Parikh's theorem

Fabian Lehr

July 7, 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 6

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 $'a\ rlexp = Var\ nat$
$\quad | \ Const\ 'a\ lang$
$\quad | \ Union\ 'a\ rlexp\ 'a\ rlexp$
$\quad | \ Concat\ 'a\ rlexp\ 'a\ rlexp$
$\quad | \ Star\ 'a\ rlexp$

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 $\forall\, i.\ v'\ i = eval\ (upd\ i)\ v$
shows $eval\ (subst\ upd\ f)\ v = eval\ f\ v'$
$\langle proof \rangle$

lemma *substitution_lemma_upd*:
$eval\ (subst\ (Var(x := f'))\ f)\ v = eval\ f\ (v(x := eval\ f'\ v))$
$\langle proof \rangle$

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

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

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

lemma *eval_vars*:
assumes $\forall\, i \in vars\ f.\ s\ i = s'\ i$
shows $eval\ f\ s = eval\ f\ s'$
$\langle proof \rangle$

lemma *eval_vars_subst*:
assumes $\forall\, i \in vars\ f.\ v\ i = eval\ (upd\ i)\ v$
shows $eval\ (subst\ upd\ f)\ v = eval\ f\ v$
$\langle proof \rangle$

$eval\ f$ is monotone:

lemma *rlexp_mono*:
assumes $\forall\, i \in vars\ f.\ v\ i \subseteq v'\ i$
shows $eval\ f\ v \subseteq eval\ f\ v'$
$\langle proof \rangle$

1.3 Continuity

lemma *lang_pow_mono*:
  fixes A :: $'a$ *lang*
  assumes $A \subseteq B$
  shows A ^^ $n \subseteq B$ ^^ n
  $\langle proof \rangle$

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 $w \in (\bigcup i.\ eval\, f\ (v\ i))$ ^^ n
   shows $w \in (\bigcup i.\ eval\, f\ (v\ i)$ ^^ $n)$
$\langle proof \rangle$

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* {})
⟨*proof*⟩

lemma *epsilon_regular*: *reg_eval* (*Const* {[]})
⟨*proof*⟩

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 $\bigwedge n.\ n \in$ *vars f* $\Longrightarrow$ *regular_lang* (*v n*)
shows *regular_lang* (*eval f v*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *subst_reg_eval_update*:
assumes *reg_eval f*
and *reg_eval g*
shows *reg_eval* (*subst* (*Var*(*x* := *g*)) *f*)
⟨*proof*⟩

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* $\Longrightarrow \exists g$. *reg_eval g* $\wedge \bigcup$(*vars* ‘ *set fs*) = *vars g*
$\wedge$ ($\forall v$. ($\bigcup f \in$ *set fs*. *eval f v*) = *eval g v*)
⟨*proof*⟩

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

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* ⇒ *bool* **where**
  const_rlexp f ≡ *vars f* = {}

lemma *const_rlexp_lang*: *const_rlexp f* ⟹ ∃ *l*. ∀ *v*. *eval f v* = *l*
  ⟨*proof*⟩

lemma *const_rlexp_regular_lang*:
  assumes *const_rlexp f*
    and *reg_eval f*
  shows ∃ *l*. *regular_lang l* ∧ (∀ *v*. *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 Ψ L, is then the set of Parikh vectors of all words in the language.

definition *parikh_img* :: *'a lang* ⇒ *'a multiset set* **where**
  parikh_img L ≡ *mset* ‘ *L*

notation *parikh_img* (Ψ)

lemma *parikh_img_Un* [*simp*]: Ψ (*L1* ∪ *L2*) = Ψ *L1* ∪ Ψ *L2*
  ⟨*proof*⟩

lemma *parikh_img_UNION*: Ψ (⋃(*L* ‘ *I*)) = ⋃ ((λ*i*. Ψ (*L i*)) ‘ *I*)
  ⟨*proof*⟩

lemma *parikh_img_conc*: Ψ (*L1* @@ *L2*) = { *m1* + *m2* | *m1 m2*. *m1* ∈ Ψ *L1* ∧ *m2* ∈ Ψ *L2* }

