Chomsky-Schützenberger Representation Theorem

Moritz Roos and Tobias Nipkow

October 13, 2025

Abstract

The Chomksy-Schützenberger Representation Theorem says that any context-free language is the homomorphic image of the intersection of a regular language and a Dyck language.

Contents

theory *Chomsky_Schuetzenberger*
imports
Context_Free_Grammar.Parse_Tree
Context_Free_Grammar.Chomsky_Normal_Form
Finite_Automata_HF.Finite_Automata_HF
Dyck_Language_Syms
begin

This theory proves the Chomsky-Schützenberger representation theorem [1]. We closely follow Kozen [2] for the proof. The theorem states that every context-free language L can be written as h $(R \cap$ *Dyck_lang* $\Gamma)$, for a suitable alphabet Γ, a regular language R and a word-homomorphism h.

The Dyck language over a set Γ (also called it's bracket language) is defined as follows: The symbols of Γ are paired with [and], as in $[_g$ and $]_g$ for $g \in \Gamma$. The Dyck language over Γ is the language of correctly bracketed words. The construction of the Dyck language is found in theory *Chomsky_Schuetzenberger.Dyck_Language_Syms*.

1 Overview of the Proof

A rough proof of Chomsky-Schützenberger is as follows: Take some context-free grammar for L with productions P. Wlog assume it is in Chomsky Normal Form. Now define a new language L' with productions P' in the following way from P:

If $\pi = A \to BC$ let $\pi' = A \to [^1{}_\pi\ B\ ]^1{}_p\ [^2{}_\pi\ C\ ]^2{}_p$, if $\pi = A \to a$ let $\pi' = A \to [^1{}_\pi\ ]^1{}_p\ [^2{}_\pi\ ]^2{}_p$, where the brackets are viewed as terminals and the old variables A, B, C are again viewed as nonterminals. This transformation is implemented by the function *transform_prod* below. Note brackets are now adorned with superscripts 1 and 2 to distinguish the first and second occurrences easily. That is, we work with symbols that are triples of type $\{[,]\} \times$ *old_prod_type* $\times \{1,2\}$.

This bracketing encodes the parse tree of any old word. The old word is easily recovered by the homomorphism which sends $[^1{}_\pi$ to a if $\pi = A \to a$, and sends every other bracket to ε. Thus we have $h(L') = L$ by essentially exchanging π for π' and the other way round in the derivation. The direction $\supseteq$ is done in *transfer_parse_tree*, the direction $\subseteq$ is done directly in the proof of the main theorem.

Then all that remains to show is, that L' is of the form $R \cap$ *Dyck_lang* Γ (for $\Gamma := P \times \{1, 2\}$) and the regularity of R.

For this, $R := R_S$ is defined via an intersection of 5 following regular languages. Each of these is defined via a property on words x:

P1 x: after a $]^1{}_p$ there always immediately follows a $[^2{}_p$ in x. This especially means, that $]^1{}_p$ cannot be the end of the string.

successively P2 x: a $]^2{}_\pi$ is never directly followed by some [in x.

successively P3 x: each $[^1{}_{A \to BC}$ is directly followed by $[^1{}_{B \to \_}$ in x (last letter isn't checked).

successively P4 x: each $[^1{}_{A \to a}$ is directly followed by $]^1{}_{A \to a}$ in x and each $[^2{}_{A \to a}$ is directly followed by $]^2{}_{A \to a}$ in x (last letter isn't checked).

P5 A x: there exists some y such that the word begins with $[^1{}_{A \to y}$.

One then shows the key theorem $P' \vdash A \to^* w \quad \longleftrightarrow \quad w \in R_A \cap$ *Dyck_lang* Γ:

The $\to$-direction (see lemma *P'_imp_Reg*) is easily checked, by checking that every condition holds during all derivation steps already. For this one needs a version of R (and all the conditions) which ignores any Terminals that might still exist in such a derivation step. Since this version operates on symbols (a different type) it needs a fully new definition. Since these new versions allow more flexibility on the words, it turns out that the original 5 conditions aren't enough anymore to fully constrain to the target language. Thus we add two additional constraints *successively P7* and *successively P8* on the symbol-version of R_A that vanish when we ultimately restricts back to words consisting only of terminal symbols. With these the induction goes through:

(*successively P7_sym*) x: each *Nt Y* is directly preceded by some *Tm* $[^1{}_{A \to YC}$ or some *Tm* $[^2{}_{A \to BY}$ in x;

(*successively P8_sym*) x: each *Nt Y* is directly followed by some $]^1{}_{A \to YC}$ or some $]^2{}_{A \to BY}$ in x.

The $\leftarrow$-direction (see lemma *Reg_and_dyck_imp_P'*) is more work. This time we stick with fully terminal words, so we work with the standard version of R_A: Proceed by induction on the length of w generalized over A. For this, let $x \in R_A \cap$ *Dyck_lang* Γ, thus we have the properties *P1 x*, *successively Pi x* for $i \in \{2,3,4,7,8\}$ and *P5 A x* available. From *P5 A x* we have that there exists $\pi \in P$ s.t. *fst* $\pi = A$ and x begins with $[^1{}_\pi$. Since $x \in$ *Dyck_lang* Γ it is balanced, so it must be of the form $x = [^1{}_\pi \;\; y \;\;]^1{}_\pi \;\; r1$ for some balanced y. From *P1 x* it must then be of the form $x = [^1{}_\pi \;\; y \;\;]^1{}_\pi \;\; [^2{}_\pi \;\; r1'$. Since x is balanced it must then be of the form $x = [^1{}_\pi \;\; y \;\;]^1{}_\pi \;\; [^2{}_\pi \; z \;]^2{}_\pi \; r2$ for some balanced z. Then $r2$ must also be balanced. If $r2$ was not empty it would begin with an opening bracket, but *P2 x* makes this impossible - so $r2 = []$ and as such $x = [^1{}_\pi \;\; y \;\;]^1{}_\pi \;\; [^2{}_\pi \; z \;]^2{}_\pi$. Since our grammar is in CNF, we can consider the following case distinction on π:

Case 1: $\pi = A \to BC$. Since y,z are balanced substrings of x one easily checks $Pi\ y$ and $Pi\ z$ for $i \in \{1,2,3,4\}$. From $P3\ x$ (and $\pi = A \to BC$) we further obtain $P5\ B\ y$ and $P5\ C\ z$. So $y \in R_B \cap Dyck\_lang\ \Gamma$ and $z \in R_C \cap Dyck\_lang\ \Gamma$. From the induction hypothesis we thus obtain $P' \vdash B \to^* y$ and $P' \vdash C \to^* z$. Since $\pi = A \to BC$ we then have $A \ \to^1{}_{\pi'}\ [^1{}_\pi\ B\ ]^1{}_\pi\ [^2{}_\pi\ C\ ]^2{}_\pi\ \to^*\ [^1{}_\pi\ y\ ]^1{}_\pi\ [^2{}_\pi\ z\ ]^2{}_\pi = x$ as required.

Case 2: $\pi = A \to a$. Suppose we didn't have $y = []$. Then from $P4\ x$ (and $\pi = A \to a$) we would have $y =]^1{}_\pi$. But since y is balanced it needs to begin with an opening bracket, contradiction. So it must be that $y = []$. By the same argument we also have that $z = []$. So really $x = [^1{}_\pi\ ]^1{}_\pi\ [^2{}_\pi\ ]^2{}_\pi$ and of course from $\pi = A \to a$ it holds $A \ \to^1{}_{\pi'}\ [^1{}_\pi\ ]^1{}_\pi\ [^2{}_\pi\ ]^2{}_\pi = x$ as required.

From the key theorem we obtain (by setting $A := S$) that $L' = R_S \cap Dyck\_lang\ \Gamma$ as wanted.

Only regularity remains to be shown. For this we use that $R_S \cap Dyck\_lang\ \Gamma = (R_S \cap brackets\ \Gamma) \cap Dyck\_lang\ \Gamma$, where $brackets\ \Gamma$ ($\supseteq Dyck\_lang\ \Gamma$) is the set of words which only consist of brackets over Γ. Actually, what we defined as R_S, isn't regular, only ($R_S \cap brackets\ \Gamma$) is. The intersection restricts to a finite amount of possible brackets, that are used in states for finite automatons for the 5 languages that R_S is the intersection of.

Throughout most of the proof below, we implicitly or explicitly assume that the grammar is in CNF. This is lifted only at the very end.

2 Production Transformation and Homomorphisms

A fixed finite set of productions P, used later on:

locale *locale_P* =
fixes *P* :: (*'n*,*'t*) *Prods*
assumes *finiteP*: ‹*finite P*›

2.1 Brackets

A type with 2 elements, for creating 2 copies as needed in the proof:

datatype *version* = *One* | *Two*

type_synonym (*'n*,*'t*) *bracket3* = ((*'n*, *'t*) *prod* × *version*) *bracket*

abbreviation *open_bracket1* :: (*'n*, *'t*) *prod* ⇒ (*'n*,*'t*) *bracket3* ($[^1$_ [*1000*]) **where**
$[^1{}_p$ ≡ (*Open* (*p*, *One*))

abbreviation *close_bracket1* :: (*'n*,*'t*) *prod* ⇒ (*'n*,*'t*) *bracket3* ($]^1$_ [*1000*]) **where**

$]^1{}_p \equiv (Close\ (p,\ One))$

abbreviation *open_bracket2* :: ('n,'t) *prod* ⇒ ('n,'t) *bracket3* ([2_ [1000]) **where**
$[^2{}_p \equiv (Open\ (p,\ Two))$

abbreviation *close_bracket2* :: ('n,'t) *prod* ⇒ ('n,'t) *bracket3* (]2_ [1000]) **where**
$]^2{}_p \equiv (Close\ (p,\ Two))$

Version for p = (A, w) (multiple letters) with bsub and esub:

abbreviation *open_bracket1'* :: ('n,'t) *prod* ⇒ ('n,'t) *bracket3* ([1_) **where**
$[^1{}_p \equiv (Open\ (p,\ One))$

abbreviation *close_bracket1'* :: ('n,'t) *prod* ⇒ ('n,'t) *bracket3* (]1_) **where**
$]^1{}_p \equiv (Close\ (p,\ One))$

abbreviation *open_bracket2'* :: ('n,'t) *prod* ⇒ ('n,'t) *bracket3* ([2_) **where**
$[^2{}_p \equiv (Open\ (p,\ Two))$

abbreviation *close_bracket2'* :: ('n,'t) *prod* ⇒ ('n,'t) *bracket3* (]2_) **where**
$]^2{}_p \equiv (Close\ (p,\ Two))$

Nice LaTeX rendering:

notation (*latex* **output**) *open_bracket1* ($[^1$_)
notation (*latex* **output**) *open_bracket1'* ($[^1$_)
notation (*latex* **output**) *open_bracket2* ($[^2$_)
notation (*latex* **output**) *open_bracket2'* ($[^2$_)
notation (*latex* **output**) *close_bracket1* ($]^1$_)
notation (*latex* **output**) *close_bracket1'* ($]^1$_)
notation (*latex* **output**) *close_bracket2* ($]^2$_)
notation (*latex* **output**) *close_bracket2'* ($]^2$_)

2.2 Transformation

abbreviation *wrap1* :: ‹'n ⇒ 't ⇒ ('n, ('n,'t) *bracket3*) *syms*› **where**
‹*wrap1 A a* ≡
[$Tm\ [^1{}_{(A,\ [Tm\ a])}$,
$Tm\ ]^1{}_{(A,\ [Tm\ a])}$,
$Tm\ [^2{}_{(A,\ [Tm\ a])}$,
$Tm\ ]^2{}_{(A,\ [Tm\ a])}$]›

abbreviation *wrap2* :: ‹'n ⇒ 'n ⇒ 'n ⇒ ('n, ('n,'t) *bracket3*) *syms*› **where**
‹*wrap2 A B C* ≡
[$Tm\ [^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$,
Nt B,
$Tm\ ]^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$,

$Tm\ [^2{}_{(A,\ [Nt\ B,\ Nt\ C])}$,
Nt C,

Tm $]^2_{(A,\ [Nt\ B,\ Nt\ C])}$]›

The transformation of old productions to new productions used in the proof:

fun *transform_rhs* :: *'n* ⇒ (*'n*, *'t*) *syms* ⇒ (*'n*, (*'n*,*'t*) *bracket3*) *syms* **where**
‹*transform_rhs A* [*Tm a*] = *wrap1 A a*› |
‹*transform_rhs A* [*Nt B*, *Nt C*] = *wrap2 A B C*›

The last equation is only added to permit us to state lemmas about

fun *transform_prod* :: (*'n*, *'t*) *prod* ⇒ (*'n*, (*'n*,*'t*) *bracket3*) *prod* **where**
‹*transform_prod* (*A*, *α*) = (*A*, *transform_rhs A α*)›

2.3 Homomorphisms

Definition of a monoid-homomorphism where multiplication is (@):

definition *hom_list* :: ‹(*'a list* ⇒ *'b list*) ⇒ *bool*› **where**
‹*hom_list h* = (∀ *a b*. *h* (*a* @ *b*) = *h a* @ *h b*)›

lemma *hom_list_Nil*: *hom_list h* ⟹ *h* [] = []
unfolding *hom_list_def* **by** (*metis self_append_conv*)

The homomorphism on single brackets:

fun *the_hom1* :: ‹(*'n*,*'t*) *bracket3* ⇒ *'t list*› **where**
‹*the_hom1* $[^1_{(A,\ [Tm\ a])}$ = [*a*]› |
‹*the_hom1* _ = []›

The homomorphism on single bracket symbols:

fun *the_hom_sym* :: ‹(*'n*, (*'n*,*'t*) *bracket3*) *sym* ⇒ (*'n*,*'t*) *sym list*› **where**
‹*the_hom_sym* (*Tm* $[^1_{(A,\ [Tm\ a])}$) = [*Tm a*]› |
‹*the_hom_sym* (*Nt A*) = [*Nt A*]› |
‹*the_hom_sym* _ = []›

