

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

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

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

$\xi(p)$ for a standard-form production $p = (Y_i \rightarrow a\ Y_{j1} \dots Y_{jm})$: 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 $xi :: ('n \Rightarrow nat) \Rightarrow ('n, 't)\ prod \Rightarrow t0\ list$ **where**
 $xi\ idx\ p =$
 $popcode\ (idx\ (fst\ p)) @$
 $concat\ (map\ (\lambda s. case\ s\ of\ Nt\ B \Rightarrow pushcode\ (idx\ B) \mid Tm\ _ \Rightarrow [])\ (rev\ (tl\ (snd\ p))))$

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

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

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

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 $brks :: t0\ list \Rightarrow t0_A\ bracket\ list$ **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 $bal_stk_pushcode$: $bal_stk\ s\ (brks\ (pushcode\ i)) = (frag\ i @ s, [])$
 $\langle proof \rangle$

lemma $bal_stk_popcode$: $bal_stk\ (frag\ i @ s)\ (brks\ (popcode\ i)) = (s, [])$
 $\langle proof \rangle$

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 :
 $bal_stk\ s\ (brks\ (concat\ (map\ pushcode\ is))) = (stkenc\ (rev\ is) @ s, [])$
 $\langle proof \rangle$

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

lemma xi_eq :
 $snd\ p = Tm\ a \# map\ Nt\ Bs \implies$
 $xi\ idx\ p = popcode\ (idx\ (fst\ p)) @ concat\ (map\ pushcode\ (rev\ (map\ idx\ Bs)))$
 $\langle proof \rangle$

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 $bal_stk\ (frag\ (idx\ (fst\ p)) @ s)\ (brks\ (xi\ idx\ p)) = (stkenc\ (map\ idx\ Bs) @ s, [])$
 $\langle proof \rangle$

1.3.2 Letters occurring in the encoded words

lemma $set_pushcode$: $set\ (pushcode\ i) \subseteq bracks\ \langle proof \rangle$

lemma $set_popcode$: $set\ (popcode\ i) \subseteq bracks\ \langle proof \rangle$

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

lemma set_xi : $set\ (xi\ idx\ p) \subseteq bracks$

$\langle \text{proof} \rangle$

lemma *set_xihat*: $\text{set } (xihat \text{ idx } S \ p) \subseteq \text{insert } Ce \text{ bracks}$
 $\langle \text{proof} \rangle$

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

lemma *map_xihat_no_S*:
 $\forall p \in \text{set } ps. \text{fst } p \neq S \implies \text{map } (xihat \text{ idx } S) \ ps = \text{map } (xi \text{ idx}) \ ps$
 $\langle \text{proof} \rangle$

1.3.3 Block decomposition of the encoded word

lemma *set_xihat_Talph*: $\text{set } (xihat \text{ idx } S \ q) \subseteq \text{Talph}$
 $\langle \text{proof} \rangle$

lemma *set_Cc_xihat*: $\text{set } (Cc \ \# \ xihat \text{ idx } S \ q) \subseteq \text{Talph}$
 $\langle \text{proof} \rangle$

lemma *set_concat_Cc_xihat*: $\text{set } (\text{concat } (\text{map } (\lambda q. Cc \ \# \ xihat \text{ idx } S \ q) \ qs)) \subseteq \text{Talph}$
 $\langle \text{proof} \rangle$

lemma *set_concat_xihat_Cc*: $\text{set } (\text{concat } (\text{map } (\lambda q. xihat \text{ idx } S \ q \ @ \ [Cc]) \ qs)) \subseteq \text{Talph}$
 $\langle \text{proof} \rangle$

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*:
 $\text{concat } (\text{map } (\lambda p. Cc \ \# \ xihat \text{ idx } S \ p) \ qs) \ @ \ [Cc, \ Dd]$
 $= Cc \ \# \ \text{concat } (\text{map } (\lambda q. xihat \text{ idx } S \ q \ @ \ [Cc]) \ qs) \ @ \ [Dd]$
 $\langle \text{proof} \rangle$

