

Chomsky-Schützenberger Representation Theorem

Moritz Roos and Tobias Nipkow

May 18, 2026

Abstract

The Chomsky-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

1	Overview of the Proof	2
2	Production Transformation and Homomorphisms	4
2.1	Brackets	4
2.2	Transformation	5
2.3	Homomorphisms	6
3	The Regular Language	8
3.1	$P1$	8
3.2	$P2$	10
3.3	$P3$	10
3.4	$P4$	11
3.5	$P5$	12
3.6	$P6$ and $P7$	13
3.7	Reg and Reg_sym	13
4	Showing Regularity	15
4.1	An automaton for $\{xs. \textit{ successively } Q \textit{ } xs \wedge xs \in \textit{ brackets } P\}$.	16
4.2	Regularity of $P2, P3$ and $P4$	19
4.3	An automaton for $P1$	20
4.4	An automaton for $P5$	22
5	Definitions of L, Γ, P', L'	26
6	Lemmas for $P' \vdash A \Rightarrow^* x \iff x \in R_A \cap Dyck_lang \Gamma$	26
7	Showing $h(L') = L$	36

```

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 $h(R \cap \text{Dyck_lang } \Gamma)$, 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 $\pi = A \rightarrow BC$ let $\pi' = A \rightarrow [^1_\pi B]^1_p [^2_\pi C]^2_p$, if $\pi = A \rightarrow a$ let $\pi' = A \rightarrow [^1_\pi]^1_p [^2_\pi]^2_p$, 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 $\{[,]\} \times \text{old_prod_type} \times \{1,2\}$.

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 $\pi = A \rightarrow a$, 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 $R \cap \text{Dyck_lang } \Gamma$ (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 $[^1_{A \rightarrow BC}$ is directly followed by $[^1_{B \rightarrow _}$ in x (last letter isn't checked).

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

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

One then shows the key theorem $P' \vdash A \rightarrow^* w \iff w \in R_A \cap Dyck_lang \Gamma$:

The \rightarrow -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 P6* and *successively P7* 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 P6_sym*) x : each $Nt Y$ is directly preceded by some Tm $[^1_{A \rightarrow YC}$ or some Tm $]^2_{A \rightarrow BY}$ in x ;

(*successively P7_sym*) x : each $Nt Y$ is directly followed by some $]^1_{A \rightarrow YC}$ or some $]^2_{A \rightarrow BY}$ 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 $x \in R_A \cap Dyck_lang \Gamma$, 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 $x \in Dyck_lang \Gamma$ 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: $\pi = A \rightarrow BC$. 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 $\pi = A \rightarrow BC$) 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 $P' \vdash B \rightarrow^* y$ and $P' \vdash C \rightarrow^* z$. Since $\pi = A \rightarrow BC$ we then have $A \rightarrow^1_{\pi'} [^1_{\pi} B]^1_{\pi} [^2_{\pi} C]^2_{\pi} \rightarrow^* [^1_{\pi} y]^1_{\pi} [^2_{\pi} z]^2_{\pi} = x$ as required.

Case 2: $\pi = A \rightarrow a$. Suppose we didn't have $y = []$. Then from $P4 x$ (and $\pi = A \rightarrow a$) 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 $\pi = A \rightarrow a$ it holds $A \rightarrow^1_{\pi'} [^1_{\pi}]^1_{\pi} [^2_{\pi}]^2_{\pi} = x$ 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 automaton 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

```

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

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

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

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

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

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

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

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

Nice LaTeX rendering:

notation (*latex output*) $\text{open_bracket1 } ([^1_)$

notation (*latex output*) $\text{open_bracket1}' ([^1_)$

notation (*latex output*) $\text{open_bracket2 } ([^2_)$

notation (*latex output*) $\text{open_bracket2}' ([^2_)$

notation (*latex output*) $\text{close_bracket1 } (]^1_)$

notation (*latex output*) $\text{close_bracket1}' (]^1_)$

notation (*latex output*) $\text{close_bracket2 } (]^2_)$

notation (*latex output*) $\text{close_bracket2}' (]^2_)$

2.2 Transformation

abbreviation $\text{wrap1} :: \langle 'n \Rightarrow 't \Rightarrow ('n, ('n, 't) \text{bracket3}) \text{syms} \rangle$ **where**

$\langle \text{wrap1 } A \ a \equiv$
 $[\ Tm \ [^1(A, [Tm \ a]),$
 $\quad Tm \]^1(A, [Tm \ a]),$
 $\quad Tm \ [^2(A, [Tm \ a]),$
 $\quad Tm \]^2(A, [Tm \ a]) \] \rangle$

abbreviation $\text{wrap2} :: \langle 'n \Rightarrow 'n \Rightarrow 'n \Rightarrow ('n, ('n, 't) \text{bracket3}) \text{syms} \rangle$ **where**

$\langle \text{wrap2 } A \ B \ C \equiv$
 $[\ Tm \ [^1(A, [Nt \ B, Nt \ C]),$
 $\quad Nt \ B,$
 $\quad Tm \]^1(A, [Nt \ B, Nt \ C]),$
 $\quad Tm \ [^2(A, [Nt \ B, Nt \ C]),$
 $\quad Nt \ C,$

$Tm]^2(A, [Nt B, Nt C])] \rangle$

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

fun *transform_rhs* :: $\langle 'n \Rightarrow ('n, 't) \text{ syms} \Rightarrow ('n, ('n, 't) \text{ bracket3}) \text{ syms} \rangle$ **where**
 $\langle \text{transform_rhs } A [Tm a] = \text{wrap1 } A a \rangle$ |
 $\langle \text{transform_rhs } A [Nt B, Nt C] = \text{wrap2 } A B C \rangle$

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

fun *transform_prod* :: $\langle 'n, 't \rangle \text{ prod} \Rightarrow ('n, ('n, 't) \text{ bracket3}) \text{ prod}$ **where**
 $\langle \text{transform_prod } (A, \alpha) = (A, \text{transform_rhs } A \alpha) \rangle$

2.3 Homomorphisms

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

definition *hom_list* :: $\langle ('a \text{ list} \Rightarrow 'b \text{ list}) \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{hom_list } h = (\forall a b. h (a @ b) = h a @ h b) \rangle$

lemma *hom_list_Nil*: $\text{hom_list } h \Longrightarrow h [] = []$
unfolding *hom_list_def* **by** (*metis self_append_conv*)

The homomorphism on single brackets:

fun *the_hom1* :: $\langle ('n, 't) \text{ bracket3} \Rightarrow 't \text{ list} \rangle$ **where**
 $\langle \text{the_hom1 } [^1(A, [Tm a])] = [a] \rangle$ |
 $\langle \text{the_hom1 } _ = [] \rangle$

The homomorphism on single bracket symbols:

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

The homomorphism on bracket words:

fun *the_hom* :: $\langle ('n, 't) \text{ bracket3 list} \Rightarrow 't \text{ list} \rangle$ (h) **where**
 $\langle \text{the_hom } l = \text{concat } (\text{map } \text{the_hom1 } l) \rangle$

