Greibach's Hardest Context-Free Language

Tobias Nipkow

September 17, 2026

Abstract

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

1 Greibach's Hardest Context-Free Language

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

Formalization of Theorem 2.1 by Sheila Greibach [1]:

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

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

abbreviation *Cons_power* :: *'a* ⇒ *nat* ⇒ *'a list* (**infixl** $\#^\frown$ *70*) **where**
$a \#^\frown n \equiv$ *replicate n a*

lemma *bal_stk_append_split*:
assumes *bal_stk s* (*xs* @ *ys*) = (*t*, [])
obtains *s'* **where** *bal_stk s xs* = (*s'*, []) **and** *bal_stk s' ys* = (*t*, [])
⟨*proof*⟩
lemma *bal_stk_replicate_Open*: *bal_stk s* (*Open a* $\#^\frown i$) = ($a\#^\frown i$ @ *s*, [])
⟨*proof*⟩
lemma *bal_stk_replicate_Close*: *bal_stk* ($a\#^\frown i$ @ *t*) (*Close a* $\#^\frown i$) = (*t*, [])
⟨*proof*⟩
lemma *bal_stk_replicate_Close_inv*:

bal_stk t (replicate i (Close a) @ rest) = (s, []) ⟹
∃ t'. t = replicate i a @ t' ∧ bal_stk t' rest = (s, [])
⟨*proof*⟩
lemmas *derives_Nt_map_TmD = derives_start1*
lemma *Lang_lfp_unfold*:
Lang_lfp P A = (⋃α ∈ Rhss P A. inst_syms (Lang_lfp P) α)
⟨*proof*⟩
corollary *Lang_unfold*: *Lang P A = (⋃α ∈ Rhss P A. inst_syms (Lang P) α)*
⟨*proof*⟩
lemma *concats_simps*[*simp*]:
concats [] = {[]}
concats (L#Ls) = L @@ concats Ls
⟨*proof*⟩
lemma *concats_append*[*simp*]: *concats (Ls1 @ Ls2) = concats Ls1 @@ concats Ls2*
⟨*proof*⟩
lemma *inst_sym_simps*[*simp*]:
inst_sym L (Tm a) = {[a]}
inst_sym L (Nt A) = L A
⟨*proof*⟩
lemma *inst_syms_Nil*[*simp*]: *inst_syms L [] = {[]}*
⟨*proof*⟩
lemma *inst_syms_Cons*[*simp*]: *inst_syms L (s # β) = inst_sym L s @@ inst_syms L β*
⟨*proof*⟩
lemma *inst_syms_append*[*simp*]: *inst_syms L (α @ β) = inst_syms L α @@ inst_syms L β*
⟨*proof*⟩
lemma *Lang_I*: *(A, α) ∈ P ⟹ w ∈ inst_syms (Lang P) α ⟹ w ∈ Lang P A*
⟨*proof*⟩
lemma *Lang_subset_if*:
assumes *⋀A α. (A,α) ∈ P ⟹ inst_syms R α ⊆ R A*
shows *Lang P A ⊆ R A*
⟨*proof*⟩

1.1 The hardest language L_0

1.1.1 The terminal alphabet of L_0

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

datatype *t0_A = A1 | A2*

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

1.1.2 The Dyck set D on two letters

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

fun *brk* :: *t0* ⇒ *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* ⊆ *bracks* ∧ *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* × *t0 list* × *t0 list* ⇒ *t0 list* **where**
  blk = (λ(*x*,*y*,*z*). *x* @ *Cc* # *y* @ *Cc* # *z* @ [*Dd*])

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

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

1.2 Homomorphisms

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

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

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

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

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

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

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

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

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

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

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

1.3 Basic machinery

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

The bracket image of a word over *t0*:

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

A single nonterminal Y_i occupies the stack fragment *frag i*; a whole nonterminal stack (top first) is encoded by *stkenc*. With the cons-stack of *bal_stk*

(top = head), reading *pushcode i* pushes *frag i* and reading *popcode i* pops it.

