

A Codatatype of Formal Languages

Dmitriy Traytel

May 21, 2019

1 Introduction

We define formal languages as a codatatype of infinite trees branching over the alphabet $'a$. Each node in such a tree indicates whether the path to this node constitutes a word inside or outside of the language.

codatatype $'a$ language = Lang (σ : bool) (δ : $'a \Rightarrow 'a$ language)

This codatatype is isomorphic to the set of lists representation of languages, but caters for definitions by corecursion and proofs by coinduction.

Regular operations on languages are then defined by primitive corecursion. A difficulty arises here, since the standard definitions of concatenation and iteration from the coalgebraic literature are not primitively corecursive—they require guardedness up-to union/concatenation. Without support for up-to corecursion, these operation must be defined as a composition of primitive ones (and proved being equal to the standard definitions). As an exercise in coinduction we also prove the axioms of Kleene algebra for the defined regular operations.

Furthermore, a language for context-free grammars given by productions in Greibach normal form and an initial nonterminal is constructed by primitive corecursion, yielding an executable decision procedure for the word problem without further ado.

2 Regular Languages

primcorec Zero :: $'a$ language **where**

σ Zero = False

| δ Zero = ($\lambda..$ Zero)

primcorec One :: $'a$ language **where**

σ One = True

| δ One = ($\lambda..$ Zero)

primcorec Atom :: $'a \Rightarrow 'a$ language **where**

σ (Atom a) = False

| δ (Atom a) = ($\lambda b.$ if $a = b$ then One else Zero)

primcorec Plus :: $'a$ language $\Rightarrow 'a$ language $\Rightarrow 'a$ language **where**

σ (Plus r s) = (σ $r \vee \sigma$ s)

| δ (Plus r s) = ($\lambda a.$ Plus (δ r a) (δ s a))

theorem Plus_ZeroL[simp]: Plus Zero r = r

by (coinduction arbitrary: r) simp

theorem Plus_ZeroR[simp]: Plus r Zero = r

by (coinduction arbitrary: r) simp

theorem *Plus_assoc*: $Plus (Plus r s) t = Plus r (Plus s t)$
by (*coinduction arbitrary*: $r s t$) *auto*

theorem *Plus_comm*: $Plus r s = Plus s r$
by (*coinduction arbitrary*: $r s$) *auto*

lemma *Plus_rotate*: $Plus r (Plus s t) = Plus s (Plus r t)$
using *Plus_assoc Plus_comm by metis*

theorem *Plus_idem*: $Plus r r = r$
by (*coinduction arbitrary*: r) *auto*

lemma *Plus_idem_assoc*: $Plus r (Plus r s) = Plus r s$
by (*metis Plus_assoc Plus_idem*)

lemmas *Plus_ACI[simp]* = *Plus_rotate Plus_comm Plus_assoc Plus_idem_assoc Plus_idem*

lemma *Plus_OneL[simp]*: $\mathfrak{o} r \implies Plus One r = r$
by (*coinduction arbitrary*: r) *auto*

lemma *Plus_OneR[simp]*: $\mathfrak{o} r \implies Plus r One = r$
by (*coinduction arbitrary*: r) *auto*

Concatenation is not primitively corecursive—the corecursive call of its derivative is guarded by *Plus*. However, it can be defined as a composition of two primitively corecursive functions.

primcorec *TimesLR* :: ' a language \Rightarrow ' a language \Rightarrow (' $a \times bool$) language **where**
 $\mathfrak{o} (TimesLR r s) = (\mathfrak{o} r \wedge \mathfrak{o} s)$
 $|\ \mathfrak{d} (TimesLR r s) = (\lambda(a, b).$
if b then TimesLR ($\mathfrak{d} r a$) s else if $\mathfrak{o} r$ then TimesLR ($\mathfrak{d} s a$) One else Zero)

primcorec *Times_Plus* :: (' $a \times bool$) language \Rightarrow ' a language **where**
 $\mathfrak{o} (Times_Plus r) = \mathfrak{o} r$
 $|\ \mathfrak{d} (Times_Plus r) = (\lambda a. Times_Plus (Plus (\mathfrak{d} r (a, True)) (\mathfrak{d} r (a, False))))$

lemma *TimesLR_ZeroL[simp]*: $TimesLR Zero r = Zero$
by (*coinduction arbitrary*: r) *auto*

