph and
    tlb: ∀(x,y,z)∈set (tl bs). set y ⊆ bracks and
    cD: ∃v∈D. concat (map (λ(x,y,z). y) bs) = Ce # v
    by blast
  from cD obtain v where vD: v ∈ D and yv: concat (map (λ(x,y,z). y) bs) =
Ce # v by blast

  — Every middle is c-free (it is part of the 𝔠/D-word).
  have ysub_one: set y ⊆ insert Ce bracks if (x,y,z) ∈ set bs for x y z
  proof −
    from that have y ∈ set (map (λ(x,y,z). y) bs) by force
```

```
  hence set y ⊆ set (concat (map (λ(x,y,z). y) bs)) by auto
  also have ... = insert Ce (set v) using yv by simp
  also have ... ⊆ insert Ce bracks using vD by (auto simp: D_def)
  finally show ?thesis .
 qed
 have ysub: ∀(x,y,z)∈set bs. set y ⊆ insert Ce bracks using ysub_one by fast

 — Each blk-block ends in exactly one d, just like each enc_h-block.
 have blkDd: ∃p. b = p @ [Dd] ∧ Dd ∉ set p if b ∈ set (map blk bs) for b
 proof −
  from that obtain t where tin: t ∈ set bs and bt: b = blk t by auto
  obtain x y z where t: t = (x,y,z) by (cases t)
  have Dd ∉ set (x @ Cc # y @ Cc # z)
   using ysub tin t xzT by (auto simp: Talph_def bracks_def)
  moreover have b = (x @ Cc # y @ Cc # z) @ [Dd] using bt t by (simp add:
blk_def)
  ultimately show ?thesis by blast
 qed
 have align: map blk bs = map (enc_h idx S P) w
 proof (rule concat_block_align)
  show concat (map blk bs) = concat (map (enc_h idx S P) w) using cc by simp
  show ∀xs∈set (map blk bs). ∃p. xs = p @ [Dd] ∧ Dd ∉ set p using blkDd by
blast
  show ∀ys∈set (map (enc_h idx S P) w). ∃p. ys = p @ [Dd] ∧ Dd ∉ set p
   using enc_h_Dd by auto
 qed

 — Recover a matching production list ps whose ξ^-codes are the middles.
 have len: length bs = length w using align by (metis length_map)
 have la2blk: list_all2 (λb a. blk b = enc_h idx S P a) bs w
 proof (rule list_all2_all_nthI[OF len])
  fix n assume n < length bs
  thus blk (bs ! n) = enc_h idx S P (w ! n) using align len by (metis nth_map)
 qed
 from blocks_to_ps[OF la2blk ysub] obtain ps where
  la2match: list_all2 (λa p. p ∈ set P ∧ (∃Bs. snd p = Tm a # map Nt Bs)) w
ps and
  yps: map (λ(x,y,z). y) bs = map (xihat idx S) ps by blast

 have xihatps: concat (map (xihat idx S) ps) = Ce # v
 proof −
  have concat (map (xihat idx S) ps) = concat (map (λ(x,y,z). y) bs) using yps
by simp
  thus ?thesis using yv by simp
 qed

 — w (hence ps) is non-empty.
 have w_ne: w ≠ [] using ne by auto
 have ps_ne: ps ≠ [] using la2match w_ne by (cases w; cases ps) auto
```

obtain *p0 ps′* **where** *psc*: *ps* = *p0* # *ps′* **using** *ps_ne* **by** (*cases ps*) *auto*

— The first production expands *S* (its code begins with ¢).
have *fp0*: *fst p0* = *S*
proof (*rule ccontr*)
assume *ne0*: *fst p0* ≠ *S*
have *h*: *hd* (*xihat idx S p0*) = *Aa* (*Close A1*) **using** *ne0* **by** (*simp add*: *xihat_def xi_def popcode_def*)
have *nemp*: *xihat idx S p0* ≠ [] **using** *ne0* **by** (*simp add*: *xihat_def xi_def popcode_def*)
have *concat* (*map* (*xihat idx S*) *ps*) = *xihat idx S p0* @ *concat* (*map* (*xihat idx S*) *ps′*)
using *psc* **by** *simp*
hence *hd* (*concat* (*map* (*xihat idx S*) *ps*)) = *Aa* (*Close A1*) **using** *h nemp* **by** (*simp add*: *hd_append*)
with *xihatps* **show** *False* **by** *simp*
qed
have *hdps*: *fst* (*hd ps*) = *S* **using** *fp0 psc* **by** *simp*

— No later production expands *S* (their middles are bracket-only).
have *tlmap*: *map* (λ(*x*,*y*,*z*). *y*) (*tl bs*) = *map* (*xihat idx S*) *ps′*
by (*simp add*: *map_tl psc yps*)
have *tlS*: *fst p* ≠ *S* **if** *asm*: *p*∈*set ps′* **for** *p*
proof −
from *asm* **have** *xihat idx S p* ∈ *set* (*map* (*xihat idx S*) *ps′*) **by** *simp*
also have *set* (*map* (*xihat idx S*) *ps′*) = *set* (*map* (λ(*x*,*y*,*z*). *y*) (*tl bs*))
using *tlmap* **by** *simp*
finally have *xihat idx S p* ∈ *set* (*map* (λ(*x*,*y*,*z*). *y*) (*tl bs*)) **.**
then obtain *x y z* **where** *bin*: (*x*,*y*,*z*) ∈ *set* (*tl bs*) **and** *yeq*: *xihat idx S p* = *y* **by** *auto*
have *set y* ⊆ *bracks* **using** *tlb bin* **by** *auto*
hence *Ce* ∉ *set* (*xihat idx S p*) **using** *yeq* **by** (*auto simp*: *bracks_def*)
thus *fst p* ≠ *S* **by** (*auto simp*: *xihat_def*)
qed
have *tlps*: ∀ *p*∈*set* (*tl ps*). *fst p* ≠ *S* **using** *tlS psc* **by** *simp*

— Assemble the balanced run and invoke the converse central invariant.
have *Ce* # *v* = *Ce* # *pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*)
using *xihatps concat_xihat_eq_fst*[*OF ps_ne hdps tlps*] **by** *simp*
hence *v_eq*: *v* = *pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*) **by** *simp*
have *bal_stk* [] (*brks v*) = ([], [])
using *vD* **by** (*simp add*: *D_def bal_iff_bal_stk*)
hence *bal_stk* [] (*brks* (*pushcode* (*idx S*)) @ *brks* (*concat* (*map* (*xi idx*) *ps*))) = ([], [])
by (*simp add*: *v_eq*)
hence *bal_stk* (*frag* (*idx S*)) (*brks* (*concat* (*map* (*xi idx*) *ps*))) = ([], [])
by (*simp add*: *bal_stk_append bal_stk_pushcode*)
hence *run*: *bal_stk* (*stkenc* (*map idx* [*S*])) (*brks* (*concat* (*map* (*xi idx*) *ps*))) = ([], [])

```
    by (simp add: stkenc_def)
  have set [S] ⊆ insert S (Nts (set P)) by simp
  from drives_bal_stk_inv[OF la2match this run] show ?thesis ..
qed
```

1.4.5 The core of Theorem 2.1

For a Greibach-standard-form grammar, the language minus the empty word is exactly the inverse image under the encoding homomorphism *enc_h* of $L_0 - \{\varepsilon\}$.

