

Parikh's theorem

Fabian Lehr

February 6, 2026

Abstract

In formal language theory, the *Parikh image* of a language L is the set of multisets of the words in L : the order of letters becomes irrelevant, only the number of occurrences is relevant. Parikh's Theorem states that the Parikh image of a context-free language is the same as the Parikh image of some regular language. This formalization closely follows Pilling's proof [1]: It describes a context-free language as a minimal solution to a system of equations induced by a context free grammar for this language. Then it is shown that there exists a minimal solution to this system which is regular, such that the regular solution and the context-free language have the same Parikh image.

Contents

1	Regular language expressions	2
1.1	Definition	2
1.2	Basic lemmas	3
1.3	Continuity	4
1.4	Regular language expressions which evaluate to regular languages	6
1.5	Constant regular language expressions	7
2	Parikh images	8
2.1	Definition and basic lemmas	8
2.2	Monotonicity properties	9
2.3	$\Psi (A \cup B)^* = \Psi A^*B^*$	10
2.4	$\Psi (E^*F)^* = \Psi (\{\varepsilon\} \cup E^*F^*F)$	12
2.5	A homogeneous-like property for regular language expressions	14
2.6	Extension of Arden's lemma to Parikh images	15
2.7	Equivalence class of languages with identical Parikh image . .	16
3	Context free grammars and systems of equations	17
3.1	Introduction of systems of equations	17
3.2	Partial solutions of systems of equations	19

3.3	CFLs as minimal solutions to systems of equations	20
3.4	Relation between the two types of systems of equations	27
4	Pilling's proof of Parikh's theorem	28
4.1	Special representation of regular language expressions	29
4.2	Minimal solution for a single equation	33
4.3	Minimal solution of the whole system of equations	36
4.4	Parikh's theorem	40

1 Regular language expressions

```

theory Reg_Lang_Exp
  imports
    Regular-Sets.Regular_Exp
begin

```

1.1 Definition

We introduce regular language expressions which will be the building blocks of the systems of equations defined later. Regular language expressions can contain both constant languages and variable languages where variables are natural numbers for simplicity. Given a valuation, i.e. an instantiation of each variable with a language, the regular language expression can be evaluated, yielding a language.

```

datatype 'a rlexp = Var nat
  | Const 'a lang
  | Union 'a rlexp 'a rlexp
  | Concat 'a rlexp 'a rlexp
  | Star 'a rlexp

```

```

type_synonym 'a valuation = nat  $\Rightarrow$  'a lang

```

```

primrec eval :: 'a rlexp  $\Rightarrow$  'a valuation  $\Rightarrow$  'a lang where
  eval (Var n) v = v n |
  eval (Const l) _ = l |
  eval (Union f g) v = eval f v  $\cup$  eval g v |
  eval (Concat f g) v = eval f v @@ eval g v |
  eval (Star f) v = star (eval f v)

```

```

primrec vars :: 'a rlexp  $\Rightarrow$  nat set where
  vars (Var n) = {n} |
  vars (Const _) = {} |
  vars (Union f g) = vars f  $\cup$  vars g |
  vars (Concat f g) = vars f  $\cup$  vars g |
  vars (Star f) = vars f

```

Given some regular language expression, substituting each occurrence

of a variable i by the regular language expression $s i$ yields the following regular language expression:

```

primrec subst :: (nat  $\Rightarrow$  'a rlexp)  $\Rightarrow$  'a rlexp  $\Rightarrow$  'a rlexp where
  subst s (Var n) = s n |
  subst _ (Const l) = Const l |
  subst s (Union f g) = Union (subst s f) (subst s g) |
  subst s (Concat f g) = Concat (subst s f) (subst s g) |
  subst s (Star f) = Star (subst s f)

```

1.2 Basic lemmas

```

lemma substitution_lemma:
  assumes  $\forall i. v' i = \text{eval } (\text{upd } i) v$ 
  shows  $\text{eval } (\text{subst } \text{upd } f) v = \text{eval } f v'$ 
  by (induction f rule: rlexp.induct) (use assms in auto)

```

```

lemma substitution_lemma_upd:
   $\text{eval } (\text{subst } (\text{Var}(x := f')) f) v = \text{eval } f (v(x := \text{eval } f' v))$ 
  using substitution_lemma[of v(x := eval f' v)] by force

```

```

lemma subst_id:  $\text{eval } (\text{subst } \text{Var } f) v = \text{eval } f v$ 
  using substitution_lemma[of v] by simp

```

```

lemma vars_subst:  $\text{vars } (\text{subst } \text{upd } f) = (\bigcup x \in \text{vars } f. \text{vars } (\text{upd } x))$ 
  by (induction f) auto

```

```

lemma vars_subst_upd_upper:  $\text{vars } (\text{subst } (\text{Var}(x := fx)) f) \subseteq \text{vars } f - \{x\} \cup \text{vars } fx$ 

```

proof

```

  fix y
  let ?upd = Var(x := fx)
  assume  $y \in \text{vars } (\text{subst } ?\text{upd } f)$ 
  then obtain  $y'$  where  $y' \in \text{vars } f \wedge y \in \text{vars } (?\text{upd } y')$  using vars_subst by
  blast
  then show  $y \in \text{vars } f - \{x\} \cup \text{vars } fx$  by (cases x = y') auto
qed

```

```

lemma eval_vars:
  assumes  $\forall i \in \text{vars } f. s i = s' i$ 
  shows  $\text{eval } f s = \text{eval } f s'$ 
  using assms by (induction f) auto

```

```

lemma eval_vars_subst:
  assumes  $\forall i \in \text{vars } f. v i = \text{eval } (\text{upd } i) v$ 
  shows  $\text{eval } (\text{subst } \text{upd } f) v = \text{eval } f v$ 
proof -
  let ?v' =  $\lambda i. \text{if } i \in \text{vars } f \text{ then } v i \text{ else } \text{eval } (\text{upd } i) v$ 
  let ?v'' =  $\lambda i. \text{eval } (\text{upd } i) v$ 

```

have $v'_v v''$: $?v' i = ?v'' i$ **for** i **using** $assms$ **by** $simp$
then have $v_v v''$: $\forall i. ?v'' i = eval (upd i) v$ **by** $simp$
from $assms$ **have** $eval f v = eval f ?v'$ **using** $eval_vars[of f]$ **by** $simp$
also have $\dots = eval (subst upd f) v$
using $assms$ $substitution_lemma[OF v_v v'', of f]$ **by** $(simp add: eval_vars)$
finally show $?thesis$ **by** $simp$
qed

$eval f$ is monotone:

lemma $rlexp_mono$:
assumes $\forall i \in vars f. v i \subseteq v' i$
shows $eval f v \subseteq eval f v'$
using $assms$ **proof** ($induction f$ rule: $rlexp.induct$)
case $(Star x)$
then show $?case$
by $(smt (verit, best) eval.simps(5) in_star_iff_concat order_trans subsetI vars.simps(5))$
qed $fastforce+$

1.3 Continuity

lemma $rlexp_cont_aux1$:
assumes $\forall i. v i \leq v (Suc i)$
and $w \in (\bigcup i. eval f (v i))$
shows $w \in eval f (\lambda x. \bigcup i. v i x)$
proof –
from $assms(2)$ **obtain** n **where** $n_intro: w \in eval f (v n)$ **by** $auto$
have $v n x \subseteq (\bigcup i. v i x)$ **for** x **by** $auto$
with n_intro **show** $?thesis$
using $rlexp_mono$ **[where** $v=v n$ **and** $v'=\lambda x. \bigcup i. v i x$ **]** **by** $auto$
qed

lemma $langpow_Union_eval$:
assumes $\forall i. v i \leq v (Suc i)$
and $w \in (\bigcup i. eval f (v i)) \overset{\sim}{\sim} n$
shows $w \in (\bigcup i. eval f (v i)) \overset{\sim}{\sim} n$
using $assms(2)$ **proof** ($induction n$ arbitrary: w)
case 0
then show $?case$ **by** $simp$
next
case $(Suc n)$
then obtain $u u'$ **where** $w_decomp: w = u @ u'$ **and**
 $u \in (\bigcup i. eval f (v i)) \wedge u' \in (\bigcup i. eval f (v i)) \overset{\sim}{\sim} n$ **by** $fastforce$
with Suc **have** $u \in (\bigcup i. eval f (v i)) \wedge u' \in (\bigcup i. eval f (v i)) \overset{\sim}{\sim} n$ **by** $auto$
then obtain $i j$ **where** $i_intro: u \in eval f (v i)$ **and** $j_intro: u' \in eval f (v j)$
 $\overset{\sim}{\sim} n$ **by** $blast$
let $?m = max i j$
from i_intro $Suc.prem(1)$ $assms(1)$ $rlexp_mono$ **have** $1: u \in eval f (v ?m)$
by $(metis le_fun_def lift_Suc_mono_le max.cobounded1 subset_eq)$
from $Suc.prem(1)$ $assms(1)$ $rlexp_mono$ **have** $eval f (v j) \subseteq eval f (v ?m)$

by (metis le_fun_def lift_Suc_mono_le max.cobounded2)
 with j_intro lang_pow_mono have 2: $u' \in \text{eval } f (v \text{ ?}m) \rightsquigarrow n$ by auto
 from 1 2 show ?case using w_decomp by auto
 qed

lemma rlexp_cont_aux2:
 assumes $\forall i. v \ i \leq v \ (Suc \ i)$
 and $w \in \text{eval } f (\lambda x. \bigcup i. v \ i \ x)$
 shows $w \in (\bigcup i. \text{eval } f (v \ i))$
using assms(2) **proof** (induction f arbitrary: w rule: rlexp.induct)
 case (Concat f g)
 then obtain u u' where w_decomp: $w = u @ u'$
 and $u \in \text{eval } f (\lambda x. \bigcup i. v \ i \ x) \wedge u' \in \text{eval } g (\lambda x. \bigcup i. v \ i \ x)$ by auto
 with Concat have $u \in (\bigcup i. \text{eval } f (v \ i)) \wedge u' \in (\bigcup i. \text{eval } g (v \ i))$ by auto
 then obtain i j where i_intro: $u \in \text{eval } f (v \ i)$ and j_intro: $u' \in \text{eval } g (v \ j)$
 by blast
 let ?m = max i j
 from i_intro Concat.premis(1) assms(1) rlexp_mono have $u \in \text{eval } f (v \ ?m)$
 by (metis le_fun_def lift_Suc_mono_le max.cobounded1 subset_eq)
 moreover from j_intro Concat.premis(1) assms(1) rlexp_mono have $u' \in \text{eval } g (v \ ?m)$
 by (metis le_fun_def lift_Suc_mono_le max.cobounded2 subset_eq)
 ultimately show ?case using w_decomp by auto
next
 case (Star f)
 then obtain n where n_intro: $w \in (\text{eval } f (\lambda x. \bigcup i. v \ i \ x)) \rightsquigarrow n$
 using eval.simps(5) star_pow by blast
 with Star have $w \in (\bigcup i. \text{eval } f (v \ i)) \rightsquigarrow n$ using lang_pow_mono by blast
 with Star.premis assms have $w \in (\bigcup i. \text{eval } f (v \ i) \rightsquigarrow n)$ using langpow_Union_eval
 by auto
 then show ?case by (auto simp add: star_def)
 qed fastforce+

Now we prove that $\text{eval } f$ is continuous. This result is not needed in the further proof, but it is interesting anyway:

lemma rlexp_cont:
 assumes $\forall i. v \ i \leq v \ (Suc \ i)$
 shows $\text{eval } f (\lambda x. \bigcup i. v \ i \ x) = (\bigcup i. \text{eval } f (v \ i))$
proof
 from assms show $\text{eval } f (\lambda x. \bigcup i. v \ i \ x) \subseteq (\bigcup i. \text{eval } f (v \ i))$ using rlexp_cont_aux2
 by auto
 from assms show $(\bigcup i. \text{eval } f (v \ i)) \subseteq \text{eval } f (\lambda x. \bigcup i. v \ i \ x)$ using rlexp_cont_aux1
 by blast
 qed

1.4 Regular language expressions which evaluate to regular languages

Evaluating regular language expressions can yield non-regular languages even if the valuation maps each variable to a regular language. This is because *Const* may introduce non-regular languages. We therefore define the following predicate which guarantees that a regular language expression f yields a regular language if the valuation maps all variables occurring in f to some regular language. This is achieved by only allowing regular languages as constants. However, note that this predicate is just an under-approximation, i.e. there exist regular language expressions which do not satisfy this predicate but evaluate to regular languages anyway.

```
fun reg_eval :: 'a rlexp  $\Rightarrow$  bool where
  reg_eval (Var _)  $\longleftrightarrow$  True |
  reg_eval (Const l)  $\longleftrightarrow$  regular_lang l |
  reg_eval (Union f g)  $\longleftrightarrow$  reg_eval f  $\wedge$  reg_eval g |
  reg_eval (Concat f g)  $\longleftrightarrow$  reg_eval f  $\wedge$  reg_eval g |
  reg_eval (Star f)  $\longleftrightarrow$  reg_eval f
```

```
lemma emptyset_regular: reg_eval (Const {})
using lang.simps(1) reg_eval.simps(2) by blast
```

```
lemma epsilon_regular: reg_eval (Const {[]})
using lang.simps(2) reg_eval.simps(2) by blast
```

If the valuation v maps all variables occurring in the regular language expression f to a regular language, then evaluating f again yields a regular language:

```
lemma reg_eval_regular:
  assumes reg_eval f
  and  $\bigwedge n. n \in \text{vars } f \implies \text{regular\_lang } (v \ n)$ 
  shows regular_lang (eval f v)
using assms proof (induction f rule: reg_eval.induct)
  case (3 f g)
  then obtain r1 r2 where Regular_Exp.lang r1 = eval f v  $\wedge$  Regular_Exp.lang
r2 = eval g v by auto
  then have Regular_Exp.lang (Plus r1 r2) = eval (Union f g) v by simp
  then show ?case by blast
next
  case (4 f g)
  then obtain r1 r2 where Regular_Exp.lang r1 = eval f v  $\wedge$  Regular_Exp.lang
r2 = eval g v by auto
  then have Regular_Exp.lang (Times r1 r2) = eval (Concat f g) v by simp
  then show ?case by blast
next
  case (5 f)
  then obtain r where Regular_Exp.lang r = eval f v by auto
```

then have $Regular_Exp.lang (Regular_Exp.Star r) = eval (Star f) v$ **by** *simp*
then show *?case* **by** *blast*
qed *simp_all*

A *reg_eval* regular language expression stays *reg_eval* if all variables are substituted by *reg_eval* regular language expressions:

lemma *subst_reg_eval*:
assumes *reg_eval f*
and $\forall x \in vars f. reg_eval (upd x)$
shows *reg_eval (subst upd f)*
using *assms* **by** (*induction f rule: reg_eval.induct*) *simp_all*

lemma *subst_reg_eval_update*:
assumes *reg_eval f*
and *reg_eval g*
shows *reg_eval (subst (Var(x := g)) f)*
using *assms* *subst_reg_eval fun_upd_def* **by** (*metis reg_eval.simps(1)*)

