

Greibach's Hardest Context-Free Language

Tobias Nipkow

October 6, 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 $Cons_power :: 'a \Rightarrow nat \Rightarrow 'a\ list$ (**infixl** $\#^{\wedge} 70$) **where**
 $a \#^{\wedge} n \equiv replicate\ n\ a$

1.1 The hardest language L_0

1.1.1 The terminal alphabet of L_0

Greibach's alphabet is $T = \{a_1, a_2, |a_1, |a_2, c, \mathfrak{c}\}$ 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 $\mathfrak{c}$, and Dd for d .

datatype $t0_A = A1 \mid A2$

datatype $t0 = Aa \ t0_A \ bracket \mid Cc \mid Ce \mid Dd$

1.1.2 The Dyck set D on two letters

D is the Dyck set generated by $S \rightarrow SS \mid a_1 S \mid a_1 \mid a_2 S \mid a_2 \mid \varepsilon$, 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**
 $blk = (\lambda(x,y,z). \ x \ @ \ Cc \ \# \ y \ @ \ Cc \ \# \ z \ @ \ [Dd])$

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

definition $L0 :: t0 \ list \ set$ **where**
 $L0 = \{\square\} \cup$
 $\{ \ concat \ (map \ blk \ bs) \mid bs. \$
 $bs \neq \square \wedge$
 $(\forall (x,y,z) \in set \ bs. \ set \ x \subseteq Talph \wedge set \ z \subseteq Talph) \wedge$
 $(\forall (x,y,z) \in set \ (tl \ bs). \ set \ y \subseteq bracks) \wedge$
 $(\exists v \in D. \ concat \ (map \ (\lambda(x,y,z). \ y) \ bs) = Ce \ \# \ v) \}$

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 $inv_hom :: ('a \Rightarrow 'b \text{ list}) \Rightarrow 'b \text{ list set} \Rightarrow 'a \text{ list set}$ **where**
 $inv_hom \ h \ L = \{w. \text{concat} (\text{map } h \ w) \in L\}$

The nonterminals $Y_1, \dots, Y_n$ (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 $pushcode :: nat \Rightarrow t0 \text{ list}$ **where**
 $pushcode \ i = Aa \ (Open \ A1) \ \# \ (Aa \ (Open \ A2) \ \#^{\wedge} i) \ @ \ [Aa \ (Open \ A1)]$

definition $popcode :: nat \Rightarrow t0 \text{ list}$ **where**
 $popcode \ i = Aa \ (Close \ A1) \ \# \ (Aa \ (Close \ A2) \ \#^{\wedge} i) \ @ \ [Aa \ (Close \ A1)]$

$\xi(p)$ for a standard-form production $p = (Y_i \rightarrow a \ Y_{j1} \dots Y_{jm})$: 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 $Y_{j1} \dots Y_{jm}$ become the new top, Y_{j1} rightmost.

definition $xi :: ('n \Rightarrow nat) \Rightarrow ('n, 't) \text{ prod} \Rightarrow t0 \text{ list}$ **where**
 $xi \ idx \ p =$
 $\quad popcode \ (idx \ (fst \ p)) \ @$
 $\quad concat \ (\text{map} \ (\lambda s. \text{case } s \text{ of } Nt \ B \Rightarrow pushcode \ (idx \ B) \ | \ Tm \ _ \Rightarrow []) \ (\text{rev} \ (tl \ (snd \ p))))$

$\hat{\xi}(p)$: 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 $xihat :: ('n \Rightarrow nat) \Rightarrow 'n \Rightarrow ('n, 't) \text{ prod} \Rightarrow t0 \text{ list}$ **where**
 $xihat \ idx \ S \ p = (\text{if } fst \ p = S \text{ then } Cc \ \# \ pushcode \ (idx \ S) \ @ \ xi \ idx \ p \text{ else } xi \ idx \ p)$

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

definition $enc_h :: ('n \Rightarrow nat) \Rightarrow 'n \Rightarrow ('n, 't) \text{ prods} \Rightarrow 't \Rightarrow t0 \text{ list}$ **where**
 $enc_h \ idx \ S \ ps \ a =$
 $\quad concat \ (\text{map} \ (\lambda p. \ Cc \ \# \ xihat \ idx \ S \ p)$
 $\quad \quad (\text{filter} \ (\lambda p. \ \exists Bs. \ snd \ p = Tm \ a \ \# \ \text{map } Nt \ Bs) \ ps))$
 $\quad @ \ [Cc, \ Dd]$

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 $brks :: t0 \text{ list} \Rightarrow t0_A \text{ bracket list}$ **where**

$brks\ w \equiv map\ brk\ w$

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 $frag :: nat \Rightarrow t0_A\ list$ **where**
 $frag\ i = A1 \# A2 \# \hat{i} @ [A1]$

definition $stkenc :: nat\ list \Rightarrow t0_A\ list$ **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 $xi_eq:$
 $snd\ p = Tm\ a \# map\ Nt\ Bs \implies$
 $xi\ idx\ p = popcode\ (idx\ (fst\ p)) @ concat\ (map\ pushcode\ (rev\ (map\ idx\ Bs)))$
by ($simp\ add: xi_def\ rev_map\ o_def$)

Running $\xi(p)$ for a standard-form production $p = (A \rightarrow a\ B_1 \dots B_m)$ on a stack whose top is A : it pops A and pushes $B_1 \dots B_m$ (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 $set_pushcode: set\ (pushcode\ i) \subseteq bracks$ **by** ($auto\ simp: pushcode_def\ bracks_def$)

lemma *set_popcode*: $\text{set } (\text{popcode } i) \subseteq \text{bracks}$ **by** (*auto simp*: *popcode_def bracks_def*)

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

lemma *set_xi*: $\text{set } (xi \text{ } idx \text{ } p) \subseteq \text{bracks}$
unfolding *xi_def* **using** *set_popcode set_pushcode* **by** (*auto*)

lemma *set_xihat*: $\text{set } (xihat \text{ } idx \text{ } S \text{ } p) \subseteq \text{insert } C \text{ } \text{bracks}$
using *set_xi*[*of idx p*] *set_pushcode*[*of idx S*] **by** (*auto simp*: *xihat_def*)

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

lemma *map_xihat_no_S*:
 $\forall p \in \text{set } ps. \text{fst } p \neq S \implies \text{map } (xihat \text{ } idx \text{ } S) \text{ } ps = \text{map } (xi \text{ } idx) \text{ } ps$
by (*induction ps*) (*auto simp*: *xihat_def*)

1.3.3 Block decomposition of the encoded word

lemma *set_xihat_Talph*: $\text{set } (xihat \text{ } idx \text{ } S \text{ } q) \subseteq \text{Talph}$
using *set_xihat*[*of idx S q*] **by** (*auto simp*: *Talph_def bracks_def*)

lemma *set_Cc_xihat*: $\text{set } (Cc \# xihat \text{ } idx \text{ } S \text{ } q) \subseteq \text{Talph}$
using *set_xihat_Talph*[*of idx S q*] **by** (*auto simp*: *Talph_def*)

lemma *set_concat_Cc_xihat*: $\text{set } (\text{concat } (\text{map } (\lambda q. Cc \# xihat \text{ } idx \text{ } S \text{ } q) \text{ } qs)) \subseteq \text{Talph}$
using *set_Cc_xihat*[**where** *idx=idx*] **by** *auto*

lemma *set_concat_xihat_Cc*: $\text{set } (\text{concat } (\text{map } (\lambda q. xihat \text{ } idx \text{ } S \text{ } q @ [Cc]) \text{ } qs)) \subseteq \text{Talph}$
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*:
 $\text{concat } (\text{map } (\lambda p. Cc \# xihat \text{ } idx \text{ } S \text{ } p) \text{ } qs) @ [Cc, Dd]$
 $= Cc \# \text{concat } (\text{map } (\lambda q. xihat \text{ } idx \text{ } S \text{ } q @ [Cc]) \text{ } qs) @ [Dd]$
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*: $p \in \text{set } P$ **and** *snd_p*: $\text{snd } p = Tm \text{ } a \# \text{map } Nt \text{ } Bs$
shows $\exists x \text{ } z. \text{enc_h } idx \text{ } S \text{ } P \text{ } a = \text{blk } (x, xihat \text{ } idx \text{ } S \text{ } p, z) \wedge \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph}$
proof –

```

have  $p \in \text{set } (\text{filter } (\lambda p. \exists Bs. \text{snd } p = \text{Tm } a \# \text{map } \text{Nt } Bs) P)$  using  $\text{pin } \text{snd\_}p$ 
by auto
then obtain  $qs1 \ qs2$  where  $\text{split}: \text{filter } (\lambda p. \exists Bs. \text{snd } p = \text{Tm } a \# \text{map } \text{Nt } Bs)$ 
 $P = qs1 \ @ \ p \ \# \ qs2$ 
by (meson split_list)
define  $x$  where  $x = \text{concat } (\text{map } (\lambda p. Cc \# \text{xihat } \text{idx } S \ p) \ qs1)$ 
define  $z$  where  $z = \text{concat } (\text{map } (\lambda q. \text{xihat } \text{idx } S \ q \ @ \ [Cc]) \ qs2)$ 
have  $\text{enc\_}h \ \text{idx } S \ P \ a = \text{blk } (x, \text{xihat } \text{idx } S \ p, z)$  by (simp add: blk_def x_def
 $z\_def \ cc\_shift2 \ \text{enc\_}h \ \text{def } \text{split}$ )
moreover have  $\text{set } x \subseteq \text{Talph}$  unfolding  $x\_def$  by (rule set_concat_Cc_xihat)
moreover have  $\text{set } z \subseteq \text{Talph}$  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 $\hat{\xi}$ -codes of the productions.

lemma *block_decomp*:

```

 $\text{list\_all2 } (\lambda b \ p. p \in \text{set } P \wedge (\exists Bs. \text{snd } p = \text{Tm } b \# \text{map } \text{Nt } Bs)) \ w \ ps \implies$ 
 $\exists bs. \text{concat } (\text{map } (\text{enc\_}h \ \text{idx } S \ P) \ w) = \text{concat } (\text{map } \text{blk } bs)$ 
 $\wedge \text{map } (\lambda(x,y,z). y) \ bs = \text{map } (\text{xihat } \text{idx } S) \ ps$ 
 $\wedge (\forall (x,y,z) \in \text{set } bs. \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph})$ 

```

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** $\text{pin}: p \in \text{set } P$ **and** $\text{snd_}p: \text{snd } p = \text{Tm}$
 $b \# \text{map } \text{Nt } Bs$ **by** *blast*

from $\text{enc_}h \ \text{block}[OF \ \text{pin } \text{snd_}p]$ **obtain** $x \ z$ **where**

$\text{blk_eq}: \text{enc_}h \ \text{idx } S \ P \ b = \text{blk } (x, \text{xihat } \text{idx } S \ p, z)$ **and**

$xT: \text{set } x \subseteq \text{Talph}$ **and** $zT: \text{set } z \subseteq \text{Talph}$ **by** *blast*

from *Cons.IH* **obtain** bs **where**

$bs_eq: \text{concat } (\text{map } (\text{enc_}h \ \text{idx } S \ P) \ w') = \text{concat } (\text{map } \text{blk } bs)$ **and**

$bs_y: \text{map } (\lambda(x,y,z). y) \ bs = \text{map } (\text{xihat } \text{idx } S) \ ps'$ **and**

$bs_T: \forall (x,y,z) \in \text{set } bs. \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph}$ **by** *blast*

let $?bs = (x, \text{xihat } \text{idx } S \ p, z) \# bs$

have $\text{concat } (\text{map } (\text{enc_}h \ \text{idx } S \ P) \ (b \# w')) = \text{concat } (\text{map } \text{blk } ?bs)$

by (*simp add: blk_eq bs_eq*)

moreover have $\text{map } (\lambda(x,y,z). y) \ ?bs = \text{map } (\text{xihat } \text{idx } S) \ (p \# ps')$ **by** (*simp*
 $\text{add: } bs_y$)

moreover have $\forall (x,y,z) \in \text{set } ?bs. \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph}$ **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 $\text{frag } i = A1 \ A2^i \ A1$ determines i and the rest uniquely as a prefix.

lemma *frag_append_inj*:

$\text{frag } i \ @ \ xs = \text{frag } j \ @ \ ys \implies i = j \wedge xs = ys$

unfolding *frag_def* **by** (*auto simp add: append_Cons_eq_append_Cons*)

lemma *stkenc_eq_Nil*: $\text{stkenc } cs = [] \implies cs = []$

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 $\text{bal_stk } t \ (\text{brks } (\text{popcode } i)) = (s, [])$ **shows** $t = \text{frag } i \ @ \ s$

proof –

from *assms* **have** $A: \text{bal_stk } t \ (\text{replicate } 1 \ (\text{Close } A1) \ @ \ (\text{replicate } i \ (\text{Close } A2) \ @ \ [\text{Close } A1]))) = (s, [])$

by (*simp add: popcode_def*)

from *bal_stk_replicate_Close_inv[OF A]* **obtain** $t1$ **where**

$t1: t = \text{replicate } 1 \ A1 \ @ \ t1$ **and**

$B: \text{bal_stk } t1 \ (\text{replicate } i \ (\text{Close } A2) \ @ \ [\text{Close } A1]) = (s, [])$ **by** *blast*

from *bal_stk_replicate_Close_inv[OF B]* **obtain** $t2$ **where**

$t2: t1 = \text{replicate } i \ A2 \ @ \ t2$ **and** $C: \text{bal_stk } t2 \ [\text{Close } A1] = (s, [])$ **by** *blast*

have $t2 = A1 \ \# \ s$

proof (*cases t2*)

case *Nil* **thus** *?thesis* **using** C **by** *simp*

next

case ($\text{Cons } c \ t3$) **thus** *?thesis* **using** C **by** (*auto split: if_splits*)

qed

with $t1 \ t2$ **show** $t = \text{frag } i \ @ \ s$ **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*: $\text{snd } p = Tm \ a \ \# \ map \ Nt \ Bs$

and *fpN*: $\text{fst } p \in N$ **and** αN : $set \ \alpha \subseteq N$

and *run*: $\text{bal_stk } (\text{stkenc } (\text{map } idx \ \alpha)) \ (\text{brks } (\text{xi } idx \ p)) = (s, [])$

shows $\exists \alpha'. \ \alpha = \text{fst } p \ \# \ \alpha' \wedge s = \text{stkenc } (\text{map } idx \ (Bs \ @ \ \alpha'))$

proof –

from *run snd_p* **have**

$\text{bal_stk } (\text{stkenc } (\text{map } idx \ \alpha))$

$(\text{brks } (\text{popcode } (idx \ (\text{fst } p)))) \ @ \ \text{brks } (\text{concat } (\text{map } \text{pushcode } (\text{rev } (\text{map } idx \ Bs)))))) = (s, [])$

by (*simp add: xi_eq*)

then obtain s' **where**

```

    pop: bal_stk (stkenc (map idx  $\alpha$ )) (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  $\alpha$ ) = 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  $\alpha \neq []$ 
    proof
      assume  $\alpha = []$  with stk_eq show False by (simp add: frag_def stkenc_def)
    qed
    then obtain  $A' \alpha'$  where  $\alpha c$ :  $\alpha = A' \# \alpha'$  by (cases  $\alpha$ ) auto
    with stk_eq have frag (idx  $A'$ ) @ stkenc (map idx  $\alpha'$ ) = 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  $\alpha'$ ) =  $s'$ 
    by auto
    from  $\alpha c$   $\alpha N$  have  $A' \in N$  by simp
    with iA inj fpN have  $A' =$  fst p by (auto dest: inj_onD)
    with  $\alpha c$  have  $\alpha =$  fst p #  $\alpha'$  by simp
    moreover have  $s =$  stkenc (map idx (Bs @  $\alpha'$ ))
    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  $\implies$ 
  ( $\forall xs \in \text{set } xss. \exists p. xs = p @ [s] \wedge s \notin \text{set } p$ )  $\implies$ 
  ( $\forall ys \in \text{set } yss. \exists p. ys = p @ [s] \wedge s \notin \text{set } p$ )  $\implies xss = yss$ 

```

proof (induction xss arbitrary: yss)

case Nil

thus ?case **by** (cases yss) auto

next

case (Cons xs xss')

from Cons.prem1(2) **obtain** p1 **where** xs: xs = p1 @ [s] **and** p1: s $\notin$ set p1

by auto

show ?case

proof (cases yss)

case Nil

with Cons.prem1(1) xs **show** ?thesis **by** simp

next

case (Cons ys yss')

from Cons.prem1(3) $\langle yss = ys \# yss' \rangle$ **obtain** p2 **where** ys: ys = p2 @ [s]

and p2: s $\notin$ set p2

```

    by auto
  from Cons.prem1 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.prem2 Cons.prem3
  ⟨yss = ys # yss'⟩
  by auto
  with ⟨p1 = p2⟩ xs ys ⟨yss = ys # yss'⟩ show ?thesis by simp
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  $c \notin \text{set } y$  and  $fc: \bigwedge q. c \notin \text{set } (f q)$ 
  shows  $x @ c \# y @ c \# z = \text{concat } (\text{map } (\lambda q. c \# f q) \text{ qs}) @ [c] \implies y \in \text{set } (\text{map } f \text{ qs})$ 
proof (induction qs arbitrary: x z)
  case Nil thus ?case by (cases x) auto
next
  case (Cons q qs')
  have rhs:  $\text{concat } (\text{map } (\lambda q. c \# f q) (q \# qs')) @ [c]$ 
    =  $(c \# f q) @ (\text{concat } (\text{map } (\lambda q. c \# f q) \text{ qs}') @ [c])$  by simp
  let ?REST =  $\text{concat } (\text{map } (\lambda q. c \# f q) \text{ qs}') @ [c]$ 
  have restCc:  $?REST = c \# \text{tl } ?REST$  by (cases qs') auto
  from Cons.prem1 rhs have eq:  $x @ c \# y @ c \# z = (c \# f q) @ ?REST$  by
simp
  show ?case
  proof (cases x)
    case Nil
    with eq have  $y @ c \# z = f q @ c \# \text{tl } ?REST$  using restCc by simp
    from this asms(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
       $x' = f q @ t \wedge t @ (c \# y @ c \# z) = ?REST \vee x' @ t = f q \wedge c \# y @ c \# z = t @ ?REST$ 
    using append_eq_append_conv2[of  $x' c \# y @ c \# z f q ?REST$ ] by auto
    thus ?thesis
  proof
    assume  $x' = f q @ t \wedge t @ (c \# y @ c \# z) = ?REST$ 
    with Cons.IH show ?thesis by auto
  next
    assume  $x' @ t = f q \wedge c \# y @ c \# z = t @ ?REST$ 
    hence fqt:  $f q = x' @ t$  and ceq:  $c \# y @ c \# z = t @ ?REST$  by auto
    have ct:  $c \notin \text{set } t$  using fqt fc[of q] by auto
  qed

```

