

Parikh's theorem

Fabian Lehr

October 13, 2025

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	4
1.5	Constant regular language expressions	5
2	Parikh images	6
2.1	Definition and basic lemmas	6
2.2	Monotonicity properties	7
2.3	$\Psi (A \cup B)^* = \Psi A^* B^*$	8
2.4	$\Psi (E^* F)^* = \Psi (\{\varepsilon\} \cup E^* F^* F)$	8
2.5	A homogeneous-like property for regular language expressions	9
2.6	Extension of Arden's lemma to Parikh images	9
2.7	Equivalence class of languages with identical Parikh image	9
3	Context free grammars and systems of equations	10
3.1	Introduction of systems of equations	10
3.2	Partial solutions of systems of equations	11

3.3	CFLs as minimal solutions to systems of equations	12
3.4	Relation between the two types of systems of equations	15
4	Pilling's proof of Parikh's theorem	16
4.1	Special representation of regular language expressions	17
4.2	Minimal solution for a single equation	18
4.3	Minimal solution of the whole system of equations	19
4.4	Parikh's theorem	21

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 yields the following regular language expression:

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

1.2 Basic lemmas

lemma *substitution_lemma*:

assumes $\forall i. v' \text{ } i = \text{eval } (\text{upd } i) \text{ } v$
shows $\text{eval } (\text{subst } \text{upd } f) \text{ } v = \text{eval } f \text{ } v'$
 $\langle \text{proof} \rangle$

lemma *substitution_lemma_upd*:

$\text{eval } (\text{subst } (\text{Var}(x := f')) \text{ } f) \text{ } v = \text{eval } f \text{ } (v(x := \text{eval } f' \text{ } v))$
 $\langle \text{proof} \rangle$

lemma *subst_id*: $\text{eval } (\text{subst } \text{Var } f) \text{ } v = \text{eval } f \text{ } v$
 $\langle \text{proof} \rangle$

lemma *vars_subst*: $\text{vars } (\text{subst } \text{upd } f) = (\bigcup x \in \text{vars } f. \text{vars } (\text{upd } x))$
 $\langle \text{proof} \rangle$

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

lemma *eval_vars*:

assumes $\forall i \in \text{vars } f. s \text{ } i = s' \text{ } i$
shows $\text{eval } f \text{ } s = \text{eval } f \text{ } s'$
 $\langle \text{proof} \rangle$

lemma *eval_vars_subst*:

assumes $\forall i \in \text{vars } f. v \text{ } i = \text{eval } (\text{upd } i) \text{ } v$
shows $\text{eval } (\text{subst } \text{upd } f) \text{ } v = \text{eval } f \text{ } v$
 $\langle \text{proof} \rangle$

$\text{eval } f$ is monotone:

lemma *rlexp_mono*:

assumes $\forall i \in \text{vars } f. v \text{ } i \subseteq v' \text{ } i$
shows $\text{eval } f \text{ } v \subseteq \text{eval } f \text{ } v'$
 $\langle \text{proof} \rangle$

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 $w \in eval\ f\ (\lambda x. \bigcup i. v\ i\ x)$
 $\langle proof \rangle$

lemma *langpow_Union_eval*:
assumes $\forall i. v\ i \leq v\ (Suc\ i)$
and $w \in (\bigcup i. eval\ f\ (v\ i)) \rightsquigarrow n$
shows $w \in (\bigcup i. eval\ f\ (v\ i)) \rightsquigarrow n$
 $\langle proof \rangle$

lemma *rlexp_cont_aux2*:
assumes $\forall i. v\ i \leq v\ (Suc\ i)$
and $w \in eval\ f\ (\lambda x. \bigcup i. v\ i\ x)$
shows $w \in (\bigcup i. eval\ f\ (v\ i))$
 $\langle proof \rangle$

Now we prove that *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 $eval\ f\ (\lambda x. \bigcup i. v\ i\ x) = (\bigcup i. eval\ f\ (v\ i))$
 $\langle proof \rangle$

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* {})

$\langle \text{proof} \rangle$

lemma *epsilon_regular*: *reg_eval* (*Const* {[]})
 $\langle \text{proof} \rangle$