The homomorphism on bracket words:

fun *the_hom* :: ‹(*'n*,*'t*) *bracket3 list* ⇒ *'t list* › (h) **where**
‹*the_hom l* = *concat* (*map the_hom1 l*)›

The homomorphism extended to symbols:

fun *the_hom_syms* :: ‹(*'n*, (*'n*,*'t*) *bracket3*) *syms* ⇒ (*'n*,*'t*) *syms*› **where**
‹*the_hom_syms l* = *concat* (*map the_hom_sym l*)›

notation *the_hom* (h)
notation *the_hom_syms* (hs)

lemma *the_hom_syms_hom*: ‹hs (*l1* @ *l2*) = hs *l1* @ hs *l2*›
by *simp*

lemma *the_hom_syms_keep_var*: ‹hs [(*Nt A*)] = [*Nt A*]›
by *simp*

lemma *the_hom_syms_tms_inj*: ‹hs w = *map Tm m* ⟹ ∃ w'. w = *map Tm* w'›

proof(*induction w arbitrary*: *m*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a w*)
  then obtain w' **where** ‹w = *map Tm* w'›
   by (*metis* (*no_types*, *opaque_lifting*) *append_Cons append_Nil map_eq_append_conv the_hom_syms_hom*)
  then obtain a' **where** ‹a = *Tm* a'›
  proof −
    assume *a1*: ⋀a'. a = *Tm* a' ⟹ *thesis*
    have *f2*: ∀ *ss s*. [*s*::(*'a*, (*'a*,*'b*) *bracket3*) *sym*] @ *ss* = *s* # *ss*
      by *auto*
    have ∀ *ss s*. (*s*::(*'a*, *'b*) *sym*) # *ss* = [*s*] @ *ss*
      by *simp*
   then show *?thesis* **using** *f2 a1* **by** (*metis sym.exhaust sym.simps*(*4*) *Cons.prems map_eq_Cons_D the_hom_syms_hom the_hom_syms_keep_var*)
  qed
  then show ‹∃ w'. *a* # *w* = *map Tm* w'›
    by (*metis List.list.simps*(*9*) ‹w = *map Tm* w'›)
qed

Helper for showing the upcoming lemma:

lemma *helper*: ‹*the_hom_sym* (*Tm x*) = *map Tm* (*the_hom1 x*)›
  by(*induction x rule*: *the_hom1.induct*)(*auto split*: *list.splits sym.splits*)

Show that the extension really is an extension in some sense:

lemma *h_eq_h_ext*: ‹hs (*map Tm x*) = *map Tm* (h *x*)›
proof(*induction x*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a x*)
  then show *?case* **using** *helper*[*of a*] **by** *simp*
qed

lemma *the_hom1_strip*: ‹(*the_hom_sym* x') = *map Tm w* ⟹ *the_hom1* (*destTm* x') = *w*›
  by(*induction* x' *rule*: *the_hom_sym.induct*; *auto*)

lemma *the_hom1_strip2*: ‹*concat* (*map the_hom_sym* w') = *map Tm w* ⟹ *concat* (*map* (*the_hom1* ∘ *destTm*) w') = *w*›
proof(*induction* w' *arbitrary*: *w*)
  case *Nil*
  then show *?case* **by** *simp*
next

```
  case (Cons a w')
  then show ?case
    by(auto simp: the_hom1_strip map_eq_append_conv append_eq_map_conv)
qed

lemma h_eq_h_ext2:
  assumes ‹hs w' = (map Tm w)›
  shows ‹h (map destTm w') = w›
using assms by (simp add: the_hom1_strip2)
```

3 The Regular Language

The regular Language *Reg* will be an intersection of 5 Languages. The languages *2*, *3* ,*4* are defined each via a relation *P2*, *P3*, *P4* on neighbouring letters and lifted to a language via *successively*. Language *1* is an intersection of another such lifted relation *P1'* and a condition on the last letter (if existent). Language *5* is a condition on the first letter (and requires it to exist). It takes a term of type *'n* (the original variable type) as parameter.

Additionally a version of each language (taking symbols as input) is defined which allows arbitrary interspersion of nonterminals.

As this interspersion weakens the description, the symbol version of the regular language (*Reg_sym*) is defined using two additional languages lifted from *P7* and *P8*. These vanish when restricted to words only containing terminals.

As stated in the introductory text, these languages will only be regular, when constrained to a finite bracket set. The theorems about this, are in the later section *Showing Regularity*.

3.1 *P1*

P1 will define a predicate on string elements. It will be true iff each $]^1{}_p$ is directly followed by $[^2{}_p$. That also means $]^1{}_p$ cannot be the end of the string.

But first we define a helper function, that only captures the neighbouring condition for two strings:

```
fun P1' :: ‹('n,'t) bracket3 ⇒ ('n,'t) bracket3 ⇒ bool› where
  ‹P1' ]^1_p [^2_p' = (p = p')› |
  ‹P1' ]^1_p y  = False› |
  ‹P1' x y = True›
```

A version of *P1'* for symbols, i.e. strings that may still contain Nt's:

```
fun P1'_sym :: ‹('n, ('n,'t) bracket3) sym ⇒ ('n, ('n,'t) bracket3) sym ⇒ bool›
where
  ‹P1'_sym (Tm ]^1_p) (Tm [^2_p')  = (p = p')› |
  ‹P1'_sym (Tm ]^1_p) y  = False› |
  ‹P1'_sym x y = True›
```

```
lemma P1'D[simp]:
  ‹P1' ]¹p r ⟷ r = [²p›
by(induction ‹]¹p› ‹r› rule: P1'.induct) auto
```

Asserts that *P1'* holds for every pair in xs, and that xs doesnt end in $]^1{}_p$:

```
fun P1 :: ('n, 't) bracket3 list ⇒ bool where
  ‹P1 xs = ((successively P1' xs) ∧ (if xs ≠ [] then (∄p. last xs = ]¹p) else True))›
```

Asserts that *P1'* holds for every pair in xs, and that xs doesnt end in *Tm* $]^1{}_p$:

```
fun P1_sym where
  ‹P1_sym xs = ((successively P1'_sym xs) ∧ (if xs ≠ [] then (∄p. last xs = Tm
]¹p) else True))›

lemma P1_for_tm_if_P1_sym[dest!]: ‹P1_sym (map Tm x) ⟹ P1 x›
proof(induction x rule: induct_list012)
  case (3 x y zs)
  then show ?case
    by(cases ‹(Tm x :: ('a, ('a,'b)bracket3) sym, Tm y :: ('a, ('a,'b)bracket3) sym)›
rule: P1'_sym.cases) auto
qed simp_all

lemma P1I[intro]:
  assumes ‹successively P1' xs›
    and ‹∄p. last xs = ]¹p›
  shows ‹P1 xs›
proof(cases xs)
  case Nil
  then show ?thesis using assms by force
next
  case (Cons a list)
 then show ?thesis using assms by (auto split: version.splits sym.splits prod.splits)
qed

lemma P1_symI[intro]:
  assumes ‹successively P1'_sym xs›
    and ‹∄p. last xs = Tm ]¹p›
  shows ‹P1_sym xs›
proof(cases xs rule: rev_cases)
  case Nil
  then show ?thesis by auto
next
  case (snoc ys y)
  then show ?thesis
    using assms by (cases y) auto
qed
```

lemma *P1_symD*[*dest*]: ‹*P1_sym xs* ⟹ *successively P1'_sym xs*› **by** *simp*

lemma *P1D_not_empty*[*intro*]:
  assumes ‹*xs* ≠ []›
    and ‹*P1 xs*›
  shows ‹*last xs* ≠ $]^1{}_p$›
proof−
  from *assms* **have** ‹*successively P1' xs* ∧ (∄ *p*. *last xs* = $]^1{}_p$)›
    by *simp*
  then show *?thesis* **by** *blast*
qed

lemma *P1_symD_not_empty'*[*intro*]:
  assumes ‹*xs* ≠ []›
    and ‹*P1_sym xs*›
  shows ‹*last xs* ≠ *Tm* $]^1{}_p$›
proof−
  from *assms* **have** ‹*successively P1'_sym xs* ∧ (∄ *p*. *last xs* = *Tm* $]^1{}_p$)›
    by *simp*
  then show *?thesis* **by** *blast*
qed

lemma *P1_symD_not_empty*:
  assumes ‹*xs* ≠ []›
    and ‹*P1_sym xs*›
  shows ‹∄ *p*. *last xs* = *Tm* $]^1{}_p$›
  using *P1_symD_not_empty'*[*OF assms*] **by** *simp*

3.2 *P2*

A $]^2{}_\pi$ is never directly followed by some [:

fun *P2* :: ‹(*'n*,*'t*) *bracket3* ⇒ (*'n*,*'t*) *bracket3* ⇒ *bool*› **where**
  ‹*P2* (*Close* (*p*, *Two*)) (*Open* (*p'*, *v*)) = *False*› |
  ‹*P2* (*Close* (*p*, *Two*)) *y* = *True*› |
  ‹*P2 x y* = *True*›

fun *P2_sym* :: ‹(*'n*, (*'n*,*'t*) *bracket3*) *sym* ⇒ (*'n*, (*'n*,*'t*) *bracket3*) *sym* ⇒ *bool*›
where
  ‹*P2_sym* (*Tm* (*Close* (*p*, *Two*))) (*Tm* (*Open* (*p'*, *v*))) = *False*› |
  ‹*P2_sym* (*Tm* (*Close* (*p*, *Two*))) *y* = *True*› |
  ‹*P2_sym x y* = *True*›

lemma *P2_for_tm_if_P2_sym*[*dest*]: ‹*successively P2_sym* (*map Tm x*) ⟹ *successively P2 x*›
  apply(*induction x rule*: *induct_list012*)
    apply *simp*
   apply *simp*
  using *P2.elims*(*3*) **by** *fastforce*

3.3 *P3*

Each $[^1{}_{A \to BC}$ is directly followed by $[^1{}_{B \to \_}$, and each $[^2{}_{A \to BC}$ is directly followed by $[^1{}_{C \to \_}$:

fun *P3* :: ‹(′n,′t) bracket3 ⇒ (′n,′t) bracket3 ⇒ bool› **where**
‹*P3* $[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$ (p, ((X,y), t)) = (p = True ∧ t = One ∧ X = B)› |
‹*P3* $[^2{}_{(A,\ [Nt\ B,\ Nt\ C])}$ (p, ((X,y), t)) = (p = True ∧ t = One ∧ X = C)› |
‹*P3 x y = True*›

Each $[^1{}_{A \to BC}$ is directly followed $[^1{}_{B \to \_}$ or *Nt B*, and each $[^2{}_{A \to BC}$ is directly followed by $[^1{}_{C \to \_}$ or *Nt C*:

fun *P3_sym* :: ‹(′n, (′n,′t) bracket3) sym ⇒ (′n, (′n,′t) bracket3) sym ⇒ bool› **where**
‹*P3_sym* (Tm $[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$) (Tm (p, ((X,y), t))) = (p = True ∧ t = One ∧ X = B)› |
— Not obvious: the case (Tm $[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$) *Nt X* is set to True with the catch all
‹*P3_sym* (Tm $[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$) (Nt X) = (X = B)› |

‹*P3_sym* (Tm $[^2{}_{(A,\ [Nt\ B,\ Nt\ C])}$) (Tm (p, ((X,y), t))) = (p = True ∧ t = One ∧ X = C)› |
‹*P3_sym* (Tm $[^2{}_{(A,\ [Nt\ B,\ Nt\ C])}$) (Nt X) = (X = C)› |
‹*P3_sym x y = True*›

lemma *P3D1*[*dest*]:
fixes *r*::‹(′n,′t) bracket3›
assumes ‹*P3* $[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$ *r*›
shows ‹∃ *l*. *r* = $[^1{}_{(B,\ l)}$›
using *assms* **by**(*induction* ‹$[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$:: (′n,′t) bracket3› *r rule*: *P3.induct*) *auto*

lemma *P3D2*[*dest*]:
fixes *r*::‹(′n,′t) bracket3›
assumes ‹*P3* $[^2{}_{(A,\ [Nt\ B,\ Nt\ C])}$ *r*›
shows ‹∃ *l*. *r* = $[^1{}_{(C,\ l)}$›
using *assms* **by**(*induction* ‹$[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$:: (′n,′t) bracket3› *r rule*: *P3.induct*) *auto*

lemma *P3_for_tm_if_P3_sym*[*dest*]: ‹*successively P3_sym* (*map Tm x*) ⟹ *successively P3 x*›
proof(*induction x rule*: *induct_list012*)
case (*3 x y zs*)
then show *?case*
by(*cases* ‹(*Tm x* :: (′a, (′a,′b) bracket3) sym, *Tm y* :: (′a, (′a,′b) bracket3) sym)› *rule*: *P3_sym.cases*) *auto*
qed *simp_all*

3.4 *P4*

Each $[^1{}_{A\to a}$ is directly followed by $]^1{}_{A\to a}$ and each $[^2{}_{A\to a}$ is directly followed by $]^2{}_{A\to a}$:

fun *P4* :: ‹(′n,′t) bracket3 ⇒ (′n,′t) bracket3 ⇒ bool› **where**
‹*P4* (*Open* ((*A*, [*Tm a*]), *s*)) (*p*, ((*X*, *y*), *t*)) = (*p* = *False* ∧ *X* = *A* ∧ *y* = [*Tm a*] ∧ *s* = *t*)› |
‹*P4 x y* = *True*›