```

    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:  $\exists p. enc\_h\ idx\ S\ P\ a = p @ [Dd] \wedge Dd \notin set\ p$ 
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 ξ -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:  $set\ y \subseteq insert\ Ce\ bracks$ 
  shows  $\exists p. p \in set\ P \wedge (\exists Bs. snd\ p = Tm\ a \# map\ Nt\ Bs) \wedge y = xihat\ idx\ S\ p$ 
proof -
  let ?qs = filter ( $\lambda p. \exists Bs. snd\ p = Tm\ a \# map\ Nt\ Bs$ ) P
  have eq':  $x @ Cc \# y @ Cc \# z = concat\ (map\ (\lambda p. Cc \# xihat\ idx\ S\ p)\ ?qs)$ 
  @ [Cc]
  using eq by (simp add: blk_def enc_h_def)
  have ccy:  $Cc \notin set\ y$  using yset by (auto simp: bracks_def)
  have ccq:  $Cc \notin set\ (xihat\ idx\ S\ q)$  for q
  using set_xihat[of idx S q] by (auto simp: bracks_def)
  from cfrag_parse_gen[OF ccy ccq eq']
  show ?thesis by fastforce
qed

```

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

```

lemma concat_xihat_eqfst:
  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 ξ -codes are the middles.

```

lemma blocks_to_ps:
  list_all2 ( $\lambda b\ a. blk\ b = enc\_h\ idx\ S\ P\ a$ ) bs w  $\implies$ 
  ( $\forall (x,y,z) \in set\ bs. set\ y \subseteq insert\ Ce\ bracks$ )  $\implies$ 

```

```

 $\exists ps. list\_all2 (\lambda a p. p \in set P \wedge (\exists Bs. snd p = Tm a \# map Nt Bs)) w ps$ 
 $\wedge map (\lambda(x,y,z). y) bs = map (xihat idx S) ps$ 
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 $A \rightarrow a B_1 \dots B_m$ (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:  $\bigwedge A \alpha. (A, \alpha) \in set P \implies \exists a Bs. \alpha = Tm a \# map Nt Bs$ 
  and S_notin_rhs:  $\bigwedge A a Bs. (A, Tm a \# map Nt Bs) \in set P \implies S \notin set Bs$ 
  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:  $(A, Tm a \# map Nt Bs) \in set P \implies drives ps (Bs @ \alpha) w \beta \implies$ 
  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  $\alpha$  w  $\beta \implies$ 
  bal_stk (stkenc (map idx  $\alpha$ )) (brks (concat (map (xi idx) ps))) = (stkenc (map
  idx  $\beta$ ), [])
proof (induction rule: drives.induct)
  case (drives_Nil  $\alpha$ )
  show ?case by simp

```

```

next
  case (drives_Cons A a Bs ps  $\alpha$  w  $\beta$ )
  let ?p = (A, Tm a # map Nt Bs)
  show ?case using drives_Cons bal_stk_xi[of ?p a Bs idx stkenc (map idx  $\alpha$ )]
    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 $\mathfrak{c}$ (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_derives:
  drives ps  $\alpha$  w  $\beta$   $\implies$  set P  $\vdash$  map Nt  $\alpha \Rightarrow_l^* \text{map Tm } w @ \text{map Nt } \beta$ 
proof (induction rule: drives.induct)
  case (drives_Nil  $\alpha$ )
  show ?case by simp
next
  case (drives_Cons A a Bs ps  $\alpha$  w  $\beta$ )
  have step: set P  $\vdash$  Nt A # map Nt  $\alpha \Rightarrow_l \text{Tm } a \# \text{map Nt } (Bs @ \alpha)$ 
    using derivel.intros[OF drives_Cons.hyps(1), of [] map Nt  $\alpha$ ] by simp
  have set P  $\vdash$  Tm a # map Nt (Bs @  $\alpha$ )  $\Rightarrow_l^* \text{Tm } a \# (\text{map Tm } w @ \text{map Nt } \beta)$ 
    using drives_Cons.IH by (simp add: derivels_Tm_Cons)
  with step have set P  $\vdash$  Nt A # map Nt  $\alpha \Rightarrow_l^* \text{Tm } a \# (\text{map Tm } w @ \text{map Nt } \beta)$ 
    by (rule converse_rtranclp_into_rtranclp)
  thus ?case by simp
qed

```

Hard direction: a leftmost derivation to a terminal word yields a *drives* sequence. 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  $\alpha \Rightarrow_l(n) \text{map Tm } w \implies \exists ps. \text{drives } ps \alpha w []$ 
proof (induction n arbitrary:  $\alpha$  w)
  case 0
  hence map Nt  $\alpha = \text{map Tm } w$  by simp
  hence  $\alpha = [] \wedge w = []$  by (cases  $\alpha$ ; cases w; auto)
  thus ?case using drives_Nil by auto
next

```

```

case (Suc n)
show ?case
proof (cases  $\alpha$ )
  case Nil
  with Suc.premis have set  $P \vdash [] \Rightarrow l(\text{Suc } n) \text{ map } Tm \ w$  by simp
  hence False by (simp del: relpowp.simps(2))
  thus ?thesis ..
next
case (Cons A  $\alpha'$ )
  with Suc.premis have set  $P \vdash Nt \ A \ \# \ \text{map } Nt \ \alpha' \Rightarrow l(\text{Suc } n) \text{ map } Tm \ w$  by
simp
  then obtain  $\gamma$  where  $\gamma: (A, \gamma) \in \text{set } P$  and  $\text{der}: \text{set } P \vdash \gamma @ \text{map } Nt \ \alpha' \Rightarrow l(n) \text{ map } Tm \ w$ 
  by (auto simp: deriveln_Nt_Cons simp del: relpowp.simps(2))
  from std[OF  $\gamma$ ] obtain a Bs where  $\gamma \text{eq}: \gamma = Tm \ a \ \# \ \text{map } Nt \ Bs$  by blast
  from der  $\gamma \text{eq}$  have set  $P \vdash Tm \ a \ \# \ \text{map } Nt \ (Bs @ \alpha') \Rightarrow l(n) \text{ map } Tm \ w$  by
simp
  then obtain  $w'$  where  $\text{weq}: \text{map } Tm \ w = Tm \ a \ \# \ w'$  and  $\text{der}': \text{set } P \vdash \text{map } Nt \ (Bs @ \alpha') \Rightarrow l(n) \text{ map } Tm \ w'$ 
  by (auto simp: deriveln_Tm_Cons simp del: relpowp.simps(2))
  from  $\text{weq}$  obtain  $w''$  where  $w \text{eq}: w = a \ \# \ w''$  and  $w' \text{eq}: w' = \text{map } Tm \ w''$ 
  by (cases w) auto
  from  $\text{der}' \ w' \text{eq}$  have  $\text{der}'': \text{set } P \vdash \text{map } Nt \ (Bs @ \alpha') \Rightarrow l(n) \text{ map } Tm \ w''$  by
simp
  from Suc.IH[OF  $\text{der}''$ ] obtain ps where  $\text{ps}: \text{drives } ps \ (Bs @ \alpha') \ w'' []$  by blast
  have  $\text{drives } ((A, Tm \ a \ \# \ \text{map } Nt \ Bs) \ \# \ ps) \ (A \ \# \ \alpha') \ (a \ \# \ w'') []$ 
  using  $\gamma \ \gamma \text{eq} \ ps$  by (auto intro: drives_Cons)
  thus ?thesis using Cons  $w \text{eq}$  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*:

$(\exists ps. \text{drives } ps \ [S] \ w []) \longleftrightarrow w \in \text{Lang } (\text{set } P) \ S$

proof

assume $\exists ps. \text{drives } ps \ [S] \ w []$

with *drives_imp_derivels*

show $w \in \text{Lang } (\text{set } P) \ S$ **unfolding** *Lang_def*

using *derivels_iff_derives* by *fastforce*

next

assume $w \in \text{Lang } (\text{set } P) \ S$

hence $\text{set } P \vdash [Nt \ S] \Rightarrow l * \text{map } Tm \ w$ by (simp add: *Lang_def* *derivels_iff_derives*)

thus $\exists ps. \text{drives } ps \ [S] \ w []$

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*:
 $\text{drives } ps \ \alpha \ w \ \beta \implies S \notin \text{set } \alpha \implies (\forall p \in \text{set } ps. \text{fst } p \neq S)$
proof (*induction rule: drives.induct*)
 case (*drives_Nil* α) **show** ?case **by** *simp*
next
 case (*drives_Cons* $A \ a \ Bs \ ps \ \alpha \ w \ \beta$)
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 $\mathfrak{c}$ marker and the initial push of S ; all later productions contribute pure ξ -codes.

lemma *concat_xihat_eq*:
assumes $\text{drives } ps \ [S] \ w \ []$
shows $\text{concat } (\text{map } (\text{xihat } \text{idx } S) \ ps) = Ce \ \# \ \text{pushcode } (\text{idx } S) \ @ \ \text{concat } (\text{map } (\text{xihat } \text{idx } S) \ ps)$
proof –
 from *assms* **obtain** $a \ Bs \ ps' \ w'$ **where**
 $ps_eq: ps = (S, \text{Tm } a \ \# \ \text{map } \text{Nt } Bs) \ \# \ ps'$
and $\text{mem}: (S, \text{Tm } a \ \# \ \text{map } \text{Nt } Bs) \in \text{set } P$
and $\text{rest}: \text{drives } ps' \ Bs \ w' \ []$
by (*auto elim: drives.cases*)
have $S \notin \text{set } Bs$ **using** $S_notin_rhs[OF \ \text{mem}]$ **by** *simp*
with $\text{drives_no_S}[OF \ \text{rest}]$ **have** $\forall p \in \text{set } (\text{tl } ps). \text{fst } p \neq S$ **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*:
 $\text{drives } ps \ \alpha \ w \ \beta \implies \text{list_all2 } (\lambda b \ p. p \in \text{set } P \wedge (\exists Bs. \text{snd } p = \text{Tm } b \ \# \ \text{map } \text{Nt } Bs)) \ w \ ps$
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 $\mathfrak{c}$ followed by a balanced ($\in D$) word.