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*: $\text{snd } p = Tm \ a \ \# \ \text{map } Nt \ Bs$
shows $\exists x \ z. \text{enc_h idx } S \ P \ a = \text{blk } (x, xihat \text{ idx } S \ p, z) \wedge \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph}$
 $\langle \text{proof} \rangle$

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*:
 $\text{list_all2 } (\lambda b \ p. p \in \text{set } P \wedge (\exists Bs. \text{snd } p = Tm \ b \ \# \ \text{map } Nt \ Bs)) \ w \ ps \implies$

$\exists bs. \text{concat} (\text{map} (\text{enc_h } idx \ S \ P) \ w) = \text{concat} (\text{map } blk \ bs)$
 $\wedge \text{map} (\lambda(x,y,z). y) \ bs = \text{map} (xihat \ idx \ S) \ ps$
 $\wedge (\forall (x,y,z) \in \text{set } bs. \text{set } x \subseteq \text{Talph} \wedge \text{set } z \subseteq \text{Talph})$
 $\langle \text{proof} \rangle$

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*:
 $\text{frag } i \ @ \ xs = \text{frag } j \ @ \ ys \implies i = j \wedge xs = ys$
 $\langle \text{proof} \rangle$

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

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** $t = \text{frag } i \ @ \ s$
 $\langle \text{proof} \rangle$

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*: $\text{inj_on } idx \ N$ **and** *snd_p*: $\text{snd } p = Tm \ a \ \# \ \text{map } Nt \ Bs$
and *fpN*: $\text{fst } p \in N$ **and** αN : $\text{set } \alpha \subseteq N$
and *run*: $\text{bal_stk} (\text{stkenc} (\text{map } idx \ \alpha)) (\text{brks} (xi \ idx \ p)) = (s, [])$
shows $\exists \alpha'. \alpha = \text{fst } p \ \# \ \alpha' \wedge s = \text{stkenc} (\text{map } idx \ (Bs \ @ \ \alpha'))$
 $\langle \text{proof} \rangle$

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*:
 $\text{concat } xss = \text{concat } yss \implies$
 $(\forall xs \in \text{set } xss. \exists p. xs = p \ @ \ [s] \wedge s \notin \text{set } p) \implies$
 $(\forall ys \in \text{set } yss. \exists p. ys = p \ @ \ [s] \wedge s \notin \text{set } p) \implies xss = yss$
 $\langle \text{proof} \rangle$

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 $x @ c \# y @ c \# z = \text{concat } (\text{map } (\lambda q. c \# f \ q) \ qs) @ [c] \implies y \in \text{set } (\text{map } f \ qs)$
 $\langle \text{proof} \rangle$

Each block $\text{enc_h } idx \ S \ P \ a$ ends in exactly one d .

lemma $\text{enc_h_Dd}: \exists p. \text{enc_h } idx \ S \ P \ a = p @ [Dd] \wedge Dd \notin \text{set } p$
 $\langle \text{proof} \rangle$

Inverting one block: if $\text{blk } (x,y,z)$ equals an encoded block $\text{enc_h } idx \ S \ P \ a$ 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 $\text{eq}: \text{blk } (x,y,z) = \text{enc_h } idx \ S \ P \ a$
and $yset: \text{set } y \subseteq \text{insert } Ce \ \text{bracks}$
shows $\exists p. p \in \text{set } P \wedge (\exists Bs. \text{snd } p = Tm \ a \ \# \ \text{map } Nt \ Bs) \wedge y = \text{xihat } idx \ S \ p$
 $\langle \text{proof} \rangle$

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 $\text{concat_xihat_eq_fst}$:
assumes $ne: ps \neq []$ **and** $fp0: \text{fst } (hd \ ps) = S$ **and** $tlS: \forall p \in \text{set } (tl \ ps). \text{fst } p \neq S$
shows $\text{concat } (\text{map } (\text{xihat } idx \ S) \ ps) = Ce \ \# \ \text{pushcode } (idx \ S) @ \text{concat } (\text{map } (\text{xi } idx) \ ps)$
 $\langle \text{proof} \rangle$