For any finite union of *reg_eval* regular language expressions exists a *reg_eval* regular language expression:

lemma *finite_Union_regular_aux*:
 $\forall f \in set fs. reg_eval f \implies \exists g. reg_eval g \wedge \bigcup (vars \text{ ` } set fs) = vars g$
 $\wedge (\forall v. (\bigcup f \in set fs. eval f v) = eval g v)$

proof (*induction fs*)
case *Nil*
then show *?case* **using** *emptyset_regular* **by** *fastforce*
next
case (*Cons f1 fs*)
then obtain *g* **where** $*$: $reg_eval g \wedge \bigcup (vars \text{ ` } set fs) = vars g$
 $\wedge (\forall v. (\bigcup f \in set fs. eval f v) = eval g v)$ **by** *auto*
let $?g' = Union f1 g$
from *Cons.prem*s $*$ **have** $reg_eval ?g' \wedge \bigcup (vars \text{ ` } set (f1 \# fs)) = vars ?g'$
 $\wedge (\forall v. (\bigcup f \in set (f1 \# fs). eval f v) = eval ?g' v)$ **by** *simp*
then show *?case* **by** *blast*
qed

lemma *finite_Union_regular*:
assumes *finite F*
and $\forall f \in F. reg_eval f$
shows $\exists g. reg_eval g \wedge \bigcup (vars \text{ ` } F) = vars g \wedge (\forall v. (\bigcup f \in F. eval f v) = eval g v)$
using *assms* *finite_Union_regular_aux* *finite_list* **by** *metis*

1.5 Constant regular language expressions

We call a regular language expression constant if it contains no variables. A constant regular language expression always evaluates to the same language, independent on the valuation. Thus, if the constant regular language expression is *reg_eval*, then it evaluates to some regular language, independent

on the valuation.

abbreviation $const_rlexp :: 'a\ rlexp \Rightarrow bool$ **where**
 $const_rlexp\ f \equiv vars\ f = \{\}$

lemma $const_rlexp_lang: const_rlexp\ f \Longrightarrow \exists l. \forall v. eval\ f\ v = l$
by (*induction f*) *auto*

lemma $const_rlexp_regular_lang:$
assumes $const_rlexp\ f$
and $reg_eval\ f$
shows $\exists l. regular_lang\ l \wedge (\forall v. eval\ f\ v = l)$
using *assms const_rlexp_lang reg_eval_regular* **by** *fastforce*

end

2 Parikh images

theory *Parikh_Img*
imports
 Reg_Lang_Exp
 $HOL-Library.Multiset$
begin

2.1 Definition and basic lemmas

The Parikh vector of a finite word describes how often each symbol of the alphabet occurs in the word. We represent parikh vectors by multisets. The Parikh image of a language L , denoted by $\Psi\ L$, is then the set of Parikh vectors of all words in the language.

definition $parikh_img :: 'a\ lang \Rightarrow 'a\ multiset\ set$ **where**
 $parikh_img\ L = mset\ 'L$

notation $parikh_img\ (\Psi)$

lemma $parikh_img_Un\ [simp]: \Psi\ (L1 \cup L2) = \Psi\ L1 \cup \Psi\ L2$
by (*auto simp add: parikh_img_def*)

lemma $parikh_img_UNION: \Psi\ (\bigcup (L\ 'I)) = \bigcup ((\lambda i. \Psi\ (L\ i))\ 'I)$
by (*auto simp add: parikh_img_def*)

lemma $parikh_img_conc: \Psi\ (L1\ @@\ L2) = \{ m1 + m2 \mid m1\ m2. m1 \in \Psi\ L1 \wedge m2 \in \Psi\ L2 \}$
unfolding $parikh_img_def$ **by** *force*

lemma $parikh_img_commut: \Psi\ (L1\ @@\ L2) = \Psi\ (L2\ @@\ L1)$

proof –

have $\{ m1 + m2 \mid m1\ m2. m1 \in \Psi\ L1 \wedge m2 \in \Psi\ L2 \} =$
 $\{ m2 + m1 \mid m1\ m2. m1 \in \Psi\ L1 \wedge m2 \in \Psi\ L2 \}$

```

    using add.commute by blast
  then show ?thesis
    using parikh_img_conc[of L1] parikh_img_conc[of L2] by auto
qed

```

2.2 Monotonicity properties

```

lemma parikh_img_mono:  $A \subseteq B \implies \Psi A \subseteq \Psi B$ 
  unfolding parikh_img_def by fast

```

```

lemma parikh_conc_right_subset:  $\Psi A \subseteq \Psi B \implies \Psi (A @@@ C) \subseteq \Psi (B @@@ C)$ 
  by (auto simp add: parikh_img_conc)

```

```

lemma parikh_conc_left_subset:  $\Psi A \subseteq \Psi B \implies \Psi (C @@@ A) \subseteq \Psi (C @@@ B)$ 
  by (auto simp add: parikh_img_conc)

```

```

lemma parikh_conc_subset:
  assumes  $\Psi A \subseteq \Psi C$ 
    and  $\Psi B \subseteq \Psi D$ 
  shows  $\Psi (A @@@ B) \subseteq \Psi (C @@@ D)$ 
  using assms parikh_conc_right_subset parikh_conc_left_subset by blast

```

```

lemma parikh_conc_right:  $\Psi A = \Psi B \implies \Psi (A @@@ C) = \Psi (B @@@ C)$ 
  by (auto simp add: parikh_img_conc)

```

```

lemma parikh_conc_left:  $\Psi A = \Psi B \implies \Psi (C @@@ A) = \Psi (C @@@ B)$ 
  by (auto simp add: parikh_img_conc)

```

```

lemma parikh_pow_mono:  $\Psi A \subseteq \Psi B \implies \Psi (A \overset{\sim}{\sim} n) \subseteq \Psi (B \overset{\sim}{\sim} n)$ 
  by (induction n) (auto simp add: parikh_img_conc)

```

```

lemma parikh_star_mono:
  assumes  $\Psi A \subseteq \Psi B$ 
  shows  $\Psi (\text{star } A) \subseteq \Psi (\text{star } B)$ 

```

```

proof
  fix v
  assume v ∈  $\Psi (\text{star } A)$ 
  then obtain w where w_intro:  $mset w = v \wedge w \in \text{star } A$  unfolding parikh_img_def
  by blast
  then obtain n where  $w \in A \overset{\sim}{\sim} n$  unfolding star_def by blast
  then have  $v \in \Psi (A \overset{\sim}{\sim} n)$  using w_intro unfolding parikh_img_def by blast
  with assms have  $v \in \Psi (B \overset{\sim}{\sim} n)$  using parikh_pow_mono by blast
  then show  $v \in \Psi (\text{star } B)$  unfolding star_def using parikh_img_UNION by
fastforce
qed

```

```

lemma parikh_star_mono_eq:
  assumes  $\Psi A = \Psi B$ 

```

shows $\Psi (\text{star } A) = \Psi (\text{star } B)$
 using *parikh_star_mono* by (metis *Orderings.order_eq_iff_assms*)

lemma *parikh_img_subst_mono*:
 assumes $\forall i. \Psi (\text{eval } (A \ i) \ v) \subseteq \Psi (\text{eval } (B \ i) \ v)$
 shows $\Psi (\text{eval } (\text{subst } A \ f) \ v) \subseteq \Psi (\text{eval } (\text{subst } B \ f) \ v)$
proof (*induction f*)
 case (*Concat f1 f2*)
 then have $\Psi (\text{eval } (\text{subst } A \ f1) \ v \ @\@ \ \text{eval } (\text{subst } A \ f2) \ v)$
 $\subseteq \Psi (\text{eval } (\text{subst } B \ f1) \ v \ @\@ \ \text{eval } (\text{subst } B \ f2) \ v)$
 using *parikh_conc_subset* by blast
 then show ?case by simp
 next
 case (*Star f*)
 then have $\Psi (\text{star } (\text{eval } (\text{subst } A \ f) \ v)) \subseteq \Psi (\text{star } (\text{eval } (\text{subst } B \ f) \ v))$
 using *parikh_star_mono* by blast
 then show ?case by simp
qed (*use assms(1) in auto*)

lemma *parikh_img_subst_mono_upd*:
 assumes $\Psi (\text{eval } A \ v) \subseteq \Psi (\text{eval } B \ v)$
 shows $\Psi (\text{eval } (\text{subst } (\text{Var}(x := A)) \ f) \ v) \subseteq \Psi (\text{eval } (\text{subst } (\text{Var}(x := B)) \ f) \ v)$
 using *parikh_img_subst_mono*[of $\text{Var}(x := A) \ v \ \text{Var}(x := B)$] *assms* by auto

lemma *rlexp_mono_parikh*:
 assumes $\forall i \in \text{vars } f. \Psi (v \ i) \subseteq \Psi (v' \ i)$
 shows $\Psi (\text{eval } f \ v) \subseteq \Psi (\text{eval } f \ v')$
using *assms* **proof** (*induction f* rule: *rlexp.induct*)
 case (*Concat f1 f2*)
 then have $\Psi (\text{eval } f1 \ v \ @\@ \ \text{eval } f2 \ v) \subseteq \Psi (\text{eval } f1 \ v' \ @\@ \ \text{eval } f2 \ v')$
 using *parikh_conc_subset* by (metis *UnCI vars.simps(4)*)
 then show ?case by simp
qed (*auto simp add: SUP_mono' parikh_img_UNION parikh_star_mono*)

lemma *rlexp_mono_parikh_eq*:
 assumes $\forall i \in \text{vars } f. \Psi (v \ i) = \Psi (v' \ i)$
 shows $\Psi (\text{eval } f \ v) = \Psi (\text{eval } f \ v')$
 using *assms* *rlexp_mono_parikh* by blast

2.3 $\Psi (A \cup B)^* = \Psi A^* B^*$

This property is claimed by Pilling in [1] and will be needed later.

lemma *parikh_img_union_pow_aux1*:
 assumes $v \in \Psi ((A \cup B) \ \overset{\sim}{\sim} \ n)$
 shows $v \in \Psi (\bigcup_{i \leq n. A \ \overset{\sim}{\sim} \ i \ @\@ \ B \ \overset{\sim}{\sim} \ (n-i))$
using *assms* **proof** (*induction n* arbitrary: *v*)
 case 0
 then show ?case by simp

next
case (*Suc n*)
then obtain *w* **where** *w_intro*: $w \in (A \cup B) \overset{\sim}{\sim} (\text{Suc } n) \wedge \text{mset } w = v$
unfolding *parikh_img_def* **by** *auto*
then obtain *w1 w2* **where** *w1_w2_intro*: $w = w1 @ w2 \wedge w1 \in A \cup B \wedge w2 \in (A \cup B) \overset{\sim}{\sim} n$ **by** *fastforce*
let *?v1 = mset w1* **and** *?v2 = mset w2*
from *w1_w2_intro* **have** *?v2* $\in \Psi ((A \cup B) \overset{\sim}{\sim} n)$ **unfolding** *parikh_img_def*
by *blast*
with *Suc.IH* **have** *?v2* $\in \Psi (\bigcup i \leq n. A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i))$ **by** *auto*
then obtain *w2'* **where** *w2'_intro*: $\text{mset } w2' = \text{mset } w2 \wedge w2' \in (\bigcup i \leq n. A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i))$ **unfolding** *parikh_img_def* **by** *fastforce*
then obtain *i* **where** *i_intro*: $i \leq n \wedge w2' \in A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i)$ **by** *blast*
from *w1_w2_intro w2'_intro* **have** $\text{mset } w = \text{mset } (w1 @ w2')$
by *simp*
moreover have $\text{mset } (w1 @ w2') \in \Psi (\bigcup i \leq \text{Suc } n. A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (\text{Suc } n-i))$
proof (*cases w1* $\in A$)
case *True*
with *i_intro* **have** *Suc_i_valid*: $\text{Suc } i \leq \text{Suc } n$ **and** $w1 @ w2' \in A \overset{\sim}{\sim} (\text{Suc } i) @ @ B \overset{\sim}{\sim} (\text{Suc } n - \text{Suc } i)$
by (*auto simp add: conc_assoc*)
then have $\text{mset } (w1 @ w2') \in \Psi (A \overset{\sim}{\sim} (\text{Suc } i) @ @ B \overset{\sim}{\sim} (\text{Suc } n - \text{Suc } i))$
unfolding *parikh_img_def* **by** *blast*
with *Suc_i_valid parikh_img_UNION* **show** *?thesis* **by** *fast*
next
case *False*
with *w1_w2_intro* **have** $w1 \in B$ **by** *blast*
with *i_intro* **have** $\text{mset } (w1 @ w2') \in \Psi (B @ @ A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i))$
unfolding *parikh_img_def* **by** *blast*
then have $\text{mset } (w1 @ w2') \in \Psi (A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (\text{Suc } n-i))$
using *parikh_img_commut conc_assoc*
by (*metis Suc_diff_le conc_pow_comm i_intro lang_pow.simps(2)*)
with *i_intro parikh_img_UNION* **show** *?thesis* **by** *fastforce*
qed
ultimately show *?case* **using** *w_intro* **by** *auto*
qed

lemma *parikh_img_star_aux1*:
assumes $v \in \Psi (\text{star } (A \cup B))$
shows $v \in \Psi (\text{star } A @ @ \text{star } B)$
proof –
from *assms* **have** $v \in (\bigcup n. \Psi ((A \cup B) \overset{\sim}{\sim} n))$
unfolding *star_def* **using** *parikh_img_UNION* **by** *metis*
then obtain *n* **where** $v \in \Psi ((A \cup B) \overset{\sim}{\sim} n)$ **by** *blast*
then have $v \in \Psi (\bigcup i \leq n. A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i))$
using *parikh_img_union_pow_aux1* **by** *auto*
then have $v \in (\bigcup i \leq n. \Psi (A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i)))$ **using** *parikh_img_UNION*
by *metis*

then obtain i **where** $i \leq n \wedge v \in \Psi (A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i))$ **by** *blast*
then obtain w **where** $w_intro: mset\ w = v \wedge w \in A \overset{\sim}{\sim} i @ @ B \overset{\sim}{\sim} (n-i)$
unfolding *parikh_img_def* **by** *blast*
then obtain $w1\ w2$ **where** $w_decomp: w=w1@w2 \wedge w1 \in A \overset{\sim}{\sim} i \wedge w2 \in B \overset{\sim}{\sim} (n-i)$ **by** *blast*
then have $w1 \in star\ A$ **and** $w2 \in star\ B$ **by** *auto*
with w_decomp **have** $w \in star\ A @ @ star\ B$ **by** *auto*
with w_intro **show** *?thesis* **unfolding** *parikh_img_def* **by** *blast*
qed

lemma *parikh_img_star_aux2*:
assumes $v \in \Psi (star\ A @ @ star\ B)$
shows $v \in \Psi (star\ (A \cup B))$
proof –
from *assms* **obtain** w **where** $w_intro: mset\ w = v \wedge w \in star\ A @ @ star\ B$
unfolding *parikh_img_def* **by** *blast*
then obtain $w1\ w2$ **where** $w_decomp: w=w1@w2 \wedge w1 \in star\ A \wedge w2 \in star\ B$ **by** *blast*
then obtain $i\ j$ **where** $w1 \in A \overset{\sim}{\sim} i$ **and** $w2_intro: w2 \in B \overset{\sim}{\sim} j$ **unfolding** *star_def* **by** *blast*
then have $w1_in_union: w1 \in (A \cup B) \overset{\sim}{\sim} i$ **using** *lang_pow_mono* **by** *blast*
from $w2_intro$ **have** $w2 \in (A \cup B) \overset{\sim}{\sim} j$ **using** *lang_pow_mono* **by** *blast*
with $w1_in_union\ w_decomp$ **have** $w \in (A \cup B) \overset{\sim}{\sim} (i+j)$ **using** *lang_pow_add*
by *fast*
with w_intro **show** *?thesis* **unfolding** *parikh_img_def* **by** *auto*
qed

lemma *parikh_img_star*: $\Psi (star\ (A \cup B)) = \Psi (star\ A @ @ star\ B)$
proof
show $\Psi (star\ (A \cup B)) \subseteq \Psi (star\ A @ @ star\ B)$ **using** *parikh_img_star_aux1*
by *auto*
show $\Psi (star\ A @ @ star\ B) \subseteq \Psi (star\ (A \cup B))$ **using** *parikh_img_star_aux2*
by *auto*
qed