The homomorphism extended to symbols:

fun *the_hom_syms* :: $\langle ('n, ('n, 't) \text{ bracket3}) \text{ syms} \Rightarrow ('n, 't) \text{ syms} \rangle$ **where**
 $\langle \text{the_hom_syms } l = \text{concat } (\text{map } \text{the_hom_sym } l) \rangle$

notation *the_hom* (h)

notation *the_hom_syms* (hs)

lemma *the_hom_syms_hom*: $\langle \text{hs } (l1 @ l2) = \text{hs } l1 @ \text{hs } l2 \rangle$
by *simp*

lemma *the_hom_syms_keep_var*: $\langle \text{hs } [(Nt A)] = [Nt A] \rangle$
by *simp*

```

lemma the_hom_syms_tms_inj: ⟨hs  $w = \text{map } Tm\ m \implies \exists w'. w = \text{map } Tm\ w'$ ⟩

proof(induction w arbitrary: m)
  case Nil
  then show ?case by simp
next
  case (Cons a w)
  then obtain  $w'$  where  $\langle w = \text{map } Tm\ w' \rangle$ 
  by (metis (no_types, opaque_lifting) append_Cons append_Nil map_eq_append_conv
the_hom_syms_hom)
  then obtain  $a'$  where  $\langle a = Tm\ a' \rangle$ 
  proof –
    assume  $a1: \bigwedge a'. a = Tm\ a' \implies \text{thesis}$ 
    have  $f2: \forall ss\ s. [s::('a, ('a,'b)\ \text{bracket3})\ \text{sym}] @\ ss = s \# ss$ 
    by auto
    have  $\forall ss\ s. (s::('a, 'b)\ \text{sym}) \# ss = [s] @\ ss$ 
    by simp
    then show ?thesis using f2 a1 by (metis sym.exhaust sym.simps(4) Cons.prem
map_eq_Cons_D the_hom_syms_hom the_hom_syms_keep_var)
  qed
  then show  $\langle \exists w'. a \# w = \text{map } Tm\ w' \rangle$ 
  by (metis List.list.simps(9) \langle w = \text{map } Tm\ w' \rangle)
qed

```

Helper for showing the upcoming lemma:

```

lemma helper: ⟨the_hom_sym ( $Tm\ x$ ) =  $\text{map } Tm\ (\text{the\_hom1 } x)$ ⟩
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 ( $\text{map } Tm\ x$ ) =  $\text{map } Tm\ (h\ x)$ ⟩
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  $x'$ ) =  $\text{map } Tm\ w \implies \text{the\_hom1 } (\text{destTm } x') = w$ ⟩
by(induction x' rule: the_hom_sym.induct; auto)

```

```

lemma the_hom1_strip2: ⟨ $\text{concat } (\text{map } \text{the\_hom\_sym } w') = \text{map } Tm\ w \implies$ 
 $\text{concat } (\text{map } (\text{the\_hom1 } \circ \text{destTm}) w') = w$ ⟩
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 interspersions of nonterminals.

As this interspersions weakens the description, the symbol version of the regular language (*Reg_sym*) is defined using two additional languages lifted from $P6$ and $P7$. 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 $]_p^1$ is directly followed by $[_p^2$. That also means $]_p^1$ 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' ]_p^1 b' = (b' = [_p^2)⟩ |
  ⟨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 ]_p^1) s' = (s' = (Tm [_p^2))⟩ |
  ⟨P1'_sym x y = True⟩

```

Asserts that $P1'$ holds for every pair in xs , and that xs doesn't end in $]_p^1$:

```

fun P1 :: ('n, 't) bracket3 list  $\Rightarrow$  bool where
   $\langle P1\ u = ((successively\ P1'\ u) \wedge (u \neq [] \longrightarrow (\nexists \pi. last\ u = ]^1_\pi))) \rangle$ 

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

fun P1_sym where
   $\langle P1\_sym\ xs = ((successively\ P1'\_sym\ xs) \wedge (xs \neq [] \longrightarrow (\nexists p. last\ xs = Tm\ ]^1_p))) \rangle$ 

lemma P1_for_tm_if_P1_sym[dest]:  $\langle P1\_sym\ (map\ Tm\ x) \Longrightarrow P1\ x \rangle$ 
proof(induction x rule: induct_list012)
  case ( $\exists\ x\ y\ zs$ )
  then show ?case
  by(cases  $\langle (Tm\ x :: ('a, ('a,'b)bracket3)\ sym, Tm\ y :: ('a, ('a,'b)bracket3)\ sym) \rangle$ 
rule: P1'_sym.cases) auto
qed simp_all

lemma P1I[intro]:
  assumes  $\langle successively\ P1'\ xs \rangle$ 
  and  $\langle \nexists p. last\ xs = ]^1_p \rangle$ 
  shows  $\langle P1\ xs \rangle$ 
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  $\langle successively\ P1'\_sym\ xs \rangle$ 
  and  $\langle \nexists p. last\ xs = Tm\ ]^1_p \rangle$ 
  shows  $\langle P1\_sym\ xs \rangle$ 
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]:  $\langle P1\_sym\ xs \Longrightarrow successively\ P1'\_sym\ xs \rangle$  by simp

lemma P1D_not_empty[intro]:
  assumes  $\langle xs \neq [] \rangle$ 
  and  $\langle P1\ xs \rangle$ 
  shows  $\langle last\ xs \neq ]^1_p \rangle$ 
proof –

```

```

from assms have ⟨successively  $P1' \ xs \wedge (\#p. \text{last } xs = ]^1_p)$ ⟩
  by simp
then show ?thesis by blast
qed

```

```

lemma P1_symD_not_empty[intro]:
  assumes ⟨ $xs \neq []$ ⟩
    and ⟨P1_sym  $xs$ ⟩
  shows ⟨ $\text{last } xs \neq Tm ]^1_p$ ⟩
proof –
  from assms have ⟨successively  $P1\_sym \ xs \wedge (\#p. \text{last } xs = Tm ]^1_p)$ ⟩
    by simp
  then show ?thesis by blast
qed

```

```

lemma P1_symD_not_empty:
  assumes ⟨ $xs \neq []$ ⟩
    and ⟨P1_sym  $xs$ ⟩
  shows ⟨ $\#p. \text{last } xs = Tm ]^1_p$ ⟩
  using P1_symD_not_empty'[OF assms] by simp

```

3.2 $P2$

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

```

fun P2 :: ⟨ $('n, 't) \text{ bracket3} \Rightarrow ('n, 't) \text{ bracket3} \Rightarrow \text{bool}$ ⟩ where
  ⟨P2 (Close ( $p, Two$ )) (Open ( $p', v$ )) = False⟩ |
  ⟨P2 (Close ( $p, Two$ ))  $y = True$ ⟩ |
  ⟨P2  $x \ y = True$ ⟩

```

```

fun P2_sym :: ⟨ $('n, ('n, 't) \text{ bracket3}) \text{ sym} \Rightarrow ('n, ('n, 't) \text{ bracket3}) \text{ sym} \Rightarrow \text{bool}$ ⟩
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  $P2\_sym \ (\text{map } Tm \ x) \Longrightarrow \text{successively } P2 \ x$ ⟩
  apply(induction  $x$  rule: induct_list012)
    apply simp
    apply simp
  using P2.elims( $3$ ) by fastforce

```

3.3 $P3$

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

```

fun P3 :: ⟨ $('n, 't) \text{ bracket3} \Rightarrow ('n, 't) \text{ bracket3} \Rightarrow \text{bool}$ ⟩ where
  ⟨P3  $]^1(A, [Nt \ B, Nt \ C]) \ (p, ((X, y), t)) = (p = True \wedge t = One \wedge X = B)$ ⟩ |

```

$\langle P3 \ [^2(A, [Nt\ B, Nt\ C])\ (p, ((X,y), t)) = (p = True \wedge t = One \wedge X = C)\rangle \mid$
 $\langle P3\ x\ y = True\rangle$

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

fun $P3_sym :: \langle ('n, ('n, 't)\ bracket3)\ sym \Rightarrow ('n, ('n, 't)\ bracket3)\ sym \Rightarrow bool\rangle$
where