lemma *forward*:
assumes $\text{drv}: \text{drives } ps \ [S] \ w \ []$
shows $\text{concat } (\text{map } (\text{enc_h } \text{idx } S \ P) \ w) \in L_0$

proof –
from *drv* **obtain** *a Bs ps' w'* **where**
ps_eq: $ps = (S, Tm\ a\ \# \ map\ Nt\ Bs) \# ps'$
and *mem*: $(S, Tm\ a\ \# \ map\ Nt\ Bs) \in set\ P$
and *rest*: $drives\ ps'\ Bs\ w' \ []$
by (*auto elim*: *drives.cases*)
have *SnotBs*: $S \notin set\ Bs$ **using** *S_notin_rhs*[*OF mem*] **by** *simp*
have *noS*: $\forall p \in set\ ps'.\ fst\ p \neq S$ **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: $map\ (\lambda(x,y,z). y)\ bs = map\ (xihat\ idx\ S)\ ps$ **and**
bT: $\forall (x,y,z) \in set\ bs.\ set\ x \subseteq Talph \wedge set\ z \subseteq Talph$ **by** *blast*
have *bs_ne*: $bs \neq []$ **using** *by_eq ps_eq* **by** (*cases bs*) *auto*
let *?v* = *pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*)
have *ys*: $concat\ (map\ (\lambda(x,y,z). y)\ bs) = Ce \# ?v$
using *by_eq concat_xihat_eq*[*OF drv*] **by** *simp*
have *vD*: $?v \in D$
proof –
have *bal* (*brks ?v*) **by** (*rule drives_bal_complete*[*OF drv*])
moreover **have** $set\ ?v \subseteq bracks$
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*: $map\ (\lambda(x,y,z). y)\ (tl\ bs) = map\ (xihat\ idx\ S)\ ps'$
proof –
have $map\ (\lambda(x,y,z). y)\ bs = map\ (xihat\ idx\ S)\ ((S, Tm\ a\ \# \ map\ Nt\ Bs) \# ps')$
using *by_eq ps_eq* **by** *simp*
thus *?thesis* **by** (*cases bs*) *auto*
qed
have *key*: $set\ y \subseteq bracks$ **if** $(x,y,z) \in set\ (tl\ bs)$ **for** *x y z*
proof –
from *that* **have** $y \in set\ (map\ (\lambda(x,y,z). y)\ (tl\ bs))$ **by** *force*
then obtain *p* **where** $p \in set\ ps'$ **and** *yp*: $y = xihat\ idx\ S\ p$ **using** *tl_y* **by** *auto*
have *fst p* $\neq S$ **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*: $\forall (x,y,z) \in set\ (tl\ bs).\ set\ y \subseteq bracks$ **using** *key* **by** *fast*
have *exv*: $\exists v \in D.\ concat\ (map\ (\lambda(x,y,z). y)\ bs) = Ce \# v$ **using** *ys vD* **by** *blast*
have $concat\ (map\ blk\ bs) \in L0$
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 $drives\ ps\ \alpha\ w\ []$. 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 $drives_bal_stk_inv$:

$list_all2\ (\lambda b\ p.\ p \in set\ P \wedge (\exists Bs.\ snd\ p = Tm\ b\ \# \ map\ Nt\ Bs))\ w\ ps \implies$
 $set\ \alpha \subseteq insert\ S\ (Nts\ (set\ P)) \implies$
 $bal_stk\ (stkenc\ (map\ idx\ \alpha))\ (brks\ (concat\ (map\ (xi\ idx)\ ps))) = ([], []) \implies$
 $drives\ ps\ \alpha\ w\ []$

proof (*induction w ps arbitrary: α rule: list_all2_induct*)

case ($Nil\ \alpha$)

from $Nil.prem1(2)$ **have** $stkenc\ (map\ idx\ \alpha) = []$ **by** *simp*

hence $\alpha = []$ **using** $stkenc_eq_Nil$ **by** *auto*

thus $?case$ **by** (*simp add: drives.drives_Nil*)

next

case ($Cons\ b\ w'\ p\ ps'\ \alpha$)

from $Cons.hyps(1)$ **obtain** Bs **where** pin : $p \in set\ P$ **and** snd_p : $snd\ p = Tm\ b\ \# \ map\ Nt\ Bs$ **by** *blast*

have fpN : $fst\ p \in insert\ S\ (Nts\ (set\ P))$ **using** pin **by** (*force simp: Nts_Lhss_Rhs_Nts Lhss_def*)

have BsN : $set\ Bs \subseteq insert\ S\ (Nts\ (set\ P))$

proof –

have ($fst\ p, Tm\ b\ \# \ map\ Nt\ Bs$) = p **using** snd_p **by** (*cases p*) *auto*

with pin **have** ($fst\ p, Tm\ b\ \# \ map\ Nt\ Bs$) $\in set\ P$ **by** *simp*

hence $Nts_syms\ (Tm\ b\ \# \ map\ Nt\ Bs) \subseteq Rhs_Nts\ (set\ P)$ **by** (*auto simp: Rhs_Nts_def*)

moreover **have** $set\ Bs \subseteq Nts_syms\ (Tm\ b\ \# \ map\ Nt\ Bs)$ **by** (*auto simp: Nts_syms_def*)

ultimately **have** $set\ Bs \subseteq Rhs_Nts\ (set\ P)$ **by** *blast*

also **have** $Rhs_Nts\ (set\ P) \subseteq insert\ S\ (Nts\ (set\ P))$ **by** (*auto simp: Nts_Lhss_Rhs_Nts*)

finally **show** $?thesis$.

qed

from $Cons.prem1(1)$ **have** αN : $set\ \alpha \subseteq insert\ S\ (Nts\ (set\ P))$.

from $Cons.prem1(2)$ **have** $bal_stk\ (stkenc\ (map\ idx\ \alpha))$

$(brks\ (xi\ idx\ p) @ brks\ (concat\ (map\ (xi\ idx)\ ps')))) = ([], [])$ **by** *simp*

then **obtain** s' **where**

step: $bal_stk\ (stkenc\ (map\ idx\ \alpha))\ (brks\ (xi\ idx\ p)) = (s', [])$ **and**

rest: $bal_stk\ s'\ (brks\ (concat\ (map\ (xi\ idx)\ ps')))) = ([], [])$

by (*rule bal_stk_append_split*)

from $bal_stk_xi_inv[OF\ idx_inj\ snd_p\ fpN\ \alpha N\ step]$ **obtain** α' **where**

αeq : $\alpha = fst\ p\ \# \ \alpha'$ **and** $s' eq$: $s' = stkenc\ (map\ idx\ (Bs @ \alpha'))$ **by** *blast*

from $\alpha eq\ \alpha N$ **have** $set\ \alpha' \subseteq insert\ S\ (Nts\ (set\ P))$ **by** *auto*

with BsN **have** $Bs \alpha N$: $set\ (Bs @ \alpha') \subseteq insert\ S\ (Nts\ (set\ P))$ **by** *auto*

from *rest s'eq* **have**

$bal_stk (stkenc (map\ idx\ (Bs\ @\ \alpha')) (brks\ (concat\ (map\ (xi\ idx)\ ps')))) = ([],$
 $[]) \text{ by } simp$
from $Cons.IH[OF\ Bs\ \alpha\ N\ this]$ **have** $driv': drives\ ps'\ (Bs\ @\ \alpha')\ w'\ []$.
have $peq: p = (fst\ p, Tm\ b\ \# \ map\ Nt\ Bs)$ **using** snd_p **by** $(cases\ p)\ auto$
with pin **have** $memP: (fst\ p, Tm\ b\ \# \ map\ Nt\ Bs) \in set\ P$ **by** $simp$
from $drives_Cons[OF\ memP\ driv']$
have $drives\ ((fst\ p, Tm\ b\ \# \ map\ Nt\ Bs)\ \# \ ps')\ (fst\ p\ \# \ \alpha')\ (b\ \# \ w')\ []$.
thus $?case$ **using** $\alpha eq\ peq$ **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 $inL0: concat\ (map\ (enc_h\ idx\ S\ P)\ w) \in L0$
and $ne: concat\ (map\ (enc_h\ idx\ S\ P)\ w) \neq []$
shows $\exists ps. drives\ ps\ [S]\ w\ []$

proof –

from $inL0\ ne$ **have** $\exists bs. concat\ (map\ (enc_h\ idx\ S\ P)\ w) = concat\ (map\ blk\ bs)$
 $\wedge\ bs \neq [] \wedge$
 $(\forall (x,y,z) \in set\ bs. set\ x \subseteq Talph \wedge set\ z \subseteq Talph) \wedge$
 $(\forall (x,y,z) \in set\ (tl\ bs). set\ y \subseteq bracks) \wedge$
 $(\exists v \in D. concat\ (map\ (\lambda(x,y,z). y)\ bs) = Ce\ \# \ v)$
by $(auto\ simp: L0_def)$

then obtain bs where

$cc: concat\ (map\ (enc_h\ idx\ S\ P)\ w) = concat\ (map\ blk\ bs)$ **and**
 $bs_ne0: bs \neq []$ **and**
 $xzT: \forall (x,y,z) \in set\ bs. set\ x \subseteq Talph \wedge set\ z \subseteq Talph$ **and**
 $tlb: \forall (x,y,z) \in set\ (tl\ bs). set\ y \subseteq bracks$ **and**
 $cD: \exists v \in D. concat\ (map\ (\lambda(x,y,z). y)\ bs) = Ce\ \# \ v$
by $blast$

from cD **obtain** v **where** $vD: v \in D$ **and** $yv: concat\ (map\ (\lambda(x,y,z). y)\ bs) = Ce\ \# \ v$ **by** $blast$

