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* Γ), 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* Γ and $z \in R_C \cap$ *Dyck_lang* Γ. 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* Γ 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* Γ, where *brackets* Γ ($\supseteq$ *Dyck_lang* Γ) 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* Γ) 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**

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

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

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

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

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

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

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

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

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* :: ‹$'n \Rightarrow 't \Rightarrow ('n,\ ('n,'t)\ \mathit{bracket3})\ \mathit{syms}$› **where**
‹*wrap1* $A\ a \equiv$
$[\ \mathit{Tm}\ [^1(A,\ [\mathit{Tm}\ a]),$
$\mathit{Tm}\ ]^1(A,\ [\mathit{Tm}\ a]),$
$\mathit{Tm}\ [^2(A,\ [\mathit{Tm}\ a]),$
$\mathit{Tm}\ ]^2(A,\ [\mathit{Tm}\ a])\ ]$›

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

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

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* [] = []
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *the_hom_syms_keep_var*: ‹hs [(*Nt A*)] = [*Nt A*]›
⟨*proof*⟩

lemma *the_hom_syms_tms_inj*: ‹hs w = *map Tm m* $\Longrightarrow \exists\, w'.\; w$ = *map Tm* w'›

⟨*proof*⟩

Helper for showing the upcoming lemma:

lemma *helper*: ‹*the_hom_sym* (*Tm x*) = *map Tm* (*the_hom1 x*)›
⟨*proof*⟩

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*⟩

lemma *the_hom1_strip*: ‹(*the_hom_sym* x') = *map Tm w* $\Longrightarrow$ *the_hom1* (*destTm* x') = *w*›
⟨*proof*⟩

lemma *the_hom1_strip2*: ‹*concat* (*map the_hom_sym* w') = *map Tm w* $\Longrightarrow$ *concat* (*map* (*the_hom1* ∘ *destTm*) w') = *w*›
⟨*proof*⟩

lemma *h_eq_h_ext2*:
assumes ‹hs w' = (*map Tm w*)›
shows ‹h (*map destTm* w') = *w*›
⟨*proof*⟩

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* $\Rightarrow$ $('n,'t)$ *bracket3* $\Rightarrow$ *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* $\Rightarrow$ $('n, ('n,'t)$ *bracket3*$)$ *sym* $\Rightarrow$ *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*]:
‹$P1'$ $]^1{}_p$ $r \longleftrightarrow r = [^2{}_p$›
⟨*proof*⟩

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

fun *P1* :: $('n, 't)$ *bracket3 list* $\Rightarrow$ *bool* **where**
‹*P1 xs* $=$ ((*successively* $P1'$ *xs*) $\wedge$ (*if xs* $\neq$ [] *then* ($\nexists p$. *last xs* $=]^1{}_p$) *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*) $\wedge$ (*if xs* $\neq$ [] *then* ($\nexists p$. *last xs* $=$ *Tm* $]^1{}_p$) *else True*))›

lemma *P1_for_tm_if_P1_sym*[*dest*!]: ‹*P1_sym* (*map Tm x*) $\Longrightarrow$ *P1 x*›
⟨*proof*⟩

lemma *P1I*[*intro*]:
assumes ‹*successively* $P1'$ *xs*›
and ‹$\nexists p$. *last xs* $=]^1{}_p$›
shows ‹*P1 xs*›
⟨*proof*⟩

lemma *P1_symI*[*intro*]:
assumes ‹*successively* $P1'$_*sym xs*›
and ‹$\nexists p$. *last xs* $=$ *Tm* $]^1{}_p$›
shows ‹*P1_sym xs*›
⟨*proof*⟩

lemma *P1_symD*[*dest*]: ‹*P1_sym xs* $\Longrightarrow$ *successively P1'_sym xs*› ⟨*proof*⟩

lemma *P1D_not_empty*[*intro*]:
assumes ‹$xs \neq []$›
and ‹*P1 xs*›
shows ‹*last* $xs \neq]^1{}_p$›
⟨*proof*⟩

lemma *P1_symD_not_empty'*[*intro*]:
assumes ‹$xs \neq []$›
and ‹*P1_sym xs*›
shows ‹*last* $xs \neq Tm\ ]^1{}_p$›
⟨*proof*⟩

lemma *P1_symD_not_empty*:
assumes ‹$xs \neq []$›
and ‹*P1_sym xs*›
shows ‹$\nexists p.$ *last* $xs = Tm\ ]^1{}_p$›
⟨*proof*⟩