$\langle P3_sym\ (Tm\ [^1(A, [Nt\ B, Nt\ C])\ (Tm\ (p, ((X,y), t))) = (p = True \wedge t = One \wedge X = B)\rangle \mid$

— Not obvious: the case $(Tm\ [^1(A, [Nt\ B, Nt\ C])\ Nt\ X)$ is set to `True` with the catch all

$\langle P3_sym\ (Tm\ [^1(A, [Nt\ B, Nt\ C])\ (Nt\ X) = (X = B)\rangle \mid$

$\langle P3_sym\ (Tm\ [^2(A, [Nt\ B, Nt\ C])\ (Tm\ (p, ((X,y), t))) = (p = True \wedge t = One \wedge X = C)\rangle \mid$

$\langle P3_sym\ (Tm\ [^2(A, [Nt\ B, Nt\ C])\ (Nt\ X) = (X = C)\rangle \mid$

$\langle P3_sym\ x\ y = True\rangle$

lemma $P3D1[dest]$:

fixes $r :: \langle ('n, 't)\ bracket3\rangle$

assumes $\langle P3\ [^1(A, [Nt\ B, Nt\ C])\ r\rangle$

shows $\langle \exists l. r = [^1(B, l)\rangle$

using *assms* **by** $(induction\ \langle [^1(A, [Nt\ B, Nt\ C]) :: ('n, 't)\ bracket3\rangle\ r\ rule:\ P3.induct)$
auto

lemma $P3D2[dest]$:

fixes $r :: \langle ('n, 't)\ bracket3\rangle$

assumes $\langle P3\ [^2(A, [Nt\ B, Nt\ C])\ r\rangle$

shows $\langle \exists l. r = [^1(C, l)\rangle$

using *assms* **by** $(induction\ \langle [^1(A, [Nt\ B, Nt\ C]) :: ('n, 't)\ bracket3\rangle\ r\ rule:\ P3.induct)$
auto

lemma $P3_for_tm_if_P3_sym[dest]$: $\langle successively\ P3_sym\ (map\ Tm\ x) \Longrightarrow successively\ P3\ x\rangle$

proof $(induction\ x\ rule:\ induct_list012)$

case $(\exists\ x\ y\ zs)$

then show *?case*

by $(cases\ \langle (Tm\ x :: ('a, ('a, 'b)\ bracket3)\ sym, Tm\ y :: ('a, ('a, 'b)\ bracket3)\ sym)\rangle\ rule:\ P3_sym.cases)\ auto$

qed *simp_all*

3.4 $P4$

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

fun $P4 :: \langle ('n, 't)\ bracket3 \Rightarrow ('n, 't)\ bracket3 \Rightarrow bool\rangle$ **where**

$\langle P4 \text{ (Open ((A, [Tm a]), s)) (p, ((X, y), t)) = (p = \text{False} \wedge X = A \wedge y = [Tm a] \wedge s = t) \rangle \mid$
 $\langle P4 \ x \ y = \text{True} \rangle$

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

fun $P4_sym :: \langle ('n, ('n, 't) \text{ bracket3}) \text{ sym} \Rightarrow ('n, ('n, 't) \text{ bracket3}) \text{ sym} \Rightarrow \text{bool} \rangle$
where
 $\langle P4_sym \text{ (Tm (Open ((A, [Tm a]), s)) (Tm (p, ((X, y), t))) = (p = \text{False} \wedge X = A \wedge y = [Tm a] \wedge s = t) \rangle \mid$
 $\langle P4_sym \text{ (Tm (Open ((A, [Tm a]), s)) (Nt X) = \text{False})} \mid$
 $\langle P4_sym \ x \ y = \text{True} \rangle$

lemma $P4D[\text{dest}]$:

fixes $r :: \langle ('n, 't) \text{ bracket3} \rangle$
assumes $\langle P4 \text{ (Open ((A, [Tm a]), v))} \ r \rangle$
shows $\langle r = \text{Close ((A, [Tm a]), v) \rangle$
using $\text{assms by (induction } \langle (\text{Open ((A, [Tm a]), v)) :: ('n, 't) \text{ bracket3} \rangle \ r \ \text{rule: } P4.\text{induct}) \ \text{auto}}$

lemma $P4_for_tm_if_P4_sym[\text{dest}]$: $\langle \text{successively } P4_sym \text{ (map Tm } x) \Longrightarrow \text{successively } P4 \ x \rangle$

proof $(\text{induction } x \ \text{rule: } \text{induct_list012})$

case $(\exists \ x \ y \ zs)$
then show $?case$
by $(\text{cases } \langle (\text{Tm } x :: ('a, ('a, 'b) \text{ bracket3}) \text{ sym}, \text{Tm } y :: ('a, ('a, 'b) \text{ bracket3}) \text{ sym}) \rangle \ \text{rule: } P4_sym.\text{cases}) \ \text{auto}$
qed simp_all

3.5 $P5$

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

fun $P5 :: \langle 'n \Rightarrow ('n, 't) \text{ bracket3 list} \Rightarrow \text{bool} \rangle$ **where**

$\langle P5 \ A \ [] = \text{False} \rangle \mid$
 $\langle P5 \ A \ ([^1_{(X,x)} \# \ xs) = (X = A) \rangle \mid$
 $\langle P5 \ A \ (x \# \ xs) = \text{False} \rangle$

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

fun $P5_sym :: \langle 'n \Rightarrow ('n, ('n, 't) \text{ bracket3}) \text{ syms} \Rightarrow \text{bool} \rangle$ **where**

$\langle P5_sym \ A \ [] = \text{False} \rangle \mid$
 $\langle P5_sym \ A \ (\text{Tm } [^1_{(X,x)} \# \ xs) = (X = A) \rangle \mid$
 $\langle P5_sym \ A \ ((Nt \ X) \# \ xs) = (X = A) \rangle \mid$
 $\langle P5_sym \ A \ (x \# \ xs) = \text{False} \rangle$

lemma $P5D[\text{dest}]$:

assumes $\langle P5 \ A \ x \rangle$
shows $\langle \exists \ y. \text{hd } x = [^1_{(A,y)} \rangle$

using *assms* **by**(*induction* A x *rule*: $P5.induct$) *auto*

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

lemma $P5_for_tm_if_P5_sym[dest]$: $\langle P5_sym\ A\ (map\ Tm\ x) \implies P5\ A\ x \rangle$
by(*induction* x) *auto*

3.6 $P6$ and $P7$

(*successively* $P6_sym$) w iff $Nt\ Y$ is directly preceded by some $Tm\ [^1_{A \rightarrow YC}$ or $Tm\ [^2_{A \rightarrow BY}$ in w :

fun $P6_sym :: \langle ('n, ('n, 't)\ bracket3)\ sym \Rightarrow ('n, ('n, 't)\ bracket3)\ sym \Rightarrow bool \rangle$
where
 $\langle P6_sym\ (Tm\ (b, (A, [Nt\ B, Nt\ C]), v))\ (Nt\ Y) = (b = True \wedge ((Y = B \wedge v = One) \vee (Y = C \wedge v = Two))) \rangle |$
 $\langle P6_sym\ x\ (Nt\ Y) = False \rangle |$
 $\langle P6_sym\ x\ y = True \rangle$

lemma $P6_symD[dest]$:
fixes $x :: \langle ('n, ('n, 't)\ bracket3)\ sym \rangle$
assumes $\langle P6_sym\ x\ (Nt\ Y) \rangle$
shows $\langle (\exists A\ C. x = Tm\ [^1_{(A, [Nt\ Y, Nt\ C])}] \vee (\exists A\ B. x = Tm\ [^2_{(A, [Nt\ B, Nt\ Y])}]) \rangle$
using *assms* **by**(*induction* x $\langle Nt\ Y :: ('n, ('n, 't)\ bracket3)\ sym \rangle$ *rule*: $P6_sym.induct$) *auto*