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* $f \ v$)
 $\langle \text{proof} \rangle$

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 $\forall x \in \text{vars } f. \text{reg_eval } (\text{upd } x)$
shows *reg_eval* (*subst* $\text{upd } f$)
 $\langle \text{proof} \rangle$

lemma *subst_reg_eval_update*:
assumes *reg_eval* f
and *reg_eval* g
shows *reg_eval* (*subst* (*Var* ($x := g$)) f)
 $\langle \text{proof} \rangle$

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

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

lemma *finite_Union_regular*:
assumes *finite* F
and $\forall f \in F. \text{reg_eval } f$
shows $\exists g. \text{reg_eval } g \wedge \bigcup (\text{vars } ' F) = \text{vars } g \wedge (\forall v. (\bigcup f \in F. \text{eval } f \ v) = \text{eval } g \ v)$
 $\langle \text{proof} \rangle$

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* $\implies \exists l. \forall v. \text{eval } f \ v = l$
 <proof>

lemma *const_rlexp_regular_lang*:
assumes *const_rlexp* *f*
and *reg_eval* *f*
shows $\exists l. \text{regular_lang } l \wedge (\forall v. \text{eval } f \ v = l)$
 <proof>

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 $L \equiv \text{mset } 'L$

notation *parikh_img* (Ψ)

lemma *parikh_img_Un* [*simp*]: $\Psi \ (L1 \cup L2) = \Psi \ L1 \cup \Psi \ L2$
 <proof>

lemma *parikh_img_UNION*: $\Psi \ (\bigcup (L \ 'I)) = \bigcup ((\lambda i. \Psi \ (L \ i)) \ 'I)$
 <proof>

lemma *parikh_img_conc*: $\Psi \ (L1 \ @\@ \ L2) = \{ m1 + m2 \mid m1 \ m2. m1 \in \Psi \ L1 \wedge m2 \in \Psi \ L2 \}$
 <proof>

lemma *parikh_img_commut*: $\Psi \ (L1 \ @\@ \ L2) = \Psi \ (L2 \ @\@ \ L1)$
 <proof>

2.2 Monotonicity properties

lemma *parikh_img_mono*: $A \subseteq B \implies \Psi A \subseteq \Psi B$
 $\langle proof \rangle$

lemma *parikh_conc_right_subset*: $\Psi A \subseteq \Psi B \implies \Psi (A @@@ C) \subseteq \Psi (B @@@ C)$
 $\langle proof \rangle$

lemma *parikh_conc_left_subset*: $\Psi A \subseteq \Psi B \implies \Psi (C @@@ A) \subseteq \Psi (C @@@ B)$
 $\langle proof \rangle$

lemma *parikh_conc_subset*:
 assumes $\Psi A \subseteq \Psi C$
 and $\Psi B \subseteq \Psi D$
 shows $\Psi (A @@@ B) \subseteq \Psi (C @@@ D)$
 $\langle proof \rangle$

lemma *parikh_conc_right*: $\Psi A = \Psi B \implies \Psi (A @@@ C) = \Psi (B @@@ C)$
 $\langle proof \rangle$

lemma *parikh_conc_left*: $\Psi A = \Psi B \implies \Psi (C @@@ A) = \Psi (C @@@ B)$
 $\langle proof \rangle$

lemma *parikh_pow_mono*: $\Psi A \subseteq \Psi B \implies \Psi (A \rightsquigarrow n) \subseteq \Psi (B \rightsquigarrow n)$
 $\langle proof \rangle$

