Parikh's theorem

Fabian Lehr

June 10, 2025

Abstract

This library introduces Parikh images of formal languages and proves Parikh's theorem. The proof 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

theory *Reg-Lang-Exp*
  imports
    Regular−Sets.Regular-Set
    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* ⇒ *'a lang*

primrec *eval* :: *'a rlexp* ⇒ *'a valuation* ⇒ *'a lang* **where**
  eval (*Var n*) *v* = *v n* |
  eval (*Const l*) - = *l* |
  eval (*Union f g*) *v* = *eval f v* ∪ *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* ⇒ *nat set* **where**
  vars (*Var n*) = {*n*} |
  vars (*Const* -) = {} |
  vars (*Union f g*) = *vars f* ∪ *vars g* |
  vars (*Concat f g*) = *vars f* ∪ *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 $eval\ (subst\ upd\ f)\ v = eval\ f\ v'$
  using *assms* **by** (*induction rule*: *rlexp.induct*) *auto*

lemma *substitution-lemma-update*:
  $eval\ (subst\ (Var(x := f'))\ f)\ v = eval\ f\ (v(x := eval\ f'\ v))$
  using *substitution-lemma*[*of* $v(x := eval\ f'\ v)$] **by** *force*

lemma *subst-id*: $eval\ (subst\ Var\ f)\ v = eval\ f\ v$
  using *substitution-lemma*[*of* v] **by** *simp*

lemma *vars-subst*: $vars\ (subst\ upd\ f) = (\bigcup x \in vars\ f.\ vars\ (upd\ x))$
  by (*induction* f) *auto*

lemma *vars-subst-upper*: $vars\ (subst\ upd\ f) \subseteq (\bigcup x.\ vars\ (upd\ x))$
  using *vars-subst* **by** *force*

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

lemma *vars-subst-upd-aux*:
  assumes $x \in vars\ f$
  shows $vars\ f - \{x\} \cup vars\ fx \subseteq vars\ (subst\ (Var(x := fx))\ f)$
proof
  fix y
  let $?upd = Var(x := fx)$
  assume *as*: $y \in vars\ f - \{x\} \cup vars\ fx$
  then show $y \in vars\ (subst\ ?upd\ f)$
  proof (*cases* $y \in vars\ f - \{x\}$)
    case *True*
    then show *?thesis* **using** *vars-subst* **by** *fastforce*

  next
    case *False*
    with *as* **have** $y \in$ *vars fx* **by** *blast*
    with *assms* **show** *?thesis* **using** *vars-subst* **by** *fastforce*
  qed
qed

lemma *vars-subst-upd*:
  assumes $x \in$ *vars f*
  shows *vars* (*subst* (*Var*(x := *fx*)) *f*) = *vars f* − {x} ∪ *vars fx*
  using *assms vars-subst-upd-upper vars-subst-upd-aux* **by** *blast*

lemma *eval-vars*:
  assumes ∀ $i \in$ *vars f*. $s\ i = s'\ i$
  shows *eval f s* = *eval f s*$'$
  using *assms* **by** (*induction f*) *auto*

lemma *eval-vars-subst*:
  assumes ∀ $i \in$ *vars f*. $v\ i$ = *eval* (*upd i*) v
  shows *eval* (*subst upd f*) v = *eval f v*
proof −
  let *?v*$'$ = λ*i*. *if* $i \in$ *vars f then v i else eval* (*upd i*) v
  let *?v*$''$ = λ*i*. *eval* (*upd i*) v
  have *v*$'$-*v*$''$: *?v*$'$ *i* = *?v*$''$ *i* **for** *i* **using** *assms* **by** *simp*
  then have *v-v*$''$: ∀ *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 ... = *eval* (*subst upd f*) v
    using *assms substitution-lemma*[*OF v-v*$''$, *of f*] **by** (*simp add*: *eval-vars*)
  finally show *?thesis* **by** *simp*
qed

1.3 Monotonicity

lemma *rlexp-mono-aux*:
  assumes ∀ $i \in$ *vars f*. $v\ i \subseteq v'\ i$
  shows *eval f v* ⊆ *eval f v*$'$
using *assms* **proof** (*induction 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*+

lemma *rlexp-mono*:
  fixes *f* :: $'$*a rlexp*
  shows *mono* (*eval f*)
  using *rlexp-mono-aux* **by** (*metis le-funD monoI*)

1.4 Continuity

lemma *langpow-mono*:

```
  fixes A :: 'a lang
  assumes A ⊆ B
  shows A ^^ n ⊆ B ^^ n
using assms conc-mono[of A B] by (induction n) auto

lemma rlexp-cont-aux1:
  assumes ∀ i. v i ≤ v (Suc i)
      and w ∈ (⋃ i. eval f (v i))
    shows w ∈ eval f (λx. ⋃ i. v i x)
proof −
  from assms(2) obtain n where n-intro: w ∈ eval f (v n) by auto
  have v n x ⊆ (⋃ i. v i x) for x by auto
  with n-intro show ?thesis
    using rlexp-mono-aux[where v=v n and v'=λx. ⋃ i. v i x] by auto
qed

lemma langpow-Union-eval:
  assumes ∀ i. v i ≤ v (Suc i)
      and w ∈ (⋃ i. eval f (v i)) ^^ n
    shows w ∈ (⋃ i. eval f (v i) ^^ n)
using assms 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 ∈ (⋃ i. eval f (v i)) ∧ u' ∈ (⋃ i. eval f (v i)) ^^ n by fastforce
  with Suc have u ∈ (⋃ i. eval f (v i)) ∧ u' ∈ (⋃ i. eval f (v i) ^^ n) by auto
  then obtain i j where i-intro: u ∈ eval f (v i) and j-intro: u' ∈ eval f (v j) ^^
n by blast
  let ?m = max i j
  from i-intro Suc.prems(1) rlexp-mono-aux have 1: u ∈ eval f (v ?m)
    by (metis le-fun-def lift-Suc-mono-le max.cobounded1 subset-eq)
  from Suc.prems(1) rlexp-mono-aux have eval f (v j) ⊆ eval f (v ?m)
    by (metis le-fun-def lift-Suc-mono-le max.cobounded2)
  with j-intro langpow-mono have 2: u' ∈ eval f (v ?m) ^^ n by auto
  from 1 2 show ?case using w-decomp by auto
qed

lemma rlexp-cont-aux2:
  assumes ∀ i. v i ≤ v (Suc i)
      and w ∈ eval f (λx. ⋃ i. v i x)
    shows w ∈ (⋃ i. eval f (v i))
using assms proof (induction arbitrary: w rule: rlexp.induct)
  case (Concat f g)
  then obtain u u' where w-decomp: w = u@u'
    and u ∈ eval f (λx. ⋃ i. v i x) ∧ u' ∈ eval g (λx. ⋃ i. v i x) by auto
  with Concat have u ∈ (⋃ i. eval f (v i)) ∧ u' ∈ (⋃ i. eval g (v i)) by auto
  then obtain i j where i-intro: u ∈ eval f (v i) and j-intro: u' ∈ eval g (v j) by
```

blast
let *?m* = *max i j*
from *i-intro Concat.prems(1) rlexp-mono-aux* **have** $u \in$ *eval f* (*v ?m*)
by (*metis le-fun-def lift-Suc-mono-le max.cobounded1 subset-eq*)
moreover from *j-intro Concat.prems(1) rlexp-mono-aux* **have** $u' \in$ *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*: $w \in$ (*eval f* ($\lambda x. \bigcup i.\ v\ i\ x$)) ^^ *n*
using *eval.simps(5) star-pow* **by** *blast*
with *Star* **have** $w \in$ ($\bigcup i.$ *eval f* (*v i*)) ^^ *n* **using** *langpow-mono* **by** *blast*
with *Star.prems* **have** $w \in$ ($\bigcup i.$ *eval f* (*v i*) ^^ *n*) **using** *langpow-Union-eval* **by** *auto*
then show *?case* **by** (*auto simp add: star-def*)
qed *fastforce+*

lemma *rlexp-cont*:
assumes $\forall i.\ v\ i \le v\ (Suc\ i)$
shows *eval f* ($\lambda x. \bigcup i.\ v\ i\ x$) = ($\bigcup i.$ *eval f* (*v i*))
proof
from *assms* **show** *eval f* ($\lambda x. \bigcup i.\ v\ i\ x$) $\subseteq$ ($\bigcup i.$ *eval f* (*v i*)) **using** *rlexp-cont-aux2* **by** *auto*
from *assms* **show** ($\bigcup i.$ *eval f* (*v i*)) $\subseteq$ *eval f* ($\lambda x. \bigcup i.\ v\ i\ x$) **using** *rlexp-cont-aux1* **by** *blast*
qed