— Every middle is c -free (it is part of the $\mathfrak{c}/D$ -word).

have $y_{sub_one}: set\ y \subseteq insert\ Ce\ bracks$ **if** $(x,y,z) \in set\ bs$ **for** $x\ y\ z$

proof –

from $that$ **have** $y \in set\ (map\ (\lambda(x,y,z). y)\ bs)$ **by** $force$
hence $set\ y \subseteq set\ (concat\ (map\ (\lambda(x,y,z). y)\ bs))$ **by** $auto$
also $have\ \dots = insert\ Ce\ (set\ v)$ **using** yv **by** $simp$
also $have\ \dots \subseteq insert\ Ce\ bracks$ **using** vD **by** $(auto\ simp: D_def)$
finally $show\ ?thesis$.

qed

have $y_{sub}: \forall (x,y,z) \in set\ bs. set\ y \subseteq insert\ Ce\ bracks$ **using** y_{sub_one} **by** $fast$

— Each *blk*-block ends in exactly one d , just like each *enc_h*-block.

have $blkDd$: $\exists p. b = p @ [Dd] \wedge Dd \notin set\ p$ **if** $b \in set\ (map\ blk\ bs)$ **for** b
proof –
from *that* **obtain** t **where** tin : $t \in set\ bs$ **and** bt : $b = blk\ t$ **by** *auto*
obtain $x\ y\ z$ **where** t : $t = (x, y, z)$ **by** (*cases* t)
have $Dd \notin set\ (x @ Cc \# y @ Cc \# z)$
using $y_{sub}\ tin\ t\ xzT$ **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 $\forall xs \in set\ (map\ blk\ bs). \exists p. xs = p @ [Dd] \wedge Dd \notin set\ p$ **using** $blkDd$ **by** *blast*
show $\forall ys \in set\ (map\ (enc_h\ idx\ S\ P)\ w). \exists p. ys = p @ [Dd] \wedge Dd \notin set\ p$
using enc_h_Dd **by** *auto*
qed

— Recover a matching production list ps whose $\hat{\xi}$ -codes are the middles.
have len : $length\ bs = length\ w$ **using** $align$ **by** (*metis length_map*)
have $la2blk$: $list_all2\ (\lambda b\ a. blk\ b = enc_h\ idx\ S\ P\ a)\ bs\ w$
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 $blocks_to_ps[OF\ la2blk\ y_{sub}]$ **obtain** ps **where**
 $la2match$: $list_all2\ (\lambda a\ p. p \in set\ P \wedge (\exists Bs. snd\ p = Tm\ a \# map\ Nt\ Bs))\ w$
 ps **and**
 y_{ps} : $map\ (\lambda(x, y, z). y)\ bs = map\ (xihat\ idx\ S)\ ps$ **by** *blast*

have $xihatps$: $concat\ (map\ (xihat\ idx\ S)\ ps) = Ce \# v$
proof –
have $concat\ (map\ (xihat\ idx\ S)\ ps) = concat\ (map\ (\lambda(x, y, z). y)\ bs)$ **using** y_{ps}
by *simp*
thus *?thesis* **using** yv **by** *simp*
qed

— w (hence ps) is non-empty.
have w_ne : $w \neq []$ **using** ne **by** *auto*
have ps_ne : $ps \neq []$ **using** $la2match\ w_ne$ **by** (*cases* w ; *cases* ps) *auto*
obtain $p0\ ps'$ **where** p_{sc} : $ps = p0 \# ps'$ **using** ps_ne **by** (*cases* ps) *auto*

— The first production expands S (its code begins with c).
have $fp0$: $fst\ p0 = S$
proof (*rule ccontr*)
assume $ne0$: $fst\ p0 \neq S$
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* $\neq []$ **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* ($\lambda(x,y,z). y$) (*tl bs*) = *map* (*xihat idx S*) *ps'*
by (*simp add: map_tl psc yps*)
have *tlS*: *fst p* $\neq S$ **if** *asm*: $p \in \text{set } ps'$ **for** *p*
proof —
from *asm* **have** *xihat idx S p* $\in \text{set}$ (*map* (*xihat idx S*) *ps'*) **by** *simp*
also **have** *set* (*map* (*xihat idx S*) *ps'*) = *set* (*map* ($\lambda(x,y,z). y$) (*tl bs*))
using *tlmap* **by** *simp*
finally **have** *xihat idx S p* $\in \text{set}$ (*map* ($\lambda(x,y,z). y$) (*tl bs*)) .
then **obtain** *x y z* **where** *bin*: $(x,y,z) \in \text{set} (tl bs)$ **and** *yeq*: *xihat idx S p* = *y*
by *auto*
have *set y* $\subseteq \text{bracks}$ **using** *tlb bin* **by** *auto*
hence *Ce* $\notin \text{set}$ (*xihat idx S p*) **using** *yeq* **by** (*auto simp: bracks_def*)
thus *fst p* $\neq S$ **by** (*auto simp: xihat_def*)
qed
have *tlps*: $\forall p \in \text{set} (tl ps). \text{fst } p \neq S$ **using** *tlS psc* **by** *simp*

— Assemble the balanced run and invoke the converse central invariant.
have *Ce* $\# v = Ce \# \text{pushcode} (idx S) @ \text{concat} (map (xi idx) ps)$
using *xihatps concat_xihat_eqfst[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]* $\subseteq \text{insert } S (Nts (\text{set } P))$ **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*: $Lang (set P) S - \{\varepsilon\} = inv_hom (enc_h \text{ idx } S P) (L_0 - \{\varepsilon\})$

proof (*rule set_eqI*)

fix w

show $(w \in Lang (set P) S - \{\varepsilon\}) = (w \in inv_hom (enc_h \text{ idx } S P) (L_0 - \{\varepsilon\}))$

proof

assume $w \in Lang (set P) S - \{\varepsilon\}$

hence wL : $w \in Lang (set P) S$ **and** wne : $w \neq \varepsilon$ **by** *auto*

from wL **obtain** ps **where** $drives\ ps\ [S]\ w\ []$ **using** $drives_iff_Lang$ **by** *blast*

hence $inL0$: $concat (map (enc_h \text{ idx } S P) w) \in L_0$ **by** (*rule forward*)

have $concat (map (enc_h \text{ idx } S P) w) \neq \varepsilon$

proof $-$

from wne **obtain** $a\ w'$ **where** wc : $w = a \# w'$ **by** (*cases w*) *auto*

obtain p **where** ep : $enc_h \text{ idx } S P\ a = p @ [Dd]$ **using** $enc_h_Dd[of\ \text{idx } S\ P\ a]$ **by** *blast*

show $?thesis$ **by** (*simp add: wc ep*)

qed

with $inL0$ **show** $w \in inv_hom (enc_h \text{ idx } S P) (L_0 - \{\varepsilon\})$ **by** (*simp add: inv_hom_def*)

next

assume $w \in inv_hom (enc_h \text{ idx } S P) (L_0 - \{\varepsilon\})$

hence $inL0$: $concat (map (enc_h \text{ idx } S P) w) \in L_0$

and ne : $concat (map (enc_h \text{ idx } S P) w) \neq \varepsilon$ **by** (*auto simp: inv_hom_def*)

from $backward_drives[OF\ inL0\ ne]$ **obtain** ps **where** $drives\ ps\ [S]\ w\ []$ **by** *blast*

hence $w \in Lang (set P) S$ **using** $drives_iff_Lang$ **by** *blast*

moreover **have** $w \neq \varepsilon$ **using** ne **by** *auto*

ultimately **show** $w \in Lang (set P) S - \{\varepsilon\}$ **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 $enc_h \text{ idx } S P$. 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 $\bigwedge A\ \alpha. (A, \alpha) \in set\ P \implies \exists a\ Bs. \alpha = Tm\ a \# map\ Nt\ Bs$

```

    and  $\bigwedge A \ a \ Bs. (A, Tm \ a \ \# \ map \ Nt \ Bs) \in set \ P \implies S \notin set \ Bs$ 
    and  $inj\_on \ idx \ (insert \ S \ (Nts \ (set \ P)))$ 
    shows  $Lang \ (set \ P) \ S - \{\emptyset\} = inv\_hom \ (enc\_h \ idx \ S \ P) \ (L0 - \{\emptyset\})$ 
  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  $Lang \ (R \cup \{(A', \gamma) \mid \gamma. (A, \gamma) \in R\}) \ A' = Lang \ R \ A$ 
proof -
  let ?N =  $\{(A', \gamma) \mid \gamma. (A, \gamma) \in R\}$ 
  have notinL:  $A' \notin Lhss \ R$  and notinR:  $A' \notin Rhss\_Nts \ R$ 
    using assms by (auto simp: Nts_Lhss_Rhss_Nts)
  have lhssN:  $Lhss \ ?N \subseteq \{A'\}$  by (auto simp: Lhss_def)
  have disj:  $Rhss\_Nts \ R \cap Lhss \ ?N = \{\}$  using lhssN notinR by auto
  have rhs_sub:  $A' \notin Nts\_syms \ \gamma$  if  $(A, \gamma) \in R$  for  $\gamma$ 
    using notinR that by (auto simp: Rhss_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  $\gamma$ 
    where  $\gamma R$ :  $(A', \gamma) \in R \cup ?N$  and  $d$ :  $R \cup ?N \vdash \gamma \Rightarrow^* map \ Tm \ w$  by blast
  from  $\gamma R$  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  $Nts\_syms \ \gamma \cap Lhss \ ?N = \{\}$  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  $Lang \ R \ A \subseteq Lang \ (R \cup ?N) \ A'$ 
proof
  fix  $w$  assume  $w \in Lang \ R \ A$ 
  then have der:  $R \vdash [Nt \ A] \Rightarrow^* map \ Tm \ w$  by (simp add: Lang_def)