Each $[^1{}_{A\to a}$ is directly followed by $]^1{}_{A\to a}$ and each $[^2{}_{A\to a}$ is directly followed by $]^2{}_{A\to a}$:

fun *P4_sym* :: ‹(′n, (′n,′t) bracket3) sym ⇒ (′n, (′n,′t) bracket3) sym ⇒ bool› **where**
‹*P4_sym* (*Tm* (*Open* ((*A*, [*Tm a*]), *s*))) (*Tm* (*p*, ((*X*, *y*), *t*))) = (*p* = *False* ∧ *X* = *A* ∧ *y* = [*Tm a*] ∧ *s* = *t*)› |
‹*P4_sym* (*Tm* (*Open* ((*A*, [*Tm a*]), *s*))) (*Nt X*) = *False*› |
‹*P4_sym x y* = *True*›

lemma *P4D*[*dest*]:
fixes *r*::‹(′n,′t) bracket3›
assumes ‹*P4* (*Open* ((*A*, [*Tm a*]), *v*)) *r*›
shows ‹*r* = *Close* ((*A*, [*Tm a*]), *v*)›
using *assms* **by**(*induction* ‹(*Open* ((*A*, [*Tm a*]), *v*))::(′n,′t) bracket3› *r rule*: *P4.induct*) *auto*

lemma *P4_for_tm_if_P4_sym*[*dest*]: ‹*successively P4_sym* (*map Tm x*) ⟹ *successively P4 x*›
proof(*induction x rule*: *induct_list012*)
case (*3 x y zs*)
then show *?case*
by(*cases* ‹(*Tm x* :: (′a, (′a,′b) bracket3) sym, *Tm y* :: (′a, (′a,′b) bracket3) sym)› *rule*: *P4_sym.cases*) *auto*
qed *simp_all*

3.5 *P5*

P5 A x holds, iff there exists some *y* such that *x* begins with $[^1{}_{A\to y}$:

fun *P5* :: ‹′n ⇒ (′n,′t) bracket3 list ⇒ bool› **where**
‹*P5 A* [] = *False*› |
‹*P5 A* ($[^1{}_{(X,x)}$ # *xs*) = (*X* = *A*)› |
‹*P5 A* (*x* # *xs*) = *False*›

P5_sym A x holds, iff either there exists some *y* such that *x* begins with $[^1{}_{A\to y}$, or if it begins with *Nt A*:

fun *P5_sym* :: ‹′n ⇒ (′n, (′n,′t) bracket3) syms ⇒ bool› **where**
‹*P5_sym A* [] = *False*› |
‹*P5_sym A* (*Tm* $[^1{}_{(X,x)}$ # *xs*) = (*X* = *A*)› |
‹*P5_sym A* ((*Nt X*) # *xs*) = (*X* = *A*)› |

‹P5_sym A (x # xs) = False›

lemma *P5D*[*dest*]:
assumes ‹*P5 A x*›
shows ‹$\exists$ *y*. *hd x* = $[^1{}_{(A,y)}$›
using *assms* **by**(*induction A x rule*: *P5.induct*) *auto*

lemma *P5_symD*[*dest*]:
assumes ‹*P5_sym A x*›
shows ‹($\exists$ *y*. *hd x* = *Tm* $[^1{}_{(A,y)}$) $\vee$ *hd x* = *Nt A*›
using *assms* **by**(*induction A x rule*: *P5_sym.induct*) *auto*

lemma *P5_for_tm_if_P5_sym*[*dest*]: ‹*P5_sym A* (*map Tm x*) $\Longrightarrow$ *P5 A x*›
by(*induction x*) *auto*

3.6 *P7* and *P8*

(*successively P7_sym*) *w* iff *Nt Y* is directly preceded by some *Tm* $[^1{}_{A \to YC}$ or *Tm* $[^2{}_{A \to BY}$ in w:

fun *P7_sym* :: ‹('n, ('n,'t) bracket3) sym ⇒ ('n, ('n,'t) bracket3) sym ⇒ bool›
where
‹*P7_sym* (*Tm* (*b*,(*A*, [*Nt B*, *Nt C*]), *v*)) (*Nt Y*) = (*b* = *True* ∧ ((*Y* = *B* ∧ *v* = *One*) ∨ (*Y*=*C* ∧ *v* = *Two*)))› |
‹*P7_sym x* (*Nt Y*) = *False*› |
‹*P7_sym x y* = *True*›

lemma *P7_symD*[*dest*]:
fixes *x*:: ‹('n, ('n,'t) bracket3) sym›
assumes ‹*P7_sym x* (*Nt Y*)›
shows ‹($\exists$ *A C*. *x* = *Tm* $[^1{}_{(A,[Nt\ Y,\ Nt\ C])}$) $\vee$ ($\exists$ *A B*. *x* = *Tm* $[^2{}_{(A,[Nt\ B,\ Nt\ Y])}$)›
using *assms* **by**(*induction x* ‹*Nt Y*::('n, ('n,'t) bracket3) sym› *rule*: *P7_sym.induct*) *auto*

(*successively P8_sym*) *w* iff *Nt Y* is directly followed by some $]^1{}_{A \to YC}$ or $]^2{}_{A \to BY}$ in w:

fun *P8_sym* :: ‹('n, ('n,'t) bracket3) sym ⇒ ('n, ('n,'t) bracket3) sym ⇒ bool›
where
‹*P8_sym* (*Nt Y*) (*Tm* (*b*,(*A*, [*Nt B*, *Nt C*]), *v*)) = (*b* = *False* ∧ ((*Y* = *B* ∧ *v* = *One*) ∨ (*Y*=*C* ∧ *v* = *Two*)))› |
‹*P8_sym* (*Nt Y*) *x* = *False*› |
‹*P8_sym x y* = *True*›

lemma *P8_symD*[*dest*]:
fixes *x*:: ‹('n, ('n,'t) bracket3) sym›
assumes ‹*P8_sym* (*Nt Y*) *x*›
shows ‹($\exists$ *A C*. *x* = *Tm* $]^1{}_{(A,[Nt\ Y,\ Nt\ C])}$) $\vee$ ($\exists$ *A B*. *x* = *Tm* $]^2{}_{(A,[Nt\ B,\ Nt\ Y])}$)›
using *assms* **by**(*induction* ‹*Nt Y*::('n, ('n,'t) bracket3) sym› *x rule*: *P8_sym.induct*) *auto*

3.7 *Reg* and *Reg_sym*

This is the regular language, where one takes the Start symbol as a parameter, and then has the searched for $R := R_A$:

```
definition Reg :: ‹'n ⇒ ('n,'t) bracket3 list set› where
  ‹Reg A = {x. (P1 x) ∧
    (successively P2 x) ∧
    (successively P3 x) ∧
    (successively P4 x) ∧
    (P5 A x)}›

lemma RegI[intro]:
  assumes ‹(P1 x)›
    and ‹(successively P2 x)›
    and ‹(successively P3 x)›
    and ‹(successively P4 x)›
    and ‹(P5 A x)›
  shows ‹x ∈ Reg A›
  using assms unfolding Reg_def by blast

lemma RegD[dest]:
  assumes ‹x ∈ Reg A›
  shows ‹(P1 x)›
    and ‹(successively P2 x)›
    and ‹(successively P3 x)›
    and ‹(successively P4 x)›
    and ‹(P5 A x)›
  using assms unfolding Reg_def by blast+
```

A version of *Reg* for symbols, i.e. strings that may still contain Nt's. It has 2 more Properties *P7* and *P8* that vanish for pure terminal strings:

```
definition Reg_sym :: ‹'n ⇒ ('n, ('n,'t) bracket3) syms set› where
  ‹Reg_sym A = {x. (P1_sym x) ∧
    (successively P2_sym x) ∧
    (successively P3_sym x) ∧
    (successively P4_sym x) ∧
    (P5_sym A x) ∧
    (successively P7_sym x) ∧
    (successively P8_sym x)}›

lemma Reg_symI[intro]:
  assumes ‹P1_sym x›
    and ‹successively P2_sym x›
    and ‹successively P3_sym x›
    and ‹successively P4_sym x›
    and ‹P5_sym A x›
    and ‹(successively P7_sym x)›
    and ‹(successively P8_sym x)›
  shows ‹x ∈ Reg_sym A›
```

using *assms* **unfolding** *Reg_sym_def* **by** *blast*

lemma *Reg_symD*[*dest*]:
assumes ‹$x \in$ *Reg_sym A*›
shows ‹*P1_sym x*›
and ‹*successively P2_sym x*›
and ‹*successively P3_sym x*›
and ‹*successively P4_sym x*›
and ‹*P5_sym A x*›
and ‹(*successively P7_sym x*)›
and ‹(*successively P8_sym x*)›
using *assms* **unfolding** *Reg_sym_def* **by** *blast+*

lemma *Reg_for_tm_if_Reg_sym*[*dest*]: ‹(*map Tm x*) $\in$ *Reg_sym A* $\Longrightarrow$ $x \in$ *Reg A*›
by(*rule RegI*) *auto*

4 Showing Regularity

context *locale_P*
begin

abbreviation *brackets*::‹('*n*,'*t*) *bracket3 list set*› **where**
‹*brackets* ≡ {*bs*. ∀ (_,*p*,_) ∈ *set bs*. *p* ∈ *P*}›

This is needed for the construction that shows P2,P3,P4 regular.

datatype '*a state* = *start* | *garbage* | *letter* '*a*

definition *allStates* :: ‹('*n*,'*t*) *bracket3 state set* ›**where**
‹*allStates* = { *letter* (*br*,(*p*,*v*)) | *br p v*. *p* ∈ *P* } ∪ {*start*, *garbage*}›

lemma *allStatesI*: ‹*p* ∈ *P* ⟹ *letter* (*br*,(*p*,*v*)) ∈ *allStates*›
unfolding *allStates_def* **by** *blast*

lemma *start_in_allStates*[*simp*]: ‹*start* ∈ *allStates*›
unfolding *allStates_def* **by** *blast*

lemma *garbage_in_allStates*[*simp*]: ‹*garbage* ∈ *allStates*›
unfolding *allStates_def* **by** *blast*

lemma *finite_allStates_if*:
shows ‹*finite*(*allStates*)›
proof −
define *S*::‹('*n*,'*t*) *bracket3 state set*› **where** *S* = {*letter* (*br*, (*p*, *v*)) | *br p v*. *p* ∈ *P*}
have *1*:*S* = (λ(*br*, *p*, *v*). *letter* (*br*, (*p*, *v*))) ‘ ({*True*, *False*} × *P* × {*One*, *Two*})

unfolding *S_def* **by** (*auto simp*: *image_iff intro*: *version.exhaust*)
have *finite* ({*True*, *False*} × *P* × {*One*, *Two*})

```
    using finiteP by simp
  then have ‹finite ((λ(br, p, v). letter (br, (p, v))) ‘ ({True, False} × P × {One,
Two}))›
    by blast
  then have ‹finite S›
    unfolding 1 by blast
  then have finite (S ∪ {start, garbage})
    by simp
  then show ‹finite (allStates)›
    unfolding allStates_def S_def by blast
qed

end
```

4.1 An automaton for $\{xs.\ successively\ Q\ xs \land xs \in brackets\ P\}$

```
locale successivelyConstruction = locale_P where P = P for P :: ('n,'t) Prods
+
fixes Q :: ('n,'t) bracket3 ⇒ ('n,'t) bracket3 ⇒ bool — e.g. P2
begin

fun succNext :: ‹('n,'t) bracket3 state ⇒ ('n,'t) bracket3 ⇒ ('n,'t) bracket3 state›
where
  ‹succNext garbage _ = garbage› |
  ‹succNext start (br', p', v') = (if p' ∈ P then letter (br', p',v') else garbage )› |
  ‹succNext (letter (br, p, v)) (br', p', v') =  (if Q (br,p,v) (br',p',v') ∧ p ∈ P ∧
p' ∈ P then letter (br',p',v') else garbage)›

theorem succNext_induct[case_names garbage startp startnp letterQ letternQ]:
  fixes R :: ('n,'t) bracket3 state ⇒ ('n,'t) bracket3 ⇒ bool
  fixes a0 :: ('n,'t) bracket3 state
    and a1 :: ('n,'t) bracket3
  assumes ⋀u. R garbage u
    and ⋀br' p' v'. p' ∈ P ⟹ R state.start (br', p', v')
    and ⋀br' p' v'. p' ∉ P ⟹ R state.start (br', p', v')
    and ⋀br p v br' p' v'. Q (br,p,v) (br',p',v') ∧ p ∈ P ∧ p' ∈ P ⟹ R (letter
(br, p, v)) (br', p', v')
    and ⋀br p v br' p' v'. ¬(Q (br,p,v) (br',p',v') ∧ p ∈ P ∧ p' ∈ P) ⟹ R (letter
(br, p, v)) (br', p', v')
  shows R a0 a1
by (metis assms prod_cases3 state.exhaust)

abbreviation aut where ‹aut ≡ ⦇dfa.states = allStates,
                        init  = start,
                        final = (allStates − {garbage}),
                        nxt   = succNext ⦈›

interpretation aut : dfa aut
proof(unfold_locales, goal_cases)
```

```
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by simp
next
  case (3 q x)
  then show ?case
    by(induction rule: succNext_induct[of _ q x]) (auto simp: allStatesI)
next
  case 4
  then show ?case
    using finiteP by (simp add: finite_allStates_if)
qed

lemma nextl_in_allStates[intro,simp]: ‹q ∈ allStates ⟹ aut.nextl q ys ∈ all-
States›
  using aut.nxt by(induction ys arbitrary: q) auto

lemma nextl_garbage[simp]: ‹aut.nextl garbage xs = garbage›
by(induction xs) auto

lemma drop_right: ‹xs@ys ∈ aut.language ⟹ xs ∈ aut.language›
proof(induction ys)
  case (Cons a ys)
  then have ‹xs @ [a] ∈ aut.language›
    using aut.language_def aut.nextl_app by fastforce
  then have ‹xs ∈ aut.language›
    using aut.language_def by force
  then show ?case by blast
qed auto

lemma state_after1[iff]: ‹(succNext q a ≠ garbage) = (succNext q a = letter a)›
by(induction q a rule: succNext.induct) (auto split: if_splits)

lemma state_after_in_P[intro]: ‹succNext q (br, p, v) ≠ garbage ⟹ p ∈ P›
by(induction q ‹(br, p, v)› rule: succNext_induct) auto

lemma drop_left_general: ‹aut.nextl start ys = garbage ⟹ aut.nextl q ys =
garbage›
proof(induction ys)
  case Nil
  then show ?case by simp
next
  case (Cons a ys)
  show ?case
    by(rule succNext.elims[of q a])(use Cons.prems in auto)
qed
```

```
lemma drop_left: ‹xs@ys ∈ aut.language ⟹ ys ∈ aut.language›
  unfolding aut.language_def
  by(induction xs arbitrary: ys) (auto dest: drop_left_general)

lemma empty_in_aut: ‹[] ∈ aut.language›
  unfolding aut.language_def by simp

lemma singleton_in_aut_iff: ‹[(br, p, v)] ∈ aut.language ⟷ p ∈ P›
  unfolding aut.language_def by simp

lemma duo_in_aut_iff: ‹[(br, p, v), (br', p', v')] ∈ aut.language ⟷ Q (br,p,v)
(br',p',v') ∧ p ∈ P ∧ p' ∈ P›
  unfolding aut.language_def by auto

lemma trio_in_aut_iff: ‹(br, p, v) # (br', p', v') # zs ∈ aut.language ⟷ Q
(br,p,v) (br',p',v') ∧ p ∈ P ∧ p' ∈ P ∧ (br',p',v') # zs ∈ aut.language›
proof(standard, goal_cases)
  case 1
  with drop_left have *:‹(br', p', v') # zs ∈ aut.language›
    by (metis append_Cons append_Nil)
  from drop_right 1 have ‹[(br, p, v), (br', p', v')] ∈ aut.language›
    by simp
  with duo_in_aut_iff have **:‹Q (br,p,v) (br',p',v') ∧ p ∈ P ∧ p' ∈ P›
    by blast
  from * ** show ?case by simp
next
  case 2
  then show ?case unfolding aut.language_def by auto
qed

lemma aut_lang_iff_succ_Q: ‹(successively Q xs ∧ xs ∈ brackets) ⟷ (xs ∈
aut.language)›
proof(induction xs rule: induct_list012)
  case 1
  then show ?case using empty_in_aut by auto
next
  case (2 x)
  then show ?case
    using singleton_in_aut_iff by auto
next
  case (3 x y zs)
  show ?case
  proof(cases x)
    case (fields br p v)
    then have x_eq: ‹x = (br, p, v)›
      by simp
    then show ?thesis
    proof(cases y)
      case (fields br' p' v')
```