1.5 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 underapproximation, 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 function 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*)
using *assms* **proof** (*induction 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* $\Longrightarrow \exists g$. *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 (*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* ‘ *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.prems* $*$ **have** *reg-eval ?g′* $\wedge \bigcup$ (*vars* ‘ *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* ‘ *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*

1.6 Constant regular language functions

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 Ψ L, is then the set of Parikh vectors of all words in the language.

abbreviation *parikh-vec* **where**
parikh-vec $\equiv$ *mset*

definition *parikh-img* :: *'a lang* $\Rightarrow$ *'a multiset set* (Ψ) **where**
Ψ L $\equiv$ *parikh-vec* ‘ L

lemma *parikh-img-Un* [*simp*]: Ψ ($L1 \cup L2$) = Ψ $L1 \cup \Psi$ $L2$
by (*auto simp add*: *parikh-img-def*)

lemma *parikh-img-UNION*: Ψ ($\bigcup$(L ‘ I)) = $\bigcup$ (($\lambda i.$ Ψ (L i)) ‘ I)
by (*auto simp add*: *parikh-img-def*)

lemma *parikh-img-Star-pow*: $m \in \Psi$ (*eval* (*Star f*) *v*) $\Longrightarrow \exists n.$ $m \in \Psi$ (*eval f v* ^^ *n*)
proof −
assume $m \in \Psi$ (*eval* (*rlexp.Star f*) *v*)
then have $m \in \Psi$ (*star* (*eval f v*)) **by** *simp*
then show *?thesis* **unfolding** *star-def* **by** (*simp add*: *parikh-img-UNION*)
qed

lemma *parikh-img-conc*: Ψ ($L1$ @@ $L2$) = { $m1 + m2$ | $m1$ $m2$. $m1 \in \Psi$ $L1 \wedge m2 \in \Psi$ $L2$ }
unfolding *parikh-img-def* **by** *force*

lemma *parikh-img-commut*: Ψ ($L1$ @@ $L2$) = Ψ ($L2$ @@ $L1$)
proof −
have { $m1 + m2$ | $m1$ $m2$. $m1 \in \Psi$ $L1 \wedge m2 \in \Psi$ $L2$ } =
{ $m2 + m1$ | $m1$ $m2$. $m1 \in \Psi$ $L1 \wedge m2 \in \Psi$ $L2$ }
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*: $A \subseteq B \Longrightarrow \Psi$ $A \subseteq \Psi$ B
unfolding *parikh-img-def* **by** *fast*

lemma *parikh-img-mono-eq*: $A = B \Longrightarrow \Psi$ $A = \Psi$ B
using *parikh-img-mono* **by** *blast*

lemma *parikh-conc-right-subset*: $\Psi\ A \subseteq \Psi\ B \Longrightarrow \Psi\ (A\ @@\ C) \subseteq \Psi\ (B\ @@\ C)$
  by (*auto simp add*: *parikh-img-conc*)

lemma *parikh-conc-left-subset*: $\Psi\ A \subseteq \Psi\ B \Longrightarrow \Psi\ (C\ @@\ A) \subseteq \Psi\ (C\ @@\ B)$
  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*: $\Psi\ A = \Psi\ B \Longrightarrow \Psi\ (A\ @@\ C) = \Psi\ (B\ @@\ C)$
  by (*auto simp add*: *parikh-img-conc*)

lemma *parikh-conc-left*: $\Psi\ A = \Psi\ B \Longrightarrow \Psi\ (C\ @@\ A) = \Psi\ (C\ @@\ B)$
  by (*auto simp add*: *parikh-img-conc*)

lemma *parikh-pow-mono*: $\Psi\ A \subseteq \Psi\ B \Longrightarrow \Psi\ (A\ \widehat{}\widehat{}\ n) \subseteq \Psi\ (B\ \widehat{}\widehat{}\ n)$
  by (*induction n*) (*auto simp add*: *parikh-img-conc*)

lemma *parikh-star-mono*:
  assumes $\Psi\ A \subseteq \Psi\ B$
  shows $\Psi\ (star\ A) \subseteq \Psi\ (star\ B)$
proof
  fix v
  assume $v \in \Psi\ (star\ A)$
  then obtain w **where** *w-intro*: $parikh\text{-}vec\ w = v \wedge w \in star\ A$ **unfolding** *parikh-img-def* **by** *blast*
  then obtain n **where** $w \in A\ \widehat{}\widehat{}\ n$ **unfolding** *star-def* **by** *blast*
  then have $v \in \Psi\ (A\ \widehat{}\widehat{}\ n)$ **using** *w-intro* **unfolding** *parikh-img-def* **by** *blast*
  with *assms* **have** $v \in \Psi\ (B\ \widehat{}\widehat{}\ n)$ **using** *parikh-pow-mono* **by** *blast*
  then show $v \in \Psi\ (star\ B)$ **unfolding** *star-def* **using** *parikh-img-UNION* **by** *fastforce*
qed

lemma *parikh-star-mono-eq*:
  assumes $\Psi\ A = \Psi\ B$
  shows $\Psi\ (star\ A) = \Psi\ (star\ B)$
  using *parikh-star-mono* **by** (*metis Orderings.order-eq-iff assms*)

lemma *parikh-img-subst-mono*:
  assumes $\forall\, i.\ \Psi\ (eval\ (A\ i)\ v) \subseteq \Psi\ (eval\ (B\ i)\ v)$
  shows $\Psi\ (eval\ (subst\ A\ f)\ v) \subseteq \Psi\ (eval\ (subst\ B\ f)\ v)$
using *assms* **proof** (*induction f*)
  case (*Concat f1 f2*)

then have Ψ (*eval* (*subst A f1*) *v* @@ *eval* (*subst A f2*) *v*)
$\subseteq$ Ψ (*eval* (*subst B f1*) *v* @@ *eval* (*subst B f2*) *v*)
using *parikh-conc-subset* **by** *blast*
then show *?case* **by** *simp*
next
case (*Star f*)
then have Ψ (*star* (*eval* (*subst A f*) *v*)) $\subseteq$ Ψ (*star* (*eval* (*subst B f*) *v*))
using *parikh-star-mono* **by** *blast*
then show *?case* **by** *simp*
qed *auto*

lemma *parikh-img-subst-mono-upd*:
assumes Ψ (*eval A v*) $\subseteq$ Ψ (*eval B v*)
shows Ψ (*eval* (*subst* (*Var*($x := A$)) *f*) *v*) $\subseteq$ Ψ (*eval* (*subst* (*Var*($x := B$)) *f*) *v*)
using *parikh-img-subst-mono*[*of Var*($x := A$) *v Var*($x := B$)] *assms* **by** *auto*

lemma *parikh-img-subst-mono-eq*:
assumes $\forall i$. Ψ (*eval* (*A i*) *v*) = Ψ (*eval* (*B i*) *v*)
shows Ψ (*eval* (*subst* (λi. *A i*) *f*) *v*) = Ψ (*eval* (*subst* (λi. *B i*) *f*) *v*)
using *parikh-img-subst-mono assms* **by** *blast*

lemma *rlexp-mono-parikh*:
assumes $\forall i \in$ *vars f*. Ψ (*v i*) $\subseteq$ Ψ (v' *i*)
shows Ψ (*eval f v*) $\subseteq$ Ψ (*eval f* v')
using *assms* **proof** (*induction rule*: *rlexp.induct*)
case (*Concat f1 f2*)
then have Ψ (*eval f1 v* @@ *eval f2 v*) $\subseteq$ Ψ (*eval f1* v' @@ *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 $\forall i \in$ *vars f*. Ψ (*v i*) = Ψ (v' *i*)
shows Ψ (*eval f v*) = Ψ (*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 $v \in \Psi$ (($A \cup B$) ^^ *n*)
shows $v \in \Psi$ ($\bigcup i \le n$. *A* ^^ *i* @@ *B* ^^ ($n-i$))
using *assms* **proof** (*induction n arbitrary*: *v*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then obtain *w* **where** *w-intro*: $w \in (A \cup B)$ ^^ (*Suc n*) $\wedge$ *parikh-vec* $w = v$