```
lemma core: Lang (set P) S − {[]} = inv_hom (enc_h idx S P) (L0 − {[]})
proof (rule set_eqI)
  fix w
  show (w ∈ Lang (set P) S − {[]}) = (w ∈ inv_hom (enc_h idx S P) (L0 −
{[]}))
  proof
    assume w ∈ Lang (set P) S − {[]}
    hence wL: w ∈ Lang (set P) S and wne: w ≠ [] by auto
    from wL obtain ps where drives ps [S] w [] using drives_iff_Lang by blast
    hence inL0: concat (map (enc_h idx S P) w) ∈ L0 by (rule forward)
    have concat (map (enc_h idx S P) w) ≠ []
    proof −
      from wne obtain a w' where wc: w = a # w' by (cases w) auto
      obtain p where ep: enc_h idx S P a = p @ [Dd] using enc_h_Dd[of idx S
P a] by blast
      show ?thesis by (simp add: wc ep)
    qed
    with inL0 show w ∈ inv_hom (enc_h idx S P) (L0 − {[]}) by (simp add:
inv_hom_def)
  next
    assume w ∈ inv_hom (enc_h idx S P) (L0 − {[]})
    hence inL0: concat (map (enc_h idx S P) w) ∈ L0
      and ne: concat (map (enc_h idx S P) w) ≠ [] by (auto simp: inv_hom_def)
    from backward_drives[OF inL0 ne] obtain ps where drives ps [S] w [] by
blast
    hence w ∈ Lang (set P) S using drives_iff_Lang by blast
    moreover have w ≠ [] using ne by auto
    ultimately show w ∈ Lang (set P) S − {[]} by simp
  qed
qed

end
```

The locale-free form of *greibach_std.core*: for *any* production list P in Greibach standard form (every right-hand side a terminal followed by nonterminals, the start symbol S on no right-hand side) together with an injective index *idx*, the language with the empty word removed is exactly the inverse image of $L_0 - \{\varepsilon\}$ under the concrete encoding homomorphism *enc_h idx S P*.

This is the full content of Greibach's construction; the general *Greibach_2_1* below would follow by reducing an arbitrary grammar to this form.

theorem *Greibach_2_1_std*:
fixes $P :: ('n,'t)$ *prods* **and** $S :: 'n$ **and** *idx* $:: 'n \Rightarrow nat$
assumes $\bigwedge A\ \alpha.\ (A, \alpha) \in set\ P \Longrightarrow \exists a\ Bs.\ \alpha = Tm\ a\ \#\ map\ Nt\ Bs$
and $\bigwedge A\ a\ Bs.\ (A, Tm\ a\ \#\ map\ Nt\ Bs) \in set\ P \Longrightarrow S \notin set\ Bs$
and *inj_on idx* (*insert S* (*Nts* (*set P*)))
shows *Lang* (*set P*) *S* − {[]} = *inv_hom* (*enc_h idx S P*) (*L0* − {[]})
proof −
interpret *greibach_std P S idx* **using** *assms* **by** *unfold_locales*
show *?thesis* **by** (*rule core*)
qed

1.4.6 Reduction of an arbitrary grammar to the standard form

Adding a fresh start symbol A' (not occurring in R) that copies all productions of A preserves the language and makes A' occur on no right-hand side. This is the standard "new start symbol" trick, needed to meet the *greibach_std* side condition *S_notin_rhs*.

lemma *Lang_fresh_start*:
assumes $A' \notin Nts\ R$
shows *Lang* $(R \cup \{(A', \gamma)\ |\gamma.\ (A, \gamma) \in R\})\ A' = Lang\ R\ A$
proof −
let *?N* = $\{(A', \gamma)\ |\gamma.\ (A, \gamma) \in R\}$
have *notinL*: $A' \notin Lhss\ R$ **and** *notinR*: $A' \notin Rhs\_Nts\ R$
using *assms* **by** (*auto simp*: *Nts_Lhss_Rhs_Nts*)
have *lhssN*: *Lhss ?N* $\subseteq \{A'\}$ **by** (*auto simp*: *Lhss_def*)
have *disj*: *Rhs_Nts R* $\cap$ *Lhss ?N* = {} **using** *lhssN notinR* **by** *auto*
have *rhs_sub*: $A' \notin Nts\_syms\ \gamma$ **if** $(A, \gamma) \in R$ **for** γ
using *notinR that* **by** (*auto simp*: *Rhs_Nts_def*)
show *?thesis*
proof
show *Lang* $(R \cup ?N)\ A' \subseteq Lang\ R\ A$
proof
fix *w* **assume** $w \in Lang\ (R \cup ?N)\ A'$
then have *der*: $R \cup ?N \vdash [Nt\ A'] \Rightarrow* map\ Tm\ w$ **by** (*simp add*: *Lang_def*)
from *derives_Nt_map_TmD*[*OF der*] **obtain** γ
where γR: $(A', \gamma) \in R \cup ?N$ **and** *d*: $R \cup ?N \vdash \gamma \Rightarrow* map\ Tm\ w$ **by** *blast*
from γR *notinL* **have** $(A', \gamma) \in ?N$ **by** (*auto simp*: *Lhss_def*)
then have $A\gamma R$: $(A, \gamma) \in R$ **by** *auto*
hence $A' \notin Nts\_syms\ \gamma$ **by** (*rule rhs_sub*)
with *lhssN* **have** *Nts_syms* γ $\cap$ *Lhss ?N* = {} **by** *auto*
from *derives_disj_Un_iff*[*OF disj this*] *d* **have** *dR*: $R \vdash \gamma \Rightarrow* map\ Tm\ w$
by *simp*
have $R \vdash [Nt\ A] \Rightarrow \gamma$ **using** $A\gamma R$ **by** (*simp add*: *derive_singleton*)
from *this dR* **have** $R \vdash [Nt\ A] \Rightarrow* map\ Tm\ w$ **by** (*rule converse_rtranclp_into_rtranclp*)
then show $w \in Lang\ R\ A$ **by** (*simp add*: *Lang_def*)
qed

```
  next
    show Lang R A ⊆ Lang (R ∪ ?N) A'
    proof
      fix w assume w ∈ Lang R A
      then have der: R ⊢ [Nt A] ⇒* map Tm w by (simp add: Lang_def)
      from derives_Nt_map_TmD[OF der] obtain γ
        where AγR: (A, γ) ∈ R and d: R ⊢ γ ⇒* map Tm w by blast
      from AγR have (A', γ) ∈ R ∪ ?N by auto
      then have R ∪ ?N ⊢ [Nt A'] ⇒ γ by (simp add: derive_singleton)
      moreover from d have R ∪ ?N ⊢ γ ⇒* map Tm w by (meson Un_upper1
derives_mono)
    ultimately have R ∪ ?N ⊢ [Nt A'] ⇒* map Tm w by (rule converse_rtranclp_into_rtranclp)
      then show w ∈ Lang (R ∪ ?N) A' by (simp add: Lang_def)
    qed
  qed
qed
```

A nonterminal with no productions generates the empty language; and a non-empty member of L_0 always contains the separator d. Both are needed to handle the degenerate case where the start symbol does not occur in the grammar.

```
lemma L0_Dd: x ∈ L0 ⟹ x ≠ [] ⟹ Dd ∈ set x
proof −
  assume x ∈ L0 x ≠ []
  then obtain bs where x: x = concat (map blk bs) and ne: bs ≠ [] by (auto
simp: L0_def)
  from ne obtain b bs' where bbs: bs = b # bs' by (cases bs) auto
  obtain x1 y1 z1 where b = (x1, y1, z1) by (cases b)
  hence Dd ∈ set (blk b) by (simp add: blk_def)
  with x bbs show Dd ∈ set x by simp
qed