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*) $\Longrightarrow$ *successively P2 x*›
⟨*proof*⟩

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* $[^1{}_{(A,\ [Nt\ B,\ Nt\ C])}$ $(p, ((X,y), t)) = (p = True \wedge t = One \wedge X = B)$› |
‹*P3* $[^2{}_{(A,\ [Nt\ B,\ Nt\ C])}$ $(p, ((X,y), t)) = (p = True \wedge t = One \wedge X = C)$› |
‹*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* = $[^1_{(B,\ l)}$›
⟨*proof*⟩

lemma *P3D2*[*dest*]:
fixes *r*::‹('n,'t) bracket3›
assumes ‹*P3* $[^2_{(A,\ [Nt\ B,\ Nt\ C])}$ *r*›
shows ‹∃ *l*. *r* = $[^1_{(C,\ l)}$›
⟨*proof*⟩

lemma *P3_for_tm_if_P3_sym*[*dest*]: ‹*successively P3_sym* (*map Tm x*) ⟹ *successively P3 x*›
⟨*proof*⟩

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*)›
  ⟨*proof*⟩

lemma *P4_for_tm_if_P4_sym*[*dest*]: ‹*successively P4_sym* (*map Tm x*) $\Longrightarrow$ *successively P4 x*›
⟨*proof*⟩

3.5 *P5*

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

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 $[^1{}_{A \rightarrow y}$, 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* = $[^1{}_{(A,y)}$›
  ⟨*proof*⟩

lemma *P5_symD*[*dest*]:
  assumes ‹*P5_sym A x*›
  shows ‹($\exists$ *y*. *hd x* = *Tm* $[^1{}_{(A,y)}$) $\vee$ *hd x* = *Nt A*›
  ⟨*proof*⟩

lemma *P5_for_tm_if_P5_sym*[*dest*]: ‹*P5_sym A* (*map Tm x*) $\Longrightarrow$ *P5 A x*›
  ⟨*proof*⟩

3.6 *P7* and *P8*

(*successively P7_sym*) *w* iff *Nt Y* is directly preceded by some *Tm* $[^1{}_{A \rightarrow YC}$ or *Tm* $[^2{}_{A \rightarrow BY}$ 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* $\wedge$ ((*Y* = *B* $\wedge$ *v* = *One*) $\vee$ (*Y*=*C* $\wedge$ *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 ‹$(\exists\, A\ C.\ x = Tm\ [^1{}_{(A,[Nt\ Y,\ Nt\ C])}) \vee (\exists\, A\ B.\ x = Tm\ [^2{}_{(A,[Nt\ B,\ Nt\ Y])})$›
⟨$proof$⟩

($successively$ $P8\_sym$) 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 ‹$(\exists\, A\ C.\ x = Tm\ ]^1{}_{(A,[Nt\ Y,\ Nt\ C])}) \vee (\exists\, A\ B.\ x = Tm\ ]^2{}_{(A,[Nt\ B,\ Nt\ Y])})$›
⟨$proof$⟩

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›
⟨$proof$⟩

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*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *Reg_symD*[*dest*]:
assumes ‹*x* ∈ *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*)›
⟨*proof*⟩

lemma *Reg_for_tm_if_Reg_sym*[*dest*]: ‹(*map Tm x*) ∈ *Reg_sym A* ⟹ *x* ∈ *Reg A*›
⟨*proof*⟩