⟨*proof*⟩

lemma *parikh_img_commut*: $\Psi\ (L1\ @@\ L2) = \Psi\ (L2\ @@\ L1)$
⟨*proof*⟩

2.2 Monotonicity properties

lemma *parikh_img_mono*: $A \subseteq B \Longrightarrow \Psi\ A \subseteq \Psi\ B$
⟨*proof*⟩

lemma *parikh_conc_right_subset*: $\Psi\ A \subseteq \Psi\ B \Longrightarrow \Psi\ (A\ @@\ C) \subseteq \Psi\ (B\ @@\ C)$
⟨*proof*⟩

lemma *parikh_conc_left_subset*: $\Psi\ A \subseteq \Psi\ B \Longrightarrow \Psi\ (C\ @@\ A) \subseteq \Psi\ (C\ @@\ B)$
⟨*proof*⟩

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

lemma *parikh_conc_right*: $\Psi\ A = \Psi\ B \Longrightarrow \Psi\ (A\ @@\ C) = \Psi\ (B\ @@\ C)$
⟨*proof*⟩

lemma *parikh_conc_left*: $\Psi\ A = \Psi\ B \Longrightarrow \Psi\ (C\ @@\ A) = \Psi\ (C\ @@\ B)$
⟨*proof*⟩

lemma *parikh_pow_mono*: $\Psi\ A \subseteq \Psi\ B \Longrightarrow \Psi\ (A\ \widehat{}\widehat{}\ n) \subseteq \Psi\ (B\ \widehat{}\widehat{}\ n)$
⟨*proof*⟩

lemma *parikh_star_mono*:
assumes $\Psi\ A \subseteq \Psi\ B$
shows $\Psi\ (\mathit{star}\ A) \subseteq \Psi\ (\mathit{star}\ B)$
⟨*proof*⟩

lemma *parikh_star_mono_eq*:
assumes $\Psi\ A = \Psi\ B$
shows $\Psi\ (\mathit{star}\ A) = \Psi\ (\mathit{star}\ B)$
⟨*proof*⟩

lemma *parikh_img_subst_mono*:
assumes $\forall\, i.\ \Psi\ (\mathit{eval}\ (A\ i)\ v) \subseteq \Psi\ (\mathit{eval}\ (B\ i)\ v)$
shows $\Psi\ (\mathit{eval}\ (\mathit{subst}\ A\ f)\ v) \subseteq \Psi\ (\mathit{eval}\ (\mathit{subst}\ B\ f)\ v)$
⟨*proof*⟩

lemma *parikh_img_subst_mono_upd*:

assumes Ψ $(eval\ A\ v) \subseteq \Psi$ $(eval\ B\ v)$
shows Ψ $(eval\ (subst\ (Var(x := A))\ f)\ v) \subseteq \Psi$ $(eval\ (subst\ (Var(x := B))\ f)\ v)$
$\langle proof \rangle$

lemma *rlexp_mono_parikh*:
assumes $\forall\, i \in vars\ f.\ \Psi\ (v\ i) \subseteq \Psi\ (v'\ i)$
shows Ψ $(eval\ f\ v) \subseteq \Psi$ $(eval\ f\ v')$
$\langle proof \rangle$

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

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

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

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

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

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*: Ψ $((A$ @@ $B)$ ^^ $n) \subseteq \Psi$ $(A$ ^^ n @@ B ^^ $n)$
$\langle proof \rangle$

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

lemma *parikh_img_conc_pow2*: Ψ $((A$ @@ $B)$ ^^ $Suc\ n) \subseteq \Psi$ $(star\ A$ @@ $star$ B @@ $B)$

⟨*proof*⟩

lemma *parikh_img_star2_aux1*:
 Ψ (*star* (*star* E @@ F)) $\subseteq$ Ψ ({[]} $\cup$ *star* E @@ *star* F @@ F)
⟨*proof*⟩

lemma *parikh_img_star2_aux2*: Ψ (*star* E @@ *star* F @@ F) $\subseteq$ Ψ (*star* (*star* E @@ F))
⟨*proof*⟩

lemma *parikh_img_star2*: Ψ (*star* (*star* E @@ F)) = Ψ ({[]} $\cup$ *star* E @@ *star* F @@ F)
⟨*proof*⟩