(*successively* $P7_sym$) w iff $Nt\ Y$ is directly followed by some $]^1_{A \rightarrow YC}$ or $]^2_{A \rightarrow BY}$ in w :

fun $P7_sym :: \langle ('n, ('n, 't)\ bracket3)\ sym \Rightarrow ('n, ('n, 't)\ bracket3)\ sym \Rightarrow bool \rangle$
where
 $\langle P7_sym\ (Nt\ Y)\ (Tm\ (b, (A, [Nt\ B, Nt\ C]), v)) = (b = False \wedge ((Y = B \wedge v = One) \vee (Y = C \wedge v = Two))) \rangle |$
 $\langle P7_sym\ (Nt\ Y)\ x = False \rangle |$
 $\langle P7_sym\ x\ y = True \rangle$

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

$\langle Reg\ A = \{u. P1\ u \wedge$
successively $P2\ u \wedge$
successively $P3\ u \wedge$
successively $P4\ u \wedge$
 $P5\ A\ u\} \rangle$

lemma $RegI[intro]:$

assumes $\langle P1\ u \rangle$
and $\langle \text{successively } P2\ u \rangle$
and $\langle \text{successively } P3\ u \rangle$
and $\langle \text{successively } P4\ u \rangle$
and $\langle P5\ A\ u \rangle$
shows $\langle u \in Reg\ A \rangle$
using *assms* **unfolding** Reg_def **by** *blast*

lemma $RegD[dest]:$

assumes $\langle u \in Reg\ A \rangle$
shows $\langle P1\ u \rangle$
and $\langle \text{successively } P2\ u \rangle$
and $\langle \text{successively } P3\ u \rangle$
and $\langle \text{successively } P4\ u \rangle$
and $\langle P5\ A\ u \rangle$
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 $P6$ and $P7$ that vanish for pure terminal strings:

definition $Reg_sym :: \langle 'n \Rightarrow ('n, ('n, 't) \text{ bracket3}) \text{ syms set} \rangle$ **where**

$\langle Reg_sym\ A = \{u. P1_sym\ u \wedge$
successively $P2_sym\ u \wedge$
successively $P3_sym\ u \wedge$
successively $P4_sym\ u \wedge$
 $P5_sym\ A\ u \wedge$
successively $P6_sym\ u \wedge$
successively $P7_sym\ u\} \rangle$

lemma $Reg_symI[intro]:$

assumes $\langle P1_sym\ u \rangle$
and $\langle \text{successively } P2_sym\ u \rangle$
and $\langle \text{successively } P3_sym\ u \rangle$
and $\langle \text{successively } P4_sym\ u \rangle$
and $\langle P5_sym\ A\ u \rangle$
and $\langle \text{successively } P6_sym\ u \rangle$
and $\langle \text{successively } P7_sym\ u \rangle$
shows $\langle u \in Reg_sym\ A \rangle$
using *assms* **unfolding** Reg_sym_def **by** *blast*

lemma $Reg_symD[dest]:$

assumes $\langle u \in Reg_sym\ A \rangle$

```

shows ⟨P1_sym u⟩
  and ⟨successively P2_sym u⟩
  and ⟨successively P3_sym u⟩
  and ⟨successively P4_sym u⟩
  and ⟨P5_sym A u⟩
  and ⟨successively P6_sym u⟩
  and ⟨successively P7_sym u⟩
using assms unfolding Reg_sym_def by blast+

```

```

lemma Reg_for_tm_if_Reg_sym[dest]: ⟨map Tm u ∈ Reg_sym A ⇒ u ∈ 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 $\{xs. \text{successively } Q \text{ } xs \wedge xs \in \text{brackets } P\}$

```

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]:  $\langle q \in \text{allStates} \implies \text{aut.nextl } q \text{ ys} \in \text{allStates} \rangle$ 
  using aut.nxt by(induction ys arbitrary: q) auto

lemma nextl_garbage[simp]:  $\langle \text{aut.nextl garbage } xs = \text{garbage} \rangle$ 
  by(induction xs) auto

lemma drop_right:  $\langle xs@ys \in \text{aut.language} \implies xs \in \text{aut.language} \rangle$ 
  proof(induction ys)
  case (Cons a ys)
  then have  $\langle xs @ [a] \in \text{aut.language} \rangle$ 
    using aut.language_def aut.nextl_app by fastforce
  then have  $\langle xs \in \text{aut.language} \rangle$ 
    using aut.language_def by force
  then show ?case by blast
qed auto

lemma state_after1[iff]:  $\langle (\text{succNext } q \ a \neq \text{garbage}) = (\text{succNext } q \ a = \text{letter } a) \rangle$ 
  by(induction q a rule: succNext.induct) (auto split: if_splits)

lemma state_after_in_P[intro]:  $\langle \text{succNext } q \ (br, p, v) \neq \text{garbage} \implies p \in P \rangle$ 
  by(induction q  $\langle (br, p, v) \rangle$  rule: succNext_induct) auto

lemma drop_left_general:  $\langle \text{aut.nextl start } ys = \text{garbage} \implies \text{aut.nextl } q \text{ ys} = \text{garbage} \rangle$ 
  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.prem in auto)
qed

lemma drop_left:  $\langle xs@ys \in \text{aut.language} \implies ys \in \text{aut.language} \rangle$ 
  unfolding aut.language_def
  by(induction xs arbitrary: ys) (auto dest: drop_left_general)

```

```

lemma empty_in_aut:  $\langle [] \in \text{aut.language} \rangle$ 
  unfolding aut.language_def by simp

lemma singleton_in_aut_iff:  $\langle [(br, p, v)] \in \text{aut.language} \longleftrightarrow p \in P \rangle$ 
  unfolding aut.language_def by simp

lemma duo_in_aut_iff:  $\langle [(br, p, v), (br', p', v')] \in \text{aut.language} \longleftrightarrow Q (br,p,v) (br',p',v') \wedge p \in P \wedge p' \in P \rangle$ 
  unfolding aut.language_def by auto

lemma trio_in_aut_iff:  $\langle (br, p, v) \# (br', p', v') \# zs \in \text{aut.language} \longleftrightarrow Q (br,p,v) (br',p',v') \wedge p \in P \wedge p' \in P \wedge (br',p',v') \# zs \in \text{aut.language} \rangle$ 
proof(standard, goal_cases)
  case 1
  with drop_left have *:  $\langle (br', p', v') \# zs \in \text{aut.language} \rangle$ 
    by (metis append_Cons append_Nil)
  from drop_right 1 have  $\langle [(br, p, v), (br', p', v')] \in \text{aut.language} \rangle$ 
    by simp
  with duo_in_aut_iff have **:  $\langle Q (br,p,v) (br',p',v') \wedge p \in P \wedge p' \in P \rangle$ 
    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:  $\langle (\text{successively } Q \text{ } xs \wedge xs \in \text{brackets}) \longleftrightarrow (xs \in \text{aut.language}) \rangle$ 
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:  $\langle x = (br, p, v) \rangle$ 
      by simp
    then show ?thesis
  proof(cases y)
    case (fields br' p' v')
    then have y_eq:  $\langle y = (br', p', v') \rangle$ 
      by simp
    have  $\langle (x \# y \# zs \in \text{aut.language}) \longleftrightarrow Q (br,p,v) (br',p',v') \wedge p \in P \wedge p' \in P \wedge (br',p',v') \# zs \in \text{aut.language} \rangle$ 

```