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 :
 $\text{list_all2 } (\lambda b \ a. \text{blk } b = \text{enc_h } idx \ S \ P \ a) \ bs \ w \implies$
 $(\forall (x,y,z) \in \text{set } bs. \text{set } y \subseteq \text{insert } Ce \ \text{bracks}) \implies$
 $\exists ps. \text{list_all2 } (\lambda a \ p. p \in \text{set } P \wedge (\exists Bs. \text{snd } p = Tm \ a \ \# \ \text{map } Nt \ Bs)) \ w \ ps$
 $\wedge \text{map } (\lambda (x,y,z). y) \ bs = \text{map } (\text{xihat } idx \ S) \ ps$
 $\langle \text{proof} \rangle$

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 $\text{insert } S \ (Nts \ (\text{set } P))$ matters; the specific index values, in particular $idx \ S$, are immaterial).

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

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

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*:
 $\text{drives } ps \ \alpha \ w \ \beta \implies$
 $\text{bal_stk } (\text{stkenc } (\text{map } idx \ \alpha)) \ (\text{brks } (\text{concat } (\text{map } (xi \ idx) \ ps))) = (\text{stkenc } (\text{map } idx \ \beta), [])$
<proof>

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 $\text{drives } ps \ [S] \ w \ []$
shows $\text{bal } (\text{brks } (\text{pushcode } (idx \ S) \ @ \ \text{concat } (\text{map } (xi \ idx) \ ps)))$
<proof>

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

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

lemma *drives_imp_derives*:
 $\text{drives } ps \ \alpha \ w \ \beta \implies \text{set } P \vdash \text{map } Nt \ \alpha \Rightarrow_l^* \text{map } Tm \ w \ @ \ \text{map } Nt \ \beta$
<proof>

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*:

$set\ P \vdash map\ Nt\ \alpha \Rightarrow l(n)\ map\ Tm\ w \Longrightarrow \exists ps. drives\ ps\ \alpha\ w\ []$
 $\langle proof \rangle$

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

lemma *drives_iff_Lang*:

$(\exists ps. drives\ ps\ [S]\ w\ []) \longleftrightarrow w \in Lang\ (set\ P)\ S$
 $\langle proof \rangle$

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*:

$drives\ ps\ \alpha\ w\ \beta \Longrightarrow S \notin set\ \alpha \Longrightarrow (\forall p \in set\ ps. fst\ p \neq S)$
 $\langle proof \rangle$

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 $drives\ ps\ [S]\ w\ []$
shows $concat\ (map\ (xihat\ idx\ S)\ ps) = Ce\ \# \ pushcode\ (idx\ S)\ @\ concat\ (map\ (xi\ idx)\ ps)$
 $\langle proof \rangle$

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*:

$drives\ ps\ \alpha\ w\ \beta \Longrightarrow list_all2\ (\lambda b\ p. p \in set\ P \wedge (\exists Bs. snd\ p = Tm\ b\ \# \ map\ Nt\ Bs))\ w\ ps$
 $\langle proof \rangle$

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 $drv: drives\ ps\ [S]\ w\ []$
shows $concat\ (map\ (enc_h\ idx\ S\ P)\ w) \in L0$
 $\langle proof \rangle$

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>

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 $\mathbb{C}/D$ -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>

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>

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 $\bigwedge A \alpha. (A, \alpha) \in \text{set } P \implies \exists a Bs. \alpha = Tm\ a \ \# \ \text{map } Nt\ Bs$
and $\bigwedge A a Bs. (A, Tm\ a \ \# \ \text{map } Nt\ Bs) \in \text{set } P \implies S \notin \text{set } Bs$
and $\text{inj_on } idx\ (\text{insert } S\ (Nts\ (\text{set } P)))$
shows $\text{Lang } (\text{set } P)\ S - \{\square\} = \text{inv_hom } (\text{enc_h } idx\ S\ P)\ (L0 - \{\square\})$
 $\langle \text{proof} \rangle$

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 $\text{Lang } (R \cup \{(A', \gamma) \mid \gamma. (A, \gamma) \in R\})\ A' = \text{Lang } R\ A$
 $\langle \text{proof} \rangle$