then have *y_eq*: ‹$y = (br', p', v')$›
by *simp*
have ‹(x # y # zs ∈ *aut.language*) ⟷ Q (br,p,v) (br',p',v') ∧ $p \in P$ ∧ $p' \in P$ ∧ (br',p',v') # zs ∈ *aut.language*›
unfolding *x_eq y_eq* **using** *trio_in_aut_iff* **by** *blast*
also have ‹... ⟷ Q (br,p,v) (br',p',v') ∧ $p \in P$ ∧ $p' \in P$ ∧ (*successively* Q ((br',p',v') # zs) ∧ (br',p',v') # zs ∈ *brackets*)›
using *3* **unfolding** *x_eq y_eq* **by** *blast*
also have ‹... ⟷ *successively* Q ((br,p,v) # (br',p',v') #zs) ∧ (br,p,v) # (br',p',v') # zs ∈ *brackets*›
by *force*
also have ‹... ⟷ *successively* Q (x # y #zs) ∧ x # y # zs ∈ *brackets*›
unfolding *x_eq y_eq* **by** *blast*
finally show *?thesis* **by** *blast*
qed
qed
qed

corollary *regular_successively_inter_brackets*: ‹*regular* {*xs*. *successively* Q *xs* ∧ *xs* ∈ *brackets*}›
using *aut.regular_dfa aut_lang_iff_succ_Q* **by** *auto*

end

4.2 Regularity of *P2*, *P3* and *P4*

context *locale_P*
begin

lemma *P2_regular*:
shows ‹*regular* {*xs*. *successively P2 xs* ∧ *xs* ∈ *brackets*} ›
proof−
interpret *successivelyConstruction P P2*
by(*unfold_locales*)
show *?thesis* **using** *regular_successively_inter_brackets* **by** *blast*
qed

lemma *P3_regular*:
‹*regular* {*xs*. *successively P3 xs* ∧ *xs* ∈ *brackets*} ›
proof−
interpret *successivelyConstruction P P3*
by(*unfold_locales*)
show *?thesis* **using** *regular_successively_inter_brackets* **by** *blast*
qed

lemma *P4_regular*:
‹*regular* {*xs*. *successively P4 xs* ∧ *xs* ∈ *brackets* }›
proof−
interpret *successivelyConstruction P P4*

by(*unfold_locales*)
show *?thesis* **using** *regular_successively_inter_brackets* **by** *blast*
qed

4.3 An automaton for *P1*

More Precisely, for the *if not empty, then doesnt end in* (*Close_,1*) part. Then intersect with the other construction for *P1'* to get *P1* regular.

datatype *P1_State* = *last_ok* | *last_bad* | *garbage*

The good ending letters, are those that are not of the form (*Close* _ , *1*).

fun *good* **where**
‹*good* $]^1{}_p$ = *False*› |
‹*good* (*br*, *p*, *v*) = *True*›

fun *nxt1* :: ‹*P1_State* ⇒ ('*n*,'*t*) *bracket3* ⇒ *P1_State*› **where**
‹*nxt1 garbage* _ = *garbage*› |
‹*nxt1 last_ok* (*br*, *p*, *v*) = (*if p* ∉ *P then garbage else* (*if good* (*br*, *p*, *v*) *then last_ok else last_bad*))› |
‹*nxt1 last_bad* (*br*, *p*, *v*) = (*if p* ∉ *P then garbage else* (*if good* (*br*, *p*, *v*) *then last_ok else last_bad*))›

theorem *nxt1_induct*[*case_names garbage startp startnp letterQ letternQ*]:
fixes *R* :: *P1_State* ⇒ ('*n*,'*t*) *bracket3* ⇒ *bool*
fixes *a0* :: *P1_State*
and *a1* :: ('*n*,'*t*) *bracket3*
assumes ⋀*u*. *R garbage u*
and ⋀*br p v*. *p* ∉ *P* ⟹ *R last_ok* (*br*, *p*, *v*)
and ⋀*br p v*. *p* ∈ *P* ∧ *good* (*br*, *p*, *v*) ⟹ *R last_ok* (*br*, *p*, *v*)
and ⋀*br p v*. *p* ∈ *P* ∧ ¬(*good* (*br*, *p*, *v*)) ⟹ *R last_ok* (*br*, *p*, *v*)
and ⋀*br p v*. *p* ∉ *P* ⟹ *R last_bad* (*br*, *p*, *v*)
and ⋀*br p v*. *p* ∈ *P* ∧ *good* (*br*, *p*, *v*) ⟹ *R last_bad* (*br*, *p*, *v*)
and ⋀*br p v*. *p* ∈ *P* ∧ ¬(*good* (*br*, *p*, *v*)) ⟹ *R last_bad* (*br*, *p*, *v*)
shows *R a0 a1*
by (*metis* (*full_types*) *P1_State.exhaust assms prod_induct3*)

abbreviation *p1_aut* **where** ‹*p1_aut* ≡ ⦇*dfa.states* = {*last_ok*, *last_bad*, *garbage*},
init = *last_ok*,
final = {*last_ok*},
nxt = *nxt1*⦈›

interpretation *p1_aut* : *dfa p1_aut*
proof(*unfold_locales*, *goal_cases*)
case *1*
then show *?case* **by** *simp*
next
case *2*
then show *?case* **by** *simp*

```
next
  case (3 q x)
  then show ?case
    by(induction rule: nxt1_induct[of _ q x]) auto
next
  case 4
  then show ?case by simp
qed

lemma nextl_garbage_iff[simp]: ‹p1_aut.nextl last_ok xs = garbage ⟷ xs ∉ brackets›
proof(induction xs rule: rev_induct)
  case Nil
  then show ?case by simp
next
  case (snoc x xs)
  then have ‹xs @ [x] ∉ brackets ⟷ (xs ∉ brackets ∨ [x] ∉ brackets)›
    by auto
  moreover have ‹(p1_aut.nextl last_ok (xs@[x]) = garbage) ⟷
     (p1_aut.nextl last_ok xs = garbage) ∨ ((p1_aut.nextl last_ok (xs @ [x]) = garbage) ∧ (p1_aut.nextl last_ok (xs) ≠ garbage))›
    by auto
  ultimately show ?case using snoc
    apply (cases x)
     apply (simp)
    by (smt (z3) P1_State.exhaust P1_State.simps(3,5) nxt1.simps(2,3))
qed

lemma lang_descr_full:
  ‹(p1_aut.nextl last_ok xs = last_ok ⟷ (xs = [] ∨ (xs ≠ [] ∧ good (last xs) ∧ xs ∈ brackets))) ∧
 (p1_aut.nextl last_ok xs = last_bad ⟷ ((xs ≠ [] ∧ ¬good (last xs) ∧ xs ∈ brackets)))›
proof(induction xs rule: rev_induct)
  case Nil
  then show ?case by auto
next
  case (snoc x xs)
  then show ?case
  proof(cases ‹p1_aut.nextl last_ok (xs@[x]) = garbage›)
    case True
    then show ?thesis using nextl_garbage_iff by fastforce
  next
    case False
    then have br: ‹xs ∈ brackets› ‹[x] ∈ brackets›
      using nextl_garbage_iff by fastforce+
     with snoc consider ‹(p1_aut.nextl last_ok xs = last_ok)› | ‹(p1_aut.nextl last_ok xs = last_bad)›
      using nextl_garbage_iff by blast
```

```
    then show ?thesis
    proof(cases)
      case 1
      then show ?thesis using br by(cases ‹good x›) auto
    next
      case 2
      then show ?thesis using br by(cases ‹good x›) auto
    qed
  qed
qed

lemma lang_descr: ‹xs ∈ p1_aut.language ⟷ (xs = [] ∨ (xs ≠ [] ∧ good (last
xs) ∧ xs ∈ brackets))›
  unfolding p1_aut.language_def using lang_descr_full by auto

lemma good_iff[simp]:‹(∀ a b. last xs ≠ ]¹(a, b)) = good (last xs)›
  by (metis good.simps(1) good.elims(3) split_pairs)

lemma in_P1_iff: ‹(P1 xs ∧ xs ∈ brackets) ⟷ (xs = [] ∨ (xs ≠ [] ∧ good (last
xs) ∧ xs ∈ brackets)) ∧ successively P1' xs ∧ xs ∈ brackets›
  using good_iff by auto

corollary P1_eq: ‹{xs. P1 xs ∧ xs ∈ brackets} =
  {xs. successively P1' xs ∧ xs ∈ brackets} ∩ {xs. xs = [] ∨ (xs ≠ [] ∧ good
(last xs) ∧ xs ∈ brackets)}›
  using in_P1_iff by blast

lemma P1'_regular:
  shows ‹regular {xs. successively P1' xs ∧ xs ∈ brackets} ›
proof-
  interpret successivelyConstruction P P1'
    by(unfold_locales)
  show ?thesis using regular_successively_inter_brackets by blast
qed

corollary aux_regular: ‹regular {xs. xs = [] ∨ (xs ≠ [] ∧ good (last xs) ∧ xs ∈
brackets)}›
  using lang_descr p1_aut.regular_dfa p1_aut.language_def by simp

corollary regular_P1: ‹regular {xs. P1 xs ∧ xs ∈ brackets}›
  unfolding P1_eq using P1'_regular aux_regular using regular_Int by blast

end
```

4.4 An automaton for *P5*

```
locale P5Construction = locale_P where P=P for P :: ('n,'t)Prods +
fixes A :: 'n
begin
```

```
datatype P5_State = start | first_ok | garbage
```

The good/ok ending letters, are those that are not of the form (*Close* _ , *1*).

```
fun ok where
  ‹ok (Open ((X, _), One)) = (X = A)› |
  ‹ok _ = False›

fun nxt2 :: ‹P5_State ⇒ ('n,'t) bracket3 ⇒ P5_State› where
  ‹nxt2 garbage _ = garbage› |
  ‹nxt2 start  (br, (X, r), v) = (if (X,r) ∉ P then garbage else (if ok (br, (X, r),
v) then first_ok else garbage))› |
  ‹nxt2 first_ok (br, p, v) = (if p ∉ P then garbage else first_ok)›

theorem nxt2_induct[case_names garbage startnp start_p_ok start_p_nok first_ok_np
first_ok_p]:
  fixes R :: P5_State ⇒ ('n,'t) bracket3 ⇒ bool
  fixes a0 :: P5_State
    and a1 :: ('n,'t) bracket3
  assumes ⋀u. R garbage u
    and ⋀br p v. p ∉ P ⟹ R start (br, p, v)
    and ⋀br X r v. (X, r) ∈ P ∧ ok (br, (X, r), v) ⟹ R start (br, (X, r), v)
    and ⋀br X r v. (X, r) ∈ P ∧ ¬ok (br, (X, r), v) ⟹ R start (br, (X, r), v)
    and ⋀br X r v. (X, r) ∉ P  ⟹ R first_ok (br, (X, r), v)
    and ⋀br X r v. (X, r) ∈ P  ⟹ R first_ok (br, (X, r), v)
  shows R a0 a1
by (metis (full_types, opaque_lifting) P5_State.exhaust assms surj_pair)

abbreviation p5_aut  where ‹p5_aut ≡ (|dfa.states = {start, first_ok, garbage},
                  init  = start,
                  final = {first_ok},
                  nxt   = nxt2|)›

interpretation p5_aut : dfa p5_aut
proof(unfold_locales, goal_cases)
  case 1
  then show ?case by simp
next
  case 2
  then show ?case by simp
next
  case (3 q x)
  then show ?case by(induction rule: nxt2_induct[of _ q x]) auto
next
  case 4
  then show ?case by simp
qed
```

corollary *nxt2_start_ok_iff*: ‹*ok x* ∧ *fst*(*snd x*) ∈ *P* ⟷ *nxt2 start x* = *first_ok*›
by(*auto elim*!: *nxt2.elims ok.elims split*: *if_splits*)

lemma *empty_not_in_lang*[*simp*]:‹[] ∉ *p5_aut.language*›
  unfolding *p5_aut.language_def* **by** *auto*

lemma *singleton_in_lang_iff*: ‹[*x*] ∈ *p5_aut.language* ⟷ *ok* (*hd* [*x*]) ∧ [*x*] ∈ *brackets*›
  unfolding *p5_aut.language_def* **using** *nxt2_start_ok_iff* **by** (*cases x*) *fastforce*