```

    unfolding x_eq y_eq using trio_in_out_iff by blast
    also have ⟨... ⟷ Q (br,p,v) (br',p',v') ∧ p ∈ P ∧ p' ∈ 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 P_2 , P_3 and P_4

```

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 $P1_State = last_ok \mid last_bad \mid garbage$

The good ending letters, are those that are not of the form ($Close_ , 1$).

fun *good* **where**

$\langle good \rangle_p^1 = False \mid$
 $\langle good (br, p, v) = True \rangle$

fun *next1* :: $\langle P1_State \Rightarrow ('n,'t) bracket3 \Rightarrow P1_State \rangle$ **where**

$\langle next1 garbage _ = garbage \rangle \mid$
 $\langle next1 last_ok (br, p, v) = (if\ p \notin P\ then\ garbage\ else\ (if\ good\ (br, p, v)\ then\ last_ok\ else\ last_bad)) \rangle \mid$
 $\langle next1 last_bad (br, p, v) = (if\ p \notin P\ then\ garbage\ else\ (if\ good\ (br, p, v)\ then\ last_ok\ else\ last_bad)) \rangle$

theorem *next1_induct*[*case_names garbage startp startnp letterQ letternQ*]:

fixes $R :: P1_State \Rightarrow ('n,'t) bracket3 \Rightarrow bool$

fixes $a0 :: P1_State$

and $a1 :: ('n,'t) bracket3$

assumes $\bigwedge u. R\ garbage\ u$

and $\bigwedge br\ p\ v. p \notin P \implies R\ last_ok\ (br, p, v)$

and $\bigwedge br\ p\ v. p \in P \wedge good\ (br, p, v) \implies R\ last_ok\ (br, p, v)$

and $\bigwedge br\ p\ v. p \in P \wedge \neg(good\ (br, p, v)) \implies R\ last_ok\ (br, p, v)$

and $\bigwedge br\ p\ v. p \notin P \implies R\ last_bad\ (br, p, v)$

and $\bigwedge br\ p\ v. p \in P \wedge good\ (br, p, v) \implies R\ last_bad\ (br, p, v)$

and $\bigwedge br\ p\ v. p \in P \wedge \neg(good\ (br, p, v)) \implies R\ last_bad\ (br, p, v)$

shows $R\ a0\ a1$

by (*metis (full_types) P1_State.exhaust assms prod_induct3*)

abbreviation *p1_aut* **where** $\langle p1_aut \equiv (dfa.states = \{last_ok, last_bad, garbage\},$

$init = last_ok,$

$final = \{last_ok\},$

$next = next1) \rangle$

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: next1_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) next1.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
  end
end

```

```

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 next2 :: ⟨P5_State ⇒ ('n,'t) bracket3 ⇒ P5_State⟩ where
  ⟨next2 garbage _ = garbage⟩ |
  ⟨next2 start (br, (X, r), v) = (if (X,r) ∉ P then garbage else (if ok (br, (X, r),
v) then first_ok else garbage))⟩ |
  ⟨next2 first_ok (br, p, v) = (if p ∉ P then garbage else first_ok)⟩

theorem next2_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},
  next = next2)⟩

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: next2_induct[of _ q x]) auto
next
  case 4
    then show ?case by simp
qed

corollary next2_start_ok_iff: ⟨ok x ∧ fst(snd x) ∈ P ⟷ next2 start x = first_ok⟩
by(auto elim!: next2.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: ⟨ $[x] \in p5\_aut.language \longleftrightarrow ok (hd [x]) \wedge [x] \in brackets$ ⟩
  unfolding p5_aut.language_def using next2_start_ok_iff by (cases x) fastforce

lemma singleton_first_ok_iff: ⟨ $p5\_aut.nextl\ start ([x]) = first\_ok \vee p5\_aut.nextl\ start ([x]) = garbage$ ⟩
by(cases x) (auto split: if_splits)

lemma first_ok_iff: ⟨ $xs \neq [] \implies p5\_aut.nextl\ start\ xs = first\_ok \vee p5\_aut.nextl\ start\ xs = garbage$ ⟩
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 ⟨ $p5\_aut.nextl\ start\ xs = first\_ok \vee p5\_aut.nextl\ start\ xs = garbage$ ⟩
    by blast
    then show ?thesis
    by(cases x) (auto split: if_splits)
  qed
qed

lemma lang_descr: ⟨ $xs \in p5\_aut.language \longleftrightarrow (xs \neq [] \wedge ok (hd xs) \wedge xs \in brackets)$ ⟩
proof(induction xs rule: rev_induct)
  case (snoc x xs)
  then have IH: ⟨ $xs \in p5\_aut.language = (xs \neq [] \wedge ok (hd xs) \wedge xs \in brackets)$ ⟩

  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 ⟨ $xs \in p5\_aut.language$ ⟩)
    case True
    then have ⟨ $xs \neq [] \wedge ok (hd xs) \wedge xs \in brackets$ ⟩

```

```

    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} ∩

```

$\{xs. \text{ successively } P2 \ xs \wedge xs \in \text{brackets}\} \cap$
 $\{xs. \text{ successively } P3 \ xs \wedge xs \in \text{brackets}\} \cap$
 $\{xs. \text{ successively } P4 \ xs \wedge xs \in \text{brackets}\} \cap$
 $\{xs. P5 \ A \ xs \wedge xs \in \text{brackets}\}$
 $= \text{brackets} \cap \text{Reg } A \text{ by auto}$

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*: $\langle \text{Dyck_lang } (P \times \{One, Two\}) \subseteq \text{brackets} \rangle$
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** $P :: ('n, 't) \text{Prods}$
 $+$
fixes $S :: 'n$
assumes *CNF_P*: $\langle \text{CNF } P \rangle$

begin

lemma *P_CNFE[dest]*:
assumes $\langle \pi \in P \rangle$
shows $\langle \exists A \ a \ B \ C. \pi = (A, [Nt \ B, Nt \ C]) \vee \pi = (A, [Tm \ a]) \rangle$
using *assms CNF_P* **unfolding** *CNF_def* **by fastforce**

definition *L* **where**
 $\langle L = \text{Lang } P \ S \rangle$

definition Γ **where**
 $\langle \Gamma = P \times \{One, Two\} \rangle$

definition P' **where**
 $\langle P' = \text{transform_prod } ' P \rangle$

definition L' **where**
 $\langle L' = \text{Lang } P' \ S \rangle$

6 Lemmas for $P' \vdash A \Rightarrow^* x \longleftrightarrow x \in R_A \cap \text{Dyck_lang } \Gamma$

lemma *prod1_snd_in_tm* [*intro, simp*]: $\langle (A, [Nt \ B, Nt \ C]) \in P \Rightarrow \text{snds_in_tm } \Gamma \ (\text{wrap2 } A \ B \ C) \rangle$

unfolding *snds_in_tm_def* **using** Γ_def **by** *auto*

lemma *prod2_snds_in_tm* [*intro, simp*]: $\langle (A, [Tm\ a]) \in P \implies snds_in_tm\ \Gamma\ (wrap1\ A\ a) \rangle$

unfolding *snds_in_tm_def* **using** Γ_def **by** *auto*

lemma *bal_tm_wrap1*[*iff*]: $\langle bal_tm\ (wrap1\ A\ a) \rangle$

unfolding *bal_tm_def* **by** (*simp add: bal_iff_bal_stk*)

lemma *bal_tm_wrap2*[*iff*]: $\langle bal_tm\ (wrap2\ A\ B\ C) \rangle$

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 occurring nonterminals. This will be needed for \rightarrow .