2.5 A homogeneous-like property for regular language expressions

lemma *rlexp_homogeneous_aux*:
 assumes $v\ x$ = *star* Y @@ Z
 shows Ψ (*eval f v*) $\subseteq$ Ψ (*star* Y @@ *eval f* ($v(x := Z)$))
⟨*proof*⟩

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)
 $\subseteq$ Ψ (*eval* (*Concat* (*Star* y) (*subst* (*Var*(x := z)) f)) v)
 (**is** Ψ *?L* $\subseteq$ Ψ *?R*)
⟨*proof*⟩

2.6 Extension of Arden's lemma to Parikh images

lemma *parikh_img_arden_aux*:
 assumes Ψ (A @@ X $\cup$ B) $\subseteq$ Ψ X
 shows Ψ (A ^^ n @@ B) $\subseteq$ Ψ X
⟨*proof*⟩

lemma *parikh_img_arden*:
 assumes Ψ (A @@ X $\cup$ B) $\subseteq$ Ψ X
 shows Ψ (*star* A @@ B) $\subseteq$ Ψ X
⟨*proof*⟩

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* ⇒ *'a lang set* **where**
parikh_img_eq_class $L \equiv \{L'. \ \Psi \ L' = \Psi \ L\}$

lemma *parikh_img_Union_class*: $\Psi \ A = \Psi \ (\bigcup(parikh\_img\_eq\_class \ A))$
⟨*proof*⟩

lemma *subseteq_comm_subseteq*:
assumes $\Psi \ A \subseteq \Psi \ B$
shows $A \subseteq \bigcup(parikh\_img\_eq\_class \ B)$ (**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, ... 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 $\equiv \forall\, i <$ *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 $\equiv$
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 $\equiv \Psi$ (*eval eq v*) $\subseteq \Psi$ (*v x*)

definition *solves_ineq_sys_comm* :: $'a$ *eq_sys* $\Rightarrow$ $'a$ *valuation* $\Rightarrow$ *bool* **where**
solves_ineq_sys_comm sys v $\equiv \forall\, i <$ *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 $\equiv$
solves_ineq_sys_comm sys sol $\wedge$
($\forall$ *sol*$'$. *solves_ineq_sys_comm sys sol*$'$ $\longrightarrow$ ($\forall$ *x*. Ψ (*sol x*) $\subseteq \Psi$ (*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 $\equiv$ *map* $\circ$ *subst*

lemma *subst_sys_subst*:
assumes *i* $<$ *length sys*
shows (*subst_sys s sys*) ! *i* = *subst s* (*sys* ! *i*)
$\langle$*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 *partial_sol_ineq* :: *nat* $\Rightarrow$ $'a$ *rlexp* $\Rightarrow$ $'a$ *rlexp* $\Rightarrow$ *bool* **where**
partial_sol_ineq x eq sol $\equiv \forall$ *v*. *v x* = *eval sol v* $\longrightarrow$ *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 $\equiv \forall$ *v*. ($\forall$ *x*. *v x* = *eval* (*sols x*) *v*) $\longrightarrow$ *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* ⇒ *'a rlexp* ⇒ *'a rlexp* ⇒ *bool* **where**
  partial_min_sol_one_ineq x eq sol ≡
    partial_sol_ineq x eq sol ∧
    vars sol ⊆ *vars eq* − {*x*} ∧
    (∀ *sol' v'*. *solves_ineq_comm x eq v'* ∧ *v' x* = *eval sol' v'*
              ⟶ Ψ (*eval sol v'*) ⊆ Ψ (*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 $X_n, X_{n+1}, \ldots$) 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* ⇒ *'a eq_sys* ⇒ (*nat* ⇒ *'a rlexp*) ⇒ *bool* **where**
  partial_min_sol_ineq_sys n sys sols ≡
    solution_ineq_sys (*take n sys*) *sols* ∧
    (∀ *i* ≥ *n*. *sols i* = *Var i*) ∧
    (∀ *i* < *n*. ∀ *x* ∈ *vars* (*sols i*). *x* ≥ *n* ∧ *x* < *length sys*) ∧
    (∀ *sols' v'*. (∀ *x*. *v' x* = *eval* (*sols' x*) *v'*)
                ∧ *solves_ineq_sys_comm* (*take n sys*) *v'*
                ⟶ (∀ *i*. Ψ (*eval* (*sols i*) *v'*) ⊆ Ψ (*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*: ∀ *v*. Ψ (*eval f v*) = Ψ (*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*
⟨*proof*⟩