lemma inv_hom_Cc_empty: inv_hom (λ_. [Cc]) (L0 − {[]}) = {}
proof −
  have concat (map (λ_. [Cc]) w) ∉ L0 − {[]} for w :: 'a list
  proof
    assume concat (map (λ_. [Cc]) w) ∈ L0 − {[]}
    hence A: concat (map (λ_. [Cc]) w) ∈ L0 and B: concat (map (λ_. [Cc]) w)
≠ [] by auto
    from L0_Dd[OF A B] have Dd ∈ set (concat (map (λ_. [Cc]) w)) .
    moreover have set (concat (map (λ_. [Cc]) w)) ⊆ {Cc} by auto
    ultimately show False by auto
  qed
  thus ?thesis by (auto simp: inv_hom_def)
qed
```

1.4.7 Theorem 2.1 for grammars with a fresh-symbol supply

The construction proper, for a nonterminal type with a fresh-symbol supply (sort *fresh0*, as required by *gnf_of*): the language minus ε is the inverse homomorphic image of the one fixed hardest language $L_0 - \{\varepsilon\}$. The grammar is put into Greibach standard form by *gnf_of* together with a fresh start symbol; the (finitely many) nonterminals of the resulting grammar are then indexed injectively into *nat* by *finite_imp_inj_to_nat_seg* that injection is exactly the index *idx* the encoding *enc_h* requires (no countability needed). The general *Greibach_2_1* below lifts this to an arbitrary nonterminal type by renaming.

theorem *Greibach_2_1_fresh0*:
fixes *P* :: ('*n*::*fresh0*,'*t*) *Prods*
assumes *finP*: *finite P*
shows ∃ *h*::'*t* ⇒ *t0 list*. *Lang P S* − {[]} = *inv_hom h* (*L0* − {[]})
proof (*cases S* ∈ *Nts P*)
case *False*
hence *S* ∉ *Lhss P* **by** (*simp add*: *Nts_Lhss_Rhs_Nts*)
hence *Lang P S* = {} **by** (*rule Lang_empty_if_notin_Lhss*)
hence *Lang P S* − {[]} = *inv_hom* (λ_. [*Cc*]) (*L0* − {[]}) **by** (*simp add*: *inv_hom_Cc_empty*)
thus *?thesis* **by** *blast*
next
case *True*
obtain *ps* **where** *psP*: *set ps* = *P* **using** *finP finite_list* **by** *blast*
have *Snts*: *S* ∈ *set* (*nts ps*) **using** *True psP* **by** (*simp add*: *set_nts*)
define *Pg* **where** *Pg* = *set* (*gnf_of ps*)
— *gnf_of* produces a grammar in full Greibach normal form, preserving the language (minus ε).
have *GNF_Pg*: ∃ *a Bs*. *α* = *Tm a* # *map Nt Bs* **if** (*A*, *α*) ∈ *Pg* **for** *A α*
using *gnf_gnf_of*[*of ps*] *that* **by** (*auto simp*: *Pg_def GNF_def*)
have *LangPg*: *Lang Pg S* = *Lang P S* − {[]}
using *lang_gnf_of*[*OF Snts*] *psP* **by** (*simp add*: *Pg_def*)
— A fresh start symbol *S0* copying *S*'s productions: meets *S_notin_rhs*, keeps the language.
have *finPg*: *finite* (*Nts Pg*) **by** (*simp add*: *Pg_def finite_Nts*)
define *S0* **where** *S0* = *fresh* (*Nts Pg*) *S*
have *S0_fresh*: *S0* ∉ *Nts Pg* **unfolding** *S0_def* **using** *finPg* **by** (*rule fresh_notIn*)
define *R* **where** *R* = *Pg* ∪ {(*S0*, *γ*) |*γ*. (*S*, *γ*) ∈ *Pg*}
have *LangR*: *Lang R S0* = *Lang P S* − {[]}
using *Lang_fresh_start*[*OF S0_fresh*, *of S*] *LangPg* **by** (*simp add*: *R_def*)
have *GNF_R*: ∃ *a Bs*. *α* = *Tm a* # *map Nt Bs* **if** (*A*, *α*) ∈ *R* **for** *A α*
using *that GNF_Pg* **by** (*auto simp*: *R_def*)
have *S0* ∉ *Rhs_Nts Pg* **using** *S0_fresh* **by** (*simp add*: *Nts_Lhss_Rhs_Nts*)
moreover have *Rhs_Nts* {(*S0*, *γ*) |*γ*. (*S*, *γ*) ∈ *Pg*} ⊆ *Rhs_Nts Pg* **by** (*auto simp*: *Rhs_Nts_def*)
ultimately have *notinRhsR*: *S0* ∉ *Rhs_Nts R* **by** (*auto simp*: *R_def Rhs_Nts_def*)
have *Snr*: *S0* ∉ *set Bs* **if** (*A*, *Tm a* # *map Nt Bs*) ∈ *R* **for** *A a Bs*

using *notinRhsR that* **by**(*auto simp*: *Nts_syms_def Rhs_Nts_def*)
— R is finite; pick a list representation *qs*.
have *finR*: *finite R*
proof −
have {(*S0*, γ) |γ. (*S*, γ) ∈ *Pg*} ⊆ ($\lambda\gamma$. (*S0*, γ)) ‘ (*snd* ‘ *Pg*) **by** *force*
moreover have *finite* (($\lambda\gamma$. (*S0*, γ)) ‘ (*snd* ‘ *Pg*)) **by** (*simp add*: *Pg_def*)
ultimately have *finite* {(*S0*, γ) |γ. (*S*, γ) ∈ *Pg*} **by** (*rule finite_subset*)
thus *?thesis* **by** (*simp add*: *R_def Pg_def*)
qed
obtain *qs* **where** *qsR*: *set qs* = *R* **using** *finR finite_list* **by** *blast*
have *std_qs*: (*A*, α) ∈ *set qs* ⟹ ∃ *a Bs*. α = *Tm a* # *map Nt Bs* **for** *A* α
using *GNF_R*[*of A* α] **by** (*simp add*: *qsR*)
have *Snr_qs*: (*A*, *Tm a* # *map Nt Bs*) ∈ *set qs* ⟹ *S0* ∉ *set Bs* **for** *A a Bs*
using *Snr*[*of A a Bs*] **by** (*simp add*: *qsR*)
— Index the (finite) nonterminals of the standard-form grammar injectively into *nat*; this index is exactly the *idx* the encoding requires.
have *finite* (*insert S0* (*Nts* (*set qs*))) **using** *finR qsR* **by** (*simp add*: *finite_Nts*)
from *finite_imp_inj_to_nat_seg*[*OF this*] **obtain** *idx* :: *'n* ⇒ *nat*
where *idxinj*: *inj_on idx* (*insert S0* (*Nts* (*set qs*))) **by** *blast*
have *key*: *Lang* (*set qs*) *S0* − {[]} = *inv_hom* (*enc_h idx S0 qs*) (*L0* − {[]})
by (*rule Greibach_2_1_std*[*OF std_qs Snr_qs idxinj*])
have *Lang* (*set qs*) *S0* = *Lang P S* − {[]} **using** *qsR LangR* **by** *simp*
hence *Lang* (*set qs*) *S0* − {[]} = *Lang P S* − {[]} **by** *auto*
with *key* **have** *Lang P S* − {[]} = *inv_hom* (*enc_h idx S0 qs*) (*L0* − {[]}) **by** *simp*
thus *?thesis* **by** *blast*
qed

1.4.8 Theorem 2.1 in full generality

Greibach's Theorem 2.1 for an *arbitrary* finite grammar, over *any* nonterminal type: every context-free language, minus ε, is the inverse homomorphic image of the one fixed hardest language $L_0 - \{\varepsilon\}$. Since P is finite, its (finitely many) nonterminals can be renamed injectively into *nat* by *finite_imp_inj_to_nat_seg*; the renamed grammar is over *nat*, which has a fresh-symbol supply, so *Greibach_2_1_fresh0* applies, and the renaming preserves the language by *Lang_rename_Prods*.

theorem *Greibach_2_1*:
fixes *P* :: (*'n*,*'t*) *Prods*
assumes *finP*: *finite P*
shows ∃ *h*::*'t* ⇒ *t0 list*. *Lang P S* − {[]} = *inv_hom h* (*L0* − {[]})
proof −
have *finNts*: *finite* (*Nts P* ∪ {*S*}) **using** *finP* **by** (*simp add*: *finite_Nts*)
from *finite_imp_inj_to_nat_seg*[*OF finNts*] **obtain** *f* :: *'n* ⇒ *nat*
where *f*: *inj_on f* (*Nts P* ∪ {*S*}) **by** *blast*
have *finite* (*rename_Prods f P*) **using** *finP* **by** *simp*
from *Greibach_2_1_fresh0*[*OF this*, *of f S*] **obtain** *h* :: *'t* ⇒ *t0 list*
where *Lang* (*rename_Prods f P*) (*f S*) − {[]} = *inv_hom h* (*L0* − {[]}) **by**