lemma *bal_tm_transform_rhs*[*intro!*]:

$\langle (A, \alpha) \in P \implies bal_tm\ (transform_rhs\ A\ \alpha) \rangle$

by *auto*

lemma *snds_in_tm_transform_rhs*[*intro!*]:

$\langle (A, \alpha) \in P \implies snds_in_tm\ \Gamma\ (transform_rhs\ A\ \alpha) \rangle$

using *P_CNFE* **by** (*fastforce*)

The lemma for \rightarrow

lemma *P'_imp_bal*:

assumes $\langle P' \vdash [Nt\ A] \Rightarrow * x \rangle$

shows $\langle bal_tm\ x \wedge snds_in_tm\ \Gamma\ x \rangle$

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 $\langle bal_tm\ (u\ @\ [Nt\ A]\ @\ v) \rangle$ **and** $\langle snds_in_tm\ \Gamma\ (u\ @\ [Nt\ A]\ @\ v) \rangle$

using *step.IH* *step.prem*s **by** *auto*

obtain *w'* **where** *w'_def*: $\langle w = transform_rhs\ A\ w' \rangle$ **and** *A_w'_in_P*: $\langle (A, w') \in P \rangle$

using *P'_def* *step.hyps*(2) **by** *force*

have *bal_tm_w*: $\langle bal_tm\ w \rangle$

using *bal_tm_transform_rhs*[*OF* $\langle (A, w') \in P \rangle$] *w'_def* **by** *auto*

then have $\langle bal_tm\ (u\ @\ w\ @\ v) \rangle$

using $\langle bal_tm\ (u\ @\ [Nt\ A]\ @\ v) \rangle$ **by** (*metis* *bal_tm_empty* *bal_tm_inside* *bal_tm_prepend_Nt*)

moreover have $\langle snds_in_tm\ \Gamma\ (u\ @\ w\ @\ v) \rangle$

using *snds_in_tm_transform_rhs*[*OF* $\langle (A, w') \in P \rangle$] $\langle snds_in_tm\ \Gamma\ (u\ @\ [Nt\ A]\ @\ v) \rangle$ *w'_def* **by** (*simp*)

ultimately show *?case* **using** $\langle bal_tm\ (u\ @\ w\ @\ v) \rangle$ **by** *blast*

qed

Another lemma for \rightarrow

```

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 im-
    age_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 P6_sym (u@w) ∧
    successively P7_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 P6_sym (ys @ [y] @ [Nt A] @ v)⟩
    using Reg_symD[OF uAv] snoc by auto
  then have ⟨P6_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 ⟨P6_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) ∧
      P6_sym (last (ys@[y])) (hd w) ∧
      P7_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 P6_sym (ys @ [y]) ∧
     successively P7_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 P6_sym (u@w) ∧
     successively P7_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 P6_sym (w@v) ∧
             successively P7_sym (w@v)⟩
proof(cases v)
  case Nil
  then show ?thesis using w_eq by auto
next
  case (Cons y ys)
  then have ⟨successively P7_sym ([Nt A] @ y # ys)⟩
    using Reg_symD[OF uAv] Cons using successively_append_iff by blast
  then have ⟨P7_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)) ∧
       P6_sym (last w) (hd (y#ys)) ∧
       P7_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 P6_sym (y # ys) ∧
    successively P7_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 P6_sym (w@v) ∧
    successively P7_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 succes-
sively_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 P6_sym (u@w@v) ∧
  successively P7_sym (u@w@v)⟩
  using w_eq by (metis (no_types, lifting) List.list.discI hd_append2 succes-
sively_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 ⟨ $\nexists$  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 ⟨ $\nexists$  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 \leftarrow .

lemma *transform_prod_one_step*:
assumes $\langle \pi \in P \rangle$
shows $\langle P' \vdash [Nt (fst \pi)] \Rightarrow snd (transform_prod \pi) \rangle$
proof –
obtain w' **where** $w'_def: \langle transform_prod \pi = (fst \pi, w') \rangle$
by (*metis* *fst_eqD* *transform_prod.simps* *surj_pair*)
then have $\langle (fst \pi, w') \in P' \rangle$
using *assms* **by** (*simp* *add: P'_def* *rev_image_eqI*)
then show *?thesis*
by (*simp* *add: w'_def* *derive_singleton*)
qed

The lemma for \leftarrow

lemma *Reg_and_dyck_imp_P'*:
assumes $\langle x \in (Reg A \cap Dyck_lang \Gamma) \rangle$
shows $\langle P' \vdash [Nt A] \Rightarrow * map Tm x \rangle$ **using** *assms*
proof (*induction* $\langle length (map Tm x) \rangle$ *arbitrary: A x* *rule: less_induct*)
case *less*
then have *IH*: $\langle \bigwedge w H. \llbracket length (map Tm w) < length (map Tm x); w \in Reg H \cap Dyck_lang (\Gamma) \rrbracket \Longrightarrow P' \vdash [Nt H] \Rightarrow * map Tm w \rangle$
using *less* **by** *simp*
have *xReg*: $\langle x \in Reg A \rangle$ **and** *xDL*: $\langle x \in Dyck_lang (\Gamma) \rangle$
using *less* **by** *blast+*

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

from *p5x* **obtain** πt **where** *hd_x*: $\langle hd x = [^1 \pi] \rangle$ **and** *pi_def*: $\langle \pi = (A, t) \rangle$
by (*metis* *List.list.sel(1)* *P5.elims(2)*)
with *xReg* **have** $\langle [^1 \pi \in set x \rangle$
by (*metis* *List.list.sel(1)* *List.list.set_intros(1)* *RegD(5)* *P5.elims(2)*)
then have *pi_in_P*: $\langle \pi \in P \rangle$
using *xDL* **unfolding** *Dyck_lang_def* *Gamma_def* **by** *fastforce*
have *bal_x*: $\langle bal x \rangle$
using *xDL* **by** *blast*
then have $\langle \exists y r. bal y \wedge bal r \wedge [^1 \pi \# tl x = [^1 \pi \# y @]^1 \pi \# r \rangle$
using *hd_x* *bal_x* *bal_Open_split*[*of* $\langle [^1 \pi \rangle \langle tl x \rangle$] *p5x*
by(*case_tac* *x*) *auto*
then obtain $y r1$ **where** $\langle [^1 \pi \# tl x = [^1 \pi \# y @]^1 \pi \# r1 \rangle$ **and** *bal_y*:
 $\langle bal y \rangle$ **and** *bal_r1*: $\langle bal r1 \rangle$