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

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

lemma *bal_stk_pushcode*: *bal_stk s* (*brks* (*pushcode i*)) = (*frag i* @ *s*, [])
⟨*proof*⟩

lemma *bal_stk_popcode*: *bal_stk* (*frag i* @ *s*) (*brks* (*popcode i*)) = (*s*, [])
⟨*proof*⟩

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*⟩

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

lemma *xi_eq*:
  snd p = *Tm a* # *map Nt Bs* ⟹
   xi idx p = *popcode* (*idx* (*fst p*)) @ *concat* (*map pushcode* (*rev* (*map idx Bs*)))
  ⟨*proof*⟩

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

lemma *bal_stk_xi*:
  assumes *snd p* = *Tm a* # *map Nt Bs*
  shows *bal_stk* (*frag* (*idx* (*fst p*)) @ *s*) (*brks* (*xi idx p*)) = (*stkenc* (*map idx Bs*)
@ *s*, [])
⟨*proof*⟩

1.3.2 Letters occurring in the encoded words

lemma *set_pushcode*: *set* (*pushcode i*) ⊆ *bracks* ⟨*proof*⟩
lemma *set_popcode*: *set* (*popcode i*) ⊆ *bracks* ⟨*proof*⟩

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

lemma *set_xi*: *set* (*xi idx p*) ⊆ *bracks*
  ⟨*proof*⟩

lemma *set_xihat*: *set* (*xihat idx S p*) ⊆ *insert Ce bracks*
⟨*proof*⟩

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

lemma *map_xihat_no_S*:
$\forall\, p \in set\ ps.\ fst\ p \neq S \Longrightarrow map\ (xihat\ idx\ S)\ ps = map\ (xi\ idx)\ ps$
⟨*proof*⟩

1.3.3 Block decomposition of the encoded word

lemma *set_xihat_Talph*: $set\ (xihat\ idx\ S\ q) \subseteq Talph$
⟨*proof*⟩

lemma *set_Cc_xihat*: $set\ (Cc\ \#\ xihat\ idx\ S\ q) \subseteq Talph$
⟨*proof*⟩

lemma *set_concat_Cc_xihat*: $set\ (concat\ (map\ (\lambda q.\ Cc\ \#\ xihat\ idx\ S\ q)\ qs)) \subseteq Talph$
⟨*proof*⟩

lemma *set_concat_xihat_Cc*: $set\ (concat\ (map\ (\lambda q.\ xihat\ idx\ S\ q\ @\ [Cc])\ qs)) \subseteq Talph$
⟨*proof*⟩

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

lemma *cc_shift2*:
$concat\ (map\ (\lambda p.\ Cc\ \#\ xihat\ idx\ S\ p)\ qs)\ @\ [Cc,\ Dd]$
$= Cc\ \#\ concat\ (map\ (\lambda q.\ xihat\ idx\ S\ q\ @\ [Cc])\ qs)\ @\ [Dd]$
⟨*proof*⟩

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*: $p \in set\ P$ **and** *snd_p*: $snd\ p = Tm\ a\ \#\ map\ Nt\ Bs$
shows $\exists\, x\ z.\ enc\_h\ idx\ S\ P\ a = blk\ (x,\ xihat\ idx\ S\ p,\ z) \wedge set\ x \subseteq Talph \wedge set\ z \subseteq Talph$
⟨*proof*⟩

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

lemma *block_decomp*:
$list\_all2\ (\lambda b\ p.\ p \in set\ P \wedge (\exists\, Bs.\ snd\ p = Tm\ b\ \#\ map\ Nt\ Bs))\ w\ ps \Longrightarrow$
$\exists\, bs.\ concat\ (map\ (enc\_h\ idx\ S\ P)\ w) = concat\ (map\ blk\ bs)$
$\wedge\ map\ (\lambda(x,y,z).\ y)\ bs = map\ (xihat\ idx\ S)\ ps$
$\wedge\ (\forall\,(x,y,z) \in set\ bs.\ set\ x \subseteq Talph \wedge set\ z \subseteq Talph)$
⟨*proof*⟩

1.3.4 Inverting the stack machinery (for the backward direction)

The fragment $frag\ i = A1\ A2^i\ A1$ determines i and the rest uniquely as a prefix.