unfolding *parikh-img-def* **by** *auto*
then obtain *w1 w2* **where** *w1-w2-intro*: $w = w1@w2 \wedge w1 \in A \cup B \wedge w2 \in (A \cup B)$ ^^ n **by** *fastforce*
let *?v1* = *parikh-vec w1* **and** *?v2* = *parikh-vec w2*
from *w1-w2-intro* **have** $?v2 \in \Psi\ ((A \cup B)$ ^^ $n)$ **unfolding** *parikh-img-def* **by** *blast*
with *Suc.IH* **have** $?v2 \in \Psi\ (\bigcup i \leq n.\ A$ ^^ i @@ B ^^ $(n-i))$ **by** *auto*
then obtain *w2'* **where** *w2'-intro*: *parikh-vec w2'* = *parikh-vec w2* $\wedge$
$w2' \in (\bigcup i \leq n.\ A$ ^^ i @@ B ^^ $(n-i))$ **unfolding** *parikh-img-def* **by** *fastforce*
then obtain i **where** *i-intro*: $i \leq n \wedge w2' \in A$ ^^ i @@ B ^^ $(n-i)$ **by** *blast*
from *w1-w2-intro w2'-intro* **have** *parikh-vec w* = *parikh-vec* $(w1@w2')$
by *simp*
moreover have *parikh-vec* $(w1@w2') \in \Psi\ (\bigcup i \leq Suc\ n.\ A$ ^^ i @@ B ^^ $(Suc\ n-i))$
proof (*cases* $w1 \in A$)
case *True*
with *i-intro* **have** *Suc-i-valid*: $Suc\ i \leq Suc\ n$ **and** $w1@w2' \in A$ ^^ $(Suc\ i)$ @@ B ^^ $(Suc\ n - Suc\ i)$
by (*auto simp add*: *conc-assoc*)
then have *parikh-vec* $(w1@w2') \in \Psi\ (A$ ^^ $(Suc\ i)$ @@ B ^^ $(Suc\ n - Suc\ i))$
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** *parikh-vec* $(w1@w2') \in \Psi\ (B$ @@ A ^^ i @@ B ^^ $(n-i))$
unfolding *parikh-img-def* **by** *blast*
then have *parikh-vec* $(w1@w2') \in \Psi\ (A$ ^^ i @@ B ^^ $(Suc\ n-i))$
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 $v \in \Psi\ (star\ (A \cup B))$
shows $v \in \Psi\ (star\ A$ @@ $star\ B)$
proof −
from *assms* **have** $v \in (\bigcup n.\ \Psi\ ((A \cup B)$ ^^ $n))$
unfolding *star-def* **using** *parikh-img-UNION* **by** *metis*
then obtain n **where** $v \in \Psi\ ((A \cup B)$ ^^ $n)$ **by** *blast*
then have $v \in \Psi\ (\bigcup i \leq n.\ A$ ^^ i @@ B ^^ $(n-i))$
using *parikh-img-union-pow-aux1* **by** *auto*
then have $v \in (\bigcup i \leq n.\ \Psi\ (A$ ^^ i @@ B ^^ $(n-i)))$ **using** *parikh-img-UNION* **by** *metis*
then obtain i **where** $i \leq n \wedge v \in \Psi\ (A$ ^^ i @@ B ^^ $(n-i))$ **by** *blast*
then obtain w **where** *w-intro*: *parikh-vec* $w = v \wedge w \in A$ ^^ i @@ B ^^ $(n-i)$

unfolding *parikh-img-def* by *blast*
then obtain *w1 w2* where *w-decomp*: $w{=}w1@w2 \wedge w1 \in A \frown\frown i \wedge w2 \in B \frown\frown (n{-}i)$ 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*: $parikh\text{-}vec\ w = v \wedge w \in star\ A\ @@\ star\ B$
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** $w1 \in A \frown\frown i$ **and** *w2-intro*: $w2 \in B \frown\frown j$ **unfolding** *star-def* **by** *blast*
then have *w1-in-union*: $w1 \in (A \cup B) \frown\frown i$ **using** *langpow-mono* **by** *blast*
from *w2-intro* **have** $w2 \in (A \cup B) \frown\frown j$ **using** *langpow-mono* **by** *blast*
with *w1-in-union w-decomp* **have** $w \in (A \cup B) \frown\frown (i{+}j)$ **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*: $\Psi\ ((A\ @@\ B) \frown\frown n) \subseteq \Psi\ (A \frown\frown n\ @@\ B \frown\frown n)$
proof (*induction n*)
case (*Suc n*)
then have $\Psi\ ((A\ @@\ B) \frown\frown n\ @@\ A\ @@\ B) \subseteq \Psi\ (A \frown\frown n\ @@\ B \frown\frown n\ @@\ A\ @@\ B)$
using *parikh-conc-right-subset conc-assoc* **by** *metis*
also have $\ldots = \Psi\ (A \frown\frown n\ @@\ A\ @@\ B \frown\frown n\ @@\ B)$
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*: Ψ (*star* (*A* @@ *B*)) $\subseteq$ Ψ (*star A* @@ *star B*)
proof
  fix *v*
  assume $v \in \Psi$ (*star* (*A* @@ *B*))
  then have $\exists\, n.\ v \in \Psi$ ((*A* @@ *B*) ^^ *n*) **unfolding** *star-def* **by** (*simp add*: *parikh-img-UNION*)
  then obtain *n* **where** $v \in \Psi$ ((*A* @@ *B*) ^^ *n*) **by** *blast*
  with *parikh-img-conc-pow* **have** $v \in \Psi$ (*A* ^^ *n* @@ *B* ^^ *n*) **by** *fast*
  then have $v \in \Psi$ (*A* ^^ *n* @@ *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 $v \in \Psi$ (*star A* @@ *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*: Ψ ((*A* @@ *B*) ^^ *Suc n*) $\subseteq$ Ψ (*star A* @@ *star B* @@ *B*)
proof
  fix *v*
  assume $v \in \Psi$ ((*A* @@ *B*) ^^ *Suc n*)
  with *parikh-img-conc-pow* **have** $v \in \Psi$ (*A* ^^ *Suc n* @@ *B* ^^ *n* @@ *B*)
    by (*metis conc-pow-comm lang-pow.simps*(*2*) *subsetD*)
  then have $v \in \Psi$ (*star A* @@ *B* ^^ *n* @@ *B*)
    unfolding *star-def* **using** *parikh-conc-right-subset*
    by (*metis* (*no-types*, *lifting*) *Sup-upper parikh-img-mono rangeI subset-eq*)
  then show $v \in \Psi$ (*star A* @@ *star B* @@ *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*:
  Ψ (*star* (*star E* @@ *F*)) $\subseteq$ Ψ ({[]} $\cup$ *star E* @@ *star F* @@ *F*)
proof
  fix *v*
  assume $v \in \Psi$ (*star* (*star E* @@ *F*))
  then have $\exists\, n.\ v \in \Psi$ ((*star E* @@ *F*) ^^ *n*)
    unfolding *star-def* **by** (*simp add*: *parikh-img-UNION*)
  then obtain *n* **where** *v-in-pow-n*: $v \in \Psi$ ((*star E* @@ *F*) ^^ *n*) **by** *blast*
  show $v \in \Psi$ ({[]} $\cup$ *star E* @@ *star F* @@ *F*)
  proof (*cases n*)
    case *0*
    with *v-in-pow-n* **have** $v =$ *parikh-vec* [] **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-in-pow-n* **have** $v \in \Psi$ (*star* (*star E*) @@ *star F* @@ *F*) **by** *blast*

then show *?thesis* **by** (*metis UnCI parikh-img-Un star-idemp*)
qed
qed

lemma *parikh-img-star2-aux2*: Ψ (*star E* @@ *star F* @@ *F*) $\subseteq$ Ψ (*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 Ψ (*star E* @@ *F* @@ *star F*) $\subseteq$ Ψ (*star E* @@ *F* @@ *star* (*star E* @@ *F*))
using *parikh-conc-left-subset parikh-img-mono parikh-star-mono* **by** *meson*
also have ... $\subseteq$ Ψ (*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*: Ψ (*star* (*star E* @@ *F*)) = Ψ ({[]} $\cup$ *star E* @@ *star F* @@ *F*)
proof
from *parikh-img-star2-aux1*
show Ψ (*star* (*star E* @@ *F*)) $\subseteq$ Ψ ({[]} $\cup$ *star E* @@ *star F* @@ *F*) **.**
from *parikh-img-star2-aux2*
show Ψ ({[]} $\cup$ *star E* @@ *star F* @@ *F*) $\subseteq$ Ψ (*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 functions

lemma *rlexp-homogeneous-aux*:
assumes *v x* = *star Y* @@ *Z*
shows Ψ (*eval f v*) $\subseteq$ Ψ (*star Y* @@ *eval f* (*v*(*x* := *Z*)))
using *assms* **proof** (*induction f*)
case (*Var y*)
show *?case*
proof (*cases x* = *y*)
case *True*
with *Var* **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* $\subseteq$ *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*) $\subseteq$ Ψ (*star Y* @@ *eval f* ($v(x := Z)$) $\cup$
star Y @@ *eval g* ($v(x := Z)$))
by *fastforce*
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*) $\subseteq$ 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*)) $\subseteq$ Ψ (*star* (*star Y* @@ *eval f* ($v(x := Z)$)))
using *parikh-star-mono* **by** *metis*
also from *parikh-img-conc-star* **have** ... $\subseteq$ 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:

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 –
let $?v' = v(x :=$ *star* (*eval y v*) @@ *eval z v*)
have Ψ *?L* = Ψ (*eval f* $?v'$) **using** *substitution-lemma-update*[**where** *f=f*] **by** *simp*
also have ... $\subseteq$ 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 Ψ (A @@ $X \cup B$) $\subseteq$ Ψ X
shows Ψ (A ^^ n @@ B) $\subseteq$ Ψ X
using *assms* **proof** (*induction n*)
case *0*
then show *?case* **by** *auto*

next
  case $(Suc\ n)$
  then have $\Psi\ (A\ \frown\frown (Suc\ n)\ @@\ B) \subseteq \Psi\ (A\ @@\ A\ \frown\frown n\ @@B)$
    by $(simp\ add{:}\ conc\text{-}assoc)$
  moreover from *Suc parikh-conc-left* **have** $\ldots \subseteq \Psi\ (A\ @@\ X)$
    by $(metis\ conc\text{-}Un\text{-}distrib(1)\ parikh\text{-}img\text{-}Un\ sup.orderE\ sup.orderI)$
  moreover from *Suc.prems* **have** $\ldots \subseteq \Psi\ X$ **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 $\exists\, n.\ x \in \Psi\ (A\ \frown\frown n\ @@\ B)$
   unfolding *star-def* **by** $(simp\ add{:}\ conc\text{-}UNION\text{-}distrib(2)\ parikh\text{-}img\text{-}UNION)$
  then obtain n **where** $x \in \Psi\ (A\ \frown\frown n\ @@\ B)$ **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**
  $parikh\text{-}img\text{-}eq\text{-}class\ L \equiv \{L'.\ \Psi\ L' = \Psi\ L\}$

lemma *parikh-img-Union-class*: $\Psi\ A = \Psi\ (\bigcup (parikh\text{-}img\text{-}eq\text{-}class\ A))$
proof
  let $?A' = \bigcup (parikh\text{-}img\text{-}eq\text{-}class\ A)$
  show $\Psi\ A \subseteq \Psi\ ?A'$
   unfolding *parikh-img-eq-class-def* **by** $(simp\ add{:}\ Union\text{-}upper\ parikh\text{-}img\text{-}mono)$
  show $\Psi\ ?A' \subseteq \Psi\ A$
  proof
    fix v
    assume $v \in \Psi\ ?A'$
    then obtain a **where** *a-intro*: $parikh\text{-}vec\ a = v \land a \in\ ?A'$
      unfolding *parikh-img-def* **by** *blast*
    then obtain L **where** *L-intro*: $a \in L \land L \in parikh\text{-}img\text{-}eq\text{-}class\ A$
      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 *subseteq-comm-subseteq*:

assumes $\Psi\ A \subseteq \Psi\ B$
shows $A \subseteq \bigcup(\textit{parikh-img-eq-class}\ B)$ (**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'*: *parikh-vec a* $\in \Psi$ *?B'* **unfolding** *parikh-img-def* **by** *fast*
then have $\exists\, b.$ *parikh-vec b = parikh-vec a* $\wedge$ *b* $\in$ *?B'*
unfolding *parikh-img-def* **by** *fastforce*
then obtain *b* **where** *b-intro*: *parikh-vec b = parikh-vec a* $\wedge$ *b* $\in$ *?B'* **by** *blast*
with *vec-a-in-B'* **have** Ψ (*?B'* $\cup \{a\}) = \Psi$ *?B'***unfolding** *parikh-img-def* **by** *blast*
with *parikh-img-Union-class* **have** Ψ (*?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 *Eq-Sys*
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 correspond two systems of equations - one for both of the types - and that the language defined by the CFG is a minimal solution of both systems.

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 expression 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*)
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 $\equiv \forall$ *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. A partial solution of the whole system is then defined as follows:

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* $\Rightarrow$ *'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-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 $X_n, X_{n+1}, \ldots$) 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* $\Rightarrow$ *'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*

proof −
  from *minimal-sol* **have** $vars\ sol \subseteq vars\ g - \{x\}$
    unfolding *partial-min-sol-one-ineq-def* **using** *same-vars* **by** *blast*
  moreover from *same-parikh-img minimal-sol* **have** *partial-sol-ineq x g 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** $\forall\, sol'\ v'.\ solves\text{-}ineq\text{-}comm\ x\ g\ v' \wedge v'\ x = eval\ sol'\ v'$
  $\longrightarrow \Psi\ (eval\ sol\ v') \subseteq \Psi\ (v'\ x)$
    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 solution of systems of equations

We show that each CFG induces a system of equations of the first type, i.e. without Parikh images, such that 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 $bij\text{-}Nt\text{-}Var :: 'n\ set \Rightarrow (nat \Rightarrow 'n) \Rightarrow ('n \Rightarrow nat) \Rightarrow bool$ **where**
  $bij\text{-}Nt\text{-}Var\ A\ \gamma\ \gamma' \equiv bij\text{-}betw\ \gamma\ \{..<\ card\ A\}\ A \wedge bij\text{-}betw\ \gamma'\ A\ \{..<\ card\ A\}$
  $\wedge (\forall x \in \{..<\ card\ A\}.\ \gamma'\ (\gamma\ x) = x) \wedge (\forall y \in A.\ \gamma\ (\gamma'\ y) = y)$

lemma *exists-bij-Nt-Var*: $finite\ A \Longrightarrow \exists \gamma\ \gamma'.\ bij\text{-}Nt\text{-}Var\ A\ \gamma\ \gamma'$
proof −
  assume *finite A*
  then have $\exists \gamma.\ bij\text{-}betw\ \gamma\ \{..<\ card\ A\}\ A$ **by** (*simp add*: *bij-betw-iff-card*)
  then obtain γ **where** *1*: $bij\text{-}betw\ \gamma\ \{..<\ card\ A\}\ A$ **by** *blast*
  let $?\gamma' = the\text{-}inv\text{-}into\ \{..<\ card\ A\}\ \gamma$
  from *the-inv-into-f-f 1* **have** *2*: $\forall x \in \{..<\ card\ A\}.\ ?\gamma'\ (\gamma\ x) = x$ **unfolding** *bij-betw-def* **by** *fast*
  from *bij-betw-the-inv-into*[*OF 1*] **have** *3*: $bij\text{-}betw\ ?\gamma'\ A\ \{..<\ card\ A\}$ **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-γ-γ'*: $bij\text{-}Nt\text{-}Var\ (Nts\ P)\ \gamma\ \gamma'$
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 $\forall f \in$ *set fs. reg-eval f*
shows *reg-eval* (*rlexp-concats fs*)
using *assms epsilon-regular* **unfolding** *rlexp-concats-def* **by** (*induction fs*) *auto*

lemma *rlexp-syms-reg*: *reg-eval* (*rlexp-syms w*)
proof −
from *rlexp-sym-reg* **have** $\forall s \in$ *set w. 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 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*)) $= \{\gamma'\ A\}$
using *assms* **unfolding** *rlexp-sym-def* **by** *simp*

lemma *rlexp-sym-vars-Tm*: *vars* (*rlexp-sym* (*Tm x*)) $= \{\}$
unfolding *rlexp-sym-def* **by** *simp*

lemma *rlexp-concats-vars*: *vars* (*rlexp-concats fs*) $= \bigcup$(*vars* ` *set fs*)
unfolding *rlexp-concats-def* **by** (*induction fs*) *simp-all*