```

```

from derives_Nt_map_TmD[OF der] obtain  $\gamma$ 
  where  $A\gamma R: (A, \gamma) \in R$  and  $d: R \vdash \gamma \Rightarrow^* \text{map } Tm \ w$  by blast
from  $A\gamma R$  have  $(A', \gamma) \in R \cup ?N$  by auto
then have  $R \cup ?N \vdash [Nt \ A'] \Rightarrow \gamma$  by (simp add: derive_singleton)
moreover from  $d$  have  $R \cup ?N \vdash \gamma \Rightarrow^* \text{map } Tm \ w$  by (meson Un_upper1
derives_mono)
ultimately have  $R \cup ?N \vdash [Nt \ A'] \Rightarrow^* \text{map } Tm \ w$  by (rule converse_rtranclp_into_rtranclp)
then show  $w \in \text{Lang } (R \cup ?N) \ A'$  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 $L0_Dd: x \in L0 \implies x \neq [] \implies Dd \in \text{set } x$

proof –

```

  assume  $x \in L0$   $x \neq []$ 
  then obtain  $bs$  where  $x = \text{concat } (\text{map } \text{blk } bs)$  and  $ne: bs \neq []$  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  $Dd \in \text{set } (\text{blk } b)$  by (simp add: blk_def)
  with  $x \ bbs$  show  $Dd \in \text{set } x$  by simp
qed

```

lemma $\text{inv_hom_Cc_empty}: \text{inv_hom } (\lambda_. [Cc]) (L0 - \{[]\}) = \{\}$

proof –

```

  have  $\text{concat } (\text{map } (\lambda_. [Cc]) \ w) \notin L0 - \{[]\}$  for  $w :: 'a \text{ list}$ 
  proof
    assume  $\text{concat } (\text{map } (\lambda_. [Cc]) \ w) \in L0 - \{[]\}$ 
    hence  $A: \text{concat } (\text{map } (\lambda_. [Cc]) \ w) \in L0$  and  $B: \text{concat } (\text{map } (\lambda_. [Cc]) \ w) \neq []$  by auto
    from  $L0\_Dd[OF \ A \ B]$  have  $Dd \in \text{set } (\text{concat } (\text{map } (\lambda_. [Cc]) \ w))$  .
    moreover have  $\text{set } (\text{concat } (\text{map } (\lambda_. [Cc]) \ w)) \subseteq \{Cc\}$  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  $\exists h::t \Rightarrow t0 \text{ list. } \text{Lang } P \ S - \{\emptyset\} = \text{inv\_hom } h \ (L0 - \{\emptyset\})$ 
proof (cases S  $\in$  Nts P)
  case False
  hence S  $\notin$  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 ( $\lambda\_.$  [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  $\in$  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  $\varepsilon$ ).
  have GNF_Pg:  $\exists a \ Bs. \alpha = Tm \ a \ \# \ \text{map } Nt \ Bs$  if (A,  $\alpha$ )  $\in$  Pg for A  $\alpha$ 
  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  $\notin$  Nts Pg unfolding S0_def using finPg by (rule fresh_notIn)
  define R where R = Pg  $\cup$  {(S0,  $\gamma$ ) |  $\gamma.$  (S,  $\gamma$ )  $\in$  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:  $\exists a \ Bs. \alpha = Tm \ a \ \# \ \text{map } Nt \ Bs$  if (A,  $\alpha$ )  $\in$  R for A  $\alpha$ 
  using that GNF_Pg by (auto simp: R_def)
  have S0  $\notin$  Rhs_Nts Pg using S0_fresh by (simp add: Nts_Lhss_Rhs_Nts)
  moreover have Rhs_Nts {(S0,  $\gamma$ ) |  $\gamma.$  (S,  $\gamma$ )  $\in$  Pg}  $\subseteq$  Rhs_Nts Pg by (auto
simp: Rhs_Nts_def)
  ultimately have notinRhsR: S0  $\notin$  Rhs_Nts R by (auto simp: R_def Rhs_Nts_def)
  have Snr: S0  $\notin$  set Bs if (A, Tm a  $\#$  map Nt Bs)  $\in$  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,  $\gamma$ ) |  $\gamma.$  (S,  $\gamma$ )  $\in$  Pg}  $\subseteq$  ( $\lambda \gamma.$  (S0,  $\gamma$ )) ‘ (snd ‘ Pg) by force
  moreover have finite (( $\lambda \gamma.$  (S0,  $\gamma$ )) ‘ (snd ‘ Pg)) by (simp add: Pg_def)
  ultimately have finite {(S0,  $\gamma$ ) |  $\gamma.$  (S,  $\gamma$ )  $\in$  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 : $(A, \alpha) \in set\ qs \implies \exists a\ Bs. \alpha = Tm\ a\ \# \ map\ Nt\ Bs$ **for** $A\ \alpha$
using $GNF_R[of\ A\ \alpha]$ **by** $(simp\ add: qsR)$
have Snr_qs : $(A, Tm\ a\ \# \ map\ Nt\ Bs) \in set\ qs \implies S0 \notin set\ Bs$ **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** $idx :: 'n \Rightarrow nat$
where $idxinj$: $inj_on\ idx\ (insert\ S0\ (Nts\ (set\ qs)))$ **by** $blast$
have key : $Lang\ (set\ qs)\ S0 - \{\epsilon\} = inv_hom\ (enc_h\ idx\ S0\ qs)\ (L0 - \{\epsilon\})$
by $(rule\ Greibach_2_1_std[OF\ std_qs\ Snr_qs\ idxinj])$
have $Lang\ (set\ qs)\ S0 = Lang\ P\ S - \{\epsilon\}$ **using** $qsR\ LangR$ **by** $simp$
hence $Lang\ (set\ qs)\ S0 - \{\epsilon\} = Lang\ P\ S - \{\epsilon\}$ **by** $auto$
with key **have** $Lang\ P\ S - \{\epsilon\} = inv_hom\ (enc_h\ idx\ S0\ qs)\ (L0 - \{\epsilon\})$ **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 $L_0 - \{\epsilon\}$. 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 $\exists h :: 't \Rightarrow t0\ list. Lang\ P\ S - \{\epsilon\} = inv_hom\ h\ (L0 - \{\epsilon\})$
proof —
have $finNts$: $finite\ (Nts\ P \cup \{S\})$ **using** $finP$ **by** $(simp\ add: finite_Nts)$
from $finite_imp_inj_to_nat_seg[OF\ finNts]$ **obtain** $f :: 'n \Rightarrow nat$
where f : $inj_on\ f\ (Nts\ P \cup \{S\})$ **by** $blast$
have $finite\ (rename_Prods\ f\ P)$ **using** $finP$ **by** $simp$
from $Greibach_2_1_fresh0[OF\ this, of\ f\ S]$ **obtain** $h :: 't \Rightarrow t0\ list$
where $Lang\ (rename_Prods\ f\ P)\ (f\ S) - \{\epsilon\} = inv_hom\ h\ (L0 - \{\epsilon\})$ **by**
 $blast$
moreover **have** $Lang\ (rename_Prods\ f\ P)\ (f\ S) = Lang\ P\ S$ **using** f **by** $(rule\ Lang_rename_Prods)$
ultimately **have** $Lang\ P\ S - \{\epsilon\} = inv_hom\ h\ (L0 - \{\epsilon\})$ **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 $\exists h. L - \{\epsilon\} = \text{inv_hom } h (L_0 - \{\epsilon\})$

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*) $oA1 \equiv Aa$ (*Open A1*)

abbreviation (*input*) $cA1 \equiv Aa$ (*Close A1*)

abbreviation (*input*) $oA2 \equiv Aa$ (*Open A2*)

abbreviation (*input*) $cA2 \equiv Aa$ (*Close A2*)

datatype $N = NZ \mid NF \mid NM$

definition $G :: (N, t0) \text{ Prods where}$

$$G = \{(NZ, []), \\ (NZ, [Nt NF, Tm Cc, Tm Ce, Nt NM, Tm Cc, Nt NF, Tm Dd]), \\ (NF, []), \\ (NM, []), \\ (NM, [Nt NM, Nt NM]), \\ (NM, [Tm oA1, Nt NM, Tm cA1]), \\ (NM, [Tm oA2, Nt NM, Tm cA2]), \\ (NM, [Tm Cc, Nt NF, Tm Dd, Nt NF, Tm Cc])\} \\ \cup \{(NF, [Tm t, Nt NF]) \mid t. t \in \text{Talph}\}$$

lemma *bracks_eq*: $\text{bracks} = \{oA1, cA1, oA2, cA2\}$

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  $w \subseteq Talph \implies w \in Lang\ G\ NF$ 
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:  $mbal\ w \implies mbal\ w' \implies mbal\ (w\ @\ w')$ 
| mbal_wrap:  $mbal\ w \implies mbal\ (Aa\ (Open\ a)\ \# w\ @\ [Aa\ (Close\ a)])$ 
| mbal_gap:  $set\ z \subseteq Talph \implies set\ x \subseteq Talph \implies mbal\ (Cc\ \# z\ @\ Dd\ \# x\ @\ [Cc])$ 

lemma mbal_imp_Lang_NM:  $mbal\ w \implies w \in Lang\ G\ NM$ 
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:  $(NM, [Tm\ oA2, Nt\ NM, Tm\ cA2]) \in G$  by (simp add: G_def)
  have  $Aa\ (Open\ a)\ \# w\ @\ [Aa\ (Close\ a)] \in inst\_syms\ (Lang\ G)\ [Tm\ oA2, Nt\ NM, Tm\ cA2]$ 
  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**

$$Gap = \{Cc \# z @ Dd \# x @ [Cc] \mid x z. \text{set } z \subseteq Talph \wedge \text{set } x \subseteq Talph\}$$