2.4 $\Psi (E^*F)^* = \Psi (\{\varepsilon\} \cup E^*F^*F)$

This property (where ε denotes the empty word) is claimed by Pilling as well [1]; we will use it later.

lemma *parikh_img_conc_pow*: $\Psi ((A @ @ B) \overset{\sim}{\sim} n) \subseteq \Psi (A \overset{\sim}{\sim} n @ @ B \overset{\sim}{\sim} n)$
proof (*induction n*)
case (*Suc n*)
then have $\Psi ((A @ @ B) \overset{\sim}{\sim} n @ @ A @ @ B) \subseteq \Psi (A \overset{\sim}{\sim} n @ @ B \overset{\sim}{\sim} n @ @ A @ @ B)$
using *parikh_conc_right_subset_conc_assoc* **by** *metis*
also have $\dots = \Psi (A \overset{\sim}{\sim} n @ @ A @ @ B \overset{\sim}{\sim} n @ @ B)$
by (*metis parikh_img_commut_conc_assoc parikh_conc_left*)
finally show *?case* **by** (*simp add: conc_assoc conc_pow_comm*)

qed simp

lemma parikh_img_conc_star: $\Psi (\text{star } (A \text{ @@ } B)) \subseteq \Psi (\text{star } A \text{ @@ star } B)$

proof

fix v

assume $v \in \Psi (\text{star } (A \text{ @@ } B))$

then have $\exists n. v \in \Psi ((A \text{ @@ } B) \text{ } \sim n)$ unfolding star_def by (simp add: parikh_img_UNION)

then obtain n where $v \in \Psi ((A \text{ @@ } B) \text{ } \sim n)$ by blast

with parikh_img_conc_pow have $v \in \Psi (A \text{ } \sim n \text{ @@ } B \text{ } \sim n)$ by fast

then have $v \in \Psi (A \text{ } \sim n \text{ @@ star } B)$

unfolding star_def using parikh_conc_left_subset

by (metis (no_types, lifting) Sup_upper parikh_img_mono rangeI subset_eq)

then show $v \in \Psi (\text{star } A \text{ @@ star } B)$

unfolding star_def using parikh_conc_right_subset

by (metis (no_types, lifting) Sup_upper parikh_img_mono rangeI subset_eq)

qed

lemma parikh_img_conc_pow2: $\Psi ((A \text{ @@ } B) \text{ } \sim \text{Suc } n) \subseteq \Psi (\text{star } A \text{ @@ star } B \text{ @@ } B)$

proof

fix v

assume $v \in \Psi ((A \text{ @@ } B) \text{ } \sim \text{Suc } n)$

with parikh_img_conc_pow have $v \in \Psi (A \text{ } \sim \text{Suc } n \text{ @@ } B \text{ } \sim n \text{ @@ } B)$

by (metis conc_pow_comm lang_pow.simps(2) subsetD)

then have $v \in \Psi (\text{star } A \text{ @@ } B \text{ } \sim n \text{ @@ } B)$

unfolding star_def using parikh_conc_right_subset

by (metis (no_types, lifting) Sup_upper parikh_img_mono rangeI subset_eq)

then show $v \in \Psi (\text{star } A \text{ @@ star } B \text{ @@ } B)$

unfolding star_def using parikh_conc_right_subset parikh_conc_left_subset

by (metis (no_types, lifting) Sup_upper parikh_img_mono rangeI subset_eq)

qed

lemma parikh_img_star2_aux1:

$\Psi (\text{star } (\text{star } E \text{ @@ } F)) \subseteq \Psi (\{\}\} \cup \text{star } E \text{ @@ star } F \text{ @@ } F)$

proof

fix v

assume $v \in \Psi (\text{star } (\text{star } E \text{ @@ } F))$

then have $\exists n. v \in \Psi ((\text{star } E \text{ @@ } F) \text{ } \sim n)$

unfolding star_def by (simp add: parikh_img_UNION)

then obtain n where $v \text{ in_pow_n: } v \in \Psi ((\text{star } E \text{ @@ } F) \text{ } \sim n)$ by blast

show $v \in \Psi (\{\}\} \cup \text{star } E \text{ @@ star } F \text{ @@ } F)$

proof (cases n)

case 0

with $v \text{ in_pow_n}$ have $v = \text{mset } []$ unfolding parikh_img_def by simp

then show ?thesis unfolding parikh_img_def by blast

next

case (Suc m)

with parikh_img_conc_pow2 $v \text{ in_pow_n}$ have $v \in \Psi (\text{star } (\text{star } E) \text{ @@ star } F \text{ @@ } F)$

```

F @@ F) by blast
  then show ?thesis by (metis UnCI parikh_img_Un star_idemp)
qed
qed

lemma parikh_img_star2_aux2:  $\Psi (star E @@ star F @@ F) \subseteq \Psi (star (star E @@ F))$ 
proof -
  have  $F \subseteq star E @@ F$  unfolding star_def using Nil_in_star
  by (metis concI_if_Nil1 star_def subsetI)
  then have  $\Psi (star E @@ F @@ star F) \subseteq \Psi (star E @@ F @@ star (star E @@ F))$ 
  using parikh_conc_left_subset parikh_img_mono parikh_star_mono by meson
  also have  $\dots \subseteq \Psi (star (star E @@ F))$ 
  by (metis conc_assoc_inf_sup_ord(3) parikh_img_mono star_unfold_left)
  finally show ?thesis using conc_star_comm by metis
qed

lemma parikh_img_star2:  $\Psi (star (star E @@ F)) = \Psi (\{\} \cup star E @@ star F @@ F)$ 
proof
  from parikh_img_star2_aux1
  show  $\Psi (star (star E @@ F)) \subseteq \Psi (\{\} \cup star E @@ star F @@ F)$  .
  from parikh_img_star2_aux2
  show  $\Psi (\{\} \cup star E @@ star F @@ F) \subseteq \Psi (star (star E @@ F))$ 
  by (metis le_sup_iff parikh_img_Un star_unfold_left sup.cobounded2)
qed

```

2.5 A homogeneous-like property for regular language expressions

```

lemma rlexp_homogeneous_aux:
  assumes  $v x = star Y @@ Z$ 
  shows  $\Psi (eval f v) \subseteq \Psi (star Y @@ eval f (v(x := Z)))$ 
proof (induction f)
  case (Var y)
  show ?case
  proof (cases  $x = y$ )
    case True
    with Var assms show ?thesis by simp
  next
    case False
    have  $eval (Var y) v \subseteq star Y @@ eval (Var y) v$  by (metis Nil_in_star concI_if_Nil1 subsetI)
    with False parikh_img_mono show ?thesis by auto
  qed
next
  case (Const l)

```

```

have eval (Const l) v ⊆ star Y @@ eval (Const l) v using concI_if_Nil1 by
blast
then show ?case by (simp add: parikh_img_mono)
next
case (Union f g)
then have Ψ (eval (Union f g) v) ⊆ Ψ (star Y @@ eval f (v(x := Z)) ∪
star Y @@ eval g (v(x := Z)))
by (metis eval.simps(3) parikh_img_Un sup.mono)
then show ?case by (metis conc_Un_distrib(1) eval.simps(3))
next
case (Concat f g)
then have Ψ (eval (Concat f g) v) ⊆ Ψ ((star Y @@ eval f (v(x := Z)))
@@ star Y @@ eval g (v(x := Z)))
by (metis eval.simps(4) parikh_conc_subset)
also have ... = Ψ (star Y @@ star Y @@ eval f (v(x := Z)) @@ eval g (v(x :=
Z)))
by (metis conc_assoc parikh_conc_right parikh_img_commut)
also have ... = Ψ (star Y @@ eval f (v(x := Z)) @@ eval g (v(x := Z)))
by (metis conc_assoc conc_star_star)
finally show ?case by (metis eval.simps(4))
next
case (Star f)
then have Ψ (star (eval f v)) ⊆ Ψ (star (star Y @@ eval f (v(x := Z))))
using parikh_star_mono by metis
also from parikh_img_conc_star have ... ⊆ Ψ (star Y @@ star (eval f (v(x
:= Z))))
by fastforce
finally show ?case by (metis eval.simps(5))
qed

```

Now we can prove the desired homogeneous-like property which will become useful later. Notably this property slightly differs from the property claimed in [1]. However, our property is easier to prove formally and it suffices for the rest of the proof.

```

lemma rlexp_homogeneous: Ψ (eval (subst (Var(x := Concat (Star y) z)) f) v)
⊆ Ψ (eval (Concat (Star y) (subst (Var(x := z)) f)) v)
(is Ψ ?L ⊆ Ψ ?R)

```

proof –

```

let ?v' = v(x := star (eval y v) @@ eval z v)
have Ψ ?L = Ψ (eval f ?v') using substitution_lemma_upd[where f=f] by simp
also have ... ⊆ Ψ (star (eval y v) @@ eval f (?v'(x := eval z v)))
using rlexp_homogeneous_aux[of ?v'] unfolding fun_upd_def by auto
also have ... = Ψ ?R using substitution_lemma[of v(x := eval z v)] by simp
finally show ?thesis .

```

qed

2.6 Extension of Arden's lemma to Parikh images

```

lemma parikh_img_arden_aux:

```

```

    assumes  $\Psi (A @@@ X \cup B) \subseteq \Psi X$ 
    shows  $\Psi (A \overset{\sim}{\sim} n @@@ B) \subseteq \Psi X$ 
  proof (induction n)
    case 0
    with assms show ?case by auto
  next
    case (Suc n)
    then have  $\Psi (A \overset{\sim}{\sim} (Suc\ n) @@@ B) \subseteq \Psi (A @@@ A \overset{\sim}{\sim} n @@@ B)$ 
      by (simp add: conc_assoc)
    moreover from Suc parikh_conc_left have  $\dots \subseteq \Psi (A @@@ X)$ 
      by (metis conc_Un_distrib(1) parikh_img_Un_sup.orderE sup.orderI)
    moreover from Suc.premss assms have  $\dots \subseteq \Psi X$  by auto
    ultimately show ?case by fast
  qed

```

```

lemma parikh_img_arden:
  assumes  $\Psi (A @@@ X \cup B) \subseteq \Psi X$ 
  shows  $\Psi (star\ A\ @@@\ B) \subseteq \Psi X$ 
  proof
    fix x
    assume  $x \in \Psi (star\ A\ @@@\ B)$ 
    then have  $\exists n. x \in \Psi (A \overset{\sim}{\sim} n @@@ B)$ 
      unfolding star_def by (simp add: conc_UNION_distrib(2) parikh_img_UNION)
    then obtain n where  $x \in \Psi (A \overset{\sim}{\sim} n @@@ B)$  by blast
    then show  $x \in \Psi X$  using parikh_img_arden_aux[OF assms] by fast
  qed

```

2.7 Equivalence class of languages with identical Parikh image

For a given language L , we define the equivalence class of all languages with identical Parikh image:

```

definition parikh_img_eq_class :: 'a lang  $\Rightarrow$  'a lang set where
  parikh_img_eq_class L = {L'.  $\Psi L' = \Psi L$ }

```

```

lemma parikh_img_Union_class:  $\Psi A = \Psi (\bigcup (\text{parikh\_img\_eq\_class } A))$ 
  proof
    let ?A' =  $\bigcup (\text{parikh\_img\_eq\_class } A)$ 
    show  $\Psi A \subseteq \Psi ?A'$ 
      unfolding parikh_img_eq_class_def by (simp add: Union_upper parikh_img_mono)
    show  $\Psi ?A' \subseteq \Psi A$ 
      proof
        fix v
        assume  $v \in \Psi ?A'$ 
        then obtain a where a_intro:  $mset\ a = v \wedge a \in ?A'$ 
          unfolding parikh_img_def by blast
        then obtain L where L_intro:  $a \in L \wedge L \in \text{parikh\_img\_eq\_class } A$ 
          unfolding parikh_img_eq_class_def by blast

```

```

    then have  $\Psi L = \Psi A$  unfolding parikh_img_eq_class_def by fastforce
    with a_intro L_intro show  $v \in \Psi A$  unfolding parikh_img_def by blast
  qed
qed

lemma subsetq_comm_subsetq:
  assumes  $\Psi A \subseteq \Psi B$ 
  shows  $A \subseteq \bigcup(\text{parikh\_img\_eq\_class } B)$  (is  $A \subseteq ?B'$ )
proof
  fix a
  assume a_in_A:  $a \in A$ 
  from assms have  $\Psi A \subseteq \Psi ?B'$ 
    using parikh_img_Union_class by blast
  with a_in_A have vec_a_in_B':  $\text{mset } a \in \Psi ?B'$  unfolding parikh_img_def
by fast
  then have  $\exists b. \text{mset } b = \text{mset } a \wedge b \in ?B'$ 
    unfolding parikh_img_def by fastforce
  then obtain b where b_intro:  $\text{mset } b = \text{mset } a \wedge b \in ?B'$  by blast
  with vec_a_in_B' have  $\Psi (?B' \cup \{a\}) = \Psi ?B'$  unfolding parikh_img_def by
blast
  with parikh_img_Union_class have  $\Psi (?B' \cup \{a\}) = \Psi B$  by blast
  then show  $a \in ?B'$  unfolding parikh_img_eq_class_def by blast
qed

end

```

3 Context free grammars and systems of equations

```

theory Reg_Lang_Exp_Eqns
  imports
    Parikh_Img
    Context_Free_Grammar.Context_Free_Language
begin

```

In this section, we will first introduce two types of systems of equations. Then we will show that to each CFG corresponds a system of equations of the first type and that the language defined by the CFG is a minimal solution of this systems. Lastly we prove some relations between the two types of systems of equations.