lemma *singleton_first_ok_iff*: ‹*p5_aut.nextl start* ([*x*]) = *first_ok* ∨ *p5_aut.nextl start* ([*x*]) = *garbage*›
by(*cases x*) (*auto split*: *if_splits*)

lemma *first_ok_iff*: ‹*xs*≠ [] ⟹ *p5_aut.nextl start xs* = *first_ok* ∨ *p5_aut.nextl start xs* = *garbage*›
proof(*induction xs rule*: *rev_induct*)
  case *Nil*
  then show *?case* **by** *blast*
next
  case (*snoc x xs*)
  then show *?case*
  proof(*cases* ‹*xs* = []›)
    case *True*
    then show *?thesis* **unfolding** *True* **using** *singleton_first_ok_iff* **by** *auto*
  next
    case *False*
    with *snoc* **have** ‹*p5_aut.nextl start xs* = *first_ok* ∨ *p5_aut.nextl start xs* = *garbage*›
      by *blast*
    then show *?thesis*
      by(*cases x*) (*auto split*: *if_splits*)
  qed
qed

lemma *lang_descr*: ‹*xs* ∈ *p5_aut.language* ⟷ (*xs* ≠ [] ∧ *ok* (*hd xs*) ∧ *xs* ∈ *brackets*)›
proof(*induction xs rule*: *rev_induct*)
  case (*snoc x xs*)
  then have *IH*: ‹(*xs* ∈ *p5_aut.language*) = (*xs* ≠ [] ∧ *ok* (*hd xs*) ∧ *xs* ∈ *brackets*)›

    by *blast*
  then show *?case*
  proof(*cases xs*)
    case *Nil*
    then show *?thesis* **using** *singleton_in_lang_iff* **by** *auto*
  next
    case (*Cons y ys*)
    then have *xs_eq*: ‹*xs* = *y* # *ys*›

```
      by blast
    then show ?thesis
    proof(cases ‹xs ∈ p5_aut.language›)
      case True
      then have ‹(xs ≠ [] ∧ ok (hd xs) ∧ xs ∈ brackets)›
        using IH by blast
      then show ?thesis
        using p5_aut.language_def snoc by(cases x) auto
    next
      case False
      then have ‹p5_aut.nextl start xs = garbage›
        unfolding p5_aut.language_def using first_ok_iff[of xs] Cons by auto
      then have ‹p5_aut.nextl start (xs@[x]) = garbage›
        by simp
     then show ?thesis using IH unfolding xs_eq p5_aut.language_def by auto
    qed
  qed
qed simp

lemma in_P5_iff: ‹P5 A xs ∧ xs ∈ brackets ⟷ (xs ≠ [] ∧ ok (hd xs) ∧ xs ∈
brackets)›
  using P5.elims(3) by fastforce

corollary aux_regular: ‹regular {xs. xs ≠ [] ∧ ok (hd xs) ∧ xs ∈ brackets}›
  using lang_descr p5_aut.regular_dfa p5_aut.language_def by simp

lemma regular_P5:‹regular {xs. P5 A xs ∧ xs ∈ brackets}›
  using in_P5_iff aux_regular by presburger

end


context locale_P
begin

corollary regular_Reg_inter: ‹regular (brackets ∩ Reg A)›
proof−
  interpret P5Construction P A ..
  from finiteP have regs: ‹regular {xs. P1 xs ∧ xs ∈ brackets}›
    ‹regular {xs. successively P2 xs ∧  xs ∈ brackets}›
    ‹regular {xs. successively P3 xs ∧  xs ∈ brackets}›
    ‹regular {xs. successively P4 xs ∧  xs ∈ brackets}›
    ‹regular {xs. P5 A xs ∧ xs ∈ brackets}›
    using regular_P1 P2_regular P3_regular P4_regular regular_P5
    by blast+

  hence ‹regular ({xs. P1 xs ∧ xs ∈ brackets} ∩
    {xs. successively P2 xs ∧  xs ∈ brackets} ∩
    {xs. successively P3 xs ∧  xs ∈ brackets} ∩
```

{*xs. successively P4 xs* ∧ *xs* ∈ *brackets*} ∩
{*xs. P5 A xs* ∧ *xs* ∈ *brackets*})›
by (*meson regular_Int*)

moreover have *set_eq*: ‹{*xs. P1 xs* ∧ *xs* ∈ *brackets*} ∩
{*xs. successively P2 xs* ∧ *xs* ∈ *brackets*} ∩
{*xs. successively P3 xs* ∧ *xs* ∈ *brackets*} ∩
{*xs. successively P4 xs* ∧ *xs* ∈ *brackets*} ∩
{*xs. P5 A xs* ∧ *xs* ∈ *brackets*}
= *brackets* ∩ *Reg A*› **by** *auto*

ultimately show *?thesis* **by** *argo*
qed

A lemma saying that all *Dyck_lang* words really only consist of brackets (trivial definition wrangling):

lemma *Dyck_lang_subset_brackets*: ‹*Dyck_lang* (*P* × {*One*, *Two*}) ⊆ *brackets*›
unfolding *Dyck_lang_def* **using** *Ball_set* **by** *auto*

end

5 Definitions of L, Γ, P', L'

locale *Chomsky_Schuetzenberger_locale* = *locale_P* **where** *P* = *P* **for** *P* :: ('n,'t)*Prods*
+
fixes *S* :: 'n
assumes *CNF_P*: ‹*CNF P*›

begin

lemma *P_CNFE*[*dest*]:
assumes ‹π ∈ *P*›
shows ‹∃ *A a B C*. π = (*A*, [*Nt B*, *Nt C*]) ∨ π = (*A*, [*Tm a*])›
using *assms CNF_P* **unfolding** *CNF_def* **by** *fastforce*

definition *L* **where**
‹*L* = *Lang P S*›

definition Γ **where**
‹Γ = *P* × {*One*, *Two*}›

definition *P'* **where**
‹*P'* = *transform_prod* ‘ *P*›

definition *L'* **where**
‹*L'* = *Lang P' S*›

6 Lemmas for $P' \vdash A \Rightarrow^* x \longleftrightarrow x \in R_A \cap Dyck\_lang\ \Gamma$

lemma *prod1_snds_in_tm* [*intro*, *simp*]: ‹(*A*, [*Nt B*, *Nt C*]) ∈ *P* ⟹ *snds_in_tm* Γ (*wrap2 A B C*)›
unfolding *snds_in_tm_def* **using** Γ*_def* **by** *auto*

lemma *prod2_snds_in_tm* [*intro*, *simp*]: ‹(*A*, [*Tm a*]) ∈ *P* ⟹ *snds_in_tm* Γ (*wrap1 A a*)›
unfolding *snds_in_tm_def* **using** Γ*_def* **by** *auto*

lemma *bal_tm_wrap1*[*iff*]: ‹*bal_tm* (*wrap1 A a*)›
unfolding *bal_tm_def* **by** (*simp add*: *bal_iff_bal_stk*)

lemma *bal_tm_wrap2*[*iff*]: ‹*bal_tm* (*wrap2 A B C*)›
unfolding *bal_tm_def* **by** (*simp add*: *bal_iff_bal_stk*)

This essentially says, that the right sides of productions are in the Dyck language of Γ, if one ignores any occuring nonterminals. This will be needed for →.

lemma *bal_tm_transform_rhs*[*intro*!]:
‹(*A*,α) ∈ *P* ⟹ *bal_tm* (*transform_rhs A* α)›
by *auto*

lemma *snds_in_tm_transform_rhs*[*intro*!]:
‹(*A*,α) ∈ *P* ⟹ *snds_in_tm* Γ (*transform_rhs A* α)›
using *P_CNFE* **by** (*fastforce*)

The lemma for →

lemma *P'_imp_bal*:
assumes ‹*P'* ⊢ [*Nt A*] ⇒∗ *x*›
shows ‹*bal_tm x* ∧ *snds_in_tm* Γ *x*›
using *assms* **proof**(*induction rule*: *derives_induct*)
case *base*
then show *?case* **unfolding** *snds_in_tm_def* **by** *auto*
next
case (*step u A v w*)
have ‹*bal_tm* (*u* @ [*Nt A*] @ *v*)› **and** ‹*snds_in_tm* Γ (*u* @ [*Nt A*] @ *v*)›
using *step.IH step.prems* **by** *auto*
obtain *w'* **where** *w'_def*: ‹*w* = *transform_rhs A w'*› **and** *A_w'_in_P*: ‹(*A*,*w'*) ∈ *P*›
using *P'_def step.hyps*(*2*) **by** *force*
have *bal_tm_w*: ‹*bal_tm w*›
using *bal_tm_transform_rhs*[*OF* ‹(*A*,*w'*) ∈ *P*›] *w'_def* **by** *auto*
then have ‹*bal_tm* (*u* @ *w* @ *v*)›
using ‹*bal_tm* (*u* @ [*Nt A*] @ *v*)› **by** (*metis bal_tm_empty bal_tm_inside bal_tm_prepend_Nt*)
moreover have ‹*snds_in_tm* Γ (*u* @ *w* @ *v*)›

using *snds_in_tm_transform_rhs*[*OF* ‹(*A*,*w*′) ∈ *P*›] ‹*snds_in_tm* Γ (*u* @ [*Nt* *A*] @ *v*)› *w*′*_def* **by** (*simp*)
ultimately show *?case* **using** ‹*bal_tm* (*u* @ *w* @ *v*)› **by** *blast*
qed

Another lemma for →

lemma *P*′*_imp_Reg*:
assumes ‹*P*′ ⊢ [*Nt* *T*] ⇒∗ *x*›
shows ‹*x* ∈ *Reg_sym* *T*›
using *assms* **proof**(*induction rule*: *derives_induct*)
case *base*
show *?case* **by**(*rule Reg_symI*) *simp_all*
next
case (*step u A v w*)
have *uAv*: ‹*u* @ [*Nt* *A*] @ *v* ∈ *Reg_sym* *T*›
using *step* **by** *blast*
have ‹(*A*, *w*) ∈ *P*′›
using *step* **by** *blast*
then obtain *w*′ **where** *w*′*_def*: ‹*transform_prod* (*A*, *w*′) = (*A*, *w*)› **and** ‹(*A*,*w*′) ∈ *P*›
by (*smt* (*verit*, *best*) *transform_prod.simps* *P*′*_def* *P_CNFE* *fst_conv* *image_iff*)
then obtain *B C a* **where** *w_eq*: ‹*w* = *wrap1 A a* ∨ *w* = *wrap2 A B C*› (**is** ‹*w* = *?w1* ∨ *w* = *?w2*›)
by *fastforce*
then have *w_resym*: ‹*w* ∈ *Reg_sym* *A*›
by *auto*
have *P5_uAv*: ‹*P5_sym* *T* (*u* @ [*Nt* *A*] @ *v*)›
using *Reg_symD*[*OF* *uAv*] **by** *blast*
have *P1_uAv*: ‹*P1_sym* (*u* @ [*Nt* *A*] @ *v*)›
using *Reg_symD*[*OF* *uAv*] **by** *blast*
have *left*: ‹*successively P1*′*_sym* (*u*@*w*) ∧
successively P2_sym (*u*@*w*) ∧
successively P3_sym (*u*@*w*) ∧
successively P4_sym (*u*@*w*) ∧
successively P7_sym (*u*@*w*) ∧
successively P8_sym (*u*@*w*)›
proof(*cases u rule*: *rev_cases*)
case *Nil*
then show *?thesis* **using** *w_eq* **by** *auto*
next
case (*snoc ys y*)

then have ‹*successively P7_sym* (*ys* @ [*y*] @ [*Nt* *A*] @ *v*)›
using *Reg_symD*[*OF* *uAv*] *snoc* **by** *auto*
then have ‹*P7_sym* *y* (*Nt* *A*)›
by (*simp add*: *successively_append_iff*)

then obtain *R X Y v*′ **where** *y_eq*: ‹*y* = (*Tm* (*Open*((*R*, [*Nt X*, *Nt Y*]), *v*′)))›

and ‹*v*′ = *One* ⟹ *A* = *X*› **and** ‹*v*′ = *Two* ⟹ *A* = *Y*›
by *blast*
then have ‹*P3_sym y* (*hd w*)›
using *w_eq* ‹*P7_sym y* (*Nt A*)› **by** *force*
hence ‹*P1′_sym* (*last* (*ys*@[*y*])) (*hd w*) ∧
P2_sym (*last* (*ys*@[*y*])) (*hd w*) ∧
P3_sym (*last* (*ys*@[*y*])) (*hd w*) ∧
P4_sym (*last* (*ys*@[*y*])) (*hd w*) ∧
P7_sym (*last* (*ys*@[*y*])) (*hd w*) ∧
P8_sym (*last* (*ys*@[*y*])) (*hd w*)›
unfolding *y_eq* **using** *w_eq* **by** *auto*
with *Reg_symD*[*OF uAv*] **moreover have**
‹*successively P1′_sym* (*ys* @ [*y*]) ∧
successively P2_sym (*ys* @ [*y*]) ∧
successively P3_sym (*ys* @ [*y*]) ∧
successively P4_sym (*ys* @ [*y*]) ∧
successively P7_sym (*ys* @ [*y*]) ∧
successively P8_sym (*ys* @ [*y*])›
unfolding *snoc* **using** *successively_append_iff* **by** *blast*
ultimately show
‹*successively P1′_sym* (*u*@*w*) ∧
successively P2_sym (*u*@*w*) ∧
successively P3_sym (*u*@*w*) ∧
successively P4_sym (*u*@*w*) ∧
successively P7_sym (*u*@*w*) ∧
successively P8_sym (*u*@*w*)›
unfolding *snoc* **using** *Reg_symD*[*OF w_resym*] **using** *successively_append_iff*
by *blast*
qed
have *right*: ‹*successively P1′_sym* (*w*@*v*) ∧
successively P2_sym (*w*@*v*) ∧
successively P3_sym (*w*@*v*) ∧
successively P4_sym (*w*@*v*) ∧
successively P7_sym (*w*@*v*) ∧
successively P8_sym (*w*@*v*)›
proof(*cases v*)
case *Nil*
then show *?thesis* **using** *w_eq* **by** *auto*
next
case (*Cons y ys*)
then have ‹*successively P8_sym* ([*Nt A*] @ *y* # *ys*)›
using *Reg_symD*[*OF uAv*] *Cons* **using** *successively_append_iff* **by** *blast*
then have ‹*P8_sym* (*Nt A*) *y*›
by *fastforce*
then obtain *R X Y v*′ **where** *y_eq*: ‹*y* = (*Tm* (*Close*((*R*, [*Nt X*, *Nt Y*]), *v*′)))›
and ‹*v*′ = *One* ⟹ *A* = *X*› **and** ‹*v*′ = *Two* ⟹ *A* = *Y*›
by *blast*
have ‹*P1′_sym* (*last w*) (*hd* (*y*#*ys*)) ∧
P2_sym (*last w*) (*hd* (*y*#*ys*)) ∧