lemma *frag_append_inj*:
$frag\ i\ @\ xs = frag\ j\ @\ ys \Longrightarrow i = j \wedge xs = ys$
⟨*proof*⟩

lemma *stkenc_eq_Nil*: $stkenc\ cs = [] \Longrightarrow cs = []$
⟨*proof*⟩

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

lemma *bal_stk_popcode_inv*:
assumes $bal\_stk\ t\ (brks\ (popcode\ i)) = (s, [])$ **shows** $t = frag\ i\ @\ s$
⟨*proof*⟩

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

lemma *bal_stk_xi_inv*:
assumes *inj*: $inj\_on\ idx\ N$ **and** *snd_p*: $snd\ p = Tm\ a\ \#\ map\ Nt\ Bs$
and *fpN*: $fst\ p \in N$ **and** αN: $set\ \alpha \subseteq N$
and *run*: $bal\_stk\ (stkenc\ (map\ idx\ \alpha))\ (brks\ (xi\ idx\ p)) = (s, [])$
shows $\exists \alpha'.\ \alpha = fst\ p\ \#\ \alpha' \wedge s = stkenc\ (map\ idx\ (Bs\ @\ \alpha'))$
⟨*proof*⟩

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*:
$concat\ xss = concat\ yss \Longrightarrow$
$(\forall xs{\in}set\ xss.\ \exists p.\ xs = p\ @\ [s] \wedge s \notin set\ p) \Longrightarrow$
$(\forall ys{\in}set\ yss.\ \exists p.\ ys = p\ @\ [s] \wedge s \notin set\ p) \Longrightarrow xss = yss$
⟨*proof*⟩

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 $c \notin set\ y$ **and** *fc*: $\bigwedge q.\ c \notin set\ (f\ q)$
shows $x\ @\ c\ \#\ y\ @\ c\ \#\ z = concat\ (map\ (\lambda q.\ c\ \#\ f\ q)\ qs)\ @\ [c] \Longrightarrow y \in set\ (map\ f\ qs)$
⟨*proof*⟩

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

lemma *enc_h_Dd*: $\exists\, p.\ enc\_h\ idx\ S\ P\ a = p\ @\ [Dd] \wedge Dd \notin set\ p$
⟨*proof*⟩

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

lemma *block_parse*:
assumes *eq*: $blk\ (x,y,z) = enc\_h\ idx\ S\ P\ a$
and *yset*: $set\ y \subseteq insert\ Ce\ bracks$
shows $\exists\, p.\ p \in set\ P \wedge (\exists\, Bs.\ snd\ p = Tm\ a\ \#\ map\ Nt\ Bs) \wedge y = xihat\ idx\ S\ p$
⟨*proof*⟩

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

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

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

lemma *blocks_to_ps*:
$list\_all2\ (\lambda b\ a.\ blk\ b = enc\_h\ idx\ S\ P\ a)\ bs\ w \Longrightarrow$
$(\forall\, (x,y,z){\in}set\ bs.\ set\ y \subseteq insert\ Ce\ bracks) \Longrightarrow$
$\exists\, ps.\ list\_all2\ (\lambda a\ p.\ p \in set\ P \wedge (\exists\, Bs.\ snd\ p = Tm\ a\ \#\ map\ Nt\ Bs))\ w\ ps$
$\wedge\ map\ (\lambda(x,y,z).\ y)\ bs = map\ (xihat\ idx\ S)\ ps$
⟨*proof*⟩

1.4 Theorem 2.1

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

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

and *idx_inj*: *inj_on idx* (*insert S* (*Nts* (*set P*)))
begin

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

inductive *drives* :: ('*n*,'*t*) *prod list* ⇒ '*n list* ⇒ '*t list* ⇒ '*n list* ⇒ *bool* **where**
  drives_Nil: *drives* [] *α* [] *α*
| *drives_Cons*: (*A*, *Tm a* # *map Nt Bs*) ∈ *set P* ⟹ *drives ps* (*Bs* @ *α*) *w β* ⟹
    drives ((*A*, *Tm a* # *map Nt Bs*) # *ps*) (*A* # *α*) (*a* # *w*) *β*

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

lemma *drives_bal_stk*:
  drives ps α w β ⟹
   bal_stk (*stkenc* (*map idx α*)) (*brks* (*concat* (*map* (*xi idx*) *ps*))) = (*stkenc* (*map idx β*), [])
⟨*proof*⟩