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 $'a\ state = start \mid garbage \mid\ letter\ 'a$

definition *allStates* :: ‹$('n,'t)\ bracket3\ state\ set$ ›**where**
‹$allStates = \{\ letter\ (br,(p,v)) \mid br\ p\ v.\ p \in P\ \} \cup \{start,\ garbage\}$›

lemma *allStatesI*: ‹$p \in P \Longrightarrow letter\ (br,(p,v)) \in allStates$›
⟨*proof*⟩

lemma *start_in_allStates*[*simp*]: ‹$start \in allStates$›
⟨*proof*⟩

lemma *garbage_in_allStates*[*simp*]: ‹$garbage \in allStates$›
⟨*proof*⟩

lemma *finite_allStates_if*:
shows ‹$finite(\ allStates)$›
⟨*proof*⟩

end

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

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

fun *succNext* :: ‹$('n,'t)\ bracket3\ state \Rightarrow ('n,'t)\ bracket3 \Rightarrow ('n,'t)\ bracket3\ state$›
where
‹$succNext\ garbage\ \_ = garbage$› |
‹$succNext\ start\ (br',\ p',\ v') = (if\ p' \in P\ then\ letter\ (br',\ p',v')\ else\ garbage\ )$› |
‹$succNext\ (letter\ (br,\ p,\ v))\ (br',\ p',\ v') = \ (if\ Q\ (br,p,v)\ (br',p',v') \wedge p \in P \wedge p' \in P\ then\ letter\ (br',p',v')\ else\ garbage)$›

theorem *succNext_induct*[*case_names garbage startp startnp letterQ letternQ*]:
fixes $R :: ('n,'t)\ bracket3\ state \Rightarrow ('n,'t)\ bracket3 \Rightarrow bool$
fixes $a0 :: ('n,'t)\ bracket3\ state$
and $a1 :: ('n,'t)\ bracket3$
assumes $\bigwedge u.\ R\ garbage\ u$
and $\bigwedge br'\ p'\ v'.\ p' \in P \Longrightarrow R\ state.start\ (br',\ p',\ v')$
and $\bigwedge br'\ p'\ v'.\ p' \notin P \Longrightarrow R\ state.start\ (br',\ p',\ v')$
and $\bigwedge br\ p\ v\ br'\ p'\ v'.\ Q\ (br,p,v)\ (br',p',v') \wedge p \in P \wedge p' \in P \Longrightarrow R\ (letter\ (br,\ p,\ v))\ (br',\ p',\ v')$
and $\bigwedge br\ p\ v\ br'\ p'\ v'.\ \neg(Q\ (br,p,v)\ (br',p',v') \wedge p \in P \wedge p' \in P) \Longrightarrow R\ (letter\ (br,\ p,\ v))\ (br',\ p',\ v')$
shows $R\ a0\ a1$
⟨*proof*⟩

abbreviation *aut* **where** ‹$aut \equiv$ ⦇$dfa.states = allStates$,
$init = start$,
$final = (allStates - \{garbage\})$,
$nxt = succNext$ ⦈›

interpretation *aut* : *dfa aut*
⟨*proof*⟩

lemma *nextl_in_allStates*[*intro*,*simp*]: ‹$q \in allStates \Longrightarrow aut.nextl\ q\ ys \in allStates$›
⟨*proof*⟩

lemma *nextl_garbage*[*simp*]: ‹$aut.nextl\ garbage\ xs = garbage$›
⟨*proof*⟩

lemma *drop_right*: ‹$xs@ys \in aut.language \Longrightarrow xs \in aut.language$›
⟨*proof*⟩

lemma *state_after1*[*iff*]: ‹$(succNext\ q\ a \neq garbage) = (succNext\ q\ a = letter\ a)$›
⟨*proof*⟩

lemma *state_after_in_P*[*intro*]: ‹$succNext\ q\ (br,\ p,\ v) \neq garbage \Longrightarrow p \in P$›
⟨*proof*⟩

lemma *drop_left_general*: ‹$aut.nextl\ start\ ys = garbage \Longrightarrow aut.nextl\ q\ ys = garbage$›
⟨*proof*⟩