P3_sym (*last w*) (*hd* (*y#ys*)) ∧
P4_sym (*last w*) (*hd* (*y#ys*)) ∧
P7_sym (*last w*) (*hd* (*y#ys*)) ∧
P8_sym (*last w*) (*hd* (*y#ys*))›
unfolding *y_eq* **using** *w_eq* **by** *auto*
with *Reg_symD*[*OF uAv*] **moreover have**
‹*successively P1′_sym* (*y # ys*) ∧
successively P2_sym (*y # ys*) ∧
successively P3_sym (*y # ys*) ∧
successively P4_sym (*y # ys*) ∧
successively P7_sym (*y # ys*) ∧
successively P8_sym (*y # ys*)›
unfolding *Cons* **by** (*metis P1_symD successively_append_iff*)
ultimately show ‹*successively P1′_sym* (*w@v*) ∧
successively P2_sym (*w@v*) ∧
successively P3_sym (*w@v*) ∧
successively P4_sym (*w@v*) ∧
successively P7_sym (*w@v*) ∧
successively P8_sym (*w@v*)›
unfolding *Cons* **using** *Reg_symD*[*OF w_resym*] *successively_append_iff* **by** *blast*
qed
from *left right* **have** *P1_uwv*: ‹*successively P1′_sym* (*u@w@v*)›
using *w_eq* **by** (*metis* (*no_types*, *lifting*) *List.list.discI hd_append2 successively_append_iff*)
from *left right* **have** *ch*:
‹*successively P2_sym* (*u@w@v*) ∧
successively P3_sym (*u@w@v*) ∧
successively P4_sym (*u@w@v*) ∧
successively P7_sym (*u@w@v*) ∧
successively P8_sym (*u@w@v*)›
using *w_eq* **by** (*metis* (*no_types*, *lifting*) *List.list.discI hd_append2 successively_append_iff*)

moreover have ‹*P5_sym T* (*u@w@v*)›
using *w_eq P5_uAv* **by** (*cases u*) *auto*

moreover have ‹*P1_sym* (*u@w@v*)›
proof(*cases v rule*: *rev_cases*)
case *Nil*
then have ‹∄*p*. *last* (*u@w@v*) = *Tm* (*Close*(*p*, *One*))›
using *w_eq* **by** *auto*
with *P1_uwv* **show** ‹*P1_sym* (*u* @ *w* @ *v*)›
by *blast*
next
case (*snoc vs v′*)
then have ‹∄*p*. *last v* = *Tm* (*Close*(*p*, *One*))›
using *P1_symD_not_empty*[*OF _ P1_uAv*] **by** (*metis Nil_is_append_conv last_appendR not_Cons_self2*)

then have ‹$\nexists$ *p*. *last* (*u*@*w*@*v*) = *Tm* (*Close*(*p*, *One*))›
by (*simp add*: *snoc*)
with *P1_uwv* **show** ‹*P1_sym* (*u* @ *w* @ *v*)›
by *blast*
qed
ultimately show ‹(*u*@*w*@*v*) ∈ *Reg_sym T*›
by *blast*
qed

This will be needed for the direction ←.

lemma *transform_prod_one_step*:
assumes ‹π ∈ *P*›
shows ‹*P'* ⊢ [*Nt* (*fst* π)] ⇒ *snd* (*transform_prod* π)›
proof−
obtain *w'* **where** *w'_def*: ‹*transform_prod* π = (*fst* π, *w'*)›
by (*metis fst_eqD transform_prod.simps surj_pair*)
then have ‹(*fst* π, *w'*) ∈ *P'*›
using *assms* **by** (*simp add*: *P'_def rev_image_eqI*)
then show *?thesis*
by (*simp add*: *w'_def derive_singleton*)
qed

The lemma for ←

lemma *Reg_and_dyck_imp_P'*:
assumes ‹*x* ∈ (*Reg A* ∩ *Dyck_lang* Γ)›
shows ‹*P'* ⊢ [*Nt A*] ⇒∗ *map Tm x*› **using** *assms*
proof(*induction* ‹*length* (*map Tm x*)› *arbitrary*: *A x rule*: *less_induct*)
case *less*
then have *IH*: ‹⋀*w H*. ⟦*length* (*map Tm w*) < *length* (*map Tm x*); *w* ∈ *Reg H* ∩ *Dyck_lang* (Γ)⟧ ⟹
P' ⊢ [*Nt H*] ⇒∗ *map Tm w*›
using *less* **by** *simp*
have *xReg*: ‹*x* ∈ *Reg A*› **and** *xDL*: ‹*x* ∈ *Dyck_lang* (Γ)›
using *less* **by** *blast*+

have *p1x*: ‹*P1 x*›
and *p2x*: ‹*successively P2 x*›
and *p3x*: ‹*successively P3 x*›
and *p4x*: ‹*successively P4 x*›
and *p5x*: ‹*P5 A x*›
using *RegD*[*OF xReg*] **by** *blast*+

from *p5x* **obtain** π *t* **where** *hd_x*: ‹*hd x* = $[^1{}_\pi$› **and** *pi_def*: ‹π = (*A*, *t*)›
by (*metis List.list.sel*(*1*) *P5.elims*(*2*))
with *xReg* **have** ‹$[^1{}_\pi$ ∈ *set x*›
by (*metis List.list.sel*(*1*) *List.list.set_intros*(*1*) *RegD*(*5*) *P5.elims*(*2*))
then have *pi_in_P*: ‹π ∈ *P*›
using *xDL* **unfolding** *Dyck_lang_def* Γ*_def* **by** *fastforce*
have *bal_x*: ‹*bal x*›

using *xDL* **by** *blast*
then have ‹∃ *y r*. *bal y* ∧ *bal r* ∧ $[^1{}_\pi$ # *tl x* = $[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # *r*›
using *hd_x bal_x bal_Open_split*[*of* ‹$[^1{}_\pi$ › ‹*tl x*›] *p5x*
by(*case_tac x*) *auto*
then obtain *y r1* **where** ‹$[^1{}_\pi$ # *tl x* = $[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # *r1*› **and** *bal_y*: ‹*bal y*› **and** *bal_r1*: ‹*bal r1*›
by *blast*
then have *split1*: ‹*x* = $[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # *r1*›
using *hd_x* **by** (*metis List.list.exhaust_sel List.list.set*(*1*) ‹$[^1{}_\pi$ ∈ *set x*› *empty_iff*)
have ‹*r1* ≠ []›
proof(*rule ccontr*)
assume ‹¬ *r1* ≠ []›
then have ‹*last x* = $]^1{}_\pi$ ›
using *split1* **by**(*auto*)
then show ‹*False*›
using *p1x* **using** *P1D_not_empty split1* **by** *blast*
qed
from *p1x* **have** *hd_r1*: ‹*hd r1* = $[^2{}_\pi$›
using *split1* ‹*r1* ≠ []› **by** (*metis* (*no_types*, *lifting*) *List.list.discI List.successively.elims*(*1*) *P1'D P1.simps successively_Cons successively_append_iff*)
from *bal_r1* **have** ‹∃ *z r2*. *bal z* ∧ *bal r2* ∧ $[^2{}_\pi$ # *tl r1* = $[^2{}_\pi$ # *z* @ $]^2{}_\pi$ # *r2*›
using *bal_Open_split*[*of* ‹$[^2{}_\pi$› ‹*tl r1*›] *hd_r1* ‹*r1* ≠ []›
by(*clarsimp simp add*: *neq_Nil_conv*)
then obtain *z r2* **where** *split2'*: ‹$[^2{}_\pi$ # *tl r1* = $[^2{}_\pi$ # *z* @ $]^2{}_\pi$ # *r2*› **and** *bal_z*: ‹*bal z*› **and** *bal_r2*: ‹*bal r2*›
by *blast+*
then have *split2*: ‹*x* = $[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # $[^2{}_\pi$ # *z* @ $]^2{}_\pi$ # *r2*›
by (*metis* ‹*r1* ≠ []› *hd_r1 list.exhaust_sel split1*)
have *r2_empty*: ‹*r2* = []› — prove that if r2 was not empty, it would need to start with an open bracket, else it cant be balanced. But this cant be with P2.
proof(*cases r2*)
case (*Cons r2' r2's*)
with *bal_r2* **obtain** *g* **where** *r2_begin_op*: ‹*r2'* = (*Open g*)›
using *bal_start_Open*[*of r2' r2's*] **using** *Cons* **by** *blast*
have ‹*successively P2* ($]^2{}_\pi$ # *r2'* # *r2's*)›
using *p2x* **unfolding** *split2 Cons successively_append_iff* **by** (*metis append_Cons successively_append_iff*)
then have ‹*P2* $]^2{}_\pi$ (*r2'*)›
by *fastforce*
with *r2_begin_op* **have** ‹*False*›
by (*metis P2.simps*(*1*) *split_pairs*)
then show *?thesis* **by** *blast*
qed *blast*
then have *split3*: ‹*x* = $[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # $[^2{}_\pi$ # *z* @[$]^2{}_\pi$]›
using *split2* **by** *blast*
consider (*BC*) ‹∃ *B C*. π = (*A*, [*Nt B*, *Nt C*])› | (*a*) ‹∃ *a*. π = (*A*, [*Tm a*])›
using *assms pi_in_P local.pi_def* **by** *fastforce*
then show ‹*P'* ⊢ [*Nt A*] ⇒∗ *map Tm x*›
proof(*cases*)

case *BC*
then obtain *B C* **where** *pi_eq*: ‹π = (*A*, [*Nt B*, *Nt C*])›
by *blast*
from *split3* **have** *y_successivelys*:
‹*successively P1' y* ∧
successively P2 y ∧
successively P3 y ∧
successively P4 y›
using *P1.simps p1x p2x p3x p4x* **by** (*metis List.list.simps*(*3*) *Nil_is_append_conv successively_Cons successively_append_iff*)

have *y_not_empty*: ‹*y* ≠ []›
using *p3x pi_eq split1* **by** *fastforce*
have ‹∄ *p*. *last y* = $]^1{}_p$›
proof(*rule ccontr*)
assume ‹¬ (∄ *p*. *last y* = $]^1{}_p$)›
then obtain *p* **where** *last_y*: ‹*last y* = $]^1{}_p$ ›
by *blast*
obtain *butl* **where** *butl_def*: ‹*y* = *butl* @ [*last y*]›
by (*metis append_butlast_last_id y_not_empty*)

have ‹*successively P1'* ($[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # $[^2{}_\pi$ # *z* @[$]^2{}_\pi$])›
using *p1x split3* **by** *auto*
then have ‹*successively P1'* ($[^1{}_\pi$ # (*butl*@[*last y*]) @ $]^1{}_\pi$ # $[^2{}_\pi$ # *z* @[$]^2{}_\pi$])›
using *butl_def* **by** *simp*
then have ‹*successively P1'* ((($[^1{}_\pi$ # *butl*) @ *last y* # [$]^1{}_\pi$] @ $[^2{}_\pi$ # *z* @ [$]^2{}_\pi$])›
by (*metis* (*no_types*, *opaque_lifting*) *Cons_eq_appendI append_assoc append_self_conv2*)
then have ‹*P1'* $]^1{}_p$ $]^1{}_\pi$ ›
using *last_y* **by** (*metis* (*no_types*, *lifting*) *List.successively.simps*(*3*) *append_Cons successively_append_iff*)
then show ‹*False*›
by *simp*
qed
with *y_successivelys* **have** *P1y*: ‹*P1 y*›
by *blast*
with *p3x pi_eq* **have** ‹∃ *g*. *hd y* = $[^1{}_{(B,g)}$›
using *y_not_empty split3* **by** (*metis* (*no_types*, *lifting*) *P3D1 append_is_Nil_conv hd_append2 successively_Cons*)
then have ‹*P5 B y*›
by (*metis* ‹*y* ≠ []› *P5.simps*(*2*) *hd_Cons_tl*)
with *y_successivelys P1y* **have** ‹*y* ∈ *Reg B*›
by *blast*
moreover have ‹*y* ∈ *Dyck_lang* (Γ)›
using *split3 bal_y Dyck_lang_substring* **by** (*metis append_Cons append_Nil hd_x split1 xDL*)
ultimately have ‹*y* ∈ *Reg B* ∩ *Dyck_lang* (Γ)›

```
      by force
    moreover have ‹length (map Tm y) < length (map Tm x)›
      using length_append length_map lessI split3 by fastforce
    ultimately have der_y: ‹P' ⊢ [Nt B] ⇒* map Tm y›
      using IH[of y B] split3  by blast
    from split3 have z_successivelys:
      ‹successively P1' z ∧
     successively P2 z ∧
     successively P3 z ∧
     successively P4 z›
    using P1.simps p1x p2x p3x p4x by (metis List.list.simps(3) Nil_is_append_conv
successively_Cons successively_append_iff)
    then have successively_P3: ‹successively P3 (([¹π  # y @ [ ]¹π]) @ [²π # z
@ [ ]²π ])›
      using split3 p3x by (metis List.append.assoc append_Cons append_Nil)
    have z_not_empty: ‹z ≠ []›
    using p3x pi_eq split1 successively_P3 by (metis List.list.distinct(1) List.list.sel(1)
append_Nil P3.simps(2) successively_Cons successively_append_iff)
    then have ‹P3 [²π (hd z)›
    by (metis append_is_Nil_conv hd_append2 successively_Cons successively_P3
successively_append_iff)
    with p3x pi_eq have ‹∃ g. hd z = [¹(C,g)›
      using split_pairs by blast
    then have ‹P5 C z›
      by (metis List.list.exhaust_sel ‹z ≠ []› P5.simps(2))
    moreover have ‹P1 z›
    proof−
      have ‹∄ p. last z = ]¹p›
      proof(rule ccontr)
        assume ‹¬ (∄ p. last z = ]¹p)›
        then obtain p where last_y: ‹last z = ]¹p ›
          by blast
        obtain butl where butl_def: ‹z = butl @ [last z]›
          by (metis append_butlast_last_id z_not_empty)
        have  ‹successively P1' ([¹π  # y @ ]¹π  # [²π # z @[   ]²π   ])›
          using p1x split3 by auto
         then have ‹successively P1' ([¹π  # y @ ]¹π  # [²π # butl @ [last z] @[
]²π   ])›
          using butl_def by (metis append_assoc)
         then have ‹successively P1' (([¹π  # y @ ]¹π # [²π # butl) @ last z # [
]²π ] @ []))›
            by (metis (no_types, opaque_lifting) Cons_eq_appendI append_assoc
append_self_conv2)
        then have ‹P1' ]¹p  ]²π ›
        using last_y by (metis List.append.right_neutral List.successively.simps(3)
successively_append_iff)
        then show ‹False›
          by simp
      qed
```