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*:

$$\begin{aligned}
& \text{length } ya = \text{length } gs1 \implies \\
& \text{interl } (ya @ [yl]) \text{ } gs1 @ \text{interl } (yr \# ys2) \text{ } gs2 \\
& = \text{interl } (ya @ (yl @ yr) \# ys2) \text{ } (gs1 @ gs2)
\end{aligned}$$

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**

$$\begin{aligned}
Ymid \ W \iff & (\exists \text{ys gs. } W = \text{interl } \text{ys } \text{gs} \wedge \text{length } \text{ys} = \text{Suc } (\text{length } \text{gs}) \wedge \\
& (\forall g \in \text{set } \text{gs. } g \in \text{Gap}) \wedge (\forall y \in \text{set } \text{ys. } \text{set } y \subseteq \text{bracks}) \wedge \\
& \text{bal } (\text{map } \text{brk } (\text{concat } \text{ys})))
\end{aligned}$$

lemma *Ymid_Nil*: *Ymid* []

```

unfolding Ymid_def
by (rule exI[of _ []], rule exI[of _ []]) auto

lemma Ymid_gap:  $g \in \text{Gap} \implies \text{Ymid } g$ 
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:  $\forall g \in \text{set } gs1. g \in \text{Gap}$  and y1:  $\forall y \in \text{set } ys1. \text{set } y \subseteq \text{bracks}$ 
    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:  $\forall g \in \text{set } gs2. g \in \text{Gap}$  and y2:  $\forall y \in \text{set } ys2. \text{set } y \subseteq \text{bracks}$ 
    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  $\forall g \in \text{set } ?gs. g \in \text{Gap}$  using g1 g2 by auto
  moreover have  $\forall y \in \text{set } ?ys. \text{set } y \subseteq \text{bracks}$ 
    using y1 y2 ys1 ys2 by auto
  ultimately have Wa @ Wb = interl ?ys ?gs  $\wedge$  length ?ys = Suc (length ?gs)  $\wedge$ 
    ( $\forall g \in \text{set } ?gs. g \in \text{Gap}$ )  $\wedge$  ( $\forall y \in \text{set } ?ys. \text{set } y \subseteq \text{bracks}$ )  $\wedge$  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  $\implies$  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:  $\forall g \in \text{set } gs. g \in \text{Gap}$  and yB:  $\forall y \in \text{set } ys. \text{set } y \subseteq \text{bracks}$ 
  and bl: bal (map brk (concat ys)) by (auto simp: Ymid_def)
  from len have ys  $\neq []$  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  $\forall y \in \text{set } ?ys. \text{set } y \subseteq \text{bracks}$ 
      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 $\forall y \in \text{set } ?ys. \text{set } y \subseteq \text{bracks}$
using $yB \text{ } ys \text{ } ys'$ **by** $(\text{auto simp: bracks_def})$
moreover have $\text{length } ?ys = \text{Suc } (\text{length } gs)$ **using** lenfm **by** simp
ultimately show $?thesis$ **using** $\text{eqW } gG$ **unfolding** $Ymid_def$
by $(\text{intro exI[of_ } ?ys] \text{ exI[of_ } gs]) \text{ simp}$
qed
qed

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

lemma $\text{Gap_imp_mbal}: g \in \text{Gap} \implies \text{mbal } g$
by $(\text{auto simp: Gap_def mbal_gap})$

inductive $\text{sprd} :: t0 \text{ list} \Rightarrow t0_A \text{ bracket list} \Rightarrow \text{bool}$ **where**
 $\text{sprd_Nil}: \text{sprd } [] []$
 $| \text{sprd_gap}: g \in \text{Gap} \implies \text{sprd } W \text{ } bw \implies \text{sprd } (g @ W) \text{ } bw$
 $| \text{sprd_brk}: \text{sprd } W \text{ } bw \implies \text{sprd } (Aa \text{ } b \# W) \text{ } (b \# bw)$

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

lemma sprd_brk_list :
 $\text{set } y \subseteq \text{bracks} \implies \text{sprd } Wr \text{ } bw \implies \text{sprd } (y @ Wr) \text{ } (\text{map brk } y @ bw)$
proof $(\text{induction } y)$
case Nil **thus** $?case$ **by** simp
next
case $(\text{Cons } c \text{ } y')$ **thus** $?case$ **using** sprd_brk **by** $(\text{auto simp: bracks_def})$
qed

lemma interl_sprd :
 $\text{length } ys = \text{Suc } (\text{length } gs) \implies (\forall g \in \text{set } gs. g \in \text{Gap}) \implies (\forall y \in \text{set } ys. \text{set } y \subseteq \text{bracks})$
 $\implies \text{sprd } (\text{interl } ys \text{ } gs) \text{ } (\text{map brk } (\text{concat } ys))$
proof $(\text{induction } ys \text{ arbitrary: } gs)$
case Nil **thus** $?case$ **by** simp
next
case $(\text{Cons } y \text{ } ys')$
show $?case$
proof $(\text{cases } gs)$
case Nil **then show** $?thesis$ **using** $\text{Cons.premis sprd_brk_list[OF_ } \text{sprd_Nil}]$
by auto
next
case $(\text{Cons } g \text{ } gs')$ **thus** $?thesis$ **using** $\text{sprd_brk_list sprd_gap Cons.IH Cons.premis}$
by auto
qed

qed

lemma *sprd_split*:

$sprd\ W\ bw \implies bw = bw1\ @\ bw2 \implies \exists\ W1\ W2. W = W1\ @\ W2 \wedge sprd\ W1\ bw1 \wedge sprd\ W2\ bw2$

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.premis]*

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*:

$sprd\ W\ bw \implies bw = b\ \#\ bw' \implies \exists\ U\ Wr. W = U\ @\ Aa\ b\ \#\ Wr \wedge mbal\ U \wedge sprd\ Wr\ bw'$

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*:

$sprd\ W\ bw \implies bw = bw'\ @\ [b] \implies \exists\ Wl\ U. W = Wl\ @\ Aa\ b\ \#\ U \wedge sprd\ Wl\ bw' \wedge mbal\ U$

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.premis]* **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  $\implies$  sprd W bw  $\implies$  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.prem1 refl] show ?case by (auto simp add: mbal_app)
next
case (Wrap xs a)
  from sprd_head[OF Wrap.prem1 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 Y_{mid} and the block-list form of L_0

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

lemma *mkbblocks*:

$length\ ys = Suc\ (length\ gs) \implies (\forall g \in set\ gs. g \in Gap) \implies set\ x \subseteq Talph \implies set\ z \subseteq Talph \implies$

$\exists bs. bs \neq [] \wedge (\forall (a,b,c) \in set\ bs. set\ a \subseteq Talph \wedge set\ c \subseteq Talph) \wedge$

$map\ (\lambda(a,b,c). b)\ bs = ys \wedge$

$concat\ (map\ blk\ bs) = x @ [Cc] @ interl\ ys\ gs @ [Cc] @ z @ [Dd]$

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.prem(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.prem(1) gs by simp
    have ∀g∈set gs'. g ∈ Gap using Cons.prem(2) gs by simp
    from Cons.IH[OF len' this xxT Cons.prem(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.prem(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.prem(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 $ac': \forall (a,b,c) \in \text{set } bs'. \text{ set } a \subseteq \text{Talph} \wedge \text{ set } c \subseteq \text{Talph}$ **using** $\text{Cons.prem}(2)$
by *auto*
from $\text{Cons.IH}[OF \text{ False } ac']$ **obtain** $x \text{ } ys \text{ } gs \text{ } z$ **where**
 $xT: \text{ set } x \subseteq \text{Talph}$ **and** $zT: \text{ set } z \subseteq \text{Talph}$ **and** $len: \text{ length } ys = \text{ Suc } (\text{ length } gs)$
and $gG: \forall g \in \text{set } gs. g \in \text{Gap}$ **and** $mid: \text{ map } (\lambda(a,b,c). b) \text{ } bs' = ys$
and $cc: \text{ concat } (\text{ map } \text{ blk } bs') = x @ [Cc] @ \text{ interl } ys \text{ } gs @ [Cc] @ z @ [Dd]$ **by**
blast
have $gGap: (Cc \# c @ Dd \# x @ [Cc]) \in \text{Gap}$ **using** $cT \text{ } xT$ **by** $(\text{rule } GapI)$
have $\text{ concat } (\text{ map } \text{ blk } ((a, b, c) \# bs'))$
 $= a @ [Cc] @ \text{ interl } (b \# ys) ((Cc \# c @ Dd \# x @ [Cc]) \# gs) @ [Cc] @$
 $z @ [Dd]$
using cc **by** $(\text{simp add: blk_def})$
moreover **have** $\text{ length } (b \# ys) = \text{ Suc } (\text{ length } ((Cc \# c @ Dd \# x @ [Cc]) \#$
 $gs))$ **using** len **by** *simp*
moreover **have** $\forall g \in \text{set } ((Cc \# c @ Dd \# x @ [Cc]) \# gs). g \in \text{Gap}$ **using**
 $gG \text{ } gGap$ **by** *simp*
moreover **have** $\text{ map } (\lambda(a,b,c). b) ((a, b, c) \# bs') = b \# ys$ **using** mid **by**
simp
ultimately show *?thesis* **using** $abc \text{ } aT \text{ } zT$
by $(\text{intro } exI[\text{ of } _ a] \text{ } exI[\text{ of } _ b \# ys] \text{ } exI[\text{ of } _ (Cc \# c @ Dd \# x @ [Cc])$
 $\# gs] \text{ } exI[\text{ of } _ z])$ *auto*
qed
qed

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

lemma $Ymid_imp_L0$:

assumes $Y: Ymid \text{ } W$ **and** $xT: \text{ set } x \subseteq \text{Talph}$ **and** $zT: \text{ set } z \subseteq \text{Talph}$
shows $x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd] \in L0$
proof –
from Y **obtain** $ys \text{ } gs$ **where** $W: W = \text{ interl } ys \text{ } gs$ **and** $len: \text{ length } ys = \text{ Suc } (\text{ length } gs)$
and $gG: \forall g \in \text{set } gs. g \in \text{Gap}$ **and** $yB: \forall y \in \text{set } ys. \text{ set } y \subseteq \text{bracks}$
and $bl: \text{ bal } (\text{ map } \text{ brk } (\text{ concat } ys))$ **by** $(\text{auto simp: } Ymid_def)$
from len **have** $ys \neq []$ **by** *auto*
then obtain $y0 \text{ } ys'$ **where** $ys: ys = y0 \# ys'$ **by** $(\text{cases } ys)$ *auto*
have $len2: \text{ length } ((Ce \# y0) \# ys') = \text{ Suc } (\text{ length } gs)$ **using** $len \text{ } ys$ **by** *simp*
from $\text{mkblocks}[OF \text{ len2 } gG \text{ } xT \text{ } zT]$ **obtain** bs **where** bsP :
 $bs \neq [] \wedge (\forall (a,b,c) \in \text{set } bs. \text{ set } a \subseteq \text{Talph} \wedge \text{ set } c \subseteq \text{Talph}) \wedge$
 $\text{ map } (\lambda(a,b,c). b) \text{ } bs = (Ce \# y0) \# ys' \wedge$
 $\text{ concat } (\text{ map } \text{ blk } bs) = x @ [Cc] @ \text{ interl } ((Ce \# y0) \# ys') \text{ } gs @ [Cc] @ z @$
 $[Dd] \dots$
have $ceW: \text{ interl } ((Ce \# y0) \# ys') \text{ } gs = Ce \# W$
using $\text{ interl_prepend_first}[\text{ of } [Ce] \text{ } y0 \text{ } ys' \text{ } gs] \text{ } W \text{ } ys$ **by** *simp*
have $word: \text{ concat } (\text{ map } \text{ blk } bs) = x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd]$
using $bsP \text{ } ceW$ **by** *simp*
have $midtl: \text{ map } (\lambda(a,b,c). b) (tl \text{ } bs) = ys'$ **by** $(\text{simp add: map_tl } bsP)$
have $tlmid: \text{ set } b \subseteq \text{bracks}$ **if** $tabc: (a,b,c) \in \text{set } (tl \text{ } bs)$ **for** $a \text{ } b \text{ } c$
using $tabc \text{ } midtl \text{ } yB \text{ } ys$ **by** *force*