3.1 Introduction of systems of equations

For the first type of systems, each equation is of the form

$$X_i \supseteq r_i$$

For the second type of systems, each equation is of the form

$$\Psi X_i \supseteq \Psi r_i$$

i.e. the Parikh image is applied on both sides of each equation. In both cases, we represent the whole system by a list of regular language expressions where each of the variables X_0, X_1, \dots is identified by its integer, i.e. $Var\ i$ denotes the variable X_i . The i -th item of the list then represents the right-hand side r_i of the i -th equation:

type_synonym 'a eq_sys = 'a rlexp list

Now we can define what it means for a valuation v to solve a system of equations of the first type, i.e. a system without Parikh images. Afterwards we characterize minimal solutions of such a system.

definition *solves_ineq_sys* :: 'a eq_sys \Rightarrow 'a valuation \Rightarrow bool **where**
solves_ineq_sys sys v = ($\forall i < \text{length sys}$. eval (sys ! i) v \subseteq v i)

definition *min_sol_ineq_sys* :: 'a eq_sys \Rightarrow 'a valuation \Rightarrow bool **where**
min_sol_ineq_sys sys sol =
 (solves_ineq_sys sys sol \wedge ($\forall sol'$. solves_ineq_sys sys sol' \longrightarrow ($\forall x$. sol x \subseteq sol' x)))

The previous definitions can easily be extended to the second type of systems of equations where the Parikh image is applied on both sides of each equation:

definition *solves_ineq_comm* :: nat \Rightarrow 'a rlexp \Rightarrow 'a valuation \Rightarrow bool **where**
solves_ineq_comm x eq v = (Ψ (eval eq v) \subseteq Ψ (v x))

definition *solves_ineq_sys_comm* :: 'a eq_sys \Rightarrow 'a valuation \Rightarrow bool **where**
solves_ineq_sys_comm sys v = ($\forall i < \text{length sys}$. solves_ineq_comm i (sys ! i) v)

definition *min_sol_ineq_sys_comm* :: 'a eq_sys \Rightarrow 'a valuation \Rightarrow bool **where**
min_sol_ineq_sys_comm sys sol =
 (solves_ineq_sys_comm sys sol \wedge
 ($\forall sol'$. solves_ineq_sys_comm sys sol' \longrightarrow ($\forall x$. Ψ (sol x) \subseteq Ψ (sol' x))))

Substitution into each equation of a system:

definition *subst_sys* :: (nat \Rightarrow 'a rlexp) \Rightarrow 'a eq_sys \Rightarrow 'a eq_sys **where**
subst_sys = map \circ subst

lemma *subst_sys_subst*:
assumes $i < \text{length sys}$
shows (subst_sys s sys) ! i = subst s (sys ! i)
unfolding *subst_sys_def* **by** (simp add: assms)

3.2 Partial solutions of systems of equations

We introduce partial solutions, i.e. solutions which might depend on one or multiple variables. They are therefore not represented as languages, but as regular language expressions. *sol* is a partial solution of the *x*-th equation if and only if it solves the equation independently on the values of the other variables:

definition *partial_sol_ineq* :: *nat* \Rightarrow '*a rlexp* \Rightarrow '*a rlexp* \Rightarrow *bool* **where**
partial_sol_ineq *x eq sol* = $(\forall v. v\ x = \text{eval}\ sol\ v \longrightarrow \text{solves_ineq_comm}\ x\ eq\ v)$

We generalize the previous definition to partial solutions of whole systems of equations: *sols* maps each variable *i* to a regular language expression representing the partial solution of the *i*-th equation. *sols* is then a partial solution of the whole system if it satisfies the following predicate:

definition *solution_ineq_sys* :: '*a eq_sys* \Rightarrow (*nat* \Rightarrow '*a rlexp*) \Rightarrow *bool* **where**
solution_ineq_sys *sys sols* = $(\forall v. (\forall x. v\ x = \text{eval}\ (sols\ x)\ v) \longrightarrow \text{solves_ineq_sys_comm}\ sys\ v)$

Given the *x*-th equation *eq*, *sol* is a minimal partial solution of this equation if and only if

1. *sol* is a partial solution of *eq*
2. *sol* is a proper partial solution (i.e. it does not depend on *x*) and only depends on variables occurring in the equation *eq*
3. no partial solution of the equation *eq* is smaller than *sol*

definition *partial_min_sol_one_ineq* :: *nat* \Rightarrow '*a rlexp* \Rightarrow '*a rlexp* \Rightarrow *bool* **where**
partial_min_sol_one_ineq *x eq sol* =
 $(\text{partial_sol_ineq}\ x\ eq\ sol \wedge$
 $\text{vars}\ sol \subseteq \text{vars}\ eq - \{x\} \wedge$
 $(\forall sol'\ v'. \text{solves_ineq_comm}\ x\ eq\ v' \wedge v'\ x = \text{eval}\ sol'\ v'$
 $\longrightarrow \Psi(\text{eval}\ sol\ v') \subseteq \Psi(v'\ x)))$

Given a whole system of equations *sys*, we can generalize the previous definition such that *sols* is a minimal solution (possibly dependent on the variables X_n, X_{n+1}, \dots) of the first *n* equations. Besides the three conditions described above, we introduce a fourth condition: *sols* *i* = *Var* *i* for *i* \geq *n*, i.e. *sols* assigns only spurious solutions to the equations which are not yet solved:

definition *partial_min_sol_ineq_sys* :: *nat* \Rightarrow '*a eq_sys* \Rightarrow (*nat* \Rightarrow '*a rlexp*) \Rightarrow *bool* **where**
partial_min_sol_ineq_sys *n sys sols* =
 $(\text{solution_ineq_sys}\ (\text{take}\ n\ sys)\ sols \wedge$
 $(\forall i \geq n. \text{sols}\ i = \text{Var}\ i) \wedge$
 $(\forall i < n. \forall x \in \text{vars}\ (sols\ i). x \geq n \wedge x < \text{length}\ sys) \wedge$
 $(\forall sol'\ v'. (\forall x. v'\ x = \text{eval}\ (sols'\ x)\ v'))$

$$\begin{aligned} & \wedge \text{solves_ineq_sys_comm } (\text{take } n \text{ sys}) v' \\ & \longrightarrow (\forall i. \Psi (\text{eval } (\text{sols } i) v') \subseteq \Psi (v' i))) \end{aligned}$$

If the Parikh image of two equations f and g is identical on all valuations, then their minimal partial solutions are identical, too:

lemma *same_min_sol_if_same_parikh_img*:

assumes *same_parikh_img*: $\forall v. \Psi (\text{eval } f v) = \Psi (\text{eval } g v)$

and *same_vars*: $\text{vars } f - \{x\} = \text{vars } g - \{x\}$

and *minimal_sol*: $\text{partial_min_sol_one_ineq } x f \text{ sol}$

shows $\text{partial_min_sol_one_ineq } x g \text{ sol}$

proof –

from *minimal_sol* **have** $\text{vars } \text{sol} \subseteq \text{vars } g - \{x\}$

unfolding *partial_min_sol_one_ineq_def* **using** *same_vars* **by** *blast*

moreover from *same_parikh_img* *minimal_sol* **have** $\text{partial_sol_ineq } x g \text{ sol}$

unfolding *partial_min_sol_one_ineq_def* *partial_sol_ineq_def* *solves_ineq_comm_def*

by *simp*

moreover from *same_parikh_img* *minimal_sol* **have** $\forall \text{sol}' v'. \text{solves_ineq_comm } x g v' \wedge v' x = \text{eval } \text{sol}' v'$

$\longrightarrow \Psi (\text{eval } \text{sol}' v') \subseteq \Psi (v' x)$

unfolding *partial_min_sol_one_ineq_def* *solves_ineq_comm_def* **by** *blast*

ultimately show *?thesis* **unfolding** *partial_min_sol_one_ineq_def* **by** *fast*

qed

3.3 CFLs as minimal solutions to systems of equations

We show that each CFG induces a system of equations of the first type, i.e. without Parikh images, such that each equation is *req_eval* and the CFG's language is the minimal solution of the system. First, we describe how to derive the system of equations from a CFG. This requires us to fix some bijection between the variables in the system and the non-terminals occurring in the CFG:

definition *bij_Nt_Var* :: $'n \text{ set} \Rightarrow (\text{nat} \Rightarrow 'n) \Rightarrow ('n \Rightarrow \text{nat}) \Rightarrow \text{bool}$ **where**

$\text{bij_Nt_Var } A \gamma \gamma' = (\text{bij_betw } \gamma \{.. < \text{card } A\} A \wedge \text{bij_betw } \gamma' A \{.. < \text{card } A\} \\ \wedge (\forall x \in \{.. < \text{card } A\}. \gamma' (\gamma x) = x) \wedge (\forall y \in A. \gamma (\gamma' y) = y))$

lemma *exists_bij_Nt_Var*:

assumes *finite A*

shows $\exists \gamma \gamma'. \text{bij_Nt_Var } A \gamma \gamma'$

proof –

from *assms* **have** $\exists \gamma. \text{bij_betw } \gamma \{.. < \text{card } A\} A$ **by** (*simp add: bij_betw_iff_card*)

then obtain γ **where** $1: \text{bij_betw } \gamma \{.. < \text{card } A\} A$ **by** *blast*

let $? \gamma' = \text{the_inv_into } \{.. < \text{card } A\} \gamma$

from *the_inv_into_f_f 1* **have** $2: \forall x \in \{.. < \text{card } A\}. ? \gamma' (\gamma x) = x$ **unfolding** *bij_betw_def* **by** *fast*

from *bij_betw_the_inv_into[OF 1]* **have** $3: \text{bij_betw } ? \gamma' A \{.. < \text{card } A\}$ **by** *blast*

with 1 *f* *the_inv_into_f* *bij_betw* **have** $4: \forall y \in A. \gamma (? \gamma' y) = y$ **by** *metis*

from 1 2 3 4 **show** *?thesis* **unfolding** *bij_Nt_Var_def* **by** *blast*

qed

```
locale CFG_eq_sys =  
  fixes P :: ('n,'a) Prods  
  fixes S :: 'n  
  fixes  $\gamma$  :: nat  $\Rightarrow$  'n  
  fixes  $\gamma'$  :: 'n  $\Rightarrow$  nat  
  assumes finite_P: finite P  
  assumes bij_ $\gamma$ _ $\gamma'$ : bij_Nt_Var (Nts P)  $\gamma$   $\gamma'$   
begin
```

The following definitions construct a regular language expression for a single production. This happens step by step, i.e. starting with a single symbol (terminal or non-terminal) and then extending this to a single production. The definitions closely follow the definitions *inst_sym*, *concats* and *inst_syms* in *Context_Free_Grammar.Context_Free_Language*.

```
definition rlexp_sym :: ('n, 'a) sym  $\Rightarrow$  'a rlexp where  
  rlexp_sym s = (case s of Tm a  $\Rightarrow$  Const {[a]} | Nt A  $\Rightarrow$  Var ( $\gamma'$  A))
```

```
definition rlexp_concats :: 'a rlexp list  $\Rightarrow$  'a rlexp where  
  rlexp_concats fs = foldr Concat fs (Const {})
```

```
definition rlexp_syms :: ('n, 'a) syms  $\Rightarrow$  'a rlexp where  
  rlexp_syms w = rlexp_concats (map rlexp_sym w)
```

Now it is shown that the regular language expression constructed for a single production is *reg_eval*. Again, this happens step by step:

```
lemma rlexp_sym_reg: reg_eval (rlexp_sym s)  
unfolding rlexp_sym_def proof (induction s)  
  case (Tm x)  
  have regular_lang {[x]} by (meson lang.simps(3))  
  then show ?case by auto  
qed auto
```

```
lemma rlexp_concats_reg:  
  assumes  $\forall f \in \text{set } fs. \text{reg\_eval } f$   
  shows reg_eval (rlexp_concats fs)  
  using assms unfolding rlexp_concats_def by (induction fs) (use epsilon_regular  
in auto)
```

```
lemma rlexp_syms_reg: reg_eval (rlexp_syms w)  
proof -  
  from rlexp_sym_reg have  $\forall s \in \text{set } w. \text{reg\_eval } (rlexp\_sym\ s)$  by blast  
  with rlexp_concats_reg show ?thesis unfolding rlexp_syms_def  
  by (metis (no_types, lifting) image_iff list.set_map)  
qed
```

The subsequent lemmas prove that all variables appearing in the regu-

lar language expression of a single production correspond to non-terminals appearing in the production:

lemma *rlexp_sym_vars_Nt*:
assumes $s (\gamma' A) = L A$
shows $\text{vars } (\text{rlexp_sym } (Nt A)) = \{\gamma' A\}$
using *assms unfolding rlexp_sym_def* **by** *simp*

lemma *rlexp_sym_vars_Tm*: $\text{vars } (\text{rlexp_sym } (Tm x)) = \{\}$
unfolding *rlexp_sym_def* **by** *simp*

lemma *rlexp_concats_vars*: $\text{vars } (\text{rlexp_concats } fs) = \bigcup (\text{vars } \text{' } \text{set } fs)$
unfolding *rlexp_concats_def* **by** (*induction fs*) *simp_all*

lemma *insts'_vars*: $\text{vars } (\text{rlexp_syms } w) \subseteq \gamma' \text{' } Nts_syms w$

proof

fix x
assume $x \in \text{vars } (\text{rlexp_syms } w)$
with *rlexp_concats_vars* **have** $x \in \bigcup (\text{vars } \text{' } \text{set } (\text{map } \text{rlexp_sym } w))$
unfolding *rlexp_syms_def* **by** *blast*
then obtain f **where** $*$: $f \in \text{set } (\text{map } \text{rlexp_sym } w) \wedge x \in \text{vars } f$ **by** *blast*
then obtain s **where** $**$: $s \in \text{set } w \wedge \text{rlexp_sym } s = f$ **by** *auto*
with $*$ *rlexp_sym_vars_Tm* **obtain** A **where** $***$: $s = Nt A$ **by** (*metis empty_iff sym.exhaust*)
with $**$ **have** $****$: $A \in Nts_syms w$ **unfolding** *Nts_syms_def* **by** *blast*
with *rlexp_sym_vars_Nt* **have** $\text{vars } (\text{rlexp_sym } (Nt A)) = \{\gamma' A\}$ **by** *blast*
with $*$ $***$ $****$ **show** $x \in \gamma' \text{' } Nts_syms w$ **by** *blast*

qed

Evaluating the regular language expression of a single production under a valuation corresponds to instantiating the non-terminals in the production according to the valuation:

lemma *rlexp_sym_inst_Nt*:
assumes $v (\gamma' A) = L A$
shows $\text{eval } (\text{rlexp_sym } (Nt A)) v = \text{inst_sym } L (Nt A)$
using *assms unfolding rlexp_sym_def inst_sym_def* **by** *force*

lemma *rlexp_sym_inst_Tm*: $\text{eval } (\text{rlexp_sym } (Tm a)) v = \text{inst_sym } L (Tm a)$
unfolding *rlexp_sym_def inst_sym_def* **by** *force*

lemma *rlexp_concats_concats*:
assumes $\text{length } fs = \text{length } Ls$
and $\forall i < \text{length } fs. \text{eval } (fs ! i) v = Ls ! i$
shows $\text{eval } (\text{rlexp_concats } fs) v = \text{concats } Ls$
using *assms proof* (*induction fs arbitrary: Ls*)
case *Nil*
then show $?case$ **unfolding** *rlexp_concats_def concats_def* **by** *simp*
next
case (*Cons f1 fs*)

then obtain $L1\ Lr$ **where** $*$: $Ls = L1\#Lr$ **by** (*metis length_Suc_conv*)
with *Cons* **have** $eval\ (rlexp_concats\ fs)\ v = concats\ Lr$ **by** *fastforce*
moreover from *Cons.prem*s $*$ **have** $eval\ f1\ v = L1$ **by** *force*
ultimately show *?case* **unfolding** *rlexp_concats_def concats_def* **by** (*simp*
*add: **)
qed

lemma *rlexp_syms_insts*:
assumes $\forall A \in Nts_syms\ w.\ v\ (\gamma'\ A) = L\ A$
shows $eval\ (rlexp_syms\ w)\ v = inst_syms\ L\ w$
proof –
have $\forall i < length\ w.\ eval\ (rlexp_sym\ (w!\ i))\ v = inst_sym\ L\ (w!\ i)$
proof (*rule allI, rule impI*)
fix i
assume $i < length\ w$
then show $eval\ (rlexp_sym\ (w!\ i))\ v = inst_sym\ L\ (w!\ i)$
proof (*induction w!\ i*)
case (*Nt A*)
with *assms* **have** $v\ (\gamma'\ A) = L\ A$ **unfolding** *Nts_syms_def* **by** *force*
with *rlexp_sym_inst_Nt Nt* **show** *?case* **by** *metis*
next
case (*Tm x*)
with *rlexp_sym_inst_Tm* **show** *?case* **by** *metis*
qed
qed
then show *?thesis* **unfolding** *rlexp_syms_def inst_syms_def* **using** *rlexp_concats_concats*
by (*metis (mono_tags, lifting) length_map nth_map*)
qed