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

lemma *exists_bij_Nt_Var*:
assumes *finite A*
shows $\exists\gamma\ \gamma'$. *bij_Nt_Var A* γ γ'
⟨*proof*⟩

locale *CFG_eq_sys* =
fixes *P* :: ($'n$,$'a$) *Prods*
fixes *S* :: $'n$
fixes γ :: *nat* $\Rightarrow$ $'n$
fixes γ' :: $'n$ $\Rightarrow$ *nat*
assumes *finite_P*: *finite P*
assumes *bij_*γ*_*γ': *bij_Nt_Var* (*Nts P*) γ γ'
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* (γ' *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*)
⟨*proof*⟩

lemma *rlexp_concats_reg*:
assumes $\forall$ *f* $\in$ *set fs*. *reg_eval f*
shows *reg_eval* (*rlexp_concats fs*)
⟨*proof*⟩

lemma *rlexp_syms_reg*: *reg_eval* (*rlexp_syms w*)
⟨*proof*⟩

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 $vars\ (rlexp\_sym\ (Nt\ A)) = \{\gamma'\ A\}$
  ⟨*proof*⟩

lemma *rlexp_sym_vars_Tm*: $vars\ (rlexp\_sym\ (Tm\ x)) = \{\}$
  ⟨*proof*⟩

lemma *rlexp_concats_vars*: $vars\ (rlexp\_concats\ fs) = \bigcup (vars\ `\ set\ fs)$
  ⟨*proof*⟩

lemma *insts'_vars*: $vars\ (rlexp\_syms\ w) \subseteq \gamma'\ `\ nts\_syms\ w$
⟨*proof*⟩

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 $eval\ (rlexp\_sym\ (Nt\ A))\ v = inst\_sym\ L\ (Nt\ A)$
  ⟨*proof*⟩

lemma *rlexp_sym_inst_Tm*: $eval\ (rlexp\_sym\ (Tm\ a))\ v = inst\_sym\ L\ (Tm\ a)$
  ⟨*proof*⟩

lemma *rlexp_concats_concats*:
  assumes $length\ fs = length\ Ls$
      and $\forall\, i < length\ fs.\ eval\ (fs\ !\ i)\ v = Ls\ !\ i$
    shows $eval\ (rlexp\_concats\ fs)\ v = concats\ Ls$
⟨*proof*⟩

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*⟩

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*⟩

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 $CFG\_as\_eq\_sys$: $\exists\, sys.\ CFG\_sys\ sys$
$\langle proof \rangle$

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 $sol \equiv \lambda i.\ if\ i < card\ (Nts\ P)\ then\ Lang\_lfp\ P\ (\gamma\ i)\ else\ \{\}$

lemma $CFG\_sys\_CFL\_is\_sol$:
assumes $CFG\_sys\ sys$
shows $solves\_ineq\_sys\ sys\ sol$
$\langle proof \rangle$

lemma $CFG\_sys\_CFL\_is\_min\_aux$:
assumes $CFG\_sys\ sys$
and $solves\_ineq\_sys\ sys\ sol'$
shows $Lang\_lfp\ P \le (\lambda A.\ sol'\ (\gamma'\ A))$ (**is** $\_ \le\ ?L'$)
$\langle proof \rangle$

lemma $CFG\_sys\_CFL\_is\_min$:
assumes $CFG\_sys\ sys$
and $solves\_ineq\_sys\ sys\ sol'$
shows $sol\ x \subseteq sol'\ x$
$\langle proof \rangle$