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

lemma *parikh_star_mono_eq*:
 assumes $\Psi A = \Psi B$
 shows $\Psi (\text{star } A) = \Psi (\text{star } B)$
 $\langle proof \rangle$

lemma *parikh_img_subst_mono*:
 assumes $\forall i. \Psi (\text{eval } (A \ i) \ v) \subseteq \Psi (\text{eval } (B \ i) \ v)$
 shows $\Psi (\text{eval } (\text{subst } A \ f) \ v) \subseteq \Psi (\text{eval } (\text{subst } B \ f) \ v)$
 $\langle proof \rangle$

lemma *parikh_img_subst_mono_upd*:
 assumes $\Psi (\text{eval } A \ v) \subseteq \Psi (\text{eval } B \ v)$
 shows $\Psi (\text{eval } (\text{subst } (\text{Var}(x := A)) \ f) \ v) \subseteq \Psi (\text{eval } (\text{subst } (\text{Var}(x := B)) \ f) \ v)$
 $\langle proof \rangle$

lemma *rlexp_mono_parikh*:
 assumes $\forall i \in \text{vars } f. \Psi (v \ i) \subseteq \Psi (v' \ i)$

shows $\Psi (\text{eval } f \ v) \subseteq \Psi (\text{eval } f \ v')$
 $\langle \text{proof} \rangle$

lemma *rlexp_mono_parikh_eq*:
assumes $\forall i \in \text{vars } f. \ \Psi (v \ i) = \Psi (v' \ i)$
shows $\Psi (\text{eval } f \ v) = \Psi (\text{eval } f \ v')$
 $\langle \text{proof} \rangle$

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 $v \in \Psi ((A \cup B) \sim^n)$
shows $v \in \Psi (\bigcup i \leq n. A \sim^i @@@ B \sim^{n-i})$
 $\langle \text{proof} \rangle$

lemma *parikh_img_star_aux1*:
assumes $v \in \Psi (\text{star } (A \cup B))$
shows $v \in \Psi (\text{star } A @@@ \text{star } B)$
 $\langle \text{proof} \rangle$

lemma *parikh_img_star_aux2*:
assumes $v \in \Psi (\text{star } A @@@ \text{star } B)$
shows $v \in \Psi (\text{star } (A \cup B))$
 $\langle \text{proof} \rangle$

lemma *parikh_img_star*: $\Psi (\text{star } (A \cup B)) = \Psi (\text{star } A @@@ \text{star } B)$
 $\langle \text{proof} \rangle$

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*: $\Psi ((A @@@ B) \sim^n) \subseteq \Psi (A \sim^n @@@ B \sim^n)$
 $\langle \text{proof} \rangle$

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

lemma *parikh_img_conc_pow2*: $\Psi ((A @@@ B) \sim^{Suc \ n}) \subseteq \Psi (\text{star } A @@@ \text{star } B @@@ B)$
 $\langle \text{proof} \rangle$

lemma *parikh_img_star2_aux1*:
 $\Psi (\text{star } (\text{star } E @@@ F)) \subseteq \Psi (\{\varepsilon\} \cup \text{star } E @@@ \text{star } F @@@ F)$
 $\langle \text{proof} \rangle$

lemma *parikh_img_star2_aux2*: $\Psi (star\ E\ @\@ \ star\ F\ @\@ \ F) \subseteq \Psi (star\ (star\ E\ @\@ \ F))$
 $\langle proof \rangle$

lemma *parikh_img_star2*: $\Psi (star\ (star\ E\ @\@ \ F)) = \Psi (\{\} \cup star\ E\ @\@ \ star\ F\ @\@ \ F)$
 $\langle proof \rangle$

2.5 A homogeneous-like property for regular language expressions

lemma *rlxp_homogeneous_aux*:
assumes $v\ x = star\ Y\ @\@ \ Z$
shows $\Psi (eval\ f\ v) \subseteq \Psi (star\ Y\ @\@ \ eval\ f\ (v(x := Z)))$
 $\langle proof \rangle$

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 *rlxp_homogeneous*: $\Psi (eval\ (subst\ (Var(x := Concat\ (Star\ y)\ z))\ f)\ v)$
 $\subseteq \Psi (eval\ (Concat\ (Star\ y)\ (subst\ (Var(x := z))\ f))\ v)$
 $(is\ \Psi\ ?L \subseteq \Psi\ ?R)$
 $\langle proof \rangle$

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 $\Psi (A\ \sim\ n\ @\@ \ B) \subseteq \Psi\ X$
 $\langle proof \rangle$

lemma *parikh_img_arden*:
assumes $\Psi (A\ @\@ \ X \cup B) \subseteq \Psi\ X$
shows $\Psi (star\ A\ @\@ \ B) \subseteq \Psi\ X$
 $\langle proof \rangle$