then show ‹*P1 z*›
using *z_successivelys* **by** *blast*
qed

ultimately have ‹$z \in$ *Reg C*›
using *z_successivelys* **by** *blast*
moreover have ‹$z \in$ *Dyck_lang* (Γ)›
using *xDL[simplified split3] bal_z Dyck_lang_substring[of z* $[^1{}_\pi$ # *y* @ $]^1{}_\pi$ # $[^2{}_\pi$ # [] [$]^2{}_\pi$]]
by *auto*
ultimately have ‹$z \in$ *Reg C* $\cap$ *Dyck_lang* (Γ)›
by *force*
moreover have ‹*length* (*map Tm z*) < *length* (*map Tm x*)›
using *length_append length_map lessI split3* **by** *fastforce*
ultimately have *der_z*: ‹$P' \vdash$ [*Nt C*] $\Rightarrow*$ *map Tm z*›
using *IH*[*of z C*] *split3* **by** *blast*
have ‹$P' \vdash$ [*Nt A*] $\Rightarrow*$ [*Tm* $[^1{}_\pi$] @ [(*Nt B*)] @ [*Tm* $]^1{}_\pi$, *Tm* $[^2{}_\pi$] @ [(*Nt C*)] @ [*Tm* $]^2{}_\pi$]›
using *transform_prod_one_step*[*OF pi_in_P*] **using** *pi_eq* **by** *auto*
also have ‹$P' \vdash$ [*Tm* $[^1{}_\pi$] @ [(*Nt B*)] @ [*Tm* $]^1{}_\pi$, *Tm* $[^2{}_\pi$] @ [(*Nt C*)] @ [*Tm* $]^2{}_\pi$] $\Rightarrow*$ [*Tm* $[^1{}_\pi$] @ *map Tm y* @ [*Tm* $]^1{}_\pi$, *Tm* $[^2{}_\pi$] @ [(*Nt C*)] @ [*Tm* $]^2{}_\pi$]›
using *der_y* **using** *derives_append derives_prepend* **by** *blast*
also have ‹$P' \vdash$ [*Tm* $[^1{}_\pi$] @ *map Tm y* @ [*Tm* $]^1{}_\pi$, *Tm* $[^2{}_\pi$] @ [(*Nt C*)] @ [*Tm* $]^2{}_\pi$] $\Rightarrow*$ [*Tm* $[^1{}_\pi$] @ *map Tm y* @ [*Tm* $]^1{}_\pi$, *Tm* $[^2{}_\pi$] @ (*map Tm z*) @ [*Tm* $]^2{}_\pi$]›
using *der_z* **by** (*meson derives_append derives_prepend*)
finally have ‹$P' \vdash$ [*Nt A*] $\Rightarrow*$ [*Tm* $[^1{}_\pi$] @ *map Tm y* @ [*Tm* $]^1{}_\pi$, *Tm* $[^2{}_\pi$] @ (*map Tm z*) @ [*Tm* $]^2{}_\pi$]›
.
then show *?thesis* **using** *split3* **by** *simp*
next
case *a*
then obtain *a* **where** *pi_eq*: ‹π = (*A*, [*Tm a*])›
by *blast*
have ‹*y* = []›
proof(*cases y*)
case (*Cons y' ys'*)
have ‹*P4* $[^1{}_\pi$ *y'*›
using *Cons append_Cons p4x split3* **by** (*metis List.successively.simps*(*3*))
then have ‹*y'* = *Close* (π, *One*)›
using *pi_eq P4D* **by** *auto*
moreover obtain *g* **where** ‹*y'* = (*Open g*)›
using *Cons bal_start_Open bal_y* **by** *blast*
ultimately have ‹*False*›
by *blast*
then show *?thesis* **by** *blast*
qed *blast*
have ‹*z* = []›

```
    proof(cases z)
      case (Cons z' zs')
      have ‹P4 [²π z'›
        using p4x split3 by (simp add: Cons ‹y = []›)
      then have ‹z' = Close (π, One)›
        using pi_eq bal_start_Open bal_z local.Cons by blast
      moreover obtain g where ‹z' = (Open g)›
        using Cons bal_start_Open bal_z by blast
      ultimately have ‹False›
        by blast
      then show ?thesis by blast
    qed blast
    have ‹P' ⊢ [Nt A] ⇒* [ Tm [¹π,       Tm ]¹π , Tm [²π ,       Tm ]²π    ]›
      using transform_prod_one_step[OF pi_in_P] pi_eq by auto
    then show ?thesis using ‹y = []› ‹z = []› by (simp add: split3)
  qed
qed
```

7 Showing $h(L') = L$

Particularly $\supseteq$ is formally hard. To create the witness in L' we need to use the corresponding production in P' in each step. We do this by defining the transformation on the parse tree, instead of only the word. Simple induction on the derivation wouldn't (in the induction step) get us enough information on where the corresponding production needs to be applied in the transformed version.

```
abbreviation ‹roots ts ≡ map root ts›

fun wrap1_Sym :: ‹'n ⇒ ('n,'t) sym ⇒ version ⇒ ('n,('n,'t) bracket3) tree list›
where
  wrap1_Sym A (Tm a) v = [ Sym (Tm (Open ((A, [Tm a]),v))), Sym (Tm (Close
((A, [Tm a]), v)))] |
  ‹wrap1_Sym _ _ _ = []›

fun wrap2_Sym :: ‹'n ⇒ ('n,'t) sym ⇒ ('n,'t) sym ⇒ version ⇒ ('n,('n,'t)
bracket3) tree ⇒ ('n,('n,'t) bracket3) tree list›  where
  wrap2_Sym A (Nt B) (Nt C) v t = [Sym (Tm (Open ((A, [Nt B, Nt C]), v))), t
, Sym (Tm (Close ((A, [Nt B, Nt C]), v)))] |
  ‹wrap2_Sym _ _ _ _ _ = []›

fun transform_tree :: ('n,'t) tree ⇒ ('n,('n,'t) bracket3) tree where
  ‹transform_tree (Sym (Nt A)) = (Sym (Nt A))› |
  ‹transform_tree (Sym (Tm a)) = (Sym (Tm [¹(SOME A. True, [Tm a])))› |
  ‹transform_tree (Rule A [Sym (Tm a)]) = Rule A ((wrap1_Sym A (Tm a)
One)@(wrap1_Sym A (Tm a) Two))› |
  ‹transform_tree (Rule A [t1, t2]) = Rule A ((wrap2_Sym A (root t1) (root t2)
One (transform_tree t1)) @ (wrap2_Sym A (root t1) (root t2) Two (transform_tree
```

t2)))› |
‹*transform_tree* (*Rule A y*) = (*Rule A* [])›

lemma *root_of_transform_tree*[*intro*, *simp*]: ‹*root t* = *Nt X* ⟹ *root* (*transform_tree t*) = *Nt X*›
by(*induction t rule*: *transform_tree.induct*) *auto*

lemma *transform_tree_correct*:
assumes ‹*parse_tree P t* ∧ *fringe t* = *w*›
shows ‹*parse_tree P′* (*transform_tree t*) ∧ hs (*fringe* (*transform_tree t*)) = *w*›
using *assms* **proof**(*induction t arbitrary*: *w*)
case (*Sym x*)
from *Sym* **have** *pt*: ‹*parse_tree P* (*Sym x*)› **and** ‹*fringe* (*Sym x*) = *w*›
by *blast+*
then show *?case*
proof(*cases x*)
case (*Nt x1*)
then have ‹*transform_tree* (*Sym x*) = (*Sym* (*Nt x1*))›
by *simp*
then show *?thesis* **using** *Sym* **by** (*metis Nt Parse_Tree.fringe.simps*(*1*) *Parse_Tree.parse_tree.simps*(*1*) *the_hom_syms_keep_var*)
next
case (*Tm x2*)
then obtain *a* **where** ‹*transform_tree* (*Sym x*) = (*Sym* (*Tm* [^{1}(*SOME A. True*, [*Tm a*])))›

by *simp*
then have ‹*fringe* ... = [*Tm* [^{1}(*SOME A. True*, [*Tm a*])]›
by *simp*
then have ‹hs ... = [*Tm a*]›
by *simp*
then have ‹... = *w*› **using** *Sym* **using** *Tm* ‹*transform_tree* (*Sym x*) = *Sym* (*Tm* [^{1}(*SOME A. True*, [*Tm a*]))›
by *force*
then show *?thesis* **using** *Sym* **by** (*metis Parse_Tree.parse_tree.simps*(*1*) ‹*fringe* (*Sym* (*Tm* [^{1}(*SOME A. True*, [*Tm a*]))) = [*Tm* [^{1}(*SOME A. True*, [*Tm a*])]› ‹hs [*Tm* [^{1}(*SOME A. True*, [*Tm a*])] = [*Tm a*]› ‹*transform_tree* (*Sym x*) = *Sym* (*Tm* [^{1}(*SOME A. True*, [*Tm a*]))›)
qed
next
case (*Rule A ts*)
from *Rule* **have** *pt*: ‹*parse_tree P* (*Rule A ts*)› **and** *fr*: ‹*fringe* (*Rule A ts*) = *w*›
by *blast+*
from *Rule* **have** *IH*: ‹⋀*x2a w′*. ⟦*x2a* ∈ *set ts*; *parse_tree P x2a* ∧ *fringe x2a* = *w′*⟧ ⟹ *parse_tree P′* (*transform_tree x2a*) ∧ hs (*fringe* (*transform_tree x2a*)) = *w′*›
using *P′_def* **by** *blast*
from *pt* **have** ‹(*A*, *roots ts*) ∈ *P*›

by *simp*
then obtain *B C a* where
def: ‹(*A*, *roots ts*) = (*A*, [*Nt B*, *Nt C*]) ∧ *transform_prod* (*A*, *roots ts*) = (*A*, [*Tm* [^{1}(*A*, [*Nt B*, *Nt C*]) , *Nt B*, *Tm*]^1(*A*, [*Nt B*, *Nt C*]), *Tm* [^{2}(*A*, [*Nt B*, *Nt C*]), *Nt C*, *Tm*]^2(*A*, [*Nt B*, *Nt C*])])
∨
(*A*, *roots ts*) = (*A*, [*Tm a*]) ∧ *transform_prod* (*A*, *roots ts*) = (*A*, [*Tm* [^{1}(*A*, [*Tm a*]) , *Tm*]^1(*A*, [*Tm a*]), *Tm* [^{2}(*A*, [*Tm a*]), *Tm*]^2(*A*, [*Tm a*])])›
by *fastforce*
then obtain *t1 t2 e1* where *ei_def*: ‹*ts* = [*e1*] ∨ *ts* = [*t1*, *t2*]›
by *blast*
then consider (*Tm*) ‹*roots ts* = [*Tm a*] ∧ *ts* = [*Sym* (*Tm a*)]› |
(*Nt_Nt*) ‹*roots ts* = [*Nt B*, *Nt C*] ∧ *ts* = [*t1*, *t2*]›
by (*smt* (*verit*, *best*) *def list.inject list.simps*(*8*,*9*) *not_Cons_self2 prod.inject root.elims sym.distinct*(*1*))
then show *?case*
proof(*cases*)
case *Tm*
then have *ts_eq*: ‹*ts* = [*Sym* (*Tm a*)]› and *roots*: ‹*roots ts* = [*Tm a*]›
by *blast+*
then have ‹*transform_tree* (*Rule A ts*) = *Rule A* [*Sym* (*Tm* [^{1}(*A*,[*Tm a*])), *Sym*(*Tm*]^1(*A*,[*Tm a*])), *Sym* (*Tm* [^{2}(*A*,[*Tm a*])), *Sym*(*Tm*]^2(*A*, [*Tm a*]))]›
by *simp*
then have ‹hs (*fringe* (*transform_tree* (*Rule A ts*))) = [*Tm a*]›
by *simp*
also have ‹... = *w*›
using *fr* unfolding *ts_eq* by *auto*
finally have ‹hs (*fringe* (*transform_tree* (*Rule A ts*))) = *w*› .
moreover have ‹*parse_tree* (*P'*) (*transform_tree* (*Rule A* [*Sym* (*Tm a*)]))›
using *pt roots* unfolding *P'_def* by *force*
ultimately show *?thesis* unfolding *ts_eq P'_def* by *blast*
next
case *Nt_Nt*
then have *ts_eq*: ‹*ts* = [*t1*, *t2*]› and *roots*: ‹*roots ts* = [*Nt B*, *Nt C*]›
by *blast+*
then have *root_t1_eq_B*: ‹*root t1* = *Nt B*› and *root_t2_eq_C*: ‹*root t2* = *Nt C*›
by *blast+*
then have ‹*transform_tree* (*Rule A ts*) = *Rule A* ((*wrap2_Sym A* (*Nt B*) (*Nt C*) *One* (*transform_tree t1*)) @ (*wrap2_Sym A* (*Nt B*) (*Nt C*) *Two* (*transform_tree t2*)))›
by (*simp add*: *ts_eq*)
then have ‹hs (*fringe* (*transform_tree* (*Rule A ts*))) = hs (*fringe* (*transform_tree t1*)) @ hs (*fringe* (*transform_tree t2*))›
by *auto*
also have ‹... = *fringe t1* @ *fringe t2*›
using *IH pt ts_eq* by *force*
also have ‹... = *fringe* (*Rule A ts*)›
using *ts_eq* by *simp*