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*: $x \in L0 \implies x \neq \square \implies Dd \in \text{set } x$
 $\langle \text{proof} \rangle$

lemma *inv_hom_Cc_empty*: $\text{inv_hom } (\lambda_. [Cc])\ (L0 - \{\square\}) = \{\}$
 $\langle \text{proof} \rangle$

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 $P :: ('n::\text{fresh0}, 't)\ \text{Prods}$
assumes $\text{finP}: \text{finite } P$
shows $\exists h::'t \Rightarrow t0\ \text{list}. \text{Lang } P\ S - \{\square\} = \text{inv_hom } h\ (L0 - \{\square\})$
 $\langle \text{proof} \rangle$

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 $\exists h :: 't \Rightarrow t0 \text{ list. } Lang\ P\ S - \{\square\} = inv_hom\ h\ (L0 - \{\square\})$

<proof>

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 $\exists h. L - \{\square\} = inv_hom\ h\ (L0 - \{\square\})$

<proof>

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 $G :: (N, t0) Prods$ **where**

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

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

lemma *finite_Talph*: *finite Talph*
 <proof>

lemma *finite_G*: *finite G*
 <proof>

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

lemma *Talph_sub_Lang_NF*:
 $\text{set } w \subseteq \text{Talph} \implies w \in \text{Lang } G \text{ NF}$
 <proof>

1.5.3 The “middle” language generated by NM

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

lemma *mbal_imp_Lang_NM*: $\text{mbal } w \implies w \in \text{Lang } G \text{ NM}$
 <proof>

1.5.4 Gaps and the interleaving of bracket blocks with gaps

definition *Gap* :: *t0 list set* **where**
 $\text{Gap} = \{Cc \# z @ Dd \# x @ [Cc] \mid x \text{ z. set } z \subseteq \text{Talph} \wedge \text{set } x \subseteq \text{Talph}\}$

lemma *GapI*: $\text{set } z \subseteq \text{Talph} \implies \text{set } x \subseteq \text{Talph} \implies Cc \# z @ Dd \# x @ [Cc] \in \text{Gap}$
 <proof>

fun *interl* :: '*a list list* $\Rightarrow$ '*a list list* $\Rightarrow$ '*a list* **where**
 $\text{interl } [] _ = []$
 $| \text{interl } (y \# ys) [] = y @ \text{interl } ys []$
 $| \text{interl } (y \# ys) (g \# gs) = y @ g @ \text{interl } ys gs$

lemma *interl_merge*:
 $\text{length } ya = \text{length } gs1 \implies$
 $\text{interl } (ya @ [yl]) gs1 @ \text{interl } (yr \# ys2) gs2$
 $= \text{interl } (ya @ (yl @ yr) \# ys2) (gs1 @ gs2)$
 <proof>

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

definition $Ymid :: t0\ list \Rightarrow bool$ **where**

$$Ymid\ W \longleftrightarrow (\exists\ ys\ gs.\ W = interl\ ys\ gs \wedge length\ ys = Suc\ (length\ gs) \wedge$$

$$(\forall\ g \in set\ gs.\ g \in Gap) \wedge (\forall\ y \in set\ ys.\ set\ y \subseteq bracks) \wedge$$

$$bal\ (map\ brk\ (concat\ ys)))$$

lemma $Ymid_Nil$: $Ymid\ []$
 $\langle proof \rangle$

lemma $Ymid_gap$: $g \in Gap \Longrightarrow Ymid\ g$
 $\langle proof \rangle$

lemma $Ymid_app$:
assumes $Ymid\ Wa$ **and** $Ymid\ Wb$ **shows** $Ymid\ (Wa\ @\ Wb)$
 $\langle proof \rangle$

lemma $interl_prepend_first$: $interl\ ((p\ @\ y)\ \# \ ys)\ gs = p\ @\ interl\ (y\ \# \ ys)\ gs$
 $\langle proof \rangle$

lemma $interl_append_last$:
 $length\ ys = length\ gs \Longrightarrow interl\ (ys\ @\ [yl\ @\ s])\ gs = interl\ (ys\ @\ [yl])\ gs\ @\ s$
 $\langle proof \rangle$

lemma $Ymid_wrap$:
assumes $Ymid\ W$ **shows** $Ymid\ (Aa\ (Open\ a)\ \# \ W\ @\ [Aa\ (Close\ a)])$
 $\langle proof \rangle$