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**
 $parikh_img_eq_class\ L \equiv \{L'.\ \Psi\ L' = \Psi\ L\}$

lemma *parikh_img_Union_class*: $\Psi\ A = \Psi\ (\bigcup (parikh_img_eq_class\ A))$
 $\langle proof \rangle$

```

lemma subseteq_comm_subseteq:
  assumes  $\Psi \ A \subseteq \Psi \ B$ 
  shows  $A \subseteq \bigcup (\text{parikh\_img\_eq\_class } B)$  (is  $A \subseteq ?B'$ )
  <proof>

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. $\text{Var } i$ 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  $\equiv \forall i < \text{length sys. eval (sys ! i) } v \subseteq v \ i$ 

```

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

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 $\text{solves_ineq_comm} :: \text{nat} \Rightarrow 'a \text{ rlexp} \Rightarrow 'a \text{ valuation} \Rightarrow \text{bool}$ **where**
 $\text{solves_ineq_comm } x \text{ eq } v \equiv \Psi (\text{eval eq } v) \subseteq \Psi (v \text{ } x)$

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

definition $\text{min_sol_ineq_sys_comm} :: 'a \text{ eq_sys} \Rightarrow 'a \text{ valuation} \Rightarrow \text{bool}$ **where**
 $\text{min_sol_ineq_sys_comm sys sol} \equiv$
 $\text{solves_ineq_sys_comm sys sol} \wedge$
 $(\forall \text{ sol}'. \text{solves_ineq_sys_comm sys sol}' \longrightarrow (\forall x. \Psi (\text{sol } x) \subseteq \Psi (\text{sol}' x)))$

Substitution into each equation of a system:

definition $\text{subst_sys} :: (\text{nat} \Rightarrow 'a \text{ rlexp}) \Rightarrow 'a \text{ eq_sys} \Rightarrow 'a \text{ eq_sys}$ **where**
 $\text{subst_sys} \equiv \text{map} \circ \text{subst}$

lemma subst_sys_subst :
assumes $i < \text{length sys}$
shows $(\text{subst_sys } s \text{ sys}) ! i = \text{subst } s (\text{sys } ! i)$
 $\langle \text{proof} \rangle$

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

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

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**
 $partial_min_sol_one_ineq\ x\ eq\ sol \equiv$
 $partial_sol_ineq\ x\ eq\ sol \wedge$
 $vars\ sol \subseteq vars\ eq - \{x\} \wedge$
 $(\forall sol'\ v'.\ solves_one_ineq_comm\ x\ eq\ v' \wedge v'\ x = eval\ sol'\ v'$
 $\longrightarrow \Psi\ (eval\ sol\ v') \subseteq \Psi\ (v'\ x))$

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 forth 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**
 $partial_min_sol_ineq_sys\ n\ sys\ sols \equiv$
 $solution_ineq_sys\ (take\ n\ sys)\ sols \wedge$
 $(\forall i \geq n.\ sols\ i = Var\ i) \wedge$
 $(\forall i < n.\ \forall x \in vars\ (sols\ i). x \geq n \wedge x < length\ sys) \wedge$
 $(\forall sols'\ v'. (\forall x. v'\ x = eval\ (sols'\ x)\ v')$
 $\wedge solves_ineq_sys_comm\ (take\ n\ sys)\ v'$
 $\longrightarrow (\forall i. \Psi\ (eval\ (sols\ i)\ v') \subseteq \Psi\ (v'\ i)))$

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 $same_parikh_img$: $\forall v. \Psi\ (eval\ f\ v) = \Psi\ (eval\ g\ v)$
and $same_vars$: $vars\ f - \{x\} = vars\ g - \{x\}$
and $minimal_sol$: $partial_min_sol_one_ineq\ x\ f\ sol$
shows $partial_min_sol_one_ineq\ x\ g\ sol$
 $\langle proof \rangle$