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

lemma *drives_bal_complete*:
  assumes *drives ps* [*S*] *w* []
  shows *bal* (*brks* (*pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*)))
⟨*proof*⟩

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

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

lemma *drives_imp_derivels*:
  drives ps α w β ⟹ *set P* ⊢ *map Nt α* ⇒*l** *map Tm w* @ *map Nt β*
⟨*proof*⟩

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 ⇒*l*(*Suc n*) fire before the relation power is unfolded.)

lemma *deriveln_imp_drives*:
  set P ⊢ *map Nt α* ⇒*l*(*n*) *map Tm w* ⟹ ∃ *ps*. *drives ps α w* []
⟨*proof*⟩

Combining both directions with the leftmost/standard derivation equivalence *derivels_iff_derives*: membership in the grammar's language is exactly

the existence of a *drives* sequence from $[S]$ to the empty stack.

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

1.4.2 Block structure of the encoded word

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

lemma *drives_no_S*:
$drives\ ps\ \alpha\ w\ \beta \Longrightarrow S \notin set\ \alpha \Longrightarrow (\forall\, p \in set\ ps.\ fst\ p \neq S)$
⟨*proof*⟩

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

lemma *concat_xihat_eq*:
assumes *drives ps* $[S]$ *w* []
shows *concat* (*map* (*xihat idx S*) *ps*) = *Ce* # *pushcode* (*idx S*) @ *concat* (*map* (*xi idx*) *ps*)
⟨*proof*⟩

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*:
$drives\ ps\ \alpha\ w\ \beta \Longrightarrow list\_all2\ (\lambda b\ p.\ p \in set\ P \wedge (\exists\, Bs.\ snd\ p = Tm\ b\ \#\ map\ Nt\ Bs))\ w\ ps$
⟨*proof*⟩

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

lemma *forward*:
assumes *drv*: *drives ps* $[S]$ *w* []
shows *concat* (*map* (*enc_h idx S P*) *w*) ∈ *L0*
⟨*proof*⟩

1.4.4 Backward direction of Theorem 2.1

Converse of the central invariant. If the encoded ξ-brackets of a production list *ps* (whose productions are standard-form and match the letters of w) run successfully on the encoded stack α (consuming everything, ending empty), then *ps* really is a leftmost derivation *drives ps* α *w* []. Each

step is inverted by *bal_stk_xi_inv*, which (using injectivity of *idx*) forces the production's left-hand side to be the current stack top.

lemma *drives_bal_stk_inv*:
list_all2 (λ*b p*. *p* ∈ *set P* ∧ (∃ *Bs*. *snd p* = *Tm b* # *map Nt Bs*)) *w ps* ⟹
set α ⊆ *insert S* (*Nts* (*set P*)) ⟹
bal_stk (*stkenc* (*map idx* α)) (*brks* (*concat* (*map* (*xi idx*) *ps*))) = ([], []) ⟹
drives ps α *w* []
⟨*proof*⟩

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 $\xi^\frown$-code (*block_parse*); the ¢/*D*-condition gives the balanced run feeding *drives_bal_stk_inv*.

lemma *backward_drives*:
assumes *inL0*: *concat* (*map* (*enc_h idx S P*) *w*) ∈ *L0*
and *ne*: *concat* (*map* (*enc_h idx S P*) *w*) ≠ []
shows ∃ *ps*. *drives ps* [*S*] *w* []
⟨*proof*⟩

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* ⇒ *nat*
assumes ⋀*A* α. (*A*, α) ∈ *set P* ⟹ ∃ *a Bs*. α = *Tm a* # *map Nt Bs*
and ⋀*A a Bs*. (*A*, *Tm a* # *map Nt Bs*) ∈ *set P* ⟹ *S* ∉ *set Bs*
and *inj_on idx* (*insert S* (*Nts* (*set P*)))
shows *Lang* (*set P*) *S* − {[]} = *inv_hom* (*enc_h idx S P*) (*L0* − {[]})
⟨*proof*⟩