```

  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(simp add: 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 ap-
pend_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.  $\pi = (A, [Nt B, Nt C])$ ⟩ | (a) ⟨∃ a.  $\pi = (A, [Tm a])$ ⟩
  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: ⟨ $\pi = (A, [Nt B, Nt C])$ ⟩
    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 = ]1p⟩
  proof(rule ccontr)
    assume ⟨¬ (#p. last y = ]1p)⟩
    then obtain p where last_y: ⟨last y = ]1p ⟩
      by blast
    obtain butl where butl_def: ⟨y = butl @ [last y]⟩
      by (metis append_butlast_last_id y_not_empty)

    have ⟨successively P1' ([1π # y @ ]1π # [2π # z @ [ ]2π ])]⟩
      using p1x split3 by auto
    then have ⟨successively P1' ([1π # (butl@[last y]) @ ]1π # [2π # z @ [ ]2π ])]⟩
      using butl_def by simp
    then have ⟨successively P1' (([1π # butl) @ last y # [ ]1π) @ [2π # z @ [ ]2π ])]⟩
      by (metis (no_types, opaque_lifting) Cons_eq_appendI append_assoc append_self_conv2)
    then have ⟨P1' ]1p ]1π ⟩
      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_successively by blast
  qed

  ultimately have ⟨z ∈ Reg C⟩
  using z_successively by blast
  moreover have ⟨z ∈ 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 ∈ Reg C ∩ 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' ⊢ [Nt C] ⇒* map Tm z⟩
    using IH[of z C] split3 by blast
    have ⟨P' ⊢ [Nt A] ⇒* [ 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' ⊢ [ Tm [1 $\pi$  ] @ [(Nt B)] @ [Tm ]1 $\pi$  , Tm [2 $\pi$  ] @ [(Nt C)] @ [
Tm ]2 $\pi$  ] ⇒* [ 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' ⊢ [ Tm [1 $\pi$  ] @ map Tm y @ [Tm ]1 $\pi$  , Tm [2 $\pi$  ] @ [(Nt C)] @
[ Tm ]2 $\pi$  ] ⇒* [ 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' ⊢ [Nt A] ⇒* [ 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 [2 $\pi$  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 [1_π,      Tm ]1_π , Tm [2_π ,      Tm ]2_π  ]⟩
  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 $\langle roots\ ts \equiv map\ root\ ts \rangle$

```

fun wrap1_Sym :: ⟨'n ⇒ ('n,'t) sym ⇒ version ⇒ ('n,('n,'t) bracket3) ptree 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) ptree ⇒ ('n,('n,'t) bracket3) ptree 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) ptree ⇒ ('n,('n,'t) bracket3) ptree where
  ⟨transform_tree (Sym (Nt A)) = (Sym (Nt A))⟩ |
  ⟨transform_tree (Sym (Tm a)) = (Sym (Tm [1_(SOME A. True, [Tm a]))))⟩ |
  ⟨transform_tree (Prod A [Sym (Tm a)]) = Prod A ((wrap1_Sym A (Tm a)
One)@(wrap1_Sym A (Tm a) Two))⟩ |
  ⟨transform_tree (Prod A [t1, t2]) = Prod 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 (Prod A y) = (Prod A [])⟩

```

lemma *root_of_transform_tree*[*intro, simp*]: $\langle root\ t = Nt\ X \implies root\ (transform_tree\ t) = Nt\ X \rangle$
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 (Prod A ts)
    from Prod have pt: ⟨parse_tree P (Prod A ts)⟩ and fr: ⟨fringe (Prod A ts) =
w⟩
      by blast+
    from Prod have IH: ⟨ $\bigwedge x2a w'. \llbracket x2a \in \text{set } ts; \text{parse\_tree } P \ x2a \wedge \text{fringe } x2a = w \rrbracket \implies \text{parse\_tree } P' (\text{transform\_tree } x2a) \wedge \text{hs } (\text{fringe } (\text{transform\_tree } x2a)) = w'$ ⟩
      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

```

```

 $[^1(A, [Tm\ a]), Tm\ ]^1(A, [Tm\ a]), Tm\ ]^2(A, [Tm\ a]), Tm\ ]^2(A, [Tm\ a]) ]\rangle$ 
  by fastforce
  then obtain t1 t2 e1 where ei_def:  $\langle ts = [e1] \vee ts = [t1, t2] \rangle$ 
    by blast
  then consider (Tm)  $\langle roots\ ts = [Tm\ a] \quad \wedge\ ts = [Sym\ (Tm\ a)] \rangle |$ 
    (Nt_Nt)  $\langle roots\ ts = [Nt\ B, Nt\ C] \wedge ts = [t1, t2] \rangle$ 
    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:  $\langle ts = [Sym\ (Tm\ a)] \rangle$  and roots:  $\langle roots\ ts = [Tm\ a] \rangle$ 
        by blast+
      then have  $\langle transform\_tree\ (Prod\ A\ ts) = Prod\ A\ [Sym\ (Tm\ ]^1(A, [Tm\ a]),$ 
Sym(Tm ]^1(A, [Tm a]), Sym (Tm ]^2(A, [Tm a]), Sym(Tm ]^2(A, [Tm a]) ]\rangle
        by simp
      then have  $\langle hs\ (fringe\ (transform\_tree\ (Prod\ A\ ts))) = [Tm\ a] \rangle$ 
        by simp
      also have  $\langle \dots = w \rangle$ 
        using fr unfolding ts_eq by auto
      finally have  $\langle hs\ (fringe\ (transform\_tree\ (Prod\ A\ ts))) = w \rangle .$ 
      moreover have  $\langle parse\_tree\ (P')\ (transform\_tree\ (Prod\ A\ [Sym\ (Tm\ a)]) \rangle$ 
        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:  $\langle ts = [t1, t2] \rangle$  and roots:  $\langle roots\ ts = [Nt\ B, Nt\ C] \rangle$ 
          by blast+
        then have root_t1_eq_B:  $\langle root\ t1 = Nt\ B \rangle$  and root_t2_eq_C:  $\langle root\ t2 =$ 
Nt C
          by blast+
        then have  $\langle transform\_tree\ (Prod\ A\ ts) = Prod\ A\ ((wrap2\_Sym\ A\ (Nt\ B)\ (Nt\ C)$ 
One (transform_tree t1)) @ (wrap2_Sym A (Nt B) (Nt C) Two (transform_tree
t2))) \rangle
          by (simp add: ts_eq)
        then have  $\langle hs\ (fringe\ (transform\_tree\ (Prod\ A\ ts))) = hs\ (fringe\ (transform\_tree$ 
t1)) @ hs (fringe (transform_tree t2)) \rangle
          by auto
        also have  $\langle \dots = fringe\ t1\ @\ fringe\ t2 \rangle$ 
          using IH pt ts_eq by force
        also have  $\langle \dots = fringe\ (Prod\ A\ ts) \rangle$ 
          using ts_eq by simp
        also have  $\langle \dots = w \rangle$ 
          using fr by blast
        ultimately have  $\langle hs\ (fringe\ (transform\_tree\ (Prod\ A\ ts))) = w \rangle$ 
          by blast