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

lemma Gap_imp_mbal : $g \in Gap \Longrightarrow mbal\ g$
 $\langle proof \rangle$

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

lemma $sprd_emptybw$: $sprd\ W\ [] \Longrightarrow mbal\ W$
 $\langle proof \rangle$

lemma $sprd_brk_list$:
 $set\ y \subseteq bracks \Longrightarrow sprd\ Wr\ bw \Longrightarrow sprd\ (y\ @\ Wr)\ (map\ brk\ y\ @\ bw)$
 $\langle proof \rangle$

lemma $interl_sprd$:
 $length\ ys = Suc\ (length\ gs) \Longrightarrow (\forall\ g \in set\ gs.\ g \in Gap) \Longrightarrow (\forall\ y \in set\ ys.\ set\ y \subseteq bracks)$
 $\Longrightarrow sprd\ (interl\ ys\ gs)\ (map\ brk\ (concat\ ys))$
 $\langle proof \rangle$

lemma *sprd_split*:

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

lemma *sprd_head*:

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

lemma *sprd_tail*:

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

lemma *sprd_bal_imp_mbal*: $bal\ bw \implies sprd\ W\ bw \implies mbal\ W$

$\langle proof \rangle$

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 *mkbblocks*:

$length\ ys = Suc\ (length\ gs) \implies (\forall g \in set\ gs. g \in Gap) \implies set\ x \subseteq Talph \implies set\ z \subseteq Talph \implies$
 $\exists\ bs. bs \neq [] \wedge (\forall (a,b,c) \in set\ bs. set\ a \subseteq Talph \wedge set\ c \subseteq Talph) \wedge$
 $map\ (\lambda(a,b,c). b)\ bs = ys \wedge$
 $concat\ (map\ blk\ bs) = x\ @\ [Cc]\ @\ interl\ ys\ gs\ @\ [Cc]\ @\ z\ @\ [Dd]$
 $\langle proof \rangle$

The inverse: every L_0 block list arises this way.

lemma *blocks_decomp*:

$bs \neq [] \implies (\forall (a,b,c) \in set\ bs. set\ a \subseteq Talph \wedge set\ c \subseteq Talph) \implies$
 $\exists\ x\ ys\ gs\ z. set\ x \subseteq Talph \wedge set\ z \subseteq Talph \wedge length\ ys = Suc\ (length\ gs) \wedge$
 $(\forall g \in set\ gs. g \in Gap) \wedge$
 $map\ (\lambda(a,b,c). b)\ bs = ys \wedge$
 $concat\ (map\ blk\ bs) = x\ @\ [Cc]\ @\ interl\ ys\ gs\ @\ [Cc]\ @\ z\ @\ [Dd]$
 $\langle proof \rangle$

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

lemma *Ymid_imp_L0*:

assumes Y : $Ymid\ W$ **and** xT : $set\ x \subseteq Talph$ **and** zT : $set\ z \subseteq Talph$
shows $x\ @\ [Cc,\ Ce]\ @\ W\ @\ [Cc]\ @\ z\ @\ [Dd] \in L0$
 $\langle proof \rangle$

lemma *L0_imp_Ymid*:

assumes $w \in L0$ **and** $w \neq []$
shows $\exists\ x\ z\ W. w = x\ @\ [Cc,\ Ce]\ @\ W\ @\ [Cc]\ @\ z\ @\ [Dd] \wedge set\ x \subseteq Talph \wedge set\ z \subseteq Talph \wedge Ymid\ W$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

The $\supseteq$ direction for the start symbol.

lemma $Ymid_imp_Lang_NM: Ymid\ W \implies W \in Lang\ G\ NM$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $Lang_G_NZ: Lang\ G\ NZ = L0$

$\langle proof \rangle$

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

theorem $CFL_L0: CFL\ TYPE(N)\ L0$

$\langle proof \rangle$

end

References

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