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 $CFL\_is\_min\_sol$:
$\exists\, sys.\ (\forall\, eq \in set\ sys.\ reg\_eval\ eq) \wedge (\forall\, eq \in set\ sys.\ \forall\, x \in vars\ eq.\ x < length\ sys)$
$\wedge\ min\_sol\_ineq\_sys\ sys\ sol$
$\langle proof \rangle$

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 sytem 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* ⇒ *'a rlexp* ⇒ *bool* **where**
bipart_rlexp x f ≡ ∃ p q. *reg_eval* p ∧ *reg_eval* q ∧
f = *Union* p (*Concat* q (*Var* x)) ∧ x ∉ *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 ⟹ *reg_eval* f
⟨*proof*⟩

lemma *reg_eval_bipart_rlexp_Variable*: ∃f'. *bipart_rlexp* x f' ∧ *vars* f' = *vars* (*Var* y) ∪ {x}
∧ (∀v. Ψ (*eval* (*Var* y) v) = Ψ (*eval* f' v))
⟨*proof*⟩

lemma *reg_eval_bipart_rlexp_Const*:
assumes *regular_lang* l
shows ∃f'. *bipart_rlexp* x f' ∧ *vars* f' = *vars* (*Const* l) ∪ {x}
∧ (∀v. Ψ (*eval* (*Const* l) v) = Ψ (*eval* f' v))
⟨*proof*⟩

lemma *reg_eval_bipart_rlexp_Union*:
assumes ∃f'. *bipart_rlexp* x f' ∧ *vars* f' = *vars* *f1* ∪ {x} ∧
(∀v. Ψ (*eval* *f1* v) = Ψ (*eval* f' v))
∃f'. *bipart_rlexp* x f' ∧ *vars* f' = *vars* *f2* ∪ {x} ∧
(∀v. Ψ (*eval* *f2* v) = Ψ (*eval* f' v))
shows ∃f'. *bipart_rlexp* x f' ∧ *vars* f' = *vars* (*Union* *f1* *f2*) ∪ {x} ∧
(∀v. Ψ (*eval* (*Union* *f1* *f2*) v) = Ψ (*eval* f' v))
⟨*proof*⟩

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

lemma *reg_eval_bipart_rlexp_Star*:
assumes $\exists f'.\ bipart\_rlexp\ x\ f' \wedge vars\ f' = vars\ f \cup \{x\}$
$\wedge (\forall v.\ \Psi\ (eval\ f\ v) = \Psi\ (eval\ f'\ v))$
shows $\exists f'.\ bipart\_rlexp\ x\ f' \wedge vars\ f' = vars\ (Star\ f) \cup \{x\}$
$\wedge (\forall v.\ \Psi\ (eval\ (Star\ f)\ v) = \Psi\ (eval\ f'\ v))$
$\langle proof \rangle$

lemma *reg_eval_bipart_rlexp*: $reg\_eval\ f \Longrightarrow$
$\exists f'.\ bipart\_rlexp\ x\ f' \wedge vars\ f' = vars\ f \cup \{x\} \wedge$
$(\forall s.\ \Psi\ (eval\ f\ s) = \Psi\ (eval\ f'\ s))$
$\langle proof \rangle$

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 $x :: nat$
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*: $x \notin vars\ p$
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*: $reg\_eval\ sol$
$\langle proof \rangle$

lemma *sol_vars*: $vars\ sol \subseteq vars\ eq - \{x\}$
⟨*proof*⟩

lemma *sol_is_sol_ineq*: *partial_sol_ineq x eq sol*
⟨*proof*⟩

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*⟩

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

lemma *sol_is_minimal_reg_sol*:
  reg_eval sol ∧ *partial_min_sol_one_ineq x eq sol*
  ⟨*proof*⟩

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 ∃ *sol*. *reg_eval sol* ∧ *partial_min_sol_one_ineq x eq sol*
⟨*proof*⟩