lemma *TimesLR_ZeroR[simp]*: $TimesLR r Zero = Zero$
by (*coinduction arbitrary*: r) (*auto intro: exI[of - Zero]*)

lemma *TimesLR_PlusL[simp]*: $TimesLR (Plus r s) t = Plus (TimesLR r t) (TimesLR s t)$
by (*coinduction arbitrary*: $r s t$) *auto*

lemma *TimesLR_PlusR[simp]*: $TimesLR r (Plus s t) = Plus (TimesLR r s) (TimesLR r t)$
by (*coinduction arbitrary*: $r s t$) *auto*

lemma *Times_Plus_Zero[simp]*: $Times_Plus Zero = Zero$
by *coinduction simp*

lemma *Times_Plus_Plus[simp]*: $Times_Plus (Plus r s) = Plus (Times_Plus r) (Times_Plus s)$

proof (*coinduction arbitrary*: $r s$)
case (*Lang r s*)
then show *?case unfolding Times_Plus.sel Plus.sel*
by (*intro conjI[OF refl] (metis Plus_comm Plus_rotate)*)
qed

lemma *Times_Plus_TimesLR_One[simp]*: $Times_Plus (TimesLR r One) = r$
by (*coinduction arbitrary*: r) *simp*

lemma *Times_Plus_TimesLR_PlusL[simp]*:
 $Times_Plus (TimesLR (Plus r s) t) = Plus (Times_Plus (TimesLR r t)) (Times_Plus (TimesLR s t))$
by (*coinduction arbitrary: r s t*) *auto*

lemma *Times_Plus_TimesLR_PlusR[simp]*:
 $Times_Plus (TimesLR r (Plus s t)) = Plus (Times_Plus (TimesLR r s)) (Times_Plus (TimesLR r t))$
by (*coinduction arbitrary: r s t*) *auto*

definition *Times* :: 'a language \Rightarrow 'a language \Rightarrow 'a language **where**
 $Times r s = Times_Plus (TimesLR r s)$

lemma *o_Times[simp]*:
 $o (Times r s) = (o r \wedge o s)$
unfolding *Times_def* **by** *simp*

lemma *d_Times[simp]*:
 $d (Times r s) = (\lambda a. \text{if } o r \text{ then } Plus (Times (d r a) s) (d s a) \text{ else } Times (d r a) s)$
unfolding *Times_def* **by** *auto*

theorem *Times_ZeroL[simp]*: $Times\ Zero\ r = Zero$
by *coinduction simp*

theorem *Times_ZeroR[simp]*: $Times\ r\ Zero = Zero$
by (*coinduction arbitrary: r*) *auto*

theorem *Times_OneL[simp]*: $Times\ One\ r = r$
by (*coinduction arbitrary: r rule: language.coinduct_strong*) (*simp add: rel_fun_def*)

theorem *Times_OneR[simp]*: $Times\ r\ One = r$
by (*coinduction arbitrary: r*) *simp*

Coinduction up-to *Plus*-congruence relaxes the coinduction hypothesis by requiring membership in the congruence closure of the bisimulation rather than in the bisimulation itself.

inductive *Plus_cong* **for** *R* **where**

Refl[intro]: $x = y \Longrightarrow Plus_cong\ R\ x\ y$
| *Base[intro]*: $R\ x\ y \Longrightarrow Plus_cong\ R\ x\ y$
| *Sym*: $Plus_cong\ R\ x\ y \Longrightarrow Plus_cong\ R\ y\ x$
| *Trans[intro]*: $Plus_cong\ R\ x\ y \Longrightarrow Plus_cong\ R\ y\ z \Longrightarrow Plus_cong\ R\ x\ z$
| *Plus[intro]*: $\llbracket Plus_cong\ R\ x\ y; Plus_cong\ R\ x'\ y' \rrbracket \Longrightarrow Plus_cong\ R\ (Plus\ x\ x')\ (Plus\ y\ y')$

lemma *language.coinduct_upto_Plus[unfolded rel_fun_def, simplified, case_names Lang, consumes 1]*:

assumes *R*: $R\ L\ K$ **and** *hyp*:
 $(\bigwedge L\ K. R\ L\ K \Longrightarrow o\ L = o\ K \wedge rel_fun\ (=)\ (Plus_cong\ R)\ (d\ L)\ (d\ K))$
shows $L = K$

proof (*coinduct rule: language.coinduct[of Plus_cong R]*)