```

have midD:  $\exists v \in D. \text{concat} (\text{map} (\lambda(a,b,c). b) bs) = Ce \# v$ 
proof -
  have concat ys  $\in D$  by (simp add: D_def UN_least bl yB)
  then show ?thesis using bsP ys by simp
qed
have concat (map blk bs)  $\in L0$  unfolding L0_def using bsP tlmid midD by
blast
thus ?thesis using word by simp
qed

lemma L0_imp_Ymid:
  assumes  $w \in L0$  and  $w \neq []$ 
  shows  $\exists x z W. w = x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd] \wedge \text{set } x \subseteq \text{Talph} \wedge$ 
 $\text{set } z \subseteq \text{Talph} \wedge Ymid W$ 
proof -
  from assms obtain bs where wbs:  $w = \text{concat} (\text{map blk bs})$  and bsne:  $bs \neq []$ 
  and acT:  $\forall (x,y,z) \in \text{set } bs. \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph}$ 
  and tlB:  $\forall (x,y,z) \in \text{set} (tl bs). \text{set } y \subseteq \text{bracks}$ 
  and cD:  $\exists v \in D. \text{concat} (\text{map} (\lambda(x,y,z). y) bs) = Ce \# v$ 
  by (auto simp: L0_def)
  from blocks_decomp[OF bsne acT] obtain x ys gs z where
    xT:  $\text{set } x \subseteq \text{Talph}$  and zT:  $\text{set } z \subseteq \text{Talph}$  and len:  $\text{length } ys = \text{Suc} (\text{length } gs)$ 
    and gG:  $\forall g \in \text{set } gs. g \in \text{Gap}$  and bmid:  $\text{map} (\lambda(a,b,c). b) bs = ys$ 
    and wcc:  $\text{concat} (\text{map blk bs}) = x @ [Cc] @ \text{interl } ys gs @ [Cc] @ z @ [Dd]$  by
  blast
  from cD obtain v where vD:  $v \in D$  and cyv:  $\text{concat} (\text{map} (\lambda(x,y,z). y) bs) =$ 
 $Ce \# v$  by blast
  have ysv:  $\text{concat } ys = Ce \# v$  using cyv bmid by simp
  from len obtain y0 ys' where ys:  $ys = y0 \# ys'$  by (cases ys) auto
  have ys'brk:  $\forall y \in \text{set } ys'. \text{set } y \subseteq \text{bracks}$ 
    using bmid ys tlB by (fastforce simp add: map_tl)
  have y0ne:  $y0 \neq []$ 
  proof
    assume y0nil:  $y0 = []$ 
    have  $Ce \in \text{set} (\text{concat } ys)$  using ysv by simp
    moreover have  $\text{set} (\text{concat } ys) = \text{set} (\text{concat } ys')$  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:  $b0 @ \text{concat } ys' = v$  using ysv ys y0split by auto
  hence y0:  $y0 = Ce \# b0$  using y0split by simp
  let ?W =  $\text{interl} (b0 \# ys') gs$ 
  have ceW:  $\text{interl } ys gs = Ce \# ?W$ 
    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:  $\text{set } b0 \subseteq \text{bracks}$  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 $Rsol :: N \Rightarrow t0$ list set **where**

$Rsol\ A = (case\ A\ of\ NZ \Rightarrow L0 \mid NF \Rightarrow \{w.\ set\ w \subseteq Talph\} \mid NM \Rightarrow \{W.\ Ymid\ W\})$

Each production keeps the candidate solution closed (the $\subseteq$ -obligations).

lemma $incl_NZ_big: inst_syms\ Rsol\ [Nt\ NF, Tm\ Cc, Tm\ Ce, Nt\ NM, Tm\ Cc, Nt\ NF, Tm\ Dd] \subseteq Rsol\ NZ$

proof

fix w **assume** $w \in inst_syms\ Rsol\ [Nt\ NF, Tm\ Cc, Tm\ Ce, Nt\ NM, Tm\ Cc, Nt\ NF, Tm\ Dd]$

then obtain $x\ W\ z$ **where** $w = x @ [Cc, Ce] @ W @ [Cc] @ z @ [Dd]$

and $xT: set\ x \subseteq Talph$ **and** $YW: Ymid\ W$ **and** $zT: set\ z \subseteq Talph$

by (auto simp: $Rsol_def\ conc_def$)

show $w \in Rsol\ NZ$

using $Ymid_imp_L0[OF\ YW\ xT\ zT]$ **by** (simp add: $Rsol_def$)

qed

lemma $Lang_G_subset: Lang\ G\ A \subseteq Rsol\ A$

proof (rule $Lang_subset_if$)

fix $A\ \alpha$ **assume** $aG: (A, \alpha) \in G$

consider $(A, \alpha) = (NZ, [])$

| $(A, \alpha) = (NZ, [Nt\ NF, Tm\ Cc, Tm\ Ce, Nt\ NM, Tm\ Cc, Nt\ NF, Tm\ Dd])$

| $(A, \alpha) = (NF, [])$ | $(A, \alpha) = (NM, [])$ | $(A, \alpha) = (NM, [Nt\ NM, Nt\ NM])$

| $(A, \alpha) = (NM, [Tm\ oA1, Nt\ NM, Tm\ cA1])$ | $(A, \alpha) = (NM, [Tm\ oA2, Nt\ NM, Tm\ cA2])$

| $(A, \alpha) = (NM, [Tm\ Cc, Nt\ NF, Tm\ Dd, Nt\ NF, Tm\ Cc])$

| t **where** $t \in Talph$ $(A, \alpha) = (NF, [Tm\ t, Nt\ NF])$

using aG **unfolding** G_def **by** (elim $UnE\ insertE\ CollectE\ exE\ conjE$)

$simp_all$

then show $inst_syms\ Rsol\ \alpha \subseteq Rsol\ A$

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*: $Ymid\ W \implies W \in Lang\ G\ NZ$
using *Ymid_def interl_sprd sprd_bal_imp_mbal mbal_imp_Lang_NM* **by** *auto*

lemma *L0_subset_Lang*: $L0 \subseteq Lang\ G\ NZ$

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.