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 reg_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 $\text{bij_Nt_Var} :: 'n \text{ set} \Rightarrow (\text{nat} \Rightarrow 'n) \Rightarrow ('n \Rightarrow \text{nat}) \Rightarrow \text{bool}$ **where**
 $\text{bij_Nt_Var } A \ \gamma \ \gamma' \equiv \text{bij_betw } \gamma \ \{..< \text{card } A\} \ A \wedge \text{bij_betw } \gamma' \ A \ \{..< \text{card } A\}$
 $\wedge (\forall x \in \{..< \text{card } A\}. \gamma' (\gamma \ x) = x) \wedge (\forall y \in A. \gamma (\gamma' \ y) = y)$

lemma exists_bij_Nt_Var :
assumes $\text{finite } A$
shows $\exists \gamma \ \gamma'. \text{bij_Nt_Var } A \ \gamma \ \gamma'$
 $\langle \text{proof} \rangle$

locale $\text{CFG_eq_sys} =$
fixes $P :: ('n, 'a) \text{ Prods}$
fixes $S :: 'n$
fixes $\gamma :: \text{nat} \Rightarrow 'n$
fixes $\gamma' :: 'n \Rightarrow \text{nat}$
assumes $\text{finite } P$: $\text{finite } P$
assumes $\text{bij_}\gamma\text{-}\gamma'$: $\text{bij_Nt_Var } (\text{Nts } P) \ \gamma \ \gamma'$
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 , concat and inst_syms in $\text{Context_Free_Grammar.Context_Free_Language}$.

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

definition $\text{rlexp_concat} :: 'a \text{ rlexp list} \Rightarrow 'a \text{ rlexp}$ **where**
 $\text{rlexp_concat } fs = \text{foldr } \text{Concat } fs \ (\text{Const } \{\})$

definition $\text{rlexp_syms} :: ('n, 'a) \text{ syms} \Rightarrow 'a \text{ rlexp}$ **where**
 $\text{rlexp_syms } w = \text{rlexp_concat} \ (\text{map } \text{rlexp_sym } w)$

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 : $\text{reg_eval } (\text{rlexp_sym } s)$
 $\langle \text{proof} \rangle$

lemma rlexp_concat_reg :
assumes $\forall f \in \text{set } fs. \text{reg_eval } f$
shows $\text{reg_eval } (\text{rlexp_concat } fs)$
 $\langle \text{proof} \rangle$

lemma rlexp_syms_reg : $\text{reg_eval } (\text{rlexp_syms } w)$
 $\langle \text{proof} \rangle$

The subsequent lemmas prove that all variables appearing in the regular 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 $\text{vars } (\text{rlexp_sym } (Nt A)) = \{\gamma' A\}$
 $\langle \text{proof} \rangle$

lemma *rlexp_sym_vars_Tm*: $\text{vars } (\text{rlexp_sym } (Tm x)) = \{\}$
 $\langle \text{proof} \rangle$

lemma *rlexp_concats_vars*: $\text{vars } (\text{rlexp_concat } fs) = \bigcup (\text{vars } \text{' set } fs)$
 $\langle \text{proof} \rangle$

lemma *insts'_vars*: $\text{vars } (\text{rlexp_syms } w) \subseteq \gamma' \text{' nts_syms } w$
 $\langle \text{proof} \rangle$

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 $\text{eval } (\text{rlexp_sym } (Nt A)) v = \text{inst_sym } L (Nt A)$
 $\langle \text{proof} \rangle$

lemma *rlexp_sym_inst_Tm*: $\text{eval } (\text{rlexp_sym } (Tm a)) v = \text{inst_sym } L (Tm a)$
 $\langle \text{proof} \rangle$

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 $\text{eval } (\text{rlexp_concat } fs) v = \text{concat } Ls$
 $\langle \text{proof} \rangle$

lemma *rlexp_syms_insts*:
assumes $\forall A \in \text{nts_syms } w. v (\gamma' A) = L A$
shows $\text{eval } (\text{rlexp_syms } w) v = \text{inst_syms } L w$
 $\langle \text{proof} \rangle$