blast
moreover have *Lang* (*rename_Prods f P*) (*f S*) = *Lang P S* **using** *f* **by** (*rule Lang_rename_Prods*)
ultimately have *Lang P S* − {[]} = *inv_hom h* (*L0* − {[]}) **by** *simp*
thus *?thesis* **by** *blast*
qed

The same statement at the level of languages: every context-free language, minus ε, is the inverse homomorphic image of $L_0 - \{\varepsilon\}$. Together with *CFL_L0* below (L_0 is itself context-free) this is the precise sense in which L_0 is a *hardest* context-free language.

corollary *Greibach_2_1_CFL*:
assumes *CFL TYPE*('*n*) *L*
shows ∃ *h*. *L* − {[]} = *inv_hom h* (*L0* − {[]})
proof −
from *assms* **obtain** *P* **and** *S* :: '*n* **where** *L*: *L* = *Lang P S* **and** *finP*: *finite P*
by (*auto simp*: *CFL_def*)
from *Greibach_2_1*[*OF finP*, *of S*] **show** *?thesis* **unfolding** *L* **by** *blast*
qed

1.5 L_0 is context-free

This property is left implicit in Greibach's paper.

1.5.1 A grammar for L_0

abbreviation (*input*) *oA1* ≡ *Aa* (*Open A1*)
abbreviation (*input*) *cA1* ≡ *Aa* (*Close A1*)
abbreviation (*input*) *oA2* ≡ *Aa* (*Open A2*)
abbreviation (*input*) *cA2* ≡ *Aa* (*Close A2*)

datatype *N* = *NZ* | *NF* | *NM*

definition *G* :: (*N*, *t0*) *Prods* **where**
G = {(*NZ*, []),
(*NZ*, [*Nt NF*, *Tm Cc*, *Tm Ce*, *Nt NM*, *Tm Cc*, *Nt NF*, *Tm Dd*]),
(*NF*, []),
(*NM*, []),
(*NM*, [*Nt NM*, *Nt NM*]),
(*NM*, [*Tm oA1*, *Nt NM*, *Tm cA1*]),
(*NM*, [*Tm oA2*, *Nt NM*, *Tm cA2*]),
(*NM*, [*Tm Cc*, *Nt NF*, *Tm Dd*, *Nt NF*, *Tm Cc*])}
∪ {(*NF*, [*Tm t*, *Nt NF*]) | *t*. *t* ∈ *Talph*}

lemma *bracks_eq*: *bracks* = {*oA1*, *cA1*, *oA2*, *cA2*}
using *t0_A.exhaust* **by** (*auto simp*: *bracks_def*)

lemma *finite_Talph*: *finite Talph*

by (*simp add*: *Talph_def bracks_eq*)

lemma *finite_G*: *finite G*
using *finite_Talph* **by** (*simp add*: *G_def full_SetCompr_eq*)

1.5.2 The free language T^* generated by NF ($\supseteq$ direction)

lemma *Talph_sub_Lang_NF*:
set w $\subseteq$ *Talph* $\Longrightarrow$ *w* $\in$ *Lang G NF*
proof (*induction w*)
case *Nil*
thus *?case* **using** *Lang_I*[*of NF* [] *G* []] **by** (*simp add*: *G_def*)
next
case (*Cons t w'*)
hence *t*: *t* $\in$ *Talph* **and** *w'*: *w'* $\in$ *Lang G NF* **by** *auto*
have *prod*: (*NF*, [*Tm t*, *Nt NF*]) $\in$ *G* **using** *t* **by** (*auto simp*: *G_def*)
have *t* # *w'* $\in$ *inst_syms* (*Lang G*) [*Tm t*, *Nt NF*] **using** *w'* **by** (*auto simp*: *conc_def*)
thus *?case* **using** *Lang_I*[*OF prod*] **by** *blast*
qed

1.5.3 The "middle" language generated by NM

inductive *mbal* :: *t0 list* $\Rightarrow$ *bool* **where**
mbal_Nil: *mbal* []
| *mbal_app*: *mbal w* $\Longrightarrow$ *mbal w'* $\Longrightarrow$ *mbal* (*w* @ *w'*)
| *mbal_wrap*: *mbal w* $\Longrightarrow$ *mbal* (*Aa* (*Open a*) # *w* @ [*Aa* (*Close a*)])
| *mbal_gap*: *set z* $\subseteq$ *Talph* $\Longrightarrow$ *set x* $\subseteq$ *Talph* $\Longrightarrow$ *mbal* (*Cc* # *z* @ *Dd* # *x* @ [*Cc*])

lemma *mbal_imp_Lang_NM*: *mbal w* $\Longrightarrow$ *w* $\in$ *Lang G NM*
proof (*induction rule*: *mbal.induct*)
case *mbal_Nil*
have (*NM*, []) $\in$ *G* **by** (*simp add*: *G_def*)
thus *?case* **using** *Lang_I*[*of NM* [] *G* []] **by** *simp*
next
case (*mbal_app w w'*)
have *prod*: (*NM*, [*Nt NM*, *Nt NM*]) $\in$ *G* **by** (*simp add*: *G_def*)
have *w* @ *w'* $\in$ *inst_syms* (*Lang G*) [*Nt NM*, *Nt NM*]
using *mbal_app.IH* **by** (*auto simp*: *conc_def*)
thus *?case* **using** *Lang_I*[*OF prod*] **by** *blast*
next
case (*mbal_wrap w a*)
show *?case*
proof (*cases a*)
case *A1*
have *prod*: (*NM*, [*Tm oA1*, *Nt NM*, *Tm cA1*]) $\in$ *G* **by** (*simp add*: *G_def*)
have *Aa* (*Open a*) # *w* @ [*Aa* (*Close a*)] $\in$ *inst_syms* (*Lang G*) [*Tm oA1*, *Nt NM*, *Tm cA1*]
using *mbal_wrap.IH A1* **by** (*auto simp*: *conc_def*)

thus *?thesis* **using** *Lang_I*[*OF prod*] **by** *blast*
next
case *A2*
have *prod*: (*NM*, [*Tm oA2*, *Nt NM*, *Tm cA2*]) ∈ *G* **by** (*simp add*: *G_def*)
have *Aa* (*Open a*) # *w* @ [*Aa* (*Close a*)] ∈ *inst_syms* (*Lang G*) [*Tm oA2*, *Nt NM*, *Tm cA2*]
using *mbal_wrap.IH A2* **by** (*auto simp*: *conc_def*)
thus *?thesis* **using** *Lang_I*[*OF prod*] **by** *blast*
qed
next
case (*mbal_gap z x*)
have *prod*: (*NM*, [*Tm Cc*, *Nt NF*, *Tm Dd*, *Nt NF*, *Tm Cc*]) ∈ *G* **by** (*simp add*: *G_def*)
have *Cc* # *z* @ *Dd* # *x* @ [*Cc*] ∈ *inst_syms* (*Lang G*) [*Tm Cc*, *Nt NF*, *Tm Dd*, *Nt NF*, *Tm Cc*]
using *Talph_sub_Lang_NF*[*OF mbal_gap.hyps*(*1*)] *Talph_sub_Lang_NF*[*OF mbal_gap.hyps*(*2*)]
by (*auto simp*: *conc_def*)
thus *?case* **using** *Lang_I*[*OF prod*] **by** *blast*
qed

1.5.4 Gaps and the interleaving of bracket blocks with gaps

definition *Gap* :: *t0 list set* **where**
Gap = {*Cc* # *z* @ *Dd* # *x* @ [*Cc*] | *x z*. *set z* ⊆ *Talph* ∧ *set x* ⊆ *Talph*}

lemma *GapI*: *set z* ⊆ *Talph* ⟹ *set x* ⊆ *Talph* ⟹ *Cc* # *z* @ *Dd* # *x* @ [*Cc*] ∈ *Gap*
by (*auto simp*: *Gap_def*)

fun *interl* :: *'a list list* ⇒ *'a list list* ⇒ *'a list* **where**
interl [] _ = []
| *interl* (*y*#*ys*) [] = *y* @ *interl ys* []
| *interl* (*y*#*ys*) (*g*#*gs*) = *y* @ *g* @ *interl ys gs*

lemma *interl_merge*:
length ya = *length gs1* ⟹
interl (*ya* @ [*yl*]) *gs1* @ *interl* (*yr* # *ys2*) *gs2*
= *interl* (*ya* @ (*yl* @ *yr*) # *ys2*) (*gs1* @ *gs2*)
proof (*induction ya arbitrary*: *gs1*)
case *Nil*
thus *?case* **by** (*cases gs2*) *auto*
next
case (*Cons a ya'*)
then obtain *g gs1'* **where** *gs1* = *g* # *gs1'* **by** (*cases gs1*) *auto*
with *Cons* **show** *?case* **by** *auto*
qed

Ymid W: *W* is a sequence of bracket blocks separated by gaps, whose concatenation is balanced.

definition *Ymid* :: *t0 list* ⇒ *bool* **where**
  Ymid W ⟷ (∃ *ys gs*. *W* = *interl ys gs* ∧ *length ys* = *Suc* (*length gs*) ∧
    (∀ *g*∈*set gs*. *g* ∈ *Gap*) ∧ (∀ *y*∈*set ys*. *set y* ⊆ *bracks*) ∧
    bal (*map brk* (*concat ys*)))

lemma *Ymid_Nil*: *Ymid* []
  unfolding *Ymid_def*
  by (*rule exI*[*of* _ [[]]], *rule exI*[*of* _ []]) *auto*

lemma *Ymid_gap*: *g* ∈ *Gap* ⟹ *Ymid g*
  unfolding *Ymid_def*
  by (*rule exI*[*of* _ [[],[]]], *rule exI*[*of* _ [*g*]]) *auto*

lemma *Ymid_app*:
  assumes *Ymid Wa* **and** *Ymid Wb* **shows** *Ymid* (*Wa* @ *Wb*)
proof −
  from *assms*(*1*) **obtain** *ys1 gs1* **where** *Wa*: *Wa* = *interl ys1 gs1* **and** *len1*: *length ys1* = *Suc* (*length gs1*)
    and *g1*: ∀ *g*∈*set gs1*. *g* ∈ *Gap* **and** *y1*: ∀ *y*∈*set ys1*. *set y* ⊆ *bracks*
    and *bl1*: *bal* (*map brk* (*concat ys1*)) **by** (*auto simp*: *Ymid_def*)
  from *assms*(*2*) **obtain** *ys2 gs2* **where** *Wb*: *Wb* = *interl ys2 gs2* **and** *len2*: *length ys2* = *Suc* (*length gs2*)
    and *g2*: ∀ *g*∈*set gs2*. *g* ∈ *Gap* **and** *y2*: ∀ *y*∈*set ys2*. *set y* ⊆ *bracks*
    and *bl2*: *bal* (*map brk* (*concat ys2*)) **by** (*auto simp*: *Ymid_def*)
  from *len1* **obtain** *ya yl* **where** *ys1*: *ys1* = *ya* @ [*yl*] **and** *lenya*: *length ya* = *length gs1*
    by (*metis length_Suc_conv_rev*)
  from *len2* **obtain** *yr ys2'* **where** *ys2*: *ys2* = *yr* # *ys2'* **by** (*cases ys2*) *auto*
  let *?ys* = *ya* @ (*yl* @ *yr*) # *ys2'*
  let *?gs* = *gs1* @ *gs2*
  have *wcat*: *Wa* @ *Wb* = *interl ?ys ?gs*
    using *Wa Wb ys1 ys2 interl_merge*[*OF lenya*, *of yl yr ys2' gs2*] **by** *simp*
  have *lenc*: *length ?ys* = *Suc* (*length ?gs*)
    using *lenya len1 len2 ys1 ys2* **by** *simp*
  have *ccat*: *concat ?ys* = *concat ys1* @ *concat ys2*
    using *ys1 ys2* **by** *simp*
  have *bal* (*map brk* (*concat ?ys*))
    using *bl1 bl2 ccat* **by** *simp*
  moreover have ∀ *g*∈*set ?gs*. *g* ∈ *Gap* **using** *g1 g2* **by** *auto*
  moreover have ∀ *y*∈*set ?ys*. *set y* ⊆ *bracks*
    using *y1 y2 ys1 ys2* **by** *auto*
  ultimately have *Wa* @ *Wb* = *interl ?ys ?gs* ∧ *length ?ys* = *Suc* (*length ?gs*) ∧
     (∀ *g*∈*set ?gs*. *g* ∈ *Gap*) ∧ (∀ *y*∈*set ?ys*. *set y* ⊆ *bracks*) ∧ *bal* (*map brk* (*concat ?ys*))
    using *wcat lenc* **by** *blast*
  thus *?thesis* **unfolding** *Ymid_def* **by** *blast*
qed

lemma *interl_prepend_first*: *interl* ((*p* @ *y*) # *ys*) *gs* = *p* @ *interl* (*y* # *ys*) *gs*

by (*cases gs*) *auto*

lemma *interl_append_last*:
length ys = *length gs* ⟹ *interl* (*ys* @ [*yl* @ *s*]) *gs* = *interl* (*ys* @ [*yl*]) *gs* @ *s*
by (*metis append.right_neutral interl.simps*(*1*,*2*) *interl_merge*)

lemma *Ymid_wrap*:
assumes *Ymid W* **shows** *Ymid* (*Aa* (*Open a*) # *W* @ [*Aa* (*Close a*)])
proof −
from *assms* **obtain** *ys gs* **where** *W*: *W* = *interl ys gs* **and** *len*: *length ys* = *Suc* (*length gs*)
and *gG*: ∀ *g*∈*set gs*. *g* ∈ *Gap* **and** *yB*: ∀ *y*∈*set ys*. *set y* ⊆ *bracks*
and *bl*: *bal* (*map brk* (*concat ys*)) **by** (*auto simp*: *Ymid_def*)
from *len* **have** *ys* ≠ [] **by** *auto*
then obtain *yf ys*′ **where** *ys*: *ys* = *yf* # *ys*′ **by** (*cases ys*) *auto*
show *?thesis*
proof (*cases ys*′ = [])
case *True*
with *ys* **have** *ys1*: *ys* = [*yf*] **by** *simp*
with *len* **have** *gnil*: *gs* = [] **by** *auto*
let *?ys* = [*Aa* (*Open a*) # *yf* @ [*Aa* (*Close a*)]]
have *eqW*: *Aa* (*Open a*) # *W* @ [*Aa* (*Close a*)] = *interl ?ys* []
using *W ys1 gnil* **by** *simp*
have *mb1*: *map brk* (*concat ?ys*) = *Open a* # *map brk* (*concat ys*) @ [*Close a*]
using *ys1* **by** *simp*
have *bal* (*map brk* (*concat ?ys*)) **unfolding** *mb1* **by** (*rule bal.intros*(*3*)[*OF bl*])
moreover have ∀ *y*∈*set ?ys*. *set y* ⊆ *bracks*
using *yB ys1* **by** (*auto simp*: *bracks_def*)
ultimately show *?thesis* **using** *eqW gnil* **unfolding** *Ymid_def*
by (*intro exI*[*of* _ *?ys*] *exI*[*of* _ []]) *simp*
next
case *False*
then obtain *ysm yl* **where** *ys*′: *ys*′ = *ysm* @ [*yl*] **by** (*metis rev_exhaust*)
let *?ys* = (*Aa* (*Open a*) # *yf*) # *ysm* @ [*yl* @ [*Aa* (*Close a*)]]
have *lenfm*: *length* (*yf* # *ysm*) = *length gs*
using *len ys ys*′ **by** *simp*
have *eqW*: *Aa* (*Open a*) # *W* @ [*Aa* (*Close a*)] = *interl ?ys gs*
proof −
have *interl ?ys gs* = *Aa* (*Open a*) # *interl* (*yf* # *ysm* @ [*yl* @ [*Aa* (*Close a*)]]) *gs*
using *interl_prepend_first*[*of* [*Aa* (*Open a*)] *yf ysm* @ [*yl* @ [*Aa* (*Close a*)]] *gs*] **by** *simp*
also have *interl* (*yf* # *ysm* @ [*yl* @ [*Aa* (*Close a*)]]) *gs*
= *interl* ((*yf* # *ysm*) @ [*yl* @ [*Aa* (*Close a*)]]) *gs* **by** *simp*
also have ... = *interl* ((*yf* # *ysm*) @ [*yl*]) *gs* @ [*Aa* (*Close a*)]
using *interl_append_last*[*OF lenfm*] **by** *simp*
also have *interl* ((*yf* # *ysm*) @ [*yl*]) *gs* = *W* **using** *W ys ys*′ **by** *simp*
finally show *?thesis* **by** *simp*
qed

have *ccat*: *concat ?ys = Aa (Open a) # concat ys @ [Aa (Close a)]*
using *ys ys'* **by** *simp*
have *mb*: *map brk (concat ?ys) = Open a # map brk (concat ys) @ [Close a]*
using *ys ys'* **by** *simp*
have *bal (map brk (concat ?ys))*
unfolding *mb* **by** (*rule bal.intros(3)[OF bl]*)
moreover have ∀ *y∈set ?ys. set y ⊆ bracks*
using *yB ys ys'* **by** (*auto simp*: *bracks_def*)
moreover have *length ?ys = Suc (length gs)* **using** *lenfm* **by** *simp*
ultimately show *?thesis* **using** *eqW gG* **unfolding** *Ymid_def*
by (*intro exI*[*of _ ?ys*] *exI*[*of _ gs*]) *simp*
qed
qed

1.5.5 From *Ymid* to *mbal*: inserting gaps into a balanced skeleton

lemma *Gap_imp_mbal*: *g ∈ Gap ⟹ mbal g*
by (*auto simp*: *Gap_def mbal_gap*)

inductive *sprd* :: *t0 list ⇒ t0_A bracket list ⇒ bool* **where**
sprd_Nil: *sprd [] []*
| *sprd_gap*: *g ∈ Gap ⟹ sprd W bw ⟹ sprd (g @ W) bw*
| *sprd_brk*: *sprd W bw ⟹ sprd (Aa b # W) (b # bw)*

lemma *sprd_emptybw*: *sprd W [] ⟹ mbal W*
proof (*induction W [] :: t0_A bracket list rule*: *sprd.induct*)
case *sprd_Nil* **thus** *?case* **by** (*simp add*: *mbal_Nil*)
next
case (*sprd_gap g W*) **thus** *?case* **by** (*simp add*: *Gap_imp_mbal mbal_app*)
qed

lemma *sprd_brk_list*:
set y ⊆ bracks ⟹ sprd Wr bw ⟹ sprd (y @ Wr) (map brk y @ bw)
proof (*induction y*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons c y'*) **thus** *?case* **using** *sprd_brk* **by** (*auto simp*: *bracks_def*)
qed

lemma *interl_sprd*:
length ys = Suc (length gs) ⟹ (∀ g∈set gs. g ∈ Gap) ⟹ (∀ y∈set ys. set y ⊆ bracks)
⟹ sprd (interl ys gs) (map brk (concat ys))
proof (*induction ys arbitrary*: *gs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons y ys'*)
show *?case*
proof (*cases gs*)