1.4.6 Reduction of an arbitrary grammar to the standard form

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

lemma *Lang_fresh_start*:
  assumes $A' \notin Nts\ R$
  shows $Lang\ (R \cup \{(A', \gamma)\ |\gamma.\ (A, \gamma) \in R\})\ A' = Lang\ R\ A$
⟨*proof*⟩

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*: $x \in L0 \Longrightarrow x \neq [] \Longrightarrow Dd \in set\ x$
⟨*proof*⟩

lemma *inv_hom_Cc_empty*: $inv\_hom\ (\lambda\_.\ [Cc])\ (L0 - \{[]\}) = \{\}$
⟨*proof*⟩

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 $\exists h::'t \Rightarrow t0\ list.\ Lang\ P\ S - \{[]\} = inv\_hom\ h\ (L0 - \{[]\})$
⟨*proof*⟩

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 $\exists h::'t \Rightarrow t0\ list.\ Lang\ P\ S - \{[]\} = inv\_hom\ h\ (L0 - \{[]\})$
⟨*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 $\exists h.\ L - \{[]\} = inv\_hom\ h\ (L0 - \{[]\})$
⟨*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* ≡ *Aa* (*Open A1*)
abbreviation (*input*) *cA1* ≡ *Aa* (*Close A1*)
abbreviation (*input*) *oA2* ≡ *Aa* (*Open A2*)
abbreviation (*input*) *cA2* ≡ *Aa* (*Close A2*)

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

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

lemma *bracks_eq*: *bracks* = {*oA1*, *cA1*, *oA2*, *cA2*}
⟨*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*:
set w ⊆ *Talph* ⟹ *w* ∈ *Lang G NF*
⟨*proof*⟩

1.5.3 The "middle" language generated by NM

inductive *mbal* :: *t0 list* ⇒ *bool* **where**
mbal_Nil: *mbal* []
| *mbal_app*: *mbal w* ⟹ *mbal w′* ⟹ *mbal* (*w* @ *w′*)
| *mbal_wrap*: *mbal w* ⟹ *mbal* (*Aa* (*Open a*) # *w* @ [*Aa* (*Close a*)])
| *mbal_gap*: *set z* ⊆ *Talph* ⟹ *set x* ⊆ *Talph* ⟹ *mbal* (*Cc* # *z* @ *Dd* # *x* @ [*Cc*])

lemma *mbal_imp_Lang_NM*: *mbal w* ⟹ *w* ∈ *Lang G NM*
⟨*proof*⟩

1.5.4 Gaps and the interleaving of bracket blocks with gaps

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

lemma *GapI*: *set z* ⊆ *Talph* ⟹ *set x* ⊆ *Talph* ⟹ *Cc* # *z* @ *Dd* # *x* @ [*Cc*] ∈ *Gap*
⟨*proof*⟩

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

lemma *interl_merge*:
length ya = *length gs1* ⟹
interl (*ya* @ [*yl*]) *gs1* @ *interl* (*yr* # *ys2*) *gs2*
= *interl* (*ya* @ (*yl* @ *yr*) # *ys2*) (*gs1* @ *gs2*)
⟨*proof*⟩

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

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

lemma *Ymid_Nil*: *Ymid* []
⟨*proof*⟩

lemma *Ymid_gap*: $g \in Gap \Longrightarrow Ymid\ g$
⟨*proof*⟩

lemma *Ymid_app*:
assumes *Ymid Wa* **and** *Ymid Wb* **shows** *Ymid* (*Wa* @ *Wb*)
⟨*proof*⟩

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

lemma *interl_append_last*:
length ys = *length gs* ⟹ *interl* (*ys* @ [*yl* @ *s*]) *gs* = *interl* (*ys* @ [*yl*]) *gs* @ *s*
⟨*proof*⟩

lemma *Ymid_wrap*:
assumes *Ymid W* **shows** *Ymid* (*Aa* (*Open a*) # *W* @ [*Aa* (*Close a*)])
⟨*proof*⟩