Each non-terminal of the CFG induces some *reg_eval* equation. We do not directly construct the equation but only prove its existence:

lemma *subst_lang_rlexp*:
 $\exists eq.\ reg_eval\ eq \wedge vars\ eq \subseteq \gamma'\ ' Nts\ P$
 $\wedge (\forall v\ L.\ (\forall A \in Nts\ P.\ v\ (\gamma'\ A) = L\ A) \longrightarrow eval\ eq\ v = subst_lang\ P\ L\ A)$
proof –
let $?Insts = rlexp_syms\ ' (Rhss\ P\ A)$
from *finite_Rhss[OF finite_P]* **have** *finite ?Insts* **by** *simp*
moreover from *rlexp_syms_reg* **have** $\forall f \in ?Insts.\ reg_eval\ f$ **by** *blast*
ultimately obtain eq **where** $*$: $reg_eval\ eq \wedge \bigcup (vars\ ' ?Insts) = vars\ eq$
 $\wedge (\forall v.\ (\bigcup f \in ?Insts.\ eval\ f\ v) = eval\ eq\ v)$
using *finite_Union_regular* **by** *metis*
moreover have $vars\ eq \subseteq \gamma'\ ' Nts\ P$
proof
fix x
assume $x \in vars\ eq$
with $*$ **obtain** f **where** $**$: $f \in ?Insts \wedge x \in vars\ f$ **by** *blast*
then obtain w **where** $***$: $w \in Rhss\ P\ A \wedge f = rlexp_syms\ w$ **by** *blast*
with $**\ insts'_vars$ **have** $x \in \gamma'\ ' Nts_syms\ w$ **by** *auto*
with $***$ **show** $x \in \gamma'\ ' Nts\ P$ **unfolding** *Nts_def Rhss_def* **by** *blast*

qed
moreover have $\forall v L. (\forall A \in Nts P. v (\gamma' A) = L A) \longrightarrow eval eq v = subst_lang P L A$
proof (*rule allI* | *rule impI*)+
fix $v :: nat \Rightarrow 'a lang$ **and** $L :: 'n \Rightarrow 'a lang$
assume $state_L: \forall A \in Nts P. v (\gamma' A) = L A$
have $\forall w \in Rhss P A. eval (rlexp_syms w) v = inst_syms L w$
proof
fix w
assume $w \in Rhss P A$
with $state_L Nts_Nts_syms$ **have** $\forall A \in Nts_syms w. v (\gamma' A) = L A$ **by**
fast
from $rlexp_syms_insts[OF this]$ **show** $eval (rlexp_syms w) v = inst_syms L w$ **by** *blast*
qed
then have $subst_lang P L A = (\bigcup f \in ?Insts. eval f v)$ **unfolding** $subst_lang_def$
by *auto*
with $*$ **show** $eval eq v = subst_lang P L A$ **by** *auto*
qed
ultimately show $?thesis$ **by** *auto*
qed

The whole CFG induces a system of *reg_eval* equations. We first define which conditions this system should fulfill and show its existence in the second step:

abbreviation $CFG_sys sys \equiv$
 $length sys = card (Nts P) \wedge$
 $(\forall i < card (Nts P). reg_eval (sys ! i) \wedge (\forall x \in vars (sys ! i). x < card (Nts P)))$
 $\wedge (\forall s L. (\forall A \in Nts P. s (\gamma' A) = L A)$
 $\longrightarrow eval (sys ! i) s = subst_lang P L (\gamma i))$

lemma $CFG_as_eq_sys: \exists sys. CFG_sys sys$

proof –

from bij_gamma_gamma' **have** $*$: $\bigwedge eq. vars eq \subseteq \gamma' ' Nts P \implies \forall x \in vars eq. x < card (Nts P)$

unfolding $bij_Nt_Var_def$ bij_betw_def **by** *auto*

from $subst_lang_rlexp$ **have** $\forall A. \exists eq. reg_eval eq \wedge vars eq \subseteq \gamma' ' Nts P \wedge$
 $(\forall s L. (\forall A \in Nts P. s (\gamma' A) = L A) \longrightarrow eval eq s =$

$subst_lang P L A)$

by *blast*

with $bij_gamma_gamma' *$ **have** $\forall i < card (Nts P). \exists eq. reg_eval eq \wedge (\forall x \in vars eq. x < card (Nts P))$

$\wedge (\forall s L. (\forall A \in Nts P. s (\gamma' A) = L A) \longrightarrow eval eq s = subst_lang P L (\gamma i))$

unfolding $bij_Nt_Var_def$ **by** *metis*

with $Skolem_list_nth$ **where** $P = \lambda i eq. reg_eval eq \wedge (\forall x \in vars eq. x < card (Nts P))$

$\wedge (\forall s L. (\forall A \in Nts P. s (\gamma' A) = L A) \longrightarrow eval eq s =$

```

subst_lang P L (γ i)]
  show ?thesis by blast
qed

```

As we have proved that each CFG induces a system of *reg_eval* equations, it remains to show that the CFG's language is a minimal solution of this system. The first lemma proves that the CFG's language is a solution and the next two lemmas prove that it is minimal:

abbreviation $sol \equiv \lambda i. \text{if } i < \text{card } (Nts P) \text{ then } Lang_lfp P (\gamma i) \text{ else } \{\}$

```

lemma CFG_sys_CFL_is_sol:
  assumes CFG_sys sys
  shows solves_ineq_sys sys sol
unfolding solves_ineq_sys_def proof (rule allI, rule impI)
  fix i
  assume i < length sys
  with assms have i < card (Nts P) by argo
  from bij_γ_γ' have *: ∀ A ∈ Nts P. sol (γ' A) = Lang_lfp P A
    unfolding bij_Nt_Var_def bij_betw_def by force
  with ⟨i < card (Nts P)⟩ assms have eval (sys ! i) sol = subst_lang P (Lang_lfp
P) (γ i)
    by presburger
  with lfp_fixpoint[OF mono_if_omega_cont[OF omega_cont_Lang_lfp]] have
1: eval (sys ! i) sol = Lang_lfp P (γ i)
    unfolding Lang_lfp_def by metis
  from ⟨i < card (Nts P)⟩ bij_γ_γ' have γ i ∈ Nts P
    unfolding bij_Nt_Var_def using bij_betwE by blast
  with * have Lang_lfp P (γ i) = sol (γ' (γ i)) by auto
  also have ... = sol i using bij_γ_γ' ⟨i < card (Nts P)⟩ unfolding bij_Nt_Var_def
by auto
  finally show eval (sys ! i) sol ⊆ sol i using 1 by blast
qed

```

```

lemma CFG_sys_CFL_is_min_aux:
  assumes CFG_sys sys
    and solves_ineq_sys sys sol'
  shows Lang_lfp P ≤ (λA. sol' (γ' A)) (is _ ≤ ?L')
proof -
  have subst_lang P ?L' A ⊆ ?L' A for A
  proof (cases A ∈ Nts P)
  case True
  with assms(1) bij_γ_γ' have γ' A < length sys
    unfolding bij_Nt_Var_def bij_betw_def by fastforce
  with assms(1) bij_γ_γ' True have subst_lang P ?L' A = eval (sys ! γ' A)
sol'
    unfolding bij_Nt_Var_def by metis
  also from True assms(2) ⟨γ' A < length sys⟩ bij_γ_γ' have ... ⊆ ?L' A
    unfolding solves_ineq_sys_def bij_Nt_Var_def by blast
  finally show ?thesis .

```

```

next
  case False
  then have Rhss P A = {} unfolding Nts_def Rhss_def by blast
  with False show ?thesis unfolding subst_lang_def by simp
qed
then have subst_lang P ?L' ≤ ?L' by (simp add: le_funI)
from lfp_lowerbound[of subst_lang P, OF this] Lang_lfp_def show ?thesis by
metis
qed

```

```

lemma CFG_sys_CFL_is_min:
  assumes CFG_sys sys
    and solves_ineq_sys sys sol'
    shows sol x ⊆ sol' x
proof (cases x < card (Nts P))
  case True
  then have sol x = Lang_lfp P (γ x) by argo
  also from CFG_sys_CFL_is_min_aux[OF assms] have  $\dots \subseteq sol' (\gamma' (\gamma x))$ 
by (simp add: le_fun_def)
  finally show ?thesis using True bij_γ_γ' unfolding bij_Nt_Var_def by auto
next
  case False
  then show ?thesis by auto
qed

```

Lastly we combine all of the previous lemmas into the desired result of this section, namely that each CFG induces a system of *reg_eval* equations such that the CFG's language is a minimal solution of the system:

```

lemma CFL_is_min_sol:
   $\exists sys. (\forall eq \in set\ sys. reg\_eval\ eq) \wedge (\forall eq \in set\ sys. \forall x \in vars\ eq. x < length\ sys)$ 
   $\wedge min\_sol\_ineq\_sys\ sys\ sol$ 
proof –
  from CFG_as_eq_sys obtain sys where  $*$ : CFG_sys sys by blast
  then have length sys = card (Nts P) by blast
  moreover from  $*$  have  $\forall eq \in set\ sys. reg\_eval\ eq$  by (simp add: all_set_conv_all_nth)
  moreover from  $*$   $\langle length\ sys = card\ (Nts\ P) \rangle$  have  $\forall eq \in set\ sys. \forall x \in vars\ eq. x < length\ sys$ 
  by (simp add: all_set_conv_all_nth)
  moreover from CFG_sys_CFL_is_sol[OF *] CFG_sys_CFL_is_min[OF *]
  have min_sol_ineq_sys sys sol unfolding min_sol_ineq_sys_def by blast
  ultimately show ?thesis by blast
qed

end

```

3.4 Relation between the two types of systems of equations

One can simply convert a system sys of equations of the second type (i.e. with Parikh images) into a system of equations of the first type by dropping the Parikh images on both sides of each equation. The following lemmas describe how the two systems are related to each other.

First of all, to any solution sol of sys exists a valuation whose Parikh image is identical to that of sol and which is a solution of the other system (i.e. the system obtained by dropping all Parikh images in sys). The following proof explicitly gives such a solution, namely $\lambda x. \bigcup (parikh_img_eq_class (sol\ x))$, benefiting from the results of section 2.7:

lemma *sol_comm_sol*:

assumes *sol_is_sol_comm*: *solves_ineq_sys_comm sys sol*

shows $\exists sol'. (\forall x. \Psi (sol\ x) = \Psi (sol'\ x)) \wedge solves_ineq_sys\ sys\ sol'$

proof

let $?sol' = \lambda x. \bigcup (parikh_img_eq_class (sol\ x))$

have *sol'_sol*: $\forall x. \Psi (?sol'\ x) = \Psi (sol\ x)$

using *parikh_img_Union_class* **by** *metis*

moreover **have** *solves_ineq_sys sys ?sol'*

unfolding *solves_ineq_sys_def* **proof** (*rule allI, rule impI*)

fix i

assume $i < length\ sys$

with *sol_is_sol_comm* **have** $\Psi (eval (sys\ !\ i)\ sol) \subseteq \Psi (sol\ i)$

unfolding *solves_ineq_sys_comm_def solves_ineq_comm_def* **by** *blast*

moreover **from** *sol'_sol* **have** $\Psi (eval (sys\ !\ i)\ ?sol') = \Psi (eval (sys\ !\ i)\ sol)$

using *rlexp_mono_parikh_eq* **by** *meson*

ultimately **have** $\Psi (eval (sys\ !\ i)\ ?sol') \subseteq \Psi (sol\ i)$ **by** *simp*

then show $eval (sys\ !\ i)\ ?sol' \subseteq ?sol'\ i$ **using** *subseteq_comm_subseteq* **by**

metis

qed

ultimately show $(\forall x. \Psi (sol\ x) = \Psi (?sol'\ x)) \wedge solves_ineq_sys\ sys\ ?sol'$

by *simp*

qed

The converse works similarly: Given a minimal solution sol of the system sys of the first type, then sol is also a minimal solution to the system obtained by converting sys into a system of the second type (which can be achieved by applying the Parikh image on both sides of each equation):

lemma *min_sol_min_sol_comm*:

assumes *min_sol_ineq_sys sys sol*

shows *min_sol_ineq_sys_comm sys sol*

unfolding *min_sol_ineq_sys_comm_def* **proof**

from *assms* **show** *solves_ineq_sys_comm sys sol*

unfolding *min_sol_ineq_sys_def min_sol_ineq_sys_comm_def solves_ineq_sys_def*

solves_ineq_sys_comm_def solves_ineq_comm_def **by** (*simp add: parikh_img_mono*)

show $\forall sol'. solves_ineq_sys_comm\ sys\ sol' \longrightarrow (\forall x. \Psi (sol\ x) \subseteq \Psi (sol'\ x))$

proof (*rule allI, rule impI*)

```

fix sol'
assume solves_ineq_sys_comm sys sol'
with sol_comm_sol obtain sol'' where sol''_intro:
  ( $\forall x. \Psi (sol' x) = \Psi (sol'' x)$ )  $\wedge$  solves_ineq_sys sys sol'' by meson
with assms have  $\forall x. sol x \subseteq sol'' x$  unfolding min_sol_ineq_sys_def by
  auto
with sol''_intro show  $\forall x. \Psi (sol x) \subseteq \Psi (sol' x)$ 
  using parikh_img_mono by metis
qed
qed

```

All minimal solutions of a system of the second type have the same Parikh image:

```

lemma min_sol_comm_unique:
assumes sol1_is_min_sol: min_sol_ineq_sys_comm sys sol1
  and sol2_is_min_sol: min_sol_ineq_sys_comm sys sol2
shows  $\Psi (sol1 x) = \Psi (sol2 x)$ 
proof -
from sol1_is_min_sol sol2_is_min_sol have  $\Psi (sol1 x) \subseteq \Psi (sol2 x)$ 
  unfolding min_sol_ineq_sys_comm_def by simp
moreover from sol1_is_min_sol sol2_is_min_sol have  $\Psi (sol2 x) \subseteq \Psi (sol1 x)$ 
  unfolding min_sol_ineq_sys_comm_def by simp
ultimately show ?thesis by blast
qed
end

```

4 Pilling's proof of Parikh's theorem

```

theory Pilling
imports
  Reg_Lang_Exp_Eqns
begin

```

We prove Parikh's theorem, closely following Pilling's proof [1]. The rough idea is as follows: As seen in section 3.3, each CFG can be interpreted as a system of *reg_eval* equations of the first type and we can easily convert it into a system of the second type by applying the Parikh image on both sides of each equation. Pilling now shows that there is a regular solution to the latter system and that this solution is furthermore minimal. Using the relations explored in section 3.4 we prove that the CFG's language is a minimal solution of the same system and hence that the Parikh image of the CFG's language and of the regular solution must be identical; this finishes the proof of Parikh's theorem.