lemma *drop_left*: ‹$xs@ys \in aut.language \Longrightarrow ys \in aut.language$›
⟨*proof*⟩

lemma *empty_in_aut*: ‹$[] \in aut.language$›
⟨*proof*⟩

lemma *singleton_in_aut_iff*: ‹$[(br,\ p,\ v)] \in aut.language \longleftrightarrow p \in P$›
⟨*proof*⟩

lemma *duo_in_aut_iff*: ‹$[(br,\ p,\ v),\ (br',\ p',\ v')] \in aut.language \longleftrightarrow Q\ (br,p,v)\ (br',p',v') \wedge p \in P \wedge p' \in P$›
⟨*proof*⟩

lemma *trio_in_aut_iff*: ‹$(br,\ p,\ v)\ \#\ (br',\ p',\ v')\ \#\ zs \in aut.language \longleftrightarrow Q\ (br,p,v)\ (br',p',v') \wedge p \in P \wedge p' \in P \wedge (br',p',v')\ \#\ zs \in aut.language$›
⟨*proof*⟩

lemma *aut_lang_iff_succ_Q*: ‹$(successively\ Q\ xs \wedge xs \in brackets) \longleftrightarrow (xs \in aut.language)$›
⟨*proof*⟩

corollary *regular_successively_inter_brackets*: ‹*regular* {*xs*. *successively* *Q* *xs* ∧ *xs* ∈ *brackets*}›
⟨*proof*⟩

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*⟩

lemma *P3_regular*:
‹*regular* {*xs*. *successively* *P3* *xs* ∧ *xs* ∈ *brackets*} ›
⟨*proof*⟩

lemma *P4_regular*:
‹*regular* {*xs*. *successively* *P4* *xs* ∧ *xs* ∈ *brackets* }›
⟨*proof*⟩

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 $\bigwedge br\ p\ v.\ p \in P \wedge \neg(good\ (br,\ p,\ v)) \Longrightarrow R\ last\_ok\ (br,\ p,\ v)$
and $\bigwedge br\ p\ v.\ p \notin P \Longrightarrow R\ last\_bad\ (br,\ p,\ v)$
and $\bigwedge br\ p\ v.\ p \in P \wedge good\ (br,\ p,\ v) \Longrightarrow R\ last\_bad\ (br,\ p,\ v)$
and $\bigwedge br\ p\ v.\ p \in P \wedge \neg(good\ (br,\ p,\ v)) \Longrightarrow R\ last\_bad\ (br,\ p,\ v)$
shows $R\ a0\ a1$
⟨*proof*⟩

abbreviation *p1_aut* **where** ‹$p1\_aut \equiv (\!|dfa.states = \{last\_ok,\ last\_bad,\ garbage\},$
$init\ = last\_ok,$
$final = \{last\_ok\},$
$nxt\ \ = nxt1|\!)$›

interpretation *p1_aut* : *dfa p1_aut*
⟨*proof*⟩

lemma *nextl_garbage_iff*[*simp*]: ‹$p1\_aut.nextl\ last\_ok\ xs = garbage \longleftrightarrow xs \notin brackets$›
⟨*proof*⟩

lemma *lang_descr_full*:
‹$(p1\_aut.nextl\ last\_ok\ xs = last\_ok \longleftrightarrow (xs = [] \vee (xs \neq [] \wedge good\ (last\ xs) \wedge xs \in brackets))) \wedge$
$(p1\_aut.nextl\ last\_ok\ xs = last\_bad \longleftrightarrow ((xs \neq [] \wedge \neg good\ (last\ xs) \wedge xs \in brackets)))$›
⟨*proof*⟩

lemma *lang_descr*: ‹$xs \in p1\_aut.language \longleftrightarrow (xs = [] \vee (xs \neq [] \wedge good\ (last\ xs) \wedge xs \in brackets))$›
⟨*proof*⟩

lemma *good_iff*[*simp*]:‹$(\forall\ a\ b.\ last\ xs \neq]^1{}_{(a,\ b)}) = good\ (last\ xs)$›
⟨*proof*⟩