```
   case Nil then show ?thesis using Cons.prems sprd_brk_list[OF _ sprd_Nil]
by auto
  next
   case (Cons g gs') thus ?thesis using sprd_brk_list sprd_gap Cons.IH Cons.prems
by auto
  qed
qed

lemma sprd_split:
  sprd W bw ⟹ bw = bw1 @ bw2 ⟹ ∃ W1 W2. W = W1 @ W2 ∧ sprd W1
bw1 ∧ sprd W2 bw2
proof (induction arbitrary: bw1 bw2 rule: sprd.induct)
  case sprd_Nil
  thus ?case by (auto intro: sprd.sprd_Nil)
next
  case (sprd_gap g W bw)
  thus ?case using sprd_gap.IH[OF sprd_gap.prems]
    by (metis append.assoc sprd.sprd_gap)
next
  case (sprd_brk W bw b)
  show ?case
  proof (cases bw1)
    case Nil
    thus ?thesis using sprd_brk by (auto intro: sprd.sprd_Nil sprd.sprd_brk)
  next
    case (Cons c bw1')
    thus ?thesis
      using sprd_brk sprd.sprd_brk by (metis append_Cons list.inject)
  qed
qed

lemma sprd_head:
  sprd W bw ⟹ bw = b # bw' ⟹ ∃ U Wr. W = U @ Aa b # Wr ∧ mbal U ∧
sprd Wr bw'
proof (induction arbitrary: b bw' rule: sprd.induct)
  case sprd_Nil thus ?case by simp
next
  case (sprd_gap g W bw)
  thus ?case
    using Gap_imp_mbal mbal_app by (metis append.assoc)
next
  case (sprd_brk W bw c)
  thus ?case by (auto intro: mbal_Nil)
qed

lemma sprd_tail:
  sprd W bw ⟹ bw = bw' @ [b] ⟹ ∃ Wl U. W = Wl @ Aa b # U ∧ sprd Wl
bw' ∧ mbal U
proof (induction arbitrary: bw' b rule: sprd.induct)
```

```
  case sprd_Nil thus ?case by simp
next
  case (sprd_gap g W bw)
  with sprd_gap.IH[OF sprd_gap.prems] show ?case
    by (metis append.assoc sprd.sprd_gap)
next
  case (sprd_brk W bw c)
  show ?case
  proof (cases bw')
    case Nil
    thus ?thesis using sprd_brk sprd_emptybw
      by (auto intro: sprd.sprd_Nil)
  next
    case (Cons d bw'')
    thus ?thesis
      using sprd_brk sprd.sprd_brk by (metis append_Cons list.inject)
  qed
qed

lemma sprd_bal_imp_mbal: bal bw ⟹ sprd W bw ⟹ mbal W
proof (induction arbitrary: W rule: bal.induct[case_names Emp App Wrap])
  case Emp
  thus ?case by (rule sprd_emptybw)
next
  case (App xs ys)
  with sprd_split[OF App.prems refl] show ?case by(auto simp add: mbal_app)
next
  case (Wrap xs a)
  from sprd_head[OF Wrap.prems refl] obtain U Wr
    where Wdef: W = U @ Aa (Open a) # Wr and mU: mbal U
      and sWr: sprd Wr (xs @ [Close a]) by blast
  from sprd_tail[OF sWr refl] obtain Wl U2
    where Wrdef: Wr = Wl @ Aa (Close a) # U2 and sWl: sprd Wl xs and mU2:
mbal U2 by blast
  show ?case
    using Wdef Wrdef using Wrap.IH sWl mbal_app[OF mU mbal_app[OF _
mU2]] mbal_wrap
    by fastforce
qed
```

1.5.6 Bridging *Ymid* and the block-list form of L_0

From a balanced *Ymid*-decomposition build the L_0 block list: the gaps split into the z_i/x_{i+1} parts of adjacent blocks.