Notably, while in [1] Pilling proves an auxiliary lemma first and applies this lemma in the proof of the main theorem, we were able to complete the whole proof without using the lemma.

4.1 Special representation of regular language expressions

To each *reg_eval* regular language expression and variable x corresponds a second regular language expression with the same Parikh image and of the form depicted in equation (3) in [1]. We call regular language expressions of this form "bipartite regular language expressions" since they decompose into two subexpressions where one of them contains the variable x and the other one does not:

definition *bipart_rlexp* :: *nat* \Rightarrow *'a rlexp* \Rightarrow *bool* **where**
bipart_rlexp x $f = (\exists p$ $q. \text{reg_eval } p \wedge \text{reg_eval } q \wedge$
 $f = \text{Union } p (\text{Concat } q (\text{Var } x)) \wedge x \notin \text{vars } p)$

All bipartite regular language expressions evaluate to regular languages. Additionally, for each *reg_eval* regular language expression and variable x , there exists a bipartite regular language expression with identical Parikh image and almost identical set of variables. While the first proof is simple, the second one is more complex and needs the results of the sections 2.3 and 2.4:

lemma *bipart_rlexp* x $f \implies \text{reg_eval } f$
unfolding *bipart_rlexp_def* **by** *fastforce*

lemma *reg_eval_bipart_rlexp_Variable*: $\exists f'. \text{bipart_rlexp } x$ $f' \wedge \text{vars } f' = \text{vars}$
 $(\text{Var } y) \cup \{x\}$
 $\wedge (\forall v. \Psi (\text{eval } (\text{Var } y) v) = \Psi (\text{eval } f' v))$

proof (*cases* $x = y$)
let $?f' = \text{Union } (\text{Const } \{\}) (\text{Concat } (\text{Const } \{\}) (\text{Var } x))$
case *True*
then have *bipart_rlexp* x $?f'$
unfolding *bipart_rlexp_def* **using** *emptyset_regular_epsilon_regular* **by** *fastforce*
moreover have $\text{eval } ?f' v = \text{eval } (\text{Var } y) v$ **for** $v :: 'a$ *valuation* **using** *True* **by** *simp*
moreover have $\text{vars } ?f' = \text{vars } (\text{Var } y) \cup \{x\}$ **using** *True* **by** *simp*
ultimately show *?thesis* **by** *metis*
next
let $?f' = \text{Union } (\text{Var } y) (\text{Concat } (\text{Const } \{\}) (\text{Var } x))$
case *False*
then have *bipart_rlexp* x $?f'$
unfolding *bipart_rlexp_def* **using** *emptyset_regular_epsilon_regular* **by** *fastforce*
moreover have $\text{eval } ?f' v = \text{eval } (\text{Var } y) v$ **for** $v :: 'a$ *valuation* **using** *False* **by** *simp*
moreover have $\text{vars } ?f' = \text{vars } (\text{Var } y) \cup \{x\}$ **by** *simp*
ultimately show *?thesis* **by** *metis*
qed

lemma *reg_eval_bipart_rlexp_Const*:

assumes *regular_lang l*
shows $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } (\text{Const } l) \cup \{x\}$
 $\wedge (\forall v. \Psi (\text{eval } (\text{Const } l) v) = \Psi (\text{eval } f' v))$
proof –
let $?f' = \text{Union } (\text{Const } l) (\text{Concat } (\text{Const } \{\}) (\text{Var } x))$
have *bipart_rlexp x ?f'*
unfolding *bipart_rlexp_def* **using** *assms emptyset_regular* **by** *simp*
moreover **have** *eval ?f' v = eval (Const l) v* **for** $v :: 'a$ *valuation* **by** *simp*
moreover **have** *vars ?f' = vars (Const l) \cup {x}* **by** *simp*
ultimately show *?thesis* **by** *metis*
qed

lemma *reg_eval_bipart_rlexp_Union*:
assumes $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f1 \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } f1 v) = \Psi (\text{eval } f' v))$
 $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f2 \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } f2 v) = \Psi (\text{eval } f' v))$
shows $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } (\text{Union } f1 f2) \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } (\text{Union } f1 f2) v) = \Psi (\text{eval } f' v))$
proof –
from *assms* **obtain** $f1' f2'$ **where** $f1'_intro: \text{bipart_rlexp } x f1' \wedge \text{vars } f1' =$
 $\text{vars } f1 \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } f1 v) = \Psi (\text{eval } f1' v))$
and $f2'_intro: \text{bipart_rlexp } x f2' \wedge \text{vars } f2' = \text{vars } f2 \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } f2 v) = \Psi (\text{eval } f2' v))$
by *auto*
then **obtain** $p1 q1 p2 q2$ **where** $p1_q1_intro: \text{reg_eval } p1 \wedge \text{reg_eval } q1 \wedge$
 $f1' = \text{Union } p1 (\text{Concat } q1 (\text{Var } x)) \wedge (\forall y \in \text{vars } p1. y \neq x)$
and $p2_q2_intro: \text{reg_eval } p2 \wedge \text{reg_eval } q2 \wedge f2' = \text{Union } p2 (\text{Concat } q2$
 $(\text{Var } x)) \wedge$
 $(\forall y \in \text{vars } p2. y \neq x)$ **unfolding** *bipart_rlexp_def* **by** *auto*
let $?f' = \text{Union } (\text{Union } p1 p2) (\text{Concat } (\text{Union } q1 q2) (\text{Var } x))$
have *bipart_rlexp x ?f'* **unfolding** *bipart_rlexp_def* **using** $p1_q1_intro p2_q2_intro$
by *auto*
moreover **have** $\Psi (\text{eval } ?f' v) = \Psi (\text{eval } (\text{Union } f1 f2) v)$ **for** v
using $p1_q1_intro p2_q2_intro f1'_intro f2'_intro$
by (*simp add: conc_Un_distrib(2) sup_assoc sup_left_commute*)
moreover **from** $f1'_intro f2'_intro p1_q1_intro p2_q2_intro$
have $\text{vars } ?f' = \text{vars } (\text{Union } f1 f2) \cup \{x\}$ **by** *auto*
ultimately show *?thesis* **by** *metis*
qed

lemma *reg_eval_bipart_rlexp_Concat*:
assumes $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f1 \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } f1 v) = \Psi (\text{eval } f' v))$
 $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f2 \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } f2 v) = \Psi (\text{eval } f' v))$
shows $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } (\text{Concat } f1 f2) \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } (\text{Concat } f1 f2) v) = \Psi (\text{eval } f' v))$

proof –

from *assms* **obtain** $f1' f2'$ **where** $f1'_intro: bipart_rlexp\ x\ f1' \wedge vars\ f1' = vars\ f1 \cup \{x\} \wedge$
 $(\forall v. \Psi (eval\ f1\ v) = \Psi (eval\ f1'\ v))$
and $f2'_intro: bipart_rlexp\ x\ f2' \wedge vars\ f2' = vars\ f2 \cup \{x\} \wedge$
 $(\forall v. \Psi (eval\ f2\ v) = \Psi (eval\ f2'\ v))$
by *auto*

then obtain $p1\ q1\ p2\ q2$ **where** $p1_q1_intro: reg_eval\ p1 \wedge reg_eval\ q1 \wedge$
 $f1' = Union\ p1\ (Concat\ q1\ (Var\ x)) \wedge (\forall y \in vars\ p1. y \neq x)$
and $p2_q2_intro: reg_eval\ p2 \wedge reg_eval\ q2 \wedge f2' = Union\ p2\ (Concat\ q2$
 $(Var\ x)) \wedge$
 $(\forall y \in vars\ p2. y \neq x)$ **unfolding** *bipart_rlexp_def* **by** *auto*

let $?q' = Union\ (Concat\ q1\ (Concat\ (Var\ x)\ q2))\ (Union\ (Concat\ p1\ q2)\ (Concat$
 $q1\ p2))$
let $?f' = Union\ (Concat\ p1\ p2)\ (Concat\ ?q'\ (Var\ x))$
have $\forall v. (\Psi (eval\ (Concat\ f1\ f2)\ v) = \Psi (eval\ ?f'\ v))$
proof (*rule allI*)

fix v
have $f2_subst: \Psi (eval\ f2\ v) = \Psi (eval\ p2\ v \cup eval\ q2\ v\ @\@ v\ x)$
using $p2_q2_intro\ f2'_intro$ **by** *auto*
have $\Psi (eval\ (Concat\ f1\ f2)\ v) = \Psi ((eval\ p1\ v \cup eval\ q1\ v\ @\@ v\ x)\ @\@ eval$
 $f2\ v)$
using $p1_q1_intro\ f1'_intro$
by (*metis eval.simps(1) eval.simps(3) eval.simps(4) parikh_conc_right*)
also have $\dots = \Psi (eval\ p1\ v\ @\@ eval\ f2\ v \cup eval\ q1\ v\ @\@ v\ x\ @\@ eval\ f2\ v)$
by (*simp add: conc_Un_distrib(2) conc_assoc*)
also have $\dots = \Psi (eval\ p1\ v\ @\@ (eval\ p2\ v \cup eval\ q2\ v\ @\@ v\ x)$
 $\cup eval\ q1\ v\ @\@ v\ x\ @\@ (eval\ p2\ v \cup eval\ q2\ v\ @\@ v\ x))$
using $f2_subst$ **by** (*smt (verit, ccfv_threshold) parikh_conc_right parikh_img_Un*
 $parikh_img_commut$)
also have $\dots = \Psi (eval\ p1\ v\ @\@ eval\ p2\ v \cup (eval\ p1\ v\ @\@ eval\ q2\ v\ @\@ v\ x$
 \cup
 $eval\ q1\ v\ @\@ eval\ p2\ v\ @\@ v\ x \cup eval\ q1\ v\ @\@ v\ x\ @\@ eval\ q2\ v\ @\@ v\ x))$
using $parikh_img_commut$ **by** (*smt (z3) conc_Un_distrib(1) parikh_conc_right*
 $parikh_img_Un\ sup_assoc$)
also have $\dots = \Psi (eval\ p1\ v\ @\@ eval\ p2\ v \cup (eval\ p1\ v\ @\@ eval\ q2\ v \cup$
 $eval\ q1\ v\ @\@ eval\ p2\ v \cup eval\ q1\ v\ @\@ v\ x\ @\@ eval\ q2\ v)\ @\@ v\ x)$
by (*simp add: conc_Un_distrib(2) conc_assoc*)
also have $\dots = \Psi (eval\ ?f'\ v)$
by (*simp add: Un_commute*)
finally show $\Psi (eval\ (Concat\ f1\ f2)\ v) = \Psi (eval\ ?f'\ v)$.

qed

moreover have $bipart_rlexp\ x\ ?f'$ **unfolding** *bipart_rlexp_def* **using** $p1_q1_intro$
 $p2_q2_intro$ **by** *auto*

moreover from $f1'_intro\ f2'_intro\ p1_q1_intro\ p2_q2_intro$
have $vars\ ?f' = vars\ (Concat\ f1\ f2) \cup \{x\}$ **by** *auto*

ultimately show *?thesis* **by** *metis*

qed

lemma *reg_eval_bipart_rlexp_Star*:

assumes $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f \cup \{x\}$

$\wedge (\forall v. \Psi (\text{eval } f v) = \Psi (\text{eval } f' v))$

shows $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } (\text{Star } f) \cup \{x\}$

$\wedge (\forall v. \Psi (\text{eval } (\text{Star } f) v) = \Psi (\text{eval } f' v))$

proof –

from *assms* **obtain** *f'* **where** *f'_intro*: *bipart_rlexp* *x f' ∧ vars f' = vars f ∪ {x}* \wedge

$(\forall v. \Psi (\text{eval } f v) = \Psi (\text{eval } f' v))$ **by** *auto*

then obtain *p q* **where** *p_q_intro*: *reg_eval* *p ∧ reg_eval* *q ∧*

f' = Union *p (Concat* *q (Var* *x)) ∧ (∀ y ∈ vars p. y ≠ x)* **unfolding** *bipart_rlexp_def* **by** *auto*

let *?q_new* = *Concat (Star* *p) (Concat (Star (Concat* *q (Var* *x))) (Concat (Star* *(Concat* *q (Var* *x)))* *q))*

let *?f_new* = *Union (Star* *p) (Concat* *?q_new (Var* *x))*

have $\forall v. (\Psi (\text{eval } (\text{Star } f) v) = \Psi (\text{eval } ?f_new v))$

proof (*rule allI*)

fix *v*

have $\Psi (\text{eval } (\text{Star } f) v) = \Psi (\text{star } (\text{eval } p v \cup \text{eval } q v @@ v x))$

using *f'_intro parikh_star_mono_eq p_q_intro*

by (*metis eval.simps(1) eval.simps(3) eval.simps(4) eval.simps(5)*)

also have $\dots = \Psi (\text{star } (\text{eval } p v) @@ \text{star } (\text{eval } q v @@ v x))$

using *parikh_img_star* **by** *blast*

also have $\dots = \Psi (\text{star } (\text{eval } p v) @@$

star $(\{\}\} \cup \text{star } (\text{eval } q v @@ v x) @@ \text{eval } q v @@ v x))$

by (*metis Un_commute conc_star_comm star_idemp star_unfold_left*)

also have $\dots = \Psi (\text{star } (\text{eval } p v) @@ \text{star } (\text{star } (\text{eval } q v @@ v x) @@ \text{eval } q v @@ v x))$

by *auto*

also have $\dots = \Psi (\text{star } (\text{eval } p v) @@ (\{\}\} \cup \text{star } (\text{eval } q v @@ v x))$

$@@ \text{star } (\text{eval } q v @@ v x) @@ \text{eval } q v @@ v x))$

using *parikh_img_star2 parikh_conc_left* **by** *blast*

also have $\dots = \Psi (\text{star } (\text{eval } p v) @@ (\{\}\} \cup \text{star } (\text{eval } p v) @@ \text{star } (\text{eval } q v @@ v x))$

$@@ \text{star } (\text{eval } q v @@ v x) @@ \text{eval } q v @@ v x)$ **by** (*metis conc_Un_distrib(1)*)

also have $\dots = \Psi (\text{eval } ?f_new v)$ **by** (*simp add: conc_assoc*)

finally show $\Psi (\text{eval } (\text{Star } f) v) = \Psi (\text{eval } ?f_new v)$.