fix *L K* **assume** $Plus_cong\ R\ L\ K$

then show $o\ L = o\ K \wedge rel_fun\ (=)\ (Plus_cong\ R)\ (d\ L)\ (d\ K)$

by (*induct rule: Plus_cong.induct*) (*auto simp: rel_fun_def intro: Sym dest: hyp*)

qed (*intro Base R*)

theorem *Times_PlusL[simp]*: $Times (Plus r s) t = Plus (Times r t) (Times s t)$
by (*coinduction arbitrary: r s rule: language.coinduct_upto_Plus*) *auto*

theorem *Times_PlusR[simp]*: $Times r (Plus s t) = Plus (Times r s) (Times r t)$
by (*coinduction arbitrary: r s rule: language.coinduct_upto_Plus*) *fastforce*

theorem *Times_assoc[simp]*: $Times (Times r s) t = Times r (Times s t)$

by (coinduction arbitrary: r s t rule: language_coinduct_upto_Plus) fastforce

Similarly to *Times*, iteration is not primitively corecursive (guardedness by *Times* is required). We apply a similar trick to obtain its definition.

primcorec *StarLR* :: 'a language \Rightarrow 'a language \Rightarrow 'a language **where**
 \circ (*StarLR* r s) = \circ r
 \mid ∂ (*StarLR* r s) = ($\lambda a.$ *StarLR* (∂ (*Times* r (*Plus* One s)) a) s)

lemma *StarLR_Zero[simp]*: *StarLR* Zero r = Zero
 by coinduction auto

lemma *StarLR_Plus[simp]*: *StarLR* (*Plus* r s) t = *Plus* (*StarLR* r t) (*StarLR* s t)
 by (coinduction arbitrary: r s) (auto simp del: *Plus_ACI Times_PlusR*)

lemma *StarLR_Times_Plus_One[simp]*: *StarLR* (*Times* r (*Plus* One s)) s = *StarLR* r s

proof (coinduction arbitrary: r s)

case *Lang*

{ fix a

define *L* and *R* where *L* = *Plus* (∂ r a) (*Plus* (*Times* (∂ r a) s) (∂ s a))

and *R* = *Times* (*Plus* (∂ r a) (*Plus* (*Times* (∂ r a) s) (∂ s a))) s

have *Plus* *L* (*Plus* *R* (∂ s a)) = *Plus* (*Plus* *L* (∂ s a)) *R* by (metis *Plus_assoc Plus_comm*)

also have *Plus* *L* (∂ s a) = *L* **unfolding** *L_def* **by** *simp*

finally have *Plus* *L* (*Plus* *R* (∂ s a)) = *Plus* *L* *R* .

}

then show ?case **by** (auto simp del: *StarLR_Plus Plus_assoc Times_PlusL*)

qed

lemma *StarLR_Times*: *StarLR* (*Times* r s) t = *Times* r (*StarLR* s t)

by (coinduction arbitrary: r s t rule: language_coinduct_upto_Plus)

(fastforce simp del: *Plus_ACI Times_PlusR*)

definition *Star* :: 'a language \Rightarrow 'a language **where**

Star r = *StarLR* One r

lemma \circ _*Star*[*simp*]: \circ (*Star* r)

unfolding *Star_def* **by** *simp*

lemma ∂ _*Star*[*simp*]: ∂ (*Star* r) = ($\lambda a.$ *Times* (∂ r a) (*Star* r))

unfolding *Star_def* **by** (auto simp add: *Star_def StarLR_Times[symmetric]*)

lemma *Star_Zero[simp]*: *Star* Zero = One

by coinduction auto

lemma *Star_One[simp]*: *Star* One = One

by coinduction auto

lemma *Star_unfoldL*: *Star* r = *Plus* One (*Times* r (*Star* r))

by coinduction auto

primcorec *Inter* :: 'a language \Rightarrow 'a language \Rightarrow 'a language **where**

\circ (*Inter* r s) = (\circ r \wedge \circ s)

\mid ∂ (*Inter* r s) = ($\lambda a.$ *Inter* (∂ r a) (∂ s a))

primcorec *Not* :: 'a language \Rightarrow 'a language **where**

\circ (*Not* r) = (\neg \circ r)

\mid ∂ (*Not* r) = ($\lambda a.$ *Not* (∂ r a))

primcorec *Full* :: 'a language (Σ^*) **where**

\circ $Full = True$
 $\mid \partial Full = (\lambda_. Full)$

Shuffle product is not primitively corecursive—the corecursive call of its derivative is guarded by *Plus*. However, it can be defined as a composition of two primitively corecursive functions.