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*:
 $\exists eq. \text{reg_eval } eq \wedge \text{vars } eq \subseteq \gamma' \text{' Nts } P$
 $\wedge (\forall v L. (\forall A \in \text{Nts } P. v (\gamma' A) = L A) \longrightarrow \text{eval } eq v = \text{subst_lang } P L A)$
 $\langle \text{proof} \rangle$

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
 $\text{length } sys = \text{card } (\text{Nts } P) \wedge$

$$\begin{aligned}
& (\forall i < \text{card } (Nts \ P). \text{ reg_eval } (sys \ ! \ i) \wedge (\forall x \in \text{vars } (sys \ ! \ i). x < \text{card } (Nts \\
& P)) \\
& \wedge (\forall s \ L. (\forall A \in Nts \ P. s \ (\gamma' \ A) = L \ A) \\
& \longrightarrow \text{eval } (sys \ ! \ i) \ s = \text{subst_lang } P \ L \ (\gamma \ i)))
\end{aligned}$$

lemma *CFG_as_eq_sys*: $\exists sys. \text{CFG_sys } sys$
 $\langle \text{proof} \rangle$

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

lemma *CFG_sys_CFL_is_sol*:
assumes *CFG_sys sys*
shows *solves_ineq_sys sys sol*
 $\langle \text{proof} \rangle$

lemma *CFG_sys_CFL_is_min_aux*:
assumes *CFG_sys sys*
and *solves_ineq_sys sys sol'*
shows $\text{Lang_lfp } P \leq (\lambda A. \text{ sol' } (\gamma' \ A))$ (**is** $_ \leq ?L$)
 $\langle \text{proof} \rangle$

lemma *CFG_sys_CFL_is_min*:
assumes *CFG_sys sys*
and *solves_ineq_sys sys sol'*
shows $\text{sol } x \subseteq \text{sol' } x$
 $\langle \text{proof} \rangle$

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*:
 $\exists sys. (\forall eq \in \text{set } sys. \text{ reg_eval } eq) \wedge (\forall eq \in \text{set } sys. \forall x \in \text{vars } eq. x < \text{length } sys)$
 $\wedge \text{min_sol_ineq_sys } sys \text{ sol}$
 $\langle \text{proof} \rangle$

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>
```

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
  <proof>
```