lemma *in_P1_iff*: ‹$(P1\ xs \wedge xs \in brackets) \longleftrightarrow (xs = [] \vee (xs \neq [] \wedge good\ (last\ xs) \wedge xs \in brackets)) \wedge successively\ P1'\ xs \wedge xs \in brackets$›
⟨*proof*⟩

corollary *P1_eq*: ‹$\{xs.\ P1\ xs \wedge xs \in brackets\} =$
$\{xs.\ successively\ P1'\ xs \wedge xs \in brackets\} \cap \{xs.\ xs = [] \vee (xs \neq [] \wedge good\ (last\ xs) \wedge xs \in brackets)\}$›
⟨*proof*⟩

lemma *P1'_regular*:
shows ‹$regular\ \{xs.\ successively\ P1'\ xs \wedge xs \in brackets\}$ ›
⟨*proof*⟩

corollary *aux_regular*: ‹$regular\ \{xs.\ xs = [] \vee (xs \neq [] \wedge good\ (last\ xs) \wedge xs \in brackets)\}$›
⟨*proof*⟩

corollary *regular_P1*: ‹*regular* $\{xs.\ P1\ xs \wedge xs \in brackets\}$›
  ⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

corollary *nxt2_start_ok_iff*: ‹*ok x* ∧ *fst*(*snd x*) ∈ *P* ⟷ *nxt2 start x* = *first_ok*›

⟨*proof*⟩

lemma *empty_not_in_lang*[*simp*]:‹[] ∉ *p5_aut.language*›
⟨*proof*⟩

lemma *singleton_in_lang_iff*: ‹[*x*] ∈ *p5_aut.language* ⟷ *ok* (*hd* [*x*]) ∧ [*x*] ∈ *brackets*›
⟨*proof*⟩

lemma *singleton_first_ok_iff*: ‹*p5_aut.nextl start* ([*x*]) = *first_ok* ∨ *p5_aut.nextl start* ([*x*]) = *garbage*›
⟨*proof*⟩

lemma *first_ok_iff*: ‹*xs*≠ [] ⟹ *p5_aut.nextl start xs* = *first_ok* ∨ *p5_aut.nextl start xs* = *garbage*›
⟨*proof*⟩

lemma *lang_descr*: ‹*xs* ∈ *p5_aut.language* ⟷ (*xs* ≠ [] ∧ *ok* (*hd xs*) ∧ *xs* ∈ *brackets*)›
⟨*proof*⟩

lemma *in_P5_iff*: ‹*P5 A xs* ∧ *xs* ∈ *brackets* ⟷ (*xs* ≠ [] ∧ *ok* (*hd xs*) ∧ *xs* ∈ *brackets*)›
⟨*proof*⟩

corollary *aux_regular*: ‹*regular* {*xs*. *xs* ≠ [] ∧ *ok* (*hd xs*) ∧ *xs* ∈ *brackets*}›
⟨*proof*⟩

lemma *regular_P5*:‹*regular* {*xs*. *P5 A xs* ∧ *xs* ∈ *brackets*}›
⟨*proof*⟩

end

context *locale_P*
begin

corollary *regular_Reg_inter*: ‹*regular* (*brackets* ∩ *Reg A*)›
⟨*proof*⟩

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*›
⟨*proof*⟩

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 ‹$\pi \in P$›
  shows ‹$\exists A\ a\ B\ C.\ \pi = (A, [Nt\ B, Nt\ C]) \lor \pi = (A, [Tm\ a])$›
  ⟨*proof*⟩