primcorec $ShuffleLR :: 'a \text{ language} \Rightarrow 'a \text{ language} \Rightarrow ('a \times bool) \text{ language}$ **where**
 $\circ (ShuffleLR \ r \ s) = (\circ \ r \ \wedge \ \circ \ s)$
 $\mid \partial (ShuffleLR \ r \ s) = (\lambda(a, b). \text{ if } b \text{ then } ShuffleLR \ (\partial \ r \ a) \ s \ \text{else } ShuffleLR \ r \ (\partial \ s \ a))$

lemma $ShuffleLR_ZeroL[simp]$: $ShuffleLR \ Zero \ r = Zero$
by (*coinduction arbitrary: r*) *auto*

lemma $ShuffleLR_ZeroR[simp]$: $ShuffleLR \ r \ Zero = Zero$
by (*coinduction arbitrary: r*) (*auto intro: exI[of _ Zero]*)

lemma $ShuffleLR_PlusL[simp]$: $ShuffleLR \ (Plus \ r \ s) \ t = Plus \ (ShuffleLR \ r \ t) \ (ShuffleLR \ s \ t)$
by (*coinduction arbitrary: r s t*) *auto*

lemma $ShuffleLR_PlusR[simp]$: $ShuffleLR \ r \ (Plus \ s \ t) = Plus \ (ShuffleLR \ r \ s) \ (ShuffleLR \ r \ t)$
by (*coinduction arbitrary: r s t*) *auto*

lemma $Shuffle_Plus_ShuffleLR_One[simp]$: $Times_Plus \ (ShuffleLR \ r \ One) = r$
by (*coinduction arbitrary: r*) *simp*

lemma $Shuffle_Plus_ShuffleLR_PlusL[simp]$:
 $Times_Plus \ (ShuffleLR \ (Plus \ r \ s) \ t) = Plus \ (Times_Plus \ (ShuffleLR \ r \ t)) \ (Times_Plus \ (ShuffleLR \ s \ t))$
by (*coinduction arbitrary: r s t*) *auto*

lemma $Shuffle_Plus_ShuffleLR_PlusR[simp]$:
 $Times_Plus \ (ShuffleLR \ r \ (Plus \ s \ t)) = Plus \ (Times_Plus \ (ShuffleLR \ r \ s)) \ (Times_Plus \ (ShuffleLR \ r \ t))$
by (*coinduction arbitrary: r s t*) *auto*

definition $Shuffle :: 'a \text{ language} \Rightarrow 'a \text{ language} \Rightarrow 'a \text{ language}$ **where**
 $Shuffle \ r \ s = Times_Plus \ (ShuffleLR \ r \ s)$

lemma $\circ_Shuffle[simp]$:
 $\circ (Shuffle \ r \ s) = (\circ \ r \ \wedge \ \circ \ s)$
unfolding $Shuffle_def$ **by** *simp*

lemma $\partial_Shuffle[simp]$:
 $\partial (Shuffle \ r \ s) = (\lambda a. Plus \ (Shuffle \ (\partial \ r \ a) \ s) \ (Shuffle \ r \ (\partial \ s \ a)))$
unfolding $Shuffle_def$ **by** *auto*

theorem $Shuffle_ZeroL[simp]$: $Shuffle \ Zero \ r = Zero$
by (*coinduction arbitrary: r rule: language_coinduct_upto_Plus*) (*auto 0 4*)

theorem $Shuffle_ZeroR[simp]$: $Shuffle \ r \ Zero = Zero$
by (*coinduction arbitrary: r rule: language_coinduct_upto_Plus*) (*auto 0 4*)

theorem $Shuffle_OneL[simp]$: $Shuffle \ One \ r = r$
by (*coinduction arbitrary: r*) *simp*

theorem $Shuffle_OneR[simp]$: $Shuffle \ r \ One = r$
by (*coinduction arbitrary: r*) *simp*

theorem $Shuffle_PlusL[simp]$: $Shuffle \ (Plus \ r \ s) \ t = Plus \ (Shuffle \ r \ t) \ (Shuffle \ s \ t)$
by (*coinduction arbitrary: r s t rule: language_coinduct_upto_Plus*)
(force intro!: Trans[OF Plus[OF Base Base] Refl])