qed

moreover have *bipart_rlexp* *x ?f_new* **unfolding** *bipart_rlexp_def* **using** *p_q_intro* **by** *fastforce*

moreover from *f'_intro p_q_intro* **have** *vars ?f_new = vars (Star f) ∪ {x}*

by *auto*

ultimately show *?thesis* **by** *metis*

qed

lemma *reg_eval_bipart_rlexp*: *reg_eval* *f* \implies

$\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f \cup \{x\} \wedge$

$(\forall s. \Psi (\text{eval } f s) = \Psi (\text{eval } f' s))$

proof (*induction f rule: reg_eval.induct*)

```

case (1 uu)
from reg_eval_bipart_rlexp_Variable show ?case by blast
next
case (2 l)
then have regular_lang_l by simp
from reg_eval_bipart_rlexp_Const[OF this] show ?case by blast
next
case (3 f g)
then have  $\exists f'. \text{bipart\_rlexp } x f' \wedge \text{vars } f' = \text{vars } f \cup \{x\} \wedge (\forall v. \Psi (\text{eval } f v) = \Psi (\text{eval } f' v))$ 
 $\exists f'. \text{bipart\_rlexp } x f' \wedge \text{vars } f' = \text{vars } g \cup \{x\} \wedge (\forall v. \Psi (\text{eval } g v) = \Psi (\text{eval } f' v))$ 
by auto
from reg_eval_bipart_rlexp_Union[OF this] show ?case by blast
next
case (4 f g)
then have  $\exists f'. \text{bipart\_rlexp } x f' \wedge \text{vars } f' = \text{vars } f \cup \{x\} \wedge (\forall v. \Psi (\text{eval } f v) = \Psi (\text{eval } f' v))$ 
 $\exists f'. \text{bipart\_rlexp } x f' \wedge \text{vars } f' = \text{vars } g \cup \{x\} \wedge (\forall v. \Psi (\text{eval } g v) = \Psi (\text{eval } f' v))$ 
by auto
from reg_eval_bipart_rlexp_Concat[OF this] show ?case by blast
next
case (5 f)
then have  $\exists f'. \text{bipart\_rlexp } x f' \wedge \text{vars } f' = \text{vars } f \cup \{x\} \wedge (\forall v. \Psi (\text{eval } f v) = \Psi (\text{eval } f' v))$ 
by auto
from reg_eval_bipart_rlexp_Star[OF this] show ?case by blast
qed

```

4.2 Minimal solution for a single equation

The aim is to prove that every system of *reg_eval* equations of the second type has some minimal solution which is *reg_eval*. In this section, we prove this property only for the case of a single equation. First we assume that the equation is bipartite but later in this section we will abandon this assumption.

```

locale single_bipartite_eq =
  fixes x :: nat
  fixes p :: 'a rlexp
  fixes q :: 'a rlexp
  assumes p_reg: reg_eval p
  assumes q_reg: reg_eval q
  assumes x_not_in_p:  $x \notin \text{vars } p$ 
begin

```

The equation and the minimal solution look as follows. Here, x describes the variable whose solution is to be determined. In the subsequent lemmas, we prove that the solution is *reg_eval* and fulfills each of the three conditions

of the predicate *partial_min_sol_one_ineq*. In particular, we will use the lemmas of the sections 2.5 and 2.6 here:

abbreviation *eq* \equiv *Union p (Concat q (Var x))*
abbreviation *sol* \equiv *Concat (Star (subst (Var(x := p)) q)) p*

lemma *sol_is_reg*: *reg_eval sol*

proof –

from *p_reg q_reg* **have** *r_reg*: *reg_eval (subst (Var(x := p)) q)*
using *subst_reg_eval_update* **by** *auto*
with *p_reg* **show** *reg_eval sol* **by** *auto*
qed

lemma *sol_vars*: *vars sol* \subseteq *vars eq* – {*x*}

proof –

let *?upd* = *Var(x := p)*
let *?subst_q* = *subst ?upd q*
from *x_not_in_p* **have** *vars_p*: *vars p* \subseteq *vars eq* – {*x*} **by** *fastforce*
moreover **have** *vars p* \cup *vars q* \subseteq *vars eq* **by** *auto*
ultimately **have** *vars ?subst_q* \subseteq *vars eq* – {*x*} **using** *vars_subst_upd_upper*
by *blast*
with *x_not_in_p* **show** *?thesis* **by** *auto*
qed

lemma *sol_is_sol_ineq*: *partial_sol_ineq x eq sol*

unfolding *partial_sol_ineq_def* **proof** (*rule allI*, *rule impI*)

fix *v*

assume *x_is_sol*: *v x = eval sol v*

let *?r* = *subst (Var(x := p)) q*

let *?upd* = *Var(x := sol)*

let *?q_subst* = *subst ?upd q*

let *?eq_subst* = *subst ?upd eq*

have *homogeneous_app*: $\Psi (eval ?q_subst v) \subseteq \Psi (eval (Concat (Star ?r) ?r) v)$

using *rlxp_homogeneous* **by** *blast*

from *x_not_in_p* **have** *eval (subst ?upd p) v = eval p v* **using** *eval_vars_subst[of p]* **by** *simp*

then **have** $\Psi (eval ?eq_subst v) = \Psi (eval p v \cup eval ?q_subst v @@@ eval sol v)$
by *simp*

also **have** $\dots \subseteq \Psi (eval p v \cup eval (Concat (Star ?r) ?r) v @@@ eval sol v)$

using *homogeneous_app* **by** (*metis dual_order.refl parikh_conc_right_subset parikh_img_Un sup.mono*)

also **have** $\dots = \Psi (eval p v) \cup$

$\Psi (star (eval ?r v) @@@ eval ?r v @@@ star (eval ?r v) @@@ eval p v)$

by (*simp add: conc_assoc*)

also **have** $\dots = \Psi (eval p v) \cup$

$\Psi (eval ?r v @@@ star (eval ?r v) @@@ eval p v)$

using *parikh_img_commut conc_star_star* **by** (*smt (verit, best) conc_assoc conc_star_comm*)

also **have** $\dots = \Psi (star (eval ?r v) @@@ eval p v)$

```

    using star_unfold_left
  by (smt (verit) conc_Un_distrib(2) conc_assoc conc_epsilon(1) parikh_img_Un
sup_commute)
  finally have *:  $\Psi (eval \text{ ?eq\_subst } v) \subseteq \Psi (v \ x)$  using x_is_sol by simp
  from x_is_sol have  $v = v(x := eval \ sol \ v)$  using fun_upd_triv by metis
  then have  $eval \ eq \ v = eval (subst (Var(x := sol)) eq) \ v$ 
    using substitution_lemma_upd[where f=eq] by presburger
  with * show solves_ineq_comm x eq v unfolding solves_ineq_comm_def by
argo
qed

```

lemma sol_is_minimal:

```

  assumes is_sol: solves_ineq_comm x eq v
    and sol'_s: v x = eval sol' v
  shows  $\Psi (eval \ sol \ v) \subseteq \Psi (v \ x)$ 
proof -
  from is_sol sol'_s have is_sol':  $\Psi (eval \ q \ v \ @\@ \ v \ x \cup \ eval \ p \ v) \subseteq \Psi (v \ x)$ 
    unfolding solves_ineq_comm_def by simp
  then have 1:  $\Psi (eval (Concat (Star q) p) v) \subseteq \Psi (v \ x)$ 
    using parikh_img_arden by auto
  from is_sol' have  $\Psi (eval \ p \ v) \subseteq \Psi (eval (Var \ x) \ v)$  by auto
  then have  $\Psi (eval (subst (Var(x := p)) q) v) \subseteq \Psi (eval \ q \ v)$ 
    using parikh_img_subst_mono_upd by (metis fun_upd_triv subst_id)
  then have  $\Psi (eval (Star (subst (Var(x := p)) q)) v) \subseteq \Psi (eval (Star \ q) \ v)$ 
    using parikh_star_mono by auto
  then have  $\Psi (eval \ sol \ v) \subseteq \Psi (eval (Concat (Star \ q) \ p) \ v)$ 
    using parikh_conc_right_subset by (metis eval_simps(4))
  with 1 show ?thesis by fast
qed

```

In summary, *sol* is a minimal partial solution and it is *reg_eval*:

lemma sol_is_minimal_reg_sol:

```

  reg_eval sol  $\wedge$  partial_min_sol_one_ineq x eq sol
  unfolding partial_min_sol_one_ineq_def
  using sol_is_reg_sol_vars sol_is_sol_ineq sol_is_minimal
  by blast

```

end

As announced at the beginning of this section, we now extend the previous result to arbitrary equations, i.e. we show that each *reg_eval* equation has some minimal partial solution which is *reg_eval*:

lemma exists_minimal_reg_sol:

```

  assumes eq_reg: reg_eval eq
  shows  $\exists sol. reg\_eval \ sol \wedge partial\_min\_sol\_one\_ineq \ x \ eq \ sol$ 
proof -
  from reg_eval_bipart_rlexp[OF eq_reg] obtain eq'
    where eq'_intro: bipart_rlexp x eq'  $\wedge$  vars eq' = vars eq  $\cup$  {x}  $\wedge$ 
      ( $\forall v. \Psi (eval \ eq \ v) = \Psi (eval \ eq' \ v)$ ) by blast

```

```

then obtain  $p\ q$ 
  where  $p\_q\_intro: \text{reg\_eval } p \wedge \text{reg\_eval } q \wedge eq' = \text{Union } p (\text{Concat } q (\text{Var } x)) \wedge x \notin \text{vars } p$ 
  unfolding  $\text{bipart\_rlexp\_def}$  by  $\text{blast}$ 
  let  $?sol = \text{Concat } (\text{Star } (\text{subst } (\text{Var}(x := p))\ q))\ p$ 
  from  $p\_q\_intro$  have  $\text{sol\_prop}: \text{reg\_eval } ?sol \wedge \text{partial\_min\_sol\_one\_ineq } x\ eq'\ ?sol$ 
  using  $\text{single\_bipartite\_eq.sol\_is\_minimal\_reg\_sol}$  unfolding  $\text{single\_bipartite\_eq\_def}$ 
by  $\text{blast}$ 
  with  $eq'\_intro$  have  $\text{partial\_min\_sol\_one\_ineq } x\ eq'\ ?sol$ 
  using  $\text{same\_min\_sol\_if\_same\_parikh\_img}$  by  $\text{blast}$ 
  with  $\text{sol\_prop}$  show  $?thesis$  by  $\text{blast}$ 
qed

```

4.3 Minimal solution of the whole system of equations

In this section we will extend the last section's result to whole systems of reg_eval equations. For this purpose, we will show by induction on r that the first r equations have some minimal partial solution which is reg_eval .

We start with the centerpiece of the induction step: If a reg_eval and minimal partial solution sols exists for the first r equations and furthermore a reg_eval and minimal partial solution sol_r exists for the r -th equation, then there exists a reg_eval and minimal partial solution for the first $\text{Suc } r$ equations as well.

```

locale  $\text{min\_sol\_induction\_step} =$ 
  fixes  $r :: \text{nat}$ 
  and  $\text{sys} :: 'a\ \text{eq\_sys}$ 
  and  $\text{sols} :: \text{nat} \Rightarrow 'a\ \text{rlexp}$ 
  and  $\text{sol}_r :: 'a\ \text{rlexp}$ 
  assumes  $\text{eqs\_reg}: \forall eq \in \text{set } \text{sys}. \text{reg\_eval } eq$ 
  and  $\text{sys\_valid}: \forall eq \in \text{set } \text{sys}. \forall x \in \text{vars } eq. x < \text{length } \text{sys}$ 
  and  $\text{r\_valid}: r < \text{length } \text{sys}$ 
  and  $\text{sols\_is\_sol}: \text{partial\_min\_sol\_ineq\_sys } r\ \text{sys } \text{sols}$ 
  and  $\text{sols\_reg}: \forall i. \text{reg\_eval } (\text{sols } i)$ 
  and  $\text{sol}_r\_is\_sol: \text{partial\_min\_sol\_one\_ineq } r (\text{subst\_sys } \text{sols } \text{sys } ! r)\ \text{sol}_r$ 
  and  $\text{sol}_r\_reg: \text{reg\_eval } \text{sol}_r$ 
begin

```

Throughout the proof, a modified system of equations will be occasionally used to simplify the proof; this modified system is obtained by substituting the partial solutions of the first r equations into the original system. Additionally we retrieve a partial solution for the first $\text{Suc } r$ equations - named sols' - by substituting the partial solution of the r -th equation into the partial solutions of each of the first r equations:

```

abbreviation  $\text{sys}' \equiv \text{subst\_sys } \text{sols } \text{sys}$ 
abbreviation  $\text{sols}' \equiv \lambda i. \text{subst } (\text{Var}(r := \text{sol}_r)) (\text{sols } i)$ 

```

lemma *sols'_r*: *sols' r = sol_r*
using *sols_is_sol* **unfolding** *partial_min_sol_ineq_sys_def* **by** *simp*

The next lemmas show that *sols'* is still *reg_eval* and that it complies with each of the four conditions defined by the predicate *partial_min_sol_ineq_sys*:

lemma *sols'_reg*: $\forall i. \text{reg_eval } (sols' i)$
using *sols_reg_sol_r_reg* **using** *subst_reg_eval_update* **by** *blast*

lemma *sols'_is_sol*: *solution_ineq_sys (take (Suc r) sys) sols'*
unfolding *solution_ineq_sys_def* **proof** (*rule allI*, *rule impI*)
fix *v*
assume *s_sols'*: $\forall x. v x = \text{eval } (sols' x) v$
from *sols'_r s_sols'* **have** *s_r_sol_r*: $v r = \text{eval } sol_r v$ **by** *simp*
with *s_sols'* **have** *s_sols*: $v x = \text{eval } (sols x) v$ **for** *x*
using *substitution_lemma_upd*[**where** $f=sols\ x$] **by** (*auto simp add: fun_upd_idem*)
with *sols_is_sol* **have** *solves_r_sys*: *solves_ineq_sys_comm (take r sys) v*
unfolding *partial_min_sol_ineq_sys_def solution_ineq_sys_def* **by** *meson*
have $\text{eval } (sys ! r) (\lambda y. \text{eval } (sols y) v) = \text{eval } (sys' ! r) v$
using *substitution_lemma*[*of* $\lambda y. \text{eval } (sols y) v$]
by (*simp add: r_valid Suc_le_lessD subst_sys_subst*)
with *s_sols* **have** $\text{eval } (sys ! r) v = \text{eval } (sys' ! r) v$
by (*metis (mono_tags, lifting) eval_vars*)
with *sol_r_is_sol s_r_sol_r* **have** $\Psi (\text{eval } (sys ! r) v) \subseteq \Psi (v r)$
unfolding *partial_min_sol_one_ineq_def partial_sol_ineq_def solves_ineq_comm_def*
by *simp*
with *solves_r_sys* **show** *solves_ineq_sys_comm (take (Suc r) sys) v*
unfolding *solves_ineq_sys_comm_def solves_ineq_comm_def* **by** (*auto simp add: less_Suc_eq*)
qed

lemma *sols'_min*: $\forall sols2\ v2. (\forall x. v2\ x = \text{eval } (sols2\ x) v2) \wedge \text{solves_ineq_sys_comm } (take\ (Suc\ r)\ sys)\ v2 \rightarrow (\forall i. \Psi (\text{eval } (sols' i) v2) \subseteq \Psi (v2 i))$