lemma *insts′-vars*: *vars* (*rlexp-syms w*) ⊆ γ' ‘ *nts-syms w*
proof
  fix *x*
  assume $x \in$ *vars* (*rlexp-syms w*)
  with *rlexp-concats-vars* **have** $x \in \bigcup$(*vars* ‘ *set* (*map rlexp-sym w*))
    unfolding *rlexp-syms-def* **by** *blast*
  then obtain *f* **where** ∗: $f \in$ *set* (*map rlexp-sym w*) ∧ $x \in$ *vars f* **by** *blast*
  then obtain *s* **where** ∗∗: $s \in$ *set w* ∧ *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** *vars* (*rlexp-sym* (*Nt A*)) = {γ' *A*} **by** *blast*
  with ∗ ∗∗ ∗∗∗ ∗∗∗∗ **show** $x \in \gamma'$ ‘ *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* (γ' *A*) = *L A*
    shows *eval* (*rlexp-sym* (*Nt A*)) *v* = *inst-sym L* (*Nt A*)
  using *assms* **unfolding** *rlexp-sym-def inst-sym-def* **by** *force*

lemma *rlexp-sym-inst-Tm*: *eval* (*rlexp-sym* (*Tm a*)) *v* = *inst-sym L* (*Tm a*)
  unfolding *rlexp-sym-def inst-sym-def* **by** *force*

lemma *rlexp-concats-concats*:
  assumes *length fs* = *length Ls*
      and ∀ *i* < *length fs*. *eval* (*fs* ! *i*) *v* = *Ls* ! *i*
    shows *eval* (*rlexp-concats fs*) *v* = *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.prems* ∗ **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 ∀ $A \in$ *nts-syms w*. *v* (γ' *A*) = *L A*
    shows *eval* (*rlexp-syms w*) *v* = *inst-syms L w*
proof −
  have ∀ *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 $eval\ (rlexp\text{-}sym\ (w\ !\ i))\ v = inst\text{-}sym\ L\ (w\ !\ i)$
using *assms* **proof** (*induction w*!*i*)
case (*Nt A*)
then 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*:
$\exists\, eq.\ reg\text{-}eval\ eq \wedge vars\ eq \subseteq \gamma'\ `\ Nts\ P$
$\wedge\ (\forall\, v\ L.\ (\forall\, A \in Nts\ P.\ v\ (\gamma'\ A) = L\ A) \longrightarrow eval\ eq\ v = subst\text{-}lang\ P\ L\ A)$
proof −
let $?Insts = rlexp\text{-}syms\ `\ (Rhss\ P\ A)$
from *finite-Rhss*[*OF finite-P*] **have** *finite ?Insts* **by** *simp*
moreover from *rlexp-syms-reg* **have** $\forall f \in\ ?Insts.\ reg\text{-}eval\ f$ **by** *blast*
ultimately obtain *eq* **where** ∗: $reg\text{-}eval\ eq \wedge \bigcup (vars\ `\ ?Insts) = vars\ eq$
$\wedge\ (\forall\, v.\ (\bigcup f \in\ ?Insts.\ eval\ f\ v) = eval\ eq\ v)$
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** ∗∗∗: $w \in Rhss\ P\ A \wedge f = rlexp\text{-}syms\ w$ **by** *blast*
with ∗∗ *insts'-vars* **have** $x \in \gamma'\ `\ nts\text{-}syms\ w$ **by** *auto*
with ∗∗∗ **show** $x \in \gamma'\ `\ Nts\ P$ **unfolding** *Nts-def Rhss-def* **by** *blast*
qed
moreover have $\forall\, v\ L.\ (\forall\, A \in Nts\ P.\ v\ (\gamma'\ A) = L\ A) \longrightarrow eval\ eq\ v = subst\text{-}lang\ P\ L\ A$
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 $\forall\, w \in Rhss\ P\ A.\ eval\ (rlexp\text{-}syms\ w)\ v = inst\text{-}syms\ L\ w$
proof
fix *w*
assume $w \in Rhss\ P\ A$
with *state-L Nts-nts-syms* **have** $\forall\, A \in nts\text{-}syms\ w.\ v\ (\gamma'\ A) = L\ A$ **by** *fast*
from *rlexp-syms-insts*[*OF this*] **show** $eval\ (rlexp\text{-}syms\ w)\ v = inst\text{-}syms\ L\ w$ **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 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*-γ-γ' **have** $*$: $\bigwedge$*eq. vars eq* $\subseteq \gamma'$ ‘ *Nts P* $\Longrightarrow \forall x \in$ *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*-γ-γ' $*$ **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*=λi *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* ($\gamma\ i$))]
show *?thesis* **by** *blast*
qed

As we have proved that each CFG induces a system of 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*
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** $*$: $\forall A \in Nts\ P.\ sol\ (\gamma'\ A) = Lang\text{-}lfp\ P\ A$
unfolding *bij-Nt-Var-def bij-betw-def* **by** *force*
with ‹$i < card\ (Nts\ P)$› *assms* **have** $eval\ (sys\ !\ i)\ sol = subst\text{-}lang\ P\ (Lang\text{-}lfp\ P)\ (\gamma\ i)$
by *presburger*
with *lfp-fixpoint*[*OF mono-if-omega-cont*[*OF omega-cont-Lang-lfp*]] **have** *1*: $eval\ (sys\ !\ i)\ sol = Lang\text{-}lfp\ P\ (\gamma\ i)$
unfolding *Lang-lfp-def* **by** *metis*
from ‹$i < card\ (Nts\ P)$› *bij-γ-γ′* **have** $\gamma\ i \in Nts\ P$
unfolding *bij-Nt-Var-def* **using** *bij-betwE* **by** *blast*
with $*$ **have** $Lang\text{-}lfp\ P\ (\gamma\ i) = sol\ (\gamma'\ (\gamma\ i))$ **by** *auto*
also have $\ldots = sol\ i$ **using** *bij-γ-γ′* ‹$i < card\ (Nts\ P)$› **unfolding** *bij-Nt-Var-def* **by** *auto*
finally show $eval\ (sys\ !\ i)\ sol \subseteq sol\ i$ **using** *1* **by** *blast*
qed

lemma *CFG-sys-CFL-is-min-aux*:
assumes *CFG-sys sys*
and *solves-ineq-sys sys sol′*
shows $Lang\text{-}lfp\ P \le (\lambda A.\ sol'\ (\gamma'\ A))$ (**is** - $\le$ *?L′*)
proof −
have $subst\text{-}lang\ P\ ?L'\ A \subseteq ?L'\ A$ **for** A
proof ($cases\ A \in Nts\ P$)
case *True*
with *assms*(*1*) *bij-γ-γ′* **have** $\gamma'\ A < length\ sys$
unfolding *bij-Nt-Var-def bij-betw-def* **by** *fastforce*
with *assms*(*1*) *bij-γ-γ′ True* **have** $subst\text{-}lang\ P\ ?L'\ A = eval\ (sys\ !\ \gamma'\ A)\ sol'$
unfolding *bij-Nt-Var-def* **by** *metis*
also from *True assms*(*2*) ‹$\gamma'\ A < length\ sys$› *bij-γ-γ′* **have** $\ldots \subseteq ?L'\ A$
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 $subst\text{-}lang\ P\ ?L' \le ?L'$ **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 $sol\ x \subseteq sol'\ x$
proof ($cases\ x < card\ (Nts\ P)$)
case *True*
then have $sol\ x = Lang\text{-}lfp\ P\ (\gamma\ x)$ **by** *argo*
also from *CFG-sys-CFL-is-min-aux*[*OF assms*] **have** $\ldots \subseteq sol'\ (\gamma'\ (\gamma\ x))$ **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 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 $*$ ‹*length sys* $=$ *card* (*Nts P*)› **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 side 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 proof benefits from the result of section 2.7:

lemma *sol-comm-sol*:
assumes *sol-is-sol-comm*: *solves-ineq-sys-comm sys sol*
shows $\exists$ *sol'*. ($\forall$ *x*. Ψ (*sol x*) $=$ Ψ (*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*. Ψ (*?sol' x*) $=$ Ψ (*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** Ψ (*eval* (*sys* ! *i*) *sol*) $\subseteq$ Ψ (*sol i*)
unfolding *solves-ineq-sys-comm-def solves-ineq-comm-def* **by** *blast*
moreover from *sol′-sol* **have** Ψ (*eval* (*sys* ! *i*) *?sol′*) = Ψ (*eval* (*sys* ! *i*) *sol*)
using *rlexp-mono-parikh-eq* **by** *meson*
ultimately have Ψ (*eval* (*sys* ! *i*) *?sol′*) $\subseteq$ Ψ (*sol i*) **by** *simp*
then show *eval* (*sys* ! *i*) *?sol′* $\subseteq$ *?sol′ i* **using** *subseteq-comm-subseteq* **by** *metis*
qed
ultimately show ($\forall$ *x*. Ψ (*sol x*) = Ψ (*?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*. Ψ (*sol x*) $\subseteq$ Ψ (*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*. Ψ (*sol′ x*) = Ψ (*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*. Ψ (*sol x*) $\subseteq$ Ψ (*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 Ψ (*sol1 x*) = Ψ (*sol2 x*)
proof –
from *sol1-is-min-sol sol2-is-min-sol* **have** Ψ (*sol1 x*) $\subseteq$ Ψ (*sol2 x*)
unfolding *min-sol-ineq-sys-comm-def* **by** *simp*
moreover from *sol1-is-min-sol sol2-is-min-sol* **have** Ψ (*sol2 x*) $\subseteq$ Ψ (*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
    Eq-Sys
begin

We prove Parikh's theorem, closely following Pilling's proof [1]. The rough idea is as follows: As seen above, each CFG can be interpreted as a system of 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 this system and that this solution is furthermore minimal. Using the relations explored in the last section 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 proves Parikh's theorem.