theorem *Shuffle_PlusR[simp]*: $\text{Shuffle } r \text{ (Plus } s \text{ t)} = \text{Plus (Shuffle } r \text{ s) (Shuffle } r \text{ t)}$
by (*coinduction arbitrary: r s t rule: language_coinduct_upto_Plus*)
(force intro!: Trans[OF Plus[OF Base Base] Refl])

theorem *Shuffle_assoc[simp]*: $\text{Shuffle (Shuffle } r \text{ s) t} = \text{Shuffle } r \text{ (Shuffle s t)}$
by (*coinduction arbitrary: r s t rule: language_coinduct_upto_Plus*) *fastforce*

theorem *Shuffle_comm[simp]*: $\text{Shuffle } r \text{ s} = \text{Shuffle s r}$
by (*coinduction arbitrary: r s rule: language_coinduct_upto_Plus*)
(auto intro!: Trans[OF Plus[OF Base Base] Refl])

We generalize coinduction up-to *Plus* to coinduction up-to all previously defined concepts.

inductive *regular_cong* **for** *R* **where**

Refl[*intro*]: $x = y \implies \text{regular_cong } R \ x \ y$
| *Sym*[*intro*]: $\text{regular_cong } R \ x \ y \implies \text{regular_cong } R \ y \ x$
| *Trans*[*intro*]: $\llbracket \text{regular_cong } R \ x \ y; \text{regular_cong } R \ y \ z \rrbracket \implies \text{regular_cong } R \ x \ z$
| *Base*[*intro*]: $R \ x \ y \implies \text{regular_cong } R \ x \ y$
| *Plus*[*intro, simp*]: $\llbracket \text{regular_cong } R \ x \ y; \text{regular_cong } R \ x' \ y' \rrbracket \implies$
 $\text{regular_cong } R \ (\text{Plus } x \ x') \ (\text{Plus } y \ y')$
| *Times*[*intro, simp*]: $\llbracket \text{regular_cong } R \ x \ y; \text{regular_cong } R \ x' \ y' \rrbracket \implies$
 $\text{regular_cong } R \ (\text{Times } x \ x') \ (\text{Times } y \ y')$
| *Star*[*intro, simp*]: $\llbracket \text{regular_cong } R \ x \ y \rrbracket \implies$
 $\text{regular_cong } R \ (\text{Star } x) \ (\text{Star } y)$
| *Inter*[*intro, simp*]: $\llbracket \text{regular_cong } R \ x \ y; \text{regular_cong } R \ x' \ y' \rrbracket \implies$
 $\text{regular_cong } R \ (\text{Inter } x \ x') \ (\text{Inter } y \ y')$
| *Not*[*intro, simp*]: $\llbracket \text{regular_cong } R \ x \ y \rrbracket \implies$
 $\text{regular_cong } R \ (\text{Not } x) \ (\text{Not } y)$
| *Shuffle*[*intro, simp*]: $\llbracket \text{regular_cong } R \ x \ y; \text{regular_cong } R \ x' \ y' \rrbracket \implies$
 $\text{regular_cong } R \ (\text{Shuffle } x \ x') \ (\text{Shuffle } y \ y')$

lemma *language_coinduct_upto_regular*[*unfolded rel_fun_def, simplified, case_names Lang, consumes 1*]:

assumes *R: R L K and hyp*:
 $(\bigwedge L \ K. R \ L \ K \implies \circ \ L = \circ \ K \wedge \text{rel_fun } (=) \ (\text{regular_cong } R) \ (\text{d } L) \ (\text{d } K))$
shows $L = K$

proof (*coinduct rule: language_coinduct[of regular_cong R]*)

fix *L K* **assume** *regular_cong R L K*
then show $\circ \ L = \circ \ K \wedge \text{rel_fun } (=) \ (\text{regular_cong } R) \ (\text{d } L) \ (\text{d } K)$
by (*induct rule: regular_cong.induct*) (*auto dest: hyp simp: rel_fun_def*)
qed (*intro Base R*)

lemma *Star_unfoldR*: $\text{Star } r = \text{Plus One (Times (Star } r) \ r)$

proof (*coinduction arbitrary: r rule: language_coinduct_upto_regular*)

case *Lang*
{ **fix** *a* **have** $\text{Plus (Times (d } r \ a) \ (\text{Times (Star } r) \ r)) \ (\text{d } r \ a) =$
 $\text{Times (d } r \ a) \ (\text{Plus One (Times (Star } r) \ r))$ **by** *simp*
}
then show *?