  have  $\langle parse\_tree\ P\ t1 \rangle$  and  $\langle parse\_tree\ P\ t2 \rangle$ 
    using pt ts_eq by auto

```

then have $\langle \text{parse_tree } P' (\text{transform_tree } t1) \rangle$ **and** $\langle \text{parse_tree } P' (\text{transform_tree } t2) \rangle$
by (*simp add: IH ts_eq*)
have $\text{root1: } \langle \text{map Parse_Tree.root (wrap2_Sym } A (Nt B) (Nt C) \text{ version.One } (\text{transform_tree } t1)) = [Tm]^1(A, [Nt B, Nt C]), Nt B, Tm]^1(A, [Nt B, Nt C]) \rangle$
using *root_t1_eq_B* **by** *auto*
moreover have $\text{root2: } \langle \text{map Parse_Tree.root (wrap2_Sym } A (Nt B) (Nt C) \text{ Two } (\text{transform_tree } t2)) = [Tm]^2(A, [Nt B, Nt C]), Nt C, Tm]^2(A, [Nt B, Nt C]) \rangle$
using *root_t2_eq_C* **by** *auto*
ultimately have $\langle \text{parse_tree } P' (\text{transform_tree } (\text{Prod } A \text{ ts})) \rangle$
using $\langle \text{parse_tree } P' (\text{transform_tree } t1) \rangle$ $\langle \text{parse_tree } P' (\text{transform_tree } t2) \rangle$
 $\langle (A, \text{map Parse_Tree.root } ts) \in P \rangle$ *roots*
by (*force simp: ts_eq P'_def*)
then show *?thesis*
using $\langle \text{hs } (\text{fringe } (\text{transform_tree } (\text{Prod } A \text{ ts}))) = w \rangle$ **by** *auto*
qed
qed

lemma

transfer_parse_tree:

assumes $\langle w \in \text{Ders } P \text{ } S \rangle$

shows $\langle \exists w' \in \text{Ders } P' \text{ } S. w = \text{hs } w' \rangle$

proof–

from *assms* **obtain** t **where** $t_def: \langle \text{parse_tree } P \text{ } t \wedge \text{fringe } t = w \wedge \text{root } t = Nt \text{ } S \rangle$

using *parse_tree_if_derives DersD* **by** *meson*

then have $\text{root_tr: } \langle \text{root } (\text{transform_tree } t) = Nt \text{ } S \rangle$

by *blast*

from t_def **have** $\langle \text{parse_tree } P' (\text{transform_tree } t) \wedge \text{hs } (\text{fringe } (\text{transform_tree } t)) = w \rangle$

using *transform_tree_correct* *assms* **by** *blast*

with root_tr **have** $\langle \text{fringe } (\text{transform_tree } t) \in \text{Ders } P' \text{ } S \wedge w = \text{hs } (\text{fringe } (\text{transform_tree } t)) \rangle$

using *derives_fringe_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 $\langle w \in \text{Lang } P \text{ } S \rangle$

shows $\langle \exists w' \in L'. w = h \text{ } w' \rangle$

proof–

have $ex: \langle \exists w' \in \text{Ders } P' \text{ } S. (\text{map } Tm \text{ } w) = \text{hs } w' \rangle$

using *transfer_parse_tree* **by** (*meson Lang_Ders assms imageI subsetD*)

then obtain w' **where** $w'_def: \langle w' \in \text{Ders } P' \text{ } S \rangle$ $\langle (\text{map } Tm \text{ } w) = \text{hs } w' \rangle$

using ex **by** *blast*

moreover obtain w'' **where** $\langle w' = \text{map } Tm \text{ } w'' \rangle$

```

    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 rtrancl_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  $\pi\_def \langle \pi \in P \rangle$  derive.intros by (metis Transitive_Closure.rtranclp.rtrancl_into_rtrancl)
have  $\langle hs \ x' = hs \ (snd \ (transform\_prod \ \pi)) \rangle$ 
  by (simp add: transf_\pi_def)
also have  $\langle \dots = snd \ \pi \rangle$ 
  using hom_ext_inv  $\langle \pi \in P \rangle$  by blast
also have  $\langle \dots = x \rangle$ 
  by (simp add: \pi_def)
finally have  $\langle hs \ x' = x \rangle$ 
  by simp
with  $\langle P \vdash [Nt \ S] \Rightarrow^* hs \ u \ @ \ x \ @ \ hs \ v \rangle$  have  $\langle P \vdash [Nt \ S] \Rightarrow^* hs \ u \ @ \ hs \ x' \ @ \ hs \ v \rangle$ 
  by simp
then show ?case by auto
qed
then show  $\langle h \ w' \in Lang \ P \ S \rangle$ 
  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*:

```

 $\langle regular \ (brackets \cap Reg \ S) \rangle$ 
 $\wedge L = h \ ' \ ((brackets \cap Reg \ S) \cap Dyck\_lang \ \Gamma)$ 
 $\wedge hom\_list \ (h \ :: \ ('n, 't) \ bracket3 \ list \Rightarrow \ 't \ list)$ 
proof –
have  $\langle \forall A. \forall x. P' \vdash [Nt \ A] \Rightarrow^* (map \ Tm \ x) \longleftrightarrow x \in Dyck\_lang \ \Gamma \cap Reg \ A \rangle$ 
proof –
have  $\langle \forall A. \forall x. P' \vdash [Nt \ A] \Rightarrow^* (map \ Tm \ x) \longrightarrow x \in Dyck\_lang \ \Gamma \cap Reg \ A \rangle$ 
  using P'_imp_Reg P'_imp_bal Dyck_langI_tm by blast
moreover have  $\langle \forall A. \forall x. x \in Dyck\_lang \ \Gamma \cap Reg \ A \longrightarrow P' \vdash [Nt \ A] \Rightarrow^* (map \ Tm \ x) \rangle$ 
  using Reg_and_dyck_imp_P' by blast
  ultimately show ?thesis by blast
qed
then have  $\langle L' = Dyck\_lang \ \Gamma \cap (Reg \ S) \rangle$ 
  by (auto simp add: Lang_def L'_def)
then have  $\langle h \ ' \ (Dyck\_lang \ \Gamma \cap Reg \ S) = h \ ' \ L' \rangle$ 
  by simp
also have  $\langle \dots = Lang \ P \ S \rangle$ 
proof(standard)
  show  $\langle h \ ' \ L' \subseteq Lang \ P \ S \rangle$ 
    using L'_imp_h_P by blast
next
  show  $\langle Lang \ P \ S \subseteq h \ ' \ L' \rangle$ 
    using P_imp_h_L' by blast
qed
also have  $\langle \dots = L \rangle$ 

```

```

  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 :: fresh0, '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)⟩
  from cnf_exists obtain ps' where
    ⟨CNF ps'⟩ ⟨finite ps'⟩ and lang_ps_eq_lang_ps': ⟨Lang ps' S = Lang P S -
{}⟩
  using finiteP by blast

  interpret Chomsky_Schuetzenberger_locale ⟨ps'⟩ S
  apply unfold_locales
  using ⟨finite ps'⟩ ⟨CNF ps'⟩ by auto
  have ⟨regular (brackets ∩ Reg S) ∧ Lang 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 ps' S = L - {[]}⟩
  unfolding lang_ps_eq_lang_ps' using L_def by (simp add: asms(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 :: fresh0, 't) Prods⟩ and S :: 'n
  defines ⟨L ≡ Lang P S⟩
  assumes finite: ⟨finite P⟩
  shows ⟨∃(R::('n,'t) bracket3 list set) h  $\Gamma$ . regular R  $\wedge$  L = h ‘ (R  $\cap$  Dyck_lang  $\Gamma$ )
 $\wedge$  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  $\Gamma$  where
        reg_R: ⟨(regular R)⟩ and L_minus_eq: ⟨L - {[]} = h ‘ (R  $\cap$  Dyck_lang  $\Gamma$ )⟩
      and hom_h: ⟨hom_list h⟩
      by (metis L_def Chomsky_Schuetzenberger_not_empty finite)
      then have reg_R_union: ⟨regular(R  $\cup$  {[]})⟩
      by (meson regular_Un regular_nullstr)
      have ⟨[] = h([])⟩
      by (simp add: hom_h hom_list_Nil)
      moreover have ⟨[] ∈ Dyck_lang  $\Gamma$ ⟩
      by auto
      ultimately have ⟨[] ∈ h ‘ ((R  $\cup$  {[]})  $\cap$  Dyck_lang  $\Gamma$ )⟩
      by blast
      with True L_minus_eq have ⟨L = h ‘ ((R  $\cup$  {[]})  $\cap$  Dyck_lang  $\Gamma$ )⟩
      using ⟨[] ∈ Dyck_lang  $\Gamma$ ⟩ ⟨[] = 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.