```
lemma mkblocks:
  length ys = Suc (length gs) ⟹ (∀ g∈set gs. g ∈ Gap) ⟹ set x ⊆ Talph ⟹ set
z ⊆ Talph ⟹
  ∃ bs. bs ≠ [] ∧ (∀ (a,b,c)∈set bs. set a ⊆ Talph ∧ set c ⊆ Talph) ∧
      map (λ(a,b,c). b) bs = ys ∧
```

```
      concat (map blk bs) = x @ [Cc] @ interl ys gs @ [Cc] @ z @ [Dd]
proof (induction ys arbitrary: gs x)
  case Nil thus ?case by simp
next
  case (Cons y0 ys')
  show ?case
  proof (cases gs = [])
    case True
    thus ?thesis using Cons by (intro exI[of _ [(x, y0, z)]]) (simp add: blk_def)
  next
    case False
    then obtain g gs' where gs: gs = g # gs' by (cases gs) auto
    from Cons.prems(2) gs obtain zz xx where g: g = Cc # zz @ Dd # xx @
[Cc]
      and zzT: set zz ⊆ Talph and xxT: set xx ⊆ Talph by (auto simp: Gap_def)
    have len': length ys' = Suc (length gs') using Cons.prems(1) gs by simp
    have ∀ g∈set gs'. g ∈ Gap using Cons.prems(2) gs by simp
    from Cons.IH[OF len' this xxT Cons.prems(4)] obtain bs' where
      bs'ne: bs' ≠ [] and ac': ∀ (a,b,c)∈set bs'. set a ⊆ Talph ∧ set c ⊆ Talph
      and mid': map (λ(a,b,c). b) bs' = ys'
      and cc': concat (map blk bs') = xx @ [Cc] @ interl ys' gs' @ [Cc] @ z @ [Dd]
by blast
    have concat (map blk ((x, y0, zz) # bs')) = x @ [Cc] @ interl (y0 # ys') gs @
[Cc] @ z @ [Dd]
      using cc' g gs by (simp add: blk_def)
    moreover have ∀ (a,b,c)∈set ((x, y0, zz) # bs'). set a ⊆ Talph ∧ set c ⊆
Talph
      using ac' Cons.prems(3) zzT by auto
    moreover have map (λ(a,b,c). b) ((x, y0, zz) # bs') = y0 # ys' using mid'
by simp
    ultimately show ?thesis using bs'ne by (intro exI[of _ (x, y0, zz) # bs']) 
simp
  qed
qed
```

The inverse: every L_0 block list arises this way.

```
lemma blocks_decomp:
  bs ≠ [] ⟹ (∀ (a,b,c)∈set bs. set a ⊆ Talph ∧ set c ⊆ Talph) ⟹
   ∃ x ys gs z. set x ⊆ Talph ∧ set z ⊆ Talph ∧ length ys = Suc (length gs) ∧
(∀ g∈set gs. g ∈ Gap) ∧
     map (λ(a,b,c). b) bs = ys ∧
     concat (map blk bs) = x @ [Cc] @ interl ys gs @ [Cc] @ z @ [Dd]
proof (induction bs)
  case Nil thus ?case by simp
next
  case (Cons abc bs')
  obtain a b c where abc: abc = (a, b, c) by (cases abc)
  have aT: set a ⊆ Talph and cT: set c ⊆ Talph using Cons.prems(2) abc by
auto
```

```
  show ?case
  proof (cases bs' = [])
    case True thus ?thesis
      using abc aT cT by (auto simp add: blk_def)
  next
    case False
   have ac': ∀(a,b,c)∈set bs'. set a ⊆ Talph ∧ set c ⊆ Talph using Cons.prems(2)
by auto
    from Cons.IH[OF False ac'] obtain x ys gs z where
      xT: set x ⊆ Talph and zT: set z ⊆ Talph and len: length ys = Suc (length
gs)
      and gG: ∀g∈set gs. g ∈ Gap and mid: map (λ(a,b,c). b) bs' = ys
      and cc: concat (map blk bs') = x @ [Cc] @ interl ys gs @ [Cc] @ z @ [Dd] by
blast
    have gGap: (Cc # c @ Dd # x @ [Cc]) ∈ Gap using cT xT by (rule GapI)
    have concat (map blk ((a, b, c) # bs'))
       = a @ [Cc] @ interl (b # ys) ((Cc # c @ Dd # x @ [Cc]) # gs) @ [Cc] @
z @ [Dd]
      using cc by (simp add: blk_def)
   moreover have length (b # ys) = Suc (length ((Cc # c @ Dd # x @ [Cc]) #
gs)) using len by simp
    moreover have ∀g∈set ((Cc # c @ Dd # x @ [Cc]) # gs). g ∈ Gap using
gG gGap by simp
    moreover have map (λ(a,b,c). b) ((a, b, c) # bs') = b # ys using mid by
simp
    ultimately show ?thesis using abc aT zT
      by (intro exI[of _ a] exI[of _ b # ys] exI[of _ (Cc # c @ Dd # x @ [Cc])
# gs] exI[of _ z]) auto
  qed
qed
```

The two halves of the characterisation $L_0 - \{\varepsilon\} = T^* \; c \; ¢ \; (Ymid) \; c \; T^* \; d$.

```
lemma Ymid_imp_L0:
  assumes Y: Ymid W and xT: set x ⊆ Talph and zT: set z ⊆ Talph
  shows x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd] ∈ L0
proof −
  from Y obtain ys gs where W: W = interl ys gs and len: length ys = Suc
(length gs)
    and gG: ∀g∈set gs. g ∈ Gap and yB: ∀y∈set ys. set y ⊆ bracks
    and bl: bal (map brk (concat ys)) by (auto simp: Ymid_def)
  from len have ys ≠ [] by auto
  then obtain y0 ys' where ys: ys = y0 # ys' by (cases ys) auto
  have len2: length ((Ce # y0) # ys') = Suc (length gs) using len ys by simp
  from mkblocks[OF len2 gG xT zT] obtain bs where bsP:
    bs ≠ [] ∧ (∀(a,b,c)∈set bs. set a ⊆ Talph ∧ set c ⊆ Talph) ∧
     map (λ(a,b,c). b) bs = (Ce # y0) # ys' ∧
     concat (map blk bs) = x @ [Cc] @ interl ((Ce # y0) # ys') gs @ [Cc] @ z @
[Dd] ..
  have ceW: interl ((Ce # y0) # ys') gs = Ce # W
```

```
    using interl_prepend_first[of [Ce] y0 ys' gs] W ys by simp
  have word: concat (map blk bs) = x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd]
    using bsP ceW by simp
  have midtl: map (λ(a,b,c). b) (tl bs) = ys' by (simp add: map_tl bsP)
  have tlmid: set b ⊆ bracks if tabc: (a,b,c)∈set (tl bs) for a b c
    using tabc midtl yB ys  by force
  have midD: ∃ v∈D. concat (map (λ(a,b,c). b) bs) = Ce # v
  proof −
    have concat ys ∈ D by (simp add: D_def UN_least bl yB)
    then show ?thesis using bsP ys by simp
  qed
  have concat (map blk bs) ∈ L0 unfolding L0_def using bsP tlmid midD by
blast
  thus ?thesis using word by simp
qed