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* ⇒ *′a rlexp*
    and *sol_r* :: *′a rlexp*
  assumes *eqs_reg*: ∀ *eq* ∈ *set sys*. *reg_eval eq*
    and *sys_valid*: ∀ *eq* ∈ *set sys*. ∀ *x* ∈ *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*: ∀ *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 $Suc\ r$ 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 $sys' \equiv subst\_sys\ sols\ sys$
abbreviation $sols' \equiv \lambda i.\ subst\ (Var(r := sol\_r))\ (sols\ i)$

lemma *sols'_r*: $sols'\ r = sol\_r$
⟨*proof*⟩

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*: $\forall i.\ reg\_eval\ (sols'\ i)$
⟨*proof*⟩

lemma *sols'_is_sol*: $solution\_ineq\_sys\ (take\ (Suc\ r)\ sys)\ sols'$
⟨*proof*⟩

lemma *sols'_min*: $\forall sols2\ v2.\ (\forall x.\ v2\ x = eval\ (sols2\ x)\ v2)$
$\wedge\ solves\_ineq\_sys\_comm\ (take\ (Suc\ r)\ sys)\ v2$
$\longrightarrow (\forall i.\ \Psi\ (eval\ (sols'\ i)\ v2) \subseteq \Psi\ (v2\ i))$
⟨*proof*⟩

lemma *sols'_vars_gt_r*: $\forall i \geq Suc\ r.\ sols'\ i = Var\ i$
⟨*proof*⟩

lemma *sols'_vars_leq_r*: $\forall i < Suc\ r.\ \forall x \in vars\ (sols'\ i).\ x \geq Suc\ r \wedge x < length\ sys$
⟨*proof*⟩

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*: $partial\_min\_sol\_ineq\_sys\ (Suc\ r)\ sys\ sols'$
⟨*proof*⟩

lemma *exists_min_sol_Suc_r*:
$\exists sols'.\ partial\_min\_sol\_ineq\_sys\ (Suc\ r)\ sys\ sols' \wedge (\forall i.\ reg\_eval\ (sols'\ i))$
⟨*proof*⟩

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*: $\forall\, eq \in set\ sys.\ reg\_eval\ eq$
and *sys_valid*: $\forall\, eq \in set\ sys.\ \forall\, x \in vars\ eq.\ x < length\ sys$
and *r_valid*: $r \leq length\ sys$
shows $\exists\, sols.\ partial\_min\_sol\_ineq\_sys\ r\ sys\ sols \wedge (\forall\, i.\ reg\_eval\ (sols\ i))$
$\langle proof \rangle$

lemma *exists_minimal_reg_sol_sys*:
assumes *eqs_reg*: $\forall\, eq \in set\ sys.\ reg\_eval\ eq$
and *sys_valid*: $\forall\, eq \in set\ sys.\ \forall\, x \in vars\ eq.\ x < length\ sys$
shows $\exists\, sols.\ min\_sol\_ineq\_sys\_comm\ sys\ sols \wedge (\forall\, i.\ regular\_lang\ (sols\ i))$
$\langle proof \rangle$

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 $\exists\, L'.\ regular\_lang\ L' \wedge \Psi\ L = \Psi\ L'$
$\langle proof \rangle$

lemma *singleton_set_mset_subset*: **fixes** $X\ Y :: 'a\ list\ set$
assumes $\forall\, xs \in X.\ set\ xs \subseteq \{a\}$ $mset\ `\ X \subseteq mset\ `\ Y$
shows $X \subseteq Y$
$\langle proof \rangle$

lemma *singleton_set_mset_eq*: **fixes** $X\ Y :: 'a\ list\ set$
assumes $\forall\, xs \in X.\ set\ xs \subseteq \{a\}$ $mset\ `\ X = mset\ `\ Y$
shows $X = Y$
$\langle proof \rangle$

lemma *derives_tms_syms_subset*:
$P \vdash \alpha \Rightarrow\!*\ \gamma \Longrightarrow tms\_syms\ \gamma \subseteq tms\_syms\ \alpha \cup Tms\ P$
$\langle proof \rangle$

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

corollary *CFL_1_Tm_regular*:
assumes $CFL\ (TYPE('n))\ L$ **and** $\forall\, w \in L.\ set\ w \subseteq \{a\}$
shows $regular\_lang\ L$
$\langle proof \rangle$

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.