proof (*rule allI* | *rule impI*)
fix *sols2 v2 i*
assume *as*: $(\forall x. v2\ x = \text{eval } (sols2\ x) v2) \wedge \text{solves_ineq_sys_comm } (take\ (Suc\ r)\ sys)\ v2$
then **have** *solves_ineq_sys_comm (take r sys) v2* **unfolding** *solves_ineq_sys_comm_def*
by *fastforce*
with *as sols_is_sol* **have** *sols_s2*: $\Psi (\text{eval } (sols i) v2) \subseteq \Psi (v2 i)$ **for** *i*
unfolding *partial_min_sol_ineq_sys_def* **by** *auto*
have $\text{eval } (sys' ! r) v2 = \text{eval } (sys ! r) (\lambda i. \text{eval } (sols i) v2)$
unfolding *subst_sys_def* **using** *substitution_lemma*[**where** $f=sys\ !\ r$]
by (*simp add: r_valid Suc_le_lessD*)
with *sols_s2* **have** $\Psi (\text{eval } (sys' ! r) v2) \subseteq \Psi (\text{eval } (sys ! r) v2)$
using *rlxp_mono_parikh*[*of* $sys\ !\ r$] **by** *auto*
with *as* **have** *solves_ineq_comm r (sys' ! r) v2*
unfolding *solves_ineq_sys_comm_def solves_ineq_comm_def* **using** *r_valid*
by *force*

with *as sol_r_is_sol* **have** *sol_r_min*: $\Psi (\text{eval sol}_r v2) \subseteq \Psi (v2 r)$
unfolding *partial_min_sol_one_ineq_def* **by** *blast*
let $?v' = v2(r := \text{eval sol}_r v2)$
from *sol_r_min* **have** $\Psi (?v' i) \subseteq \Psi (v2 i)$ **for** *i* **by** *simp*
with *sols_s2* **show** $\Psi (\text{eval} (sols' i) v2) \subseteq \Psi (v2 i)$
using *substitution_lemma_upd*[**where** $f = sols\ i$] *rlexp_mono_parikh*[*of sols i*
 $?v' v2$] **by** *force*
qed

lemma *sols'_vars_gt_r*: $\forall i \geq \text{Suc } r. sols' i = \text{Var } i$
using *sols_is_sol* **unfolding** *partial_min_sol_ineq_sys_def* **by** *auto*

lemma *sols'_vars_leq_r*: $\forall i < \text{Suc } r. \forall x \in \text{vars} (sols' i). x \geq \text{Suc } r \wedge x < \text{length } sys$

proof –

from *sols_is_sol* **have** $\forall i < r. \forall x \in \text{vars} (sols\ i). x \geq r \wedge x < \text{length } sys$
unfolding *partial_min_sol_ineq_sys_def* **by** *simp*
with *sols_is_sol* **have** *vars_sols*: $\forall i < \text{length } sys. \forall x \in \text{vars} (sols\ i). x \geq r \wedge x < \text{length } sys$
unfolding *partial_min_sol_ineq_sys_def* **by** (*metis empty_iff insert_iff leI vars_simps(1)*)
with *sys_valid* **have** $\forall x \in \text{vars} (\text{subst } sols (sys\ !\ i)). x \geq r \wedge x < \text{length } sys$ **if**
 $i < \text{length } sys$ **for** *i*
using *vars_subst*[*of sols sys ! i*] **that** **by** (*metis UN_E nth_mem*)
then **have** $\forall x \in \text{vars} (sys' !\ i). x \geq r \wedge x < \text{length } sys$ **if** $i < \text{length } sys$ **for** *i*
unfolding *subst_sys_def* **using** *r_valid* **that** **by** *auto*
moreover **from** *sol_r_is_sol* **have** $\text{vars} (sol_r) \subseteq \text{vars} (sys' !\ r) - \{r\}$
unfolding *partial_min_sol_one_ineq_def* **by** *simp*
ultimately **have** *vars_sol_r*: $\forall x \in \text{vars } sol_r. x > r \wedge x < \text{length } sys$
unfolding *partial_min_sol_one_ineq_def* **using** *r_valid*
by (*metis DiffE insertCI nat_less_le subsetD*)
moreover **have** $\text{vars} (sols' i) \subseteq \text{vars} (sols\ i) - \{r\} \cup \text{vars } sol_r$ **if** $i < \text{length } sys$
for *i*
using *vars_subst_upd_upper* **by** *meson*
ultimately **have** $\forall x \in \text{vars} (sols' i). x > r \wedge x < \text{length } sys$ **if** $i < \text{length } sys$
for *i*
using *vars_sols* **that** **by** *fastforce*
then **show** *?thesis* **by** (*meson r_valid Suc_le_eq dual_order.strict_trans1*)
qed

In summary, *sols'* is a minimal partial solution of the first *Suc r* equations. This allows us to prove the centerpiece of the induction step in the next lemma, namely that there exists a *reg_eval* and minimal partial solution for the first *Suc r* equations:

lemma *sols'_is_min_sol*: *partial_min_sol_ineq_sys* (*Suc r*) *sys* *sols'*
unfolding *partial_min_sol_ineq_sys_def*
using *sols'_is_sol* *sols'_min* *sols'_vars_gt_r* *sols'_vars_leq_r*
by *blast*

lemma *exists_min_sol_Suc_r*:
 $\exists \text{sols}'. \text{partial_min_sol_ineq_sys } (\text{Suc } r) \text{ sys sols}' \wedge (\forall i. \text{reg_eval } (\text{sols}' i))$
using *sols'_reg sols'_is_min_sol* **by** *blast*

end

Now follows the actual induction proof: For every r , there exists a *reg_eval* and minimal partial solution of the first r equations. This then implies that there exists a regular and minimal (non-partial) solution of the whole system:

lemma *exists_minimal_reg_sol_sys_aux*:
assumes *eqs_reg*: $\forall eq \in \text{set sys}. \text{reg_eval } eq$
and *sys_valid*: $\forall eq \in \text{set sys}. \forall x \in \text{vars } eq. x < \text{length } sys$
and *r_valid*: $r \leq \text{length } sys$
shows $\exists \text{sols}. \text{partial_min_sol_ineq_sys } r \text{ sys sols} \wedge (\forall i. \text{reg_eval } (\text{sols } i))$
using *r_valid* **proof** (*induction r*)
case 0
have *solution_ineq_sys* (*take 0 sys*) *Var*
unfolding *solution_ineq_sys_def solves_ineq_sys_comm_def* **by** *simp*
then show *?case* **unfolding** *partial_min_sol_ineq_sys_def* **by** *auto*
next
case (*Suc r*)
then obtain *sols* **where** *sols_intro*: $\text{partial_min_sol_ineq_sys } r \text{ sys sols} \wedge (\forall i. \text{reg_eval } (\text{sols } i))$
by *auto*
let *?sys'* = *subst_sys sols sys*
from *eqs_reg Suc.prem*s **have** *reg_eval* (*sys ! r*) **by** *simp*
with *sols_intro Suc.prem*s **have** *sys_r_reg*: $\text{reg_eval } (?sys' ! r)$
using *subst_reg_eval[of sys ! r] subst_sys_subst[of r sys]* **by** *simp*
then obtain *sol_r* **where** *sol_r_intro*:
 $\text{reg_eval } sol_r \wedge \text{partial_min_sol_one_ineq } r (?sys' ! r) sol_r$
using *exists_minimal_reg_sol* **by** *blast*
with *Suc sols_intro sys_valid eqs_reg* **have** *min_sol_induction_step* $r \text{ sys sols } sol_r$
unfolding *min_sol_induction_step_def* **by** *force*
from *min_sol_induction_step.exists_min_sol_Suc_r[OF this]* **show** *?case* **by** *blast*
qed

lemma *exists_minimal_reg_sol_sys*:
assumes *eqs_reg*: $\forall eq \in \text{set sys}. \text{reg_eval } eq$
and *sys_valid*: $\forall eq \in \text{set sys}. \forall x \in \text{vars } eq. x < \text{length } sys$
shows $\exists \text{sols}. \text{min_sol_ineq_sys_comm } sys \text{ sols} \wedge (\forall i. \text{regular_lang } (\text{sols } i))$
proof –
from *eqs_reg sys_valid* **have**
 $\exists \text{sols}. \text{partial_min_sol_ineq_sys } (\text{length } sys) \text{ sys sols} \wedge (\forall i. \text{reg_eval } (\text{sols } i))$
using *exists_minimal_reg_sol_sys_aux* **by** *blast*

then obtain *sols* **where**
sols_intro: *partial_min_sol_ineq_sys* (length *sys*) *sys* *sols* \wedge ($\forall i$. *reg_eval* (*sols* *i*))
by *blast*
then have *const_rlexp* (*sols* *i*) **if** $i < \text{length } \textit{sys}$ **for** *i*
using *that unfolding partial_min_sol_ineq_sys_def* **by** (*meson equals0I leD*)
with *sols_intro* **have** $\exists l$. *regular_lang* *l* \wedge ($\forall v$. *eval* (*sols* *i*) $v = l$) **if** $i < \text{length } \textit{sys}$ **for** *i*
using *that const_rlexp_regular_lang* **by** *metis*
then obtain *ls* **where** *ls_intro*: $\forall i < \text{length } \textit{sys}$. *regular_lang* (*ls* *i*) \wedge ($\forall v$. *eval* (*sols* *i*) $v = \textit{ls } i$)
by *metis*
let $?ls' = \lambda i$. *if* $i < \text{length } \textit{sys}$ *then* *ls* *i* *else* $\{\}$
from *ls_intro* **have** *ls'_intro*:
 $(\forall i < \text{length } \textit{sys}$. *regular_lang* ($?ls' i$) \wedge ($\forall v$. *eval* (*sols* *i*) $v = ?ls' i$))
 \wedge ($\forall i \geq \text{length } \textit{sys}$. $?ls' i = \{\}$) **by** *force*
then have *ls'_regular*: *regular_lang* ($?ls' i$) **for** *i* **by** (*meson lang.simps(1)*)
from *ls'_intro* *sols_intro* **have** *solves_ineq_sys_comm_sys* $?ls'$
unfolding *partial_min_sol_ineq_sys_def solution_ineq_sys_def*
by (*smt (verit) eval.simps(1) linorder_not_less nless_le take_all_iff*)
moreover have $\forall sol'$. *solves_ineq_sys_comm_sys* *sol'* \longrightarrow ($\forall x$. Ψ ($?ls' x$) $\subseteq \Psi$ (*sol' x*))
proof (*rule allI, rule impI*)
fix *sol' x*
assume *as*: *solves_ineq_sys_comm_sys* *sol'*
let $?sol_rlexps = \lambda i$. *Const* (*sol' i*)
from *as* **have** *solves_ineq_sys_comm* (*take* (length *sys*) *sys*) *sol'* **by** *simp*
moreover have *sol' x* = *eval* ($?sol_rlexps x$) *sol'* **for** *x* **by** *simp*
ultimately show $\forall x$. Ψ ($?ls' x$) $\subseteq \Psi$ (*sol' x*)
using *sols_intro unfolding partial_min_sol_ineq_sys_def*
by (*smt (verit) empty_subsetI eval.simps(1) ls'_intro parikh_img_mono*)
qed
ultimately have *min_sol_ineq_sys_comm_sys* $?ls'$ **unfolding** *min_sol_ineq_sys_comm_def*
by *blast*
with *ls'_regular* **show** *?thesis* **by** *blast*
qed

4.4 Parikh's theorem

Finally we are able to prove Parikh's theorem, i.e. that to each context free language exists a regular language with identical Parikh image:

theorem *Parikh*:

assumes *CFL* (*TYPE*('n)) *L*

shows $\exists L'$. *regular_lang* *L'* $\wedge \Psi L = \Psi L'$

proof –

from *assms* **obtain** *P* **and** *S*::'n **where** $*$: $L = \textit{Lang } P S \wedge \textit{finite } P$ **unfolding** *CFL_def* **by** *blast*

show *?thesis*

proof (*cases* $S \in \textit{Nts } P$)

```

case True
from * finite_Nts_exists_bij_Nt_Var obtain  $\gamma \gamma'$  where **: bij_Nt_Var (Nts
P)  $\gamma \gamma'$  by metis
let  $?sol = \lambda i. \text{if } i < \text{card } (Nts \ P) \text{ then } Lang\_lfp \ P \ (\gamma \ i) \text{ else } \{\}$ 
from ** True have  $\gamma' \ S < \text{card } (Nts \ P) \ \gamma \ (\gamma' \ S) = S$ 
unfolding bij_Nt_Var_def bij_betw_def by auto
with Lang_lfp_eq_Lang have **: Lang P S =  $?sol \ (\gamma' \ S)$  by metis
from * ** CFG_eq_sys.CFL_is_min_sol obtain sys
where sys_intro:  $(\forall eq \in \text{set } sys. \text{req\_eval } eq) \wedge (\forall eq \in \text{set } sys. \forall x \in \text{vars}$ 
 $eq. x < \text{length } sys)$ 
 $\wedge \text{min\_sol\_ineq\_sys } sys \ ?sol$ 
unfolding CFG_eq_sys_def by blast
with min_sol_min_sol_comm have sol_is_min_sol: min_sol_ineq_sys_comm
 $sys \ ?sol$  by fast
from sys_intro exists_minimal_req_sol_sys obtain sol' where
 $sol'_intro$ :  $\text{min\_sol\_ineq\_sys\_comm } sys \ sol' \wedge \text{regular\_lang } (sol' \ (\gamma' \ S))$  by
fastforce
with sol_is_min_sol_min_sol_comm_unique have  $\Psi \ (?sol \ (\gamma' \ S)) = \Psi \ (sol'$ 
 $(\gamma' \ S))$ 
by blast
with * **  $sol'_intro$  show ?thesis by auto
next
case False
with Nts_Lhss_Rhs_Nts have  $S \notin Lhss \ P$  by fast
from Lang_empty_if_notin_Lhss[OF this] * show ?thesis by (metis lang.simps(1))
qed
qed

```

Corollary: Every context-free language over a single letter is regular.

```

corollary CFL_1_Tm_regular:
assumes CFL (TYPE('n)) L and  $\forall w \in L. \text{set } w \subseteq \{a\}$ 
shows regular_lang L
proof –
obtain L' where regular_lang L'  $\Psi \ L = \Psi \ L'$ 
using Parikh[OF assms(1)] by blast
have  $L = L'$ 
by (metis  $\langle \Psi \ L = \Psi \ L' \rangle \langle \forall w \in L. \text{set } w \subseteq \{a\} \rangle$  parikh_img_def singleton_set_mset_eq)
with  $\langle \text{regular\_lang } L' \rangle$  show ?thesis by blast
qed

```

```

corollary CFG_1_Tm_regular:
assumes finite P  $Tms \ P = \{a\}$ 
shows regular_lang (Lang P A)
proof –
let  $?L = Lang \ P \ A$ 
have  $\forall w \in ?L. \text{set } w \subseteq \{a\}$ 
using derives_Tms_syms_subset[of P [Nt A] map Tm _] assms(2)
unfolding Lang_def Tms_syms_def by auto
thus ?thesis

```

by (meson CFL_1_Tm_regular CFL_def assms(1))
qed

no_notation parikh_img (Ψ)

end

References

- [1] D. L. Pilling. Commutative regular equations and Parikh's theorem. *Journal of the London Mathematical Society*, s2-6(4):663–666, 1973. <https://doi.org/10.1112/jlms/s2-6.4.663>.