lemma L0_imp_Ymid:
  assumes w ∈ L0 and w ≠ []
  shows ∃ x z W. w = x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd] ∧ set x ⊆ Talph ∧
set z ⊆ Talph ∧ Ymid W
proof −
  from assms obtain bs where wbs: w = concat (map blk bs) and bsne: bs ≠ []
    and acT: ∀ (x,y,z)∈set bs. set x ⊆ Talph ∧ set z ⊆ Talph
    and tlb: ∀ (x,y,z)∈set (tl bs). set y ⊆ bracks
    and cD: ∃ v∈D. concat (map (λ(x,y,z). y) bs) = Ce # v
    by (auto simp: L0_def)
  from blocks_decomp[OF bsne acT] obtain x ys gs z where
    xT: set x ⊆ Talph and zT: set z ⊆ Talph and len: length ys = Suc (length gs)
    and gG: ∀ g∈set gs. g ∈ Gap and bmid: map (λ(a,b,c). b) bs = ys
    and wcc: concat (map blk bs) = x @ [Cc] @ interl ys gs @ [Cc] @ z @ [Dd] by
blast
  from cD obtain v where vD: v ∈ D and cyv: concat (map (λ(x,y,z). y) bs) =
Ce # v by blast
  have ysv: concat ys = Ce # v using cyv bmid by simp
  from len obtain y0 ys' where ys: ys = y0 # ys' by (cases ys) auto
  have ys'brk: ∀ y∈set ys'. set y ⊆ bracks
    using bmid ys tlb by (fastforce simp add: map_tl)
  have y0ne: y0 ≠ []
  proof
    assume y0nil: y0 = []
    have Ce ∈ set (concat ys) using ysv by simp
    moreover have set (concat ys) = set (concat ys') using ys y0nil by simp
    ultimately show False using ys y0nil ys'brk by (auto simp add: bracks_def)
  qed
  then obtain h b0 where y0split: y0 = h # b0 by (cases y0) auto
  have h = Ce and bv: b0 @ concat ys' = v using ysv ys y0split by auto
  hence y0: y0 = Ce # b0 using y0split by simp
  let ?W = interl (b0 # ys') gs
  have ceW: interl ys gs = Ce # ?W
```

using *interl_prepend_first*[*of* [*Ce*] *b0 ys' gs*] *ys y0* **by** *simp*
have *wword*: $w = x$ @ [*Cc*, *Ce*] @ *?W* @ [*Cc*] @ *z* @ [*Dd*]
using *wbs wcc ceW* **by** *simp*
have *b0brk*: *set b0* ⊆ *bracks* **using** *bv vD* **by** (*auto simp*: *D_def*)
have *Ymid ?W* **using** *bv b0brk vD ys ys'brk gG len* **unfolding** *Ymid_def D_def mem_Collect_eq*
by (*metis* (*no_types*, *lifting*) *concat.simps*(*2*) *length_Cons set_ConsD*)
thus *?thesis* **using** *wword xT zT* **by** *blast*
qed

1.6 *Lang G NZ* = L_0 and context-freeness of L_0

The candidate solution assigning each nonterminal its intended language.

definition *Rsol* :: *N* ⇒ *t0 list set* **where**
Rsol A = (*case A of NZ* ⇒ *L0* | *NF* ⇒ {*w*. *set w* ⊆ *Talph*} | *NM* ⇒ {*W*. *Ymid W*})

Each production keeps the candidate solution closed (the ⊆-obligations).

lemma *incl_NZ_big*: *inst_syms Rsol* [*Nt NF*, *Tm Cc*, *Tm Ce*, *Nt NM*, *Tm Cc*, *Nt NF*, *Tm Dd*] ⊆ *Rsol NZ*
proof
fix *w* **assume** *w* ∈ *inst_syms Rsol* [*Nt NF*, *Tm Cc*, *Tm Ce*, *Nt NM*, *Tm Cc*, *Nt NF*, *Tm Dd*]
then obtain *x W z* **where** *w*: $w = x$ @ [*Cc*, *Ce*] @ *W* @ [*Cc*] @ *z* @ [*Dd*]
and *xT*: *set x* ⊆ *Talph* **and** *YW*: *Ymid W* **and** *zT*: *set z* ⊆ *Talph*
by (*auto simp*: *Rsol_def conc_def*)
show *w* ∈ *Rsol NZ*
using *Ymid_imp_L0*[*OF YW xT zT*] *w* **by** (*simp add*: *Rsol_def*)
qed

lemma *Lang_G_subset*: *Lang G A* ⊆ *Rsol A*
proof (*rule Lang_subset_if*)
fix *A* α **assume** *aG*: (*A*, α) ∈ *G*
consider (*A*,α) = (*NZ*, [])
| (*A*,α) = (*NZ*, [*Nt NF*, *Tm Cc*, *Tm Ce*, *Nt NM*, *Tm Cc*, *Nt NF*, *Tm Dd*])
| (*A*,α) = (*NF*, []) | (*A*,α) = (*NM*, []) | (*A*,α) = (*NM*, [*Nt NM*, *Nt NM*])
| (*A*,α) = (*NM*, [*Tm oA1*, *Nt NM*, *Tm cA1*]) | (*A*,α) = (*NM*, [*Tm oA2*, *Nt NM*, *Tm cA2*])
| (*A*,α) = (*NM*, [*Tm Cc*, *Nt NF*, *Tm Dd*, *Nt NF*, *Tm Cc*])
| *t* **where** *t* ∈ *Talph* (*A*,α) = (*NF*, [*Tm t*, *Nt NF*])
using *aG* **unfolding** *G_def* **by** (*elim UnE insertE CollectE exE conjE*) *simp_all*
then show *inst_syms Rsol* α ⊆ *Rsol A*
proof *cases*
case *1* **thus** *?thesis* **by** (*auto simp*: *Rsol_def L0_def*)
next
case *2* **thus** *?thesis* **using** *incl_NZ_big* **by** *simp*
next
case *3* **thus** *?thesis* **by** (*auto simp*: *Rsol_def*)

```
 next
  case 4 thus ?thesis by (auto simp: Rsol_def Ymid_Nil)
 next
  case 5 thus ?thesis by (auto simp: Rsol_def conc_def intro: Ymid_app)
 next
  case 6 thus ?thesis by (auto simp: Rsol_def conc_def intro: Ymid_wrap)
 next
  case 7 thus ?thesis by (auto simp: Rsol_def conc_def intro: Ymid_wrap)
 next
  case 8 thus ?thesis by (auto simp: Rsol_def conc_def intro: Ymid_gap GapI)
 next
  case (9 t) thus ?thesis by (auto simp: Rsol_def conc_def)
 qed
qed
```

The $\supseteq$ direction for the start symbol.

```
lemma Ymid_imp_Lang_NM: Ymid W ⟹ W ∈ Lang G NM
using Ymid_def interl_sprd sprd_bal_imp_mbal mbal_imp_Lang_NM by auto

lemma L0_subset_Lang: L0 ⊆ Lang G NZ
proof
 fix w assume wL0: w ∈ L0
 show w ∈ Lang G NZ
 proof (cases w = [])
  case True
  thus ?thesis using Lang_I[of NZ [] G []] True by (simp add: G_def)
 next
  case False
  from L0_imp_Ymid[OF wL0 False] obtain x z W where
   w: w = x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd]
   and xT: set x ⊆ Talph and zT: set z ⊆ Talph and YW: Ymid W by blast
  have prod: (NZ, [Nt NF, Tm Cc, Tm Ce, Nt NM, Tm Cc, Nt NF, Tm Dd]) ∈
G by (simp add: G_def)
   have w ∈ inst_syms (Lang G) [Nt NF, Tm Cc, Tm Ce, Nt NM, Tm Cc, Nt
NF, Tm Dd]
  proof −
   let ?ws = [x, [Cc], [Ce], W, [Cc], z, [Dd]]
   let ?al = [Nt NF, Tm Cc, Tm Ce, Nt NM, Tm Cc, Nt NF, Tm Dd]
   have ∀ i < length ?ws. ?ws ! i ∈ inst_sym (Lang G) (?al ! i)
   using Talph_sub_Lang_NF[OF xT] Talph_sub_Lang_NF[OF zT] Ymid_imp_Lang_NM[OF
YW]
    by (auto simp: nth_Cons' split: if_splits)
   from inst_syms_decomp[OF this] show ?thesis using w by simp
  qed
  thus ?thesis using Lang_I[OF prod] by blast
 qed
qed

lemma Lang_G_NZ: Lang G NZ = L0
```

using *Lang_G_subset*[*of NZ*] *L0_subset_Lang* **by** (*auto simp*: *Rsol_def*)

Greibach's hardest language L_0 is itself context-free.

theorem *CFL_L0*: *CFL TYPE*(*N*) *L0*
using *CFL_def Lang_G_NZ finite_G* **by** *blast*

end

References

[1] S. A. Greibach. The hardest context-free language. *SIAM J. Comput.*, 2(4):304–310, 1973.