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

definition Γ **where**
  ‹$\Gamma = P \times \{One, Two\}$›

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*]: ‹$(A, [Nt\ B, Nt\ C]) \in P \Longrightarrow snds\_in\_tm\ \Gamma\ (wrap2\ A\ B\ C)$›
  ⟨*proof*⟩

lemma *prod2_snds_in_tm* [*intro*, *simp*]: ‹$(A, [Tm\ a]) \in P \Longrightarrow snds\_in\_tm\ \Gamma\ (wrap1\ A\ a)$›
  ⟨*proof*⟩

lemma *bal_tm_wrap1*[*iff*]: ‹$bal\_tm\ (wrap1\ A\ a)$›
⟨*proof*⟩

lemma *bal_tm_wrap2*[*iff*]: ‹$bal\_tm\ (wrap2\ A\ B\ C)$›
⟨*proof*⟩

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 $\rightarrow$.

lemma *bal_tm_transform_rhs*[*intro*!]:
  ‹$(A,\alpha) \in P \Longrightarrow bal\_tm\ (transform\_rhs\ A\ \alpha)$›

⟨*proof*⟩

lemma *snds_in_tm_transform_rhs*[*intro*!]:
‹$(A,\alpha) \in P \Longrightarrow$ *snds_in_tm* Γ (*transform_rhs* A α)›
⟨*proof*⟩

The lemma for $\rightarrow$

lemma *P′_imp_bal*:
assumes ‹$P' \vdash [Nt\ A] \Rightarrow\!*\ x$›
shows ‹*bal_tm* x $\wedge$ *snds_in_tm* Γ x›
⟨*proof*⟩

Another lemma for $\rightarrow$

lemma *P′_imp_Reg*:
assumes ‹$P' \vdash [Nt\ T] \Rightarrow\!*\ x$›
shows ‹$x \in$ *Reg_sym* T›
⟨*proof*⟩

This will be needed for the direction $\leftarrow$.

lemma *transform_prod_one_step*:
assumes ‹$\pi \in P$›
shows ‹$P' \vdash [Nt\ (fst\ \pi)] \Rightarrow$ *snd* (*transform_prod* π)›
⟨*proof*⟩

The lemma for $\leftarrow$

lemma *Reg_and_dyck_imp_P′*:
assumes ‹$x \in$ (*Reg* A $\cap$ *Dyck_lang* Γ)›
shows ‹$P' \vdash [Nt\ A] \Rightarrow\!*$ *map Tm* x› ⟨*proof*⟩

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*›
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma
transfer_parse_tree:
assumes ‹*w* ∈ *Ders P S*›
shows ‹∃ *w'* ∈ *Ders P' S*. *w* = hs *w'*›
⟨*proof*⟩

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

lemma *P_imp_h_L'*:
assumes ‹*w* ∈ *Lang P S*›
shows ‹∃ *w'* ∈ *L'*. *w* = h *w'*›
⟨*proof*⟩

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*⟩

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*⟩

8 The Theorem

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

lemma *Chomsky_Schuetzenberger_CNF*:
‹*regular* (*brackets* $\cap$ *Reg S*)
$\wedge$ L = h ‘ ((*brackets* $\cap$ *Reg S*) $\cap$ *Dyck_lang* Γ)
$\wedge$ *hom_list* (h :: ('*n*,'*t*) *bracket3 list* $\Rightarrow$ '*t list*)›
⟨*proof*⟩

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 ‹$L \equiv$ *Lang P S* $-$ {[]}›
assumes *finiteP*: ‹*finite P*›
shows ‹$\exists$ (*R*::('*n*,'*t*) *bracket3 list set*) *h* Γ. *regular R* $\wedge$ L = *h* ‘ (*R* $\cap$ *Dyck_lang* Γ) $\wedge$ *hom_list h*›
⟨*proof*⟩

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

theorem *Chomsky_Schuetzenberger*:
fixes P :: ‹('*n* :: *infinite*, '*t*) *Prods*› **and** S :: '*n*
defines ‹$L \equiv$ *Lang P S*›
assumes *finite*: ‹*finite P*›
shows ‹$\exists$ (*R*::('*n*,'*t*) *bracket3 list set*) *h* Γ. *regular R* $\wedge$ L = *h* ‘ (*R* $\cap$ *Dyck_lang* Γ) $\wedge$ *hom_list h*›
⟨*proof*⟩

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.