also have ‹... = *w*›
using *fr* **by** *blast*
ultimately have ‹hs (*fringe* (*transform_tree* (*Rule A ts*))) = *w*›
by *blast*

have ‹*parse_tree P t1*› **and** ‹*parse_tree P t2*›
using *pt ts_eq* **by** *auto*
then have ‹*parse_tree P′* (*transform_tree t1*)› **and** ‹*parse_tree P′* (*transform_tree t2*)›
by (*simp add*: *IH ts_eq*)+
have *root1*: ‹*map Parse_Tree.root* (*wrap2_Sym A* (*Nt B*) (*Nt C*) *version.One* (*transform_tree t1*)) = [*Tm* [^{1}(*A*, [*Nt B*, *Nt C*]) , *Nt B*, *Tm*]^1(*A*, [*Nt B*, *Nt C*])]›
using *root_t1_eq_B* **by** *auto*
moreover have *root2*: ‹*map Parse_Tree.root* (*wrap2_Sym A* (*Nt B*) (*Nt C*) *Two* (*transform_tree t2*)) = [*Tm* [^{2}(*A*, [*Nt B*, *Nt C*]), *Nt C*, *Tm*]^2(*A*, [*Nt B*, *Nt C*])] ›
using *root_t2_eq_C* **by** *auto*
ultimately have ‹*parse_tree P′* (*transform_tree* (*Rule A ts*))›
using ‹*parse_tree P′* (*transform_tree t1*)› ‹*parse_tree P′* (*transform_tree t2*)›
‹(*A*, *map Parse_Tree.root ts*) ∈ *P*› *roots*
by (*force simp*: *ts_eq P′_def*)
then show *?thesis*
using ‹hs (*fringe* (*transform_tree* (*Rule A ts*))) = *w*› **by** *auto*
qed
qed

lemma
transfer_parse_tree:
assumes ‹*w* ∈ *Ders P S*›
shows ‹∃ *w′* ∈ *Ders P′ S*. *w* = hs *w′*›
proof−
from *assms* **obtain** *t* **where** *t_def*: ‹*parse_tree P t* ∧ *fringe t* = *w* ∧ *root t* = *Nt S*›
using *parse_tree_if_derives DersD* **by** *meson*
then have *root_tr*: ‹*root* (*transform_tree t*) = *Nt S*›
by *blast*
from *t_def* **have** ‹*parse_tree P′* (*transform_tree t*) ∧ hs (*fringe* (*transform_tree t*)) = *w*›
using *transform_tree_correct assms* **by** *blast*
with *root_tr* **have** ‹*fringe* (*transform_tree t*) ∈ *Ders P′ S* ∧ *w* = hs (*fringe* (*transform_tree t*))›
using *fringe_steps_if_parse_tree* **by** (*metis DersI*)
then show *?thesis* **by** *blast*
qed

This is essentially $h(L') \supseteq L$:

lemma *P_imp_h_L′*:
assumes ‹*w* ∈ *Lang P S*›

shows ‹∃ *w'* ∈ *L'*. *w* = h *w'*›
proof−
have *ex*: ‹∃ *w'* ∈ *Ders P' S*. (*map Tm w*) = hs *w'*›
using *transfer_parse_tree* **by** (*meson Lang_Ders assms imageI subsetD*)
then obtain *w'* **where** *w'_def*: ‹*w'* ∈ *Ders P' S*› ‹(*map Tm w*) = hs *w'*›
using *ex* **by** *blast*
moreover obtain *w''* **where** ‹*w'* = *map Tm w''*›
using *w'_def the_hom_syms_tms_inj* **by** *metis*
then have ‹*w* = h *w''*›
using *h_eq_h_ext2* **by** (*metis h_eq_h_ext w'_def*(*2*))
moreover have ‹*w''* ∈ *L'*›
using *DersD L'_def Lang_def* ‹*w'* = *map Tm w''*› *w'_def*(*1*) **by** *fastforce*
ultimately show *?thesis*
by *blast*
qed

This lemma is used in the proof of the other direction ($h(L') \subseteq L$):

lemma *hom_ext_inv*[*simp*]:
assumes ‹π ∈ *P*›
shows ‹hs (*snd* (*transform_prod* π)) = *snd* π›
proof−
obtain *A a B C* **where** *pi_def*: ‹π = (*A*, [*Nt B*, *Nt C*]) ∨ π = (*A*, [*Tm a*])›
using *assms* **by** *fastforce*
then show *?thesis*
by *auto*
qed

This lemma is essentially the other direction ($h(L') \subseteq L$):

lemma *L'_imp_h_P*:
assumes ‹*w'* ∈ *L'*›
shows ‹h *w'* ∈ *Lang P S*›
proof−
from *assms L'_def* **have** ‹*w'* ∈ *Lang P' S*›
by *simp*
then have ‹*P'* ⊢ [*Nt S*] ⇒∗ *map Tm w'*›
by (*simp add*: *Lang_def*)
then obtain *n* **where** ‹*P'* ⊢ [*Nt S*] ⇒(*n*) *map Tm w'*›
by (*metis rtranclp_power*)
then have ‹*P* ⊢ [*Nt S*] ⇒∗ hs (*map Tm w'*)›
proof(*induction rule*: *deriven_induct*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n u A v x'*)
from ‹(*A*, *x'*) ∈ *P'*› **obtain** π **where** ‹π ∈ *P*› **and** *transf_π_def*: ‹(*transform_prod*
π) = (*A*, *x'*)›
using *P'_def* **by** *auto*
then obtain *x* **where** *π_def*: ‹π = (*A*, *x*)›
by *auto*

```
    have ‹hs (u @ [Nt A] @ v) = hs u @ hs [Nt A] @ hs v›
      by simp
    then have ‹ P ⊢ [Nt S] ⇒∗ hs u @ hs [Nt A] @ hs v›
      using Suc.IH by auto
    then have ‹ P ⊢ [Nt S] ⇒∗ hs u @ [Nt A] @ hs v›
      by simp
    then have ‹ P ⊢ [Nt S] ⇒∗ hs u @ x @ hs v›
     using π_def ‹π ∈ P› derive.intros by (metis Transitive_Closure.rtranclp.rtrancl_into_rtrancl)
    have ‹hs x′ = hs (snd (transform_prod π))›
      by (simp add: transf_π_def)
    also have ‹... = snd π›
      using hom_ext_inv ‹π ∈ P› by blast
    also have ‹... = x›
      by (simp add: π_def)
    finally have ‹hs x′ = x›
      by simp
     with ‹ P ⊢ [Nt S] ⇒∗ hs u @ x @ hs v› have ‹ P ⊢ [Nt S] ⇒∗ hs u @ hs x′
@ hs v›
      by simp
    then show ?case by auto
  qed
  then show ‹h w′ ∈ Lang P S›
    by (metis Lang_def h_eq_h_ext mem_Collect_eq)
qed
```

8 The Theorem

The constructive version of the Theorem, for a grammar already in CNF:

```
lemma Chomsky_Schuetzenberger_CNF:
  ‹regular (brackets ∩ Reg S)
   ∧ L = h ‘ ((brackets ∩ Reg S) ∩ Dyck_lang Γ)
   ∧ hom_list (h :: (′n,′t) bracket3 list ⇒ ′t list)›
proof −
  have ‹∀ A. ∀ x. P′ ⊢ [Nt A] ⇒∗ (map Tm x) ⟷ x ∈ Dyck_lang Γ ∩ Reg A›
  proof−
    have ‹∀ A. ∀ x. P′ ⊢ [Nt A] ⇒∗ (map Tm x) ⟶ x ∈ Dyck_lang Γ ∩ Reg A›
      using P′_imp_Reg P′_imp_bal Dyck_langI_tm by blast
   moreover have ‹∀ A. ∀ x. x ∈ Dyck_lang Γ ∩ Reg A ⟶ P′ ⊢ [Nt A] ⇒∗ (map
Tm x) ›
      using Reg_and_dyck_imp_P′ by blast
    ultimately show ?thesis by blast
  qed
  then have ‹L′ = Dyck_lang Γ ∩ (Reg S)›
    by (auto simp add: Lang_def L′_def)
  then have ‹h ‘ (Dyck_lang Γ ∩ Reg S) = h ‘ L′›
    by simp
  also have ‹... = Lang P S›
  proof(standard)
```

```
    show ‹h ` L' ⊆ Lang P S›
      using L'_imp_h_P by blast
  next
    show ‹Lang P S ⊆ h ` L'›
      using P_imp_h_L' by blast
  qed
  also have ‹... = L›
    by (simp add: L_def)
  finally have ‹h ` (Dyck_lang Γ ∩ Reg S) = L›
    by auto
  moreover have ‹Dyck_lang Γ ∩ (brackets ∩ Reg S) = Dyck_lang Γ ∩ Reg S›
    using Dyck_lang_subset_brackets unfolding Γ_def by fastforce
  moreover have hom: ‹hom_list h›
    by (simp add: hom_list_def)
  moreover from finiteP have ‹regular (brackets ∩ Reg S)›
    using regular_Reg_inter by fast
  ultimately have ‹regular (brackets ∩ Reg S) ∧ L = h ` ((brackets ∩ Reg S) ∩
Dyck_lang Γ) ∧ hom_list h›
    by (simp add: inf_commute)
  then show ?thesis unfolding Γ_def by blast
qed

end
```

Now we want to prove the theorem without assuming that P is in CNF. Of course any grammar can be converted into CNF, but this requires an infinite type of nonterminals (because the conversion to CNF may need to invent new nonterminals). Therefore we cannot just re-enter *locale_P*. Now we make all the assumption explicit.

The theorem for any grammar, but only for languages not containing ε:

```
lemma Chomsky_Schuetzenberger_not_empty:
  fixes P :: ‹('n :: infinite, 't) Prods› and S::'n
  defines ‹L ≡ Lang P S − {[]}›
  assumes finiteP: ‹finite P›
  shows ‹∃(R::('n,'t) bracket3 list set) h Γ. regular R ∧ L = h ` (R ∩ Dyck_lang
Γ) ∧ hom_list h›
proof −
  define h where ‹h = (the_hom:: ('n,'t) bracket3 list ⇒ 't list)›
  obtain ps where ps_def: ‹set ps = P›
    using ‹finite P› finite_list by auto
  from cnf_exists obtain ps' where
    ‹CNF(set ps')› and lang_ps_eq_lang_ps': ‹Lang (set ps') S = Lang (set ps)
S − {[]}›
    by blast
  then have ‹finite (set ps')›
    by auto
  interpret Chomsky_Schuetzenberger_locale ‹(set ps')› S
    apply unfold_locales
```

```
    using ‹finite (set ps')› ‹CNF (set ps')› by auto
  have ‹regular (brackets ∩ Reg S) ∧ Lang (set ps') S = h ‘ (brackets ∩ Reg S ∩
Dyck_lang Γ) ∧ hom_list h›
    using Chomsky_Schuetzenberger_CNF L_def h_def by argo
  moreover have ‹Lang (set ps') S = L − {[]}›
   unfolding lang_ps_eq_lang_ps' using L_def ps_def by (simp add: assms(1))
  ultimately have ‹regular (brackets ∩ Reg S) ∧ L − {[]} = h ‘ (brackets ∩ Reg
S ∩ Dyck_lang Γ) ∧ hom_list h›
    by presburger
  then show ?thesis
    using assms(1) by auto
qed
```

The Chomsky-Schützenberger theorem that we really want to prove:

```
theorem Chomsky_Schuetzenberger:
  fixes P :: ‹('n :: infinite, 't) Prods› and S :: 'n
  defines ‹L ≡ Lang P S›
  assumes finite: ‹finite P›
  shows ‹∃(R::('n,'t) bracket3 list set) h Γ. regular R ∧ L = h ‘ (R ∩ Dyck_lang
Γ) ∧ hom_list h›
proof(cases ‹[] ∈ L›)
  case False
  then show ?thesis
   using Chomsky_Schuetzenberger_not_empty[OF finite, of S] unfolding L_def
by auto
next
  case True
  obtain R::('n,'t) bracket3 list set and h and Γ where
     reg_R: ‹(regular R)› and L_minus_eq: ‹L−{[]} = h ‘ (R ∩ Dyck_lang Γ)›
and hom_h: ‹hom_list h›
    by (metis L_def Chomsky_Schuetzenberger_not_empty finite)
  then have reg_R_union: ‹regular(R ∪ {[]})›
    by (meson regular_Un regular_nullstr)
  have ‹[] = h([])›
    by (simp add: hom_h hom_list_Nil)
  moreover have ‹[] ∈ Dyck_lang Γ›
    by auto
  ultimately have ‹[] ∈ h ‘ ((R ∪ {[]}) ∩ Dyck_lang Γ)›
    by blast
  with True L_minus_eq have ‹L = h ‘ ((R ∪ {[]}) ∩ Dyck_lang Γ)›
    using ‹[] ∈ Dyck_lang Γ› ‹[] = h []› by auto
  then show ?thesis using reg_R_union hom_h by blast
qed

no_notation the_hom (h)
no_notation the_hom_syms (hs)

end
```

References

[1] N. Chomsky and M. Schützenberger. The algebraic theory of context-free languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 26 of *Studies in Logic and the Foundations of Mathematics*, pages 118–161. Elsevier, 1959.

[2] D. Kozen. *Automata and computability.* Undergraduate texts in computer science. Springer, 1997.