4.1 Special representation of regular language expressions

To each 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*
  unfolding *bipart-rlexp-def* **by** *fastforce*

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 (*cases x* = *y*)
let *?f'* = *Union* (*Const* {}) (*Concat* (*Const* {[]}) (*Var x*))
  case *True*
  then have *bipart-rlexp x ?f'*

unfolding *bipart-rlexp-def* **using** *emptyset-regular epsilon-regular* **by** *fastforce*
moreover have *eval ?f′ v* = *eval* (*Var y*) *v* **for** *v* :: *′a valuation* **using** *True* **by** *simp*
moreover have *vars ?f′* = *vars* (*Var y*) ∪ {*x*} **using** *True* **by** *simp*
ultimately show *?thesis* **by** *metis*
next
let *?f′* = *Union* (*Var y*) (*Concat* (*Const* {}) (*Var x*))
case *False*
then have *bipart-rlexp x ?f′*
unfolding *bipart-rlexp-def* **using** *emptyset-regular epsilon-regular* **by** *fastforce*
moreover have *eval ?f′ v* = *eval* (*Var y*) *v* **for** *v* :: *′a valuation* **using** *False* **by** *simp*
moreover have *vars ?f′* = *vars* (*Var y*) ∪ {*x*} **by** *simp*
ultimately show *?thesis* **by** *metis*
qed

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 −
let *?f′* = *Union* (*Const l*) (*Concat* (*Const* {}) (*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*) ∪ {*x*} **by** *simp*
ultimately show *?thesis* **by** *metis*
qed

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 −
from *assms* **obtain** *f1′ f2′* **where** *f1′-intro*: *bipart-rlexp x f1′* ∧ *vars f1′* = *vars f1* ∪ {*x*} ∧
(∀ *v*. Ψ (*eval f1 v*) = Ψ (*eval f1′ v*))
and *f2′-intro*: *bipart-rlexp x f2′* ∧ *vars f2′* = *vars f2* ∪ {*x*} ∧
(∀ *v*. Ψ (*eval f2 v*) = Ψ (*eval f2′ v*))
by *auto*
then obtain *p1 q1 p2 q2* **where** *p1-q1-intro*: *reg-eval p1* ∧ *reg-eval q1* ∧
f1′ = *Union p1* (*Concat q1* (*Var x*)) ∧ (∀ *y* ∈ *vars p1*. *y* ≠ *x*)
and *p2-q2-intro*: *reg-eval p2* ∧ *reg-eval q2* ∧ *f2′* = *Union p2* (*Concat q2* (*Var x*)) ∧
(∀ *y* ∈ *vars p2*. *y* ≠ *x*) **unfolding** *bipart-rlexp-def* **by** *auto*
let *?f′* = *Union* (*Union p1 p2*) (*Concat* (*Union q1 q2*) (*Var x*))

have *bipart-rlexp x ?f'* **unfolding** *bipart-rlexp-def* **using** *p1-q1-intro p2-q2-intro* **by** *auto*
moreover have Ψ (*eval ?f' v*) = Ψ (*eval* (*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 *vars ?f'* = *vars* (*Union f1 f2*) ∪ {*x*} **by** *auto*
ultimately show *?thesis* **by** *metis*
qed

lemma *reg-eval-bipart-rlexp-Concat*:
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* (*Concat f1 f2*) ∪ {*x*} ∧
(∀ *v*. Ψ (*eval* (*Concat f1 f2*) *v*) = Ψ (*eval f' v*))
proof −
from *assms* **obtain** *f1' f2'* **where** *f1'-intro*: *bipart-rlexp x f1'* ∧ *vars f1'* = *vars f1* ∪ {*x*} ∧
(∀ *v*. Ψ (*eval f1 v*) = Ψ (*eval f1' v*))
and *f2'-intro*: *bipart-rlexp x f2'* ∧ *vars f2'* = *vars f2* ∪ {*x*} ∧
(∀ *v*. Ψ (*eval f2 v*) = Ψ (*eval f2' v*))
by *auto*
then obtain *p1 q1 p2 q2* **where** *p1-q1-intro*: *reg-eval p1* ∧ *reg-eval q1* ∧
f1' = *Union p1* (*Concat q1* (*Var x*)) ∧ (∀ *y* ∈ *vars p1*. *y* ≠ *x*)
and *p2-q2-intro*: *reg-eval p2* ∧ *reg-eval q2* ∧ *f2'* = *Union p2* (*Concat q2* (*Var x*)) ∧
(∀ *y* ∈ *vars p2*. *y* ≠ *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 ∀ *v*. (Ψ (*eval* (*Concat f1 f2*) *v*) = Ψ (*eval ?f' v*))
proof (*rule allI*)
fix *v*
have *f2-subst*: Ψ (*eval f2 v*) = Ψ (*eval p2 v* ∪ *eval q2 v* @@ *v x*)
using *p2-q2-intro f2'-intro* **by** *auto*
have Ψ (*eval* (*Concat f1 f2*) *v*) = Ψ ((*eval p1 v* ∪ *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 . . . = Ψ (*eval p1 v* @@ *eval f2 v* ∪ *eval q1 v* @@ *v x* @@ *eval f2 v*)
by (*simp add*: *conc-Un-distrib*(*2*) *conc-assoc*)
also have . . . = Ψ (*eval p1 v* @@ (*eval p2 v* ∪ *eval q2 v* @@ *v x*)
∪ *eval q1 v* @@ *v x* @@ (*eval p2 v* ∪ *eval q2 v* @@ *v x*))
using *f2-subst* **by** (*smt* (*verit*, *ccfv-threshold*) *parikh-conc-right parikh-img-Un parikh-img-commut*)
also have . . . = Ψ (*eval p1 v* @@ *eval p2 v* ∪ (*eval p1 v* @@ *eval q2 v* @@ *v x* ∪

eval q1 v @@ eval p2 v @@ v x ∪ *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 ... = Ψ (*eval p1 v @@ eval p2 v* ∪ (*eval p1 v @@ eval q2 v* ∪
eval q1 v @@ eval p2 v ∪ *eval q1 v @@ v x @@ eval q2 v*) *@@ v x*)
by (*simp add*: *conc-Un-distrib*(*2*) *conc-assoc*)
also have ... = Ψ (*eval ?f′ v*)
by (*simp add*: *Un-commute*)
finally show Ψ (*eval* (*Concat f1 f2*) *v*) = Ψ (*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*) ∪ {*x*} **by** *auto*
ultimately show *?thesis* **by** *metis*
qed