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

lemma *Gap_imp_mbal*: $g \in Gap \Longrightarrow mbal\ g$
⟨*proof*⟩

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

lemma *sprd_emptybw*: *sprd W* [] ⟹ *mbal W*
⟨*proof*⟩

lemma *sprd_brk_list*:
set y ⊆ *bracks* ⟹ *sprd Wr bw* ⟹ *sprd* (*y* @ *Wr*) (*map brk y* @ *bw*)
⟨*proof*⟩

lemma *interl_sprd*:
length ys = *Suc* (*length gs*) ⟹ (∀ *g*∈*set gs*. *g* ∈ *Gap*) ⟹ (∀ *y*∈*set ys*. *set y* ⊆ *bracks*)
⟹ *sprd* (*interl ys gs*) (*map brk* (*concat ys*))
⟨*proof*⟩

lemma *sprd_split*:
sprd W bw ⟹ *bw* = *bw1* @ *bw2* ⟹ ∃ *W1 W2*. *W* = *W1* @ *W2* ∧ *sprd W1 bw1* ∧ *sprd W2 bw2*
⟨*proof*⟩

lemma *sprd_head*:
sprd W bw ⟹ *bw* = *b* # *bw′* ⟹ ∃ *U Wr*. *W* = *U* @ *Aa b* # *Wr* ∧ *mbal U* ∧ *sprd Wr bw′*

⟨*proof*⟩

lemma *sprd_tail*:
sprd W bw ⟹ *bw* = *bw′* @ [*b*] ⟹ ∃ *Wl U*. *W* = *Wl* @ *Aa b* # *U* ∧ *sprd Wl bw′* ∧ *mbal U*
⟨*proof*⟩

lemma *sprd_bal_imp_mbal*: *bal bw* ⟹ *sprd W bw* ⟹ *mbal W*
⟨*proof*⟩

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

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

lemma *mkblocks*:
length ys = *Suc* (*length gs*) ⟹ (∀ *g*∈*set gs*. *g* ∈ *Gap*) ⟹ *set x* ⊆ *Talph* ⟹ *set z* ⊆ *Talph* ⟹
∃ *bs*. *bs* ≠ [] ∧ (∀ (*a*,*b*,*c*)∈*set bs*. *set a* ⊆ *Talph* ∧ *set c* ⊆ *Talph*) ∧
map (λ(*a*,*b*,*c*). *b*) *bs* = *ys* ∧
concat (*map blk bs*) = *x* @ [*Cc*] @ *interl ys gs* @ [*Cc*] @ *z* @ [*Dd*]
⟨*proof*⟩

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*⟩

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* ⊆ *Talph* **and** *zT*: *set z* ⊆ *Talph*
shows *x* @ [*Cc*, *Ce*] @ *W* @ [*Cc*] @ *z* @ [*Dd*] ∈ *L0*
⟨*proof*⟩

lemma *L0_imp_Ymid*:
assumes *w* ∈ *L0* **and** *w* ≠ []
shows ∃ *x z W*. *w* = *x* @ [*Cc*, *Ce*] @ *W* @ [*Cc*] @ *z* @ [*Dd*] ∧ *set x* ⊆ *Talph* ∧ *set z* ⊆ *Talph* ∧ *Ymid W*
⟨*proof*⟩

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

The candidate solution assigning each nonterminal its intended language.

definition *Rsol* :: *N* ⇒ *t0 list set* **where**

Rsol A = (*case A of NZ* ⇒ *L0* | *NF* ⇒ {*w*. *set w* ⊆ *Talph*} | *NM* ⇒ {*W*. *Ymid W*})

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

lemma *incl_NZ_big*: *inst_syms Rsol* [*Nt NF*, *Tm Cc*, *Tm Ce*, *Nt NM*, *Tm Cc*, *Nt NF*, *Tm Dd*] ⊆ *Rsol NZ*
⟨*proof*⟩

lemma *Lang_G_subset*: *Lang G A* ⊆ *Rsol A*
⟨*proof*⟩

The ⊇ direction for the start symbol.

lemma *Ymid_imp_Lang_NM*: *Ymid W* ⟹ *W* ∈ *Lang G NM*
⟨*proof*⟩

lemma *L0_subset_Lang*: *L0* ⊆ *Lang G NZ*
⟨*proof*⟩

lemma *Lang_G_NZ*: *Lang G NZ* = *L0*
⟨*proof*⟩

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

theorem *CFL_L0*: *CFL TYPE*(*N*) *L0*
⟨*proof*⟩

end

References

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