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>
```

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 $f \equiv \exists p\ q. \text{reg_eval } p \wedge \text{reg_eval } q \wedge$
 $f = \text{Union } p (\text{Concat } q (\text{Var } x)) \wedge x \notin \text{vars } p$

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 $f \implies \text{reg_eval } f$
 $\langle \text{proof} \rangle$

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

lemma *reg_eval_bipart_rlexp_Const*:
assumes *regular_lang* l
shows $\exists f'. \text{bipart_rlexp } x\ f' \wedge \text{vars } f' = \text{vars } (\text{Const } l) \cup \{x\}$
 $\wedge (\forall v. \Psi (\text{eval } (\text{Const } l)\ v) = \Psi (\text{eval } f'\ v))$
 $\langle \text{proof} \rangle$

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 $\exists f'. \text{bipart_rlexp } x\ f' \wedge \text{vars } f' = \text{vars } (\text{Union } f1\ f2) \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } (\text{Union } f1\ f2)\ v) = \Psi (\text{eval } f'\ v))$
 $\langle \text{proof} \rangle$

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 $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } (\text{Concat } f1 f2) \cup \{x\} \wedge$
 $(\forall v. \Psi (\text{eval } (\text{Concat } f1 f2) v) = \Psi (\text{eval } f' v))$
 $\langle \text{proof} \rangle$

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 $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } (\text{Star } f) \cup \{x\}$
 $\wedge (\forall v. \Psi (\text{eval } (\text{Star } f) v) = \Psi (\text{eval } f' v))$
 $\langle \text{proof} \rangle$

lemma *reg_eval_bipart_rlexp*: *reg_eval* $f \implies$
 $\exists f'. \text{bipart_rlexp } x f' \wedge \text{vars } f' = \text{vars } f \cup \{x\} \wedge$
 $(\forall s. \Psi (\text{eval } f s) = \Psi (\text{eval } f' s))$
 $\langle \text{proof} \rangle$

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 $x :: \text{nat}$
fixes $p :: 'a \text{ rlexp}$
fixes $q :: 'a \text{ rlexp}$
assumes $p_reg: \text{reg_eval } p$
assumes $q_reg: \text{reg_eval } q$
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 $eq \equiv \text{Union } p (\text{Concat } q (\text{Var } x))$
abbreviation $sol \equiv \text{Concat } (\text{Star } (\text{subst } (\text{Var}(x := p)) q)) p$

lemma *sol_is_reg*: *reg_eval* sol
 $\langle \text{proof} \rangle$

lemma *sol_vars*: $\text{vars } sol \subseteq \text{vars } eq - \{x\}$

$\langle proof \rangle$

lemma *sol_is_sol_ineq*: *partial_sol_ineq x eq sol*
 $\langle proof \rangle$

lemma *sol_is_minimal*:
assumes *is_sol*: *solves_ineq_comm x eq v*
and *sol'_s*: *v x = eval sol' v*
shows $\Psi (eval\ sol\ v) \subseteq \Psi (v\ x)$
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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 $\exists sol. reg_eval\ sol \wedge partial_min_sol_one_ineq\ x\ eq\ sol$
 $\langle proof \rangle$

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 *Suc r* equations as well.

locale *min_sol_induction_step* =
fixes *r* :: *nat*
and *sys* :: '*a* *eq_sys*
and *sols* :: *nat* \Rightarrow '*a* *rlxp*
and *sol_r* :: '*a* *rlxp*
assumes *eqs_reg*: $\forall eq \in set\ sys. reg_eval\ eq$
and *sys_valid*: $\forall eq \in set\ sys. \forall x \in vars\ eq. x < length\ sys$
and *r_valid*: $r < length\ sys$
and *sols_is_sol*: *partial_min_sol_ineq_sys r sys sols*
and *sols_reg*: $\forall i. reg_eval\ (sols\ i)$
and *sol_r_is_sol*: *partial_min_sol_one_ineq r (subst_sys sols sys ! r) sol_r*
and *sol_r_reg*: *reg_eval 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 $\text{sols}' \equiv \text{subst_sys } \text{sols } \text{sys}$

abbreviation $\text{sols}' \equiv \lambda i. \text{subst } (\text{Var}(r := \text{sol_}r)) (\text{sols } i)$

lemma $\text{sols}'_r: \text{sols}' r = \text{sol_}r$

$\langle \text{proof} \rangle$

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

lemma $\text{sols}'_reg: \forall i. \text{reg_eval } (\text{sols}' i)$

$\langle \text{proof} \rangle$

lemma $\text{sols}'_is_sol: \text{solution_ineq_sys } (\text{take } (\text{Suc } r) \text{ sys}) \text{ sols}'$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma $\text{sols}'_vars_gt_r: \forall i \geq \text{Suc } r. \text{sols}' i = \text{Var } i$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

In summary, sols' is a minimal partial solution of the first $\text{Suc } r$ 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 $\text{Suc } r$ equations:

lemma $\text{sols}'_is_min_sol: \text{partial_min_sol_ineq_sys } (\text{Suc } r) \text{ sys } \text{sols}'$

$\langle \text{proof} \rangle$

lemma $\text{exists_min_sol_Suc_}r:$

$\exists \text{sols}'. \text{partial_min_sol_ineq_sys } (\text{Suc } r) \text{ sys } \text{sols}' \wedge (\forall i. \text{reg_eval } (\text{sols}' i))$

$\langle \text{proof} \rangle$

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 $\exists \text{sols. partial_min_sol_ineq_sys } r \text{ sys sols} \wedge (\forall i. \text{reg_eval } (\text{sols } i))$
<proof>

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 $\exists \text{sols. min_sol_ineq_sys_comm } sys \text{ sols} \wedge (\forall i. \text{regular_lang } (\text{sols } i))$
<proof>

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 $\exists L'. \text{regular_lang } L' \wedge \Psi L = \Psi L'$
<proof>

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>

corollary *CFG_1_Tm_regular*:
assumes *finite* *P* *Tms* *P* = $\{a\}$
shows *regular_lang* (*Lang* *P* *A*)
<proof>

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>.