lemma *reg-eval-bipart-rlexp-Star*:
assumes ∃*f′*. *bipart-rlexp x f′* ∧ *vars f′* = *vars f* ∪ {*x*}
∧ (∀ *v*. Ψ (*eval f v*) = Ψ (*eval f′ v*))
shows ∃*f′*. *bipart-rlexp x f′* ∧ *vars f′* = *vars* (*Star f*) ∪ {*x*}
∧ (∀ *v*. Ψ (*eval* (*Star f*) *v*) = Ψ (*eval f′ v*))
proof −
from *assms* **obtain** *f′* **where** *f′-intro*: *bipart-rlexp x f′* ∧ *vars f′* = *vars f* ∪ {*x*} ∧
(∀ *v*. Ψ (*eval f v*) = Ψ (*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 ∀ *v*. (Ψ (*eval* (*Star f*) *v*) = Ψ (*eval ?f-new v*))
proof (*rule allI*)
fix *v*
have Ψ (*eval* (*Star f*) *v*) = Ψ (*star* (*eval p v* ∪ *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 ... = Ψ (*star* (*eval p v*) *@@ star* (*eval q v @@ v x*))
using *parikh-img-star* **by** *blast*
also have ... = Ψ (*star* (*eval p v*) @@
star ({[]} ∪ *star* (*eval q v @@ v x*) *@@ eval q v @@ v x*))
by (*metis Un-commute conc-star-comm star-idemp star-unfold-left*)
also have ... = Ψ (*star* (*eval p v*) *@@ star* (*star* (*eval q v @@ v x*) *@@ eval q v @@ v x*))
by *auto*
also have ... = Ψ (*star* (*eval p v*) @@ ({[]} ∪ *star* (*eval q v @@ v x*)
@@ star (*eval q v @@ v x*) *@@ eval q v @@ v x*))
using *parikh-img-star2 parikh-conc-left* **by** *blast*

also have ... = Ψ (*star* (*eval p v*) @@ {[]} $\cup$ *star* (*eval p v*) @@ *star* (*eval q v* @@ *v x*)
@@ *star* (*eval q v* @@ *v x*) @@ *eval q v* @@ *v x*) **by** (*metis conc-Un-distrib*(*1*))
also have ... = Ψ (*eval ?f-new v*) **by** (*simp add*: *conc-assoc*)
finally show Ψ (*eval* (*Star f*) *v*) = Ψ (*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*) $\cup$ {*x*} **by** *auto*
ultimately show *?thesis* **by** *metis*
qed

lemma *reg-eval-bipart-rlexp*: *reg-eval f* $\Longrightarrow$
$\exists f'$. *bipart-rlexp x f′* $\wedge$ *vars f′* = *vars f* $\cup$ {*x*} $\wedge$
($\forall s$. Ψ (*eval f s*) = Ψ (*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 $\exists f'$. *bipart-rlexp x f′* $\wedge$ *vars f′* = *vars f* $\cup$ {*x*} $\wedge$ ($\forall v$. Ψ (*eval f v*) = Ψ (*eval f′ v*))
$\exists f'$. *bipart-rlexp x f′* $\wedge$ *vars f′* = *vars g* $\cup$ {*x*} $\wedge$ ($\forall v$. Ψ (*eval g v*) = Ψ (*eval f′ v*))
by *auto*
from *reg-eval-bipart-rlexp-Union*[*OF this*] **show** *?case* **by** *blast*
next
case (*4 f g*)
then have $\exists f'$. *bipart-rlexp x f′* $\wedge$ *vars f′* = *vars f* $\cup$ {*x*} $\wedge$ ($\forall v$. Ψ (*eval f v*) = Ψ (*eval f′ v*))
$\exists f'$. *bipart-rlexp x f′* $\wedge$ *vars f′* = *vars g* $\cup$ {*x*} $\wedge$ ($\forall v$. Ψ (*eval g v*) = Ψ (*eval f′ v*))
by *auto*
from *reg-eval-bipart-rlexp-Concat*[*OF this*] **show** *?case* **by** *blast*
next
case (*5 f*)
then have $\exists f'$. *bipart-rlexp x f′* $\wedge$ *vars f′* = *vars f* $\cup$ {*x*} $\wedge$ ($\forall v$. Ψ (*eval f v*) = Ψ (*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 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* ≡ *Union p* (*Concat q* (*Var x*))
abbreviation *sol* ≡ *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* ⊆ *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* ⊆ *vars eq* − $\{x\}$ **by** *fastforce*
  have *vars ?subst-q* ⊆ *vars eq* − $\{x\}$
  proof
    fix y
    assume *y-in-subst-q*: $y \in$ *vars ?subst-q*
    with *vars-subst* **obtain** y' **where** *y'-in-q*: $y' \in$ *vars q* **and** *y-in-y'*: $y \in$ *vars*
(*?upd* y')
      unfolding *fun-upd-def* **by** *force*
    show $y \in$ *vars eq* − $\{x\}$
    proof (*cases* $y' = x$)
      case *True*
      with *y-in-y' x-not-in-p* **show** *?thesis* **by** *auto*
    next
      case *False*

    with y′-in-q y-in-y′ show ?thesis by simp
  qed
qed
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*: Ψ (*eval ?q-subst v*) ⊆ Ψ (*eval (Concat (Star ?r) ?r) v*)
    using *rlexp-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 Ψ (*eval ?eq-subst v*) = Ψ (*eval p v* ∪ *eval ?q-subst v @@ eval sol v*)
    by *simp*
  also have . . . ⊆ Ψ (*eval p v* ∪ *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 . . . = Ψ (*eval p v*) ∪
      Ψ (*star (eval ?r v) @@ eval ?r v @@ star (eval ?r v) @@ eval p v*)
    by (*simp add*: *conc-assoc*)
  also have . . . = Ψ (*eval p v*) ∪
      Ψ (*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 . . . = Ψ (*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 ∗: Ψ (*eval ?eq-subst v*) ⊆ Ψ (*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-update*[**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   Ψ (*eval sol v*) ⊆ Ψ (*v x*)
proof −
  from *is-sol sol′-s* **have** *is-sol′*: Ψ (*eval q v @@ v x* ∪ *eval p v*) ⊆ Ψ (*v x*)
    unfolding *solves-ineq-comm-def* **by** *simp*
  then have *1*: Ψ (*eval (Concat (Star q) p) v*) ⊆ Ψ (*v x*)
    using *parikh-img-arden* **by** *auto*

from *is-sol'* **have** Ψ (*eval p v*) $\subseteq$ Ψ (*eval* (*Var x*) *v*) **by** *auto*
then have Ψ (*eval* (*subst* (*Var*($x := p$)) *q*) *v*) $\subseteq$ Ψ (*eval q v*)
using *parikh-img-subst-mono-upd* **by** (*metis fun-upd-triv subst-id*)
then have Ψ (*eval* (*Star* (*subst* (*Var*($x := p$)) *q*)) *v*) $\subseteq$ Ψ (*eval* (*Star q*) *v*)
using *parikh-star-mono* **by** *auto*
then have Ψ (*eval sol v*) $\subseteq$ Ψ (*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 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*. Ψ (*eval eq v*) = Ψ (*eval eq' v*)) **by** *blast*
then obtain *p q*
where *p-q-intro*: *reg-eval p* $\wedge$ *reg-eval q* $\wedge$ *eq'* = *Union p* (*Concat q* (*Var x*)) $\wedge$
$x \notin$ *vars p*
unfolding *bipart-rlexp-def* **by** *blast*
let *?sol* = *Concat* (*Star* (*subst* (*Var*($x := p$)) *q*)) *p*
from *p-q-intro* **have** *sol-prop*: *reg-eval ?sol* $\wedge$ *partial-min-sol-one-ineq x eq' ?sol*
using *single-bipartite-eq.sol-is-minimal-reg-sol* **unfolding** *single-bipartite-eq-def*
by *blast*
with *eq'-intro* **have** *partial-min-sol-one-ineq x eq ?sol*
using *same-min-sol-if-same-parikh-img* **by** *blast*
with *sol-prop* **show** *?thesis* **by** *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 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*: $\forall$ *eq* ∈ *set sys*. *reg-eval eq*
      and *sys-valid*: $\forall$ *eq* ∈ *set sys*. $\forall$ *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*: $\forall$ *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 *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'* ≡ *subst-sys sols sys*
abbreviation *sols'* ≡ λ*i*. *subst* (*Var*(*r* := *sol-r*)) (*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*. *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* = *eval* (*sols' x*) *v*
  from *sols'-r s-sols'* **have** *s-r-sol-r*: *v r* = *eval sol-r v* **by** *simp*
  with *s-sols'* **have** *s-sols*: *v x* = *eval* (*sols x*) *v* **for** *x*
   using *substitution-lemma-update*[**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 *eval* (*sys* ! *r*) (λ*y*. *eval* (*sols y*) *v*) = *eval* (*sys'* ! *r*) *v*
    using *substitution-lemma*[*of* λ*y*. *eval* (*sols y*) *v*]
    by (*simp add*: *r-valid Suc-le-lessD subst-sys-subst*)
  with *s-sols* **have** *eval* (*sys* ! *r*) *v* = *eval* (*sys'* ! *r*) *v*
    by (*metis* (*mono-tags*, *lifting*) *eval-vars*)

with *sol-r-is-sol s-r-sol-r* **have** Ψ (*eval* (*sys* ! *r*) *v*) $\subseteq$ Ψ (*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* = *eval* (*sols2 x*) *v2*)
$\wedge$ *solves-ineq-sys-comm* (*take* (*Suc r*) *sys*) *v2*
$\longrightarrow$ ($\forall$ *i*. Ψ (*eval* (*sols' i*) *v2*) $\subseteq$ Ψ (*v2 i*))
proof (*rule allI* | *rule impI*)+
fix *sols2 v2 i*
assume *as*: ($\forall$ *x*. *v2 x* = *eval* (*sols2 x*) *v2*) $\wedge$ *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*: Ψ (*eval* (*sols i*) *v2*) $\subseteq$ Ψ (*v2 i*) **for** *i*
unfolding *partial-min-sol-ineq-sys-def* **by** *auto*
have *eval* (*sys'* ! *r*) *v2* = *eval* (*sys* ! *r*) (λ*i*. *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** Ψ (*eval* (*sys'* ! *r*) *v2*) $\subseteq$ Ψ (*eval* (*sys* ! *r*) *v2*)
using *rlexp-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*: Ψ (*eval sol-r v2*) $\subseteq$ Ψ (*v2 r*)
unfolding *partial-min-sol-one-ineq-def* **by** *blast*
let *?v'* = *v2*(*r* := *eval sol-r v2*)
from *sol-r-min* **have** Ψ (*?v' i*) $\subseteq$ Ψ (*v2 i*) **for** *i* **by** *simp*
with *sols-s2* **show** Ψ (*eval* (*sols' i*) *v2*) $\subseteq$ Ψ (*v2 i*)
using *substitution-lemma-update*[**where** *f*=*sols i*] *rlexp-mono-parikh*[*of sols i ?v' v2*] **by** *force*
qed

lemma *sols'-vars-gt-r*: $\forall$ *i* $\geq$ *Suc r*. *sols' i* = *Var i*
using *sols-is-sol* **unfolding** *partial-min-sol-ineq-sys-def* **by** *auto*

lemma *sols'-vars-leq-r*: $\forall$ *i* < *Suc r*. $\forall$ *x* $\in$ *vars* (*sols' i*). *x* $\geq$ *Suc r* $\wedge$ *x* < *length sys*
proof −
from *sols-is-sol* **have** $\forall$ *i* < *r*. $\forall$ *x* $\in$ *vars* (*sols i*). *x* $\geq$ *r* $\wedge$ *x* < *length sys*
unfolding *partial-min-sol-ineq-sys-def* **by** *simp*
with *sols-is-sol* **have** *vars-sols*: $\forall$ *i* < *length sys*. $\forall$ *x* $\in$ *vars* (*sols i*). *x* $\geq$ *r* $\wedge$ *x* < *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* $\in$ *vars* (*subst sols* (*sys* ! *i*)). *x* $\geq$ *r* $\wedge$ *x* < *length sys* **if** *i* < *length sys* **for** *i*

using *vars-subst*[*of sols sys* ! *i*] *that* **by** (*metis UN-E nth-mem*)
then have $\forall x \in$ *vars* (*sys'* ! *i*). $x \geq r \wedge x <$ *length sys* **if** $i <$ *length sys* **for** *i*
unfolding *subst-sys-def* **using** *r-valid that* **by** *auto*
moreover from *sol-r-is-sol* **have** *vars* (*sol-r*) $\subseteq$ *vars* (*sys'* ! *r*) $- \{r\}$
unfolding *partial-min-sol-one-ineq-def* **by** *simp*
ultimately have *vars-sol-r*: $\forall x \in$ *vars sol-r*. $x > r \wedge x <$ *length sys*
unfolding *partial-min-sol-one-ineq-def* **using** *r-valid*
by (*metis DiffE insertCI nat-less-le subsetD*)
moreover have *vars* (*sols'* *i*) $\subseteq$ *vars* (*sols i*) $- \{r\} \cup$ *vars sol-r* **if** $i <$ *length sys* **for** *i*
using *vars-subst-upd-upper* **by** *meson*
ultimately have $\forall x \in$ *vars* (*sols'* *i*). $x > r \wedge x <$ *length sys* **if** $i <$ *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*:
$\exists$ *sols'*. *partial-min-sol-ineq-sys* (*Suc r*) *sys sols'* $\wedge$ ($\forall$ *i*. *reg-eval* (*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 also 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*))
using *assms* **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*: *partial-min-sol-ineq-sys r sys sols* $\wedge$ ($\forall$ *i*. *reg-eval* (*sols i*))
  by *auto*
let *?sys'* = *subst-sys sols sys*
from *eqs-reg Suc.prems* **have** *reg-eval* (*sys* ! *r*) **by** *simp*
with *sols-intro Suc.prems* **have** *sys-r-reg*: *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*:
  reg-eval sol-r $\wedge$ *partial-min-sol-one-ineq r* (*?sys'* ! *r*) *sol-r*
  using *exists-minimal-reg-sol* **by** *blast*
with *Suc sols-intro* **have** *min-sol-induction-step r 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* $\in$ *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*))
proof −
from *eqs-reg sys-valid* **have**
  $\exists$ *sols*. *partial-min-sol-ineq-sys* (*length sys*) *sys sols* $\wedge$ ($\forall$ *i*. *reg-eval* (*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* < *length 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* < *length sys* **for** *i*
  using *that const-rlexp-regular-lang* **by** *metis*
then obtain *ls* **where** *ls-intro*: $\forall$ *i* < *length sys*. *regular-lang* (*ls i*) $\wedge$ ($\forall$ *v*. *eval* (*sols i*) *v* = *ls i*)
  by *metis*
let *?ls'* = λ*i*. *if i* < *length sys then ls i else* {}
from *ls-intro* **have** *ls'-intro*:
  ($\forall$ *i* < *length sys*. *regular-lang* (*?ls' i*) $\wedge$ ($\forall$ *v*. *eval* (*sols i*) *v* = *?ls' i*))
  $\wedge$ ($\forall$ *i* $\geq$ *length 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*) $\subseteq$ Ψ (*sol' x*))
proof (*rule allI*, *rule impI*)
  fix *sol' x*
  assume *as*: *solves-ineq-sys-comm sys sol'*
  let *?sol-rlexps* = λ*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 $\forall x.\ \Psi$ (*?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 grammar exists a regular language with identical Parikh image:

theorem *Parikh*: *CFL* (*TYPE*(*′n*)) *L* $\Longrightarrow \exists L'.$ *regular-lang L′* $\wedge\ \Psi\ L = \Psi\ L'$
proof −
assume *CFL* (*TYPE*(*′n*)) *L*
then obtain *P* **and** *S*::*′n* **where** ∗: *L* = *Lang P S* ∧ *finite P* **unfolding** *CFL-def* **by** *blast*
show *?thesis*
proof (*cases S* ∈ *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* = $\lambda i.$ *if i* < *card* (*Nts P*) *then Lang-lfp P* (γ *i*) *else* {}
from ∗∗ *True* **have** γ' *S* < *card* (*Nts P*) γ (γ' *S*) = *S*
unfolding *bij-Nt-Var-def bij-betw-def* **by** *auto*
with *Lang-lfp-eq-Lang* **have** ∗∗∗: *Lang P S* = *?sol* (γ' *S*) **by** *metis*
from ∗ ∗∗ *CFG-eq-sys.CFL-is-min-sol* **obtain** *sys*
where *sys-intro*: (∀ *eq* ∈ *set sys. reg-eval eq*) ∧ (∀ *eq* ∈ *set sys.* ∀ *x* ∈ *vars eq. x* < *length sys*)
∧ *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-reg-sol-sys* **obtain** *sol′* **where**
sol′-intro: *min-sol-ineq-sys-comm sys sol′* ∧ *regular-lang* (*sol′* (γ' *S*)) **by** *fastforce*
with *sol-is-min-sol min-sol-comm-unique* **have** Ψ (*?sol* (γ' *S*)) = Ψ (*sol′* (γ' *S*))
by *blast*
with ∗ ∗∗∗ *sol′-intro* **show** *?thesis* **by** *auto*
next
case *False*
with *Nts-Lhss-Rhs-Nts* **have** *S* ∉ *Lhss P* **by** *fast*
from *Lang-empty-if-notin-Lhss*[*OF this*] ∗ **show** *?thesis* **by** (*metis lang.simps*(*1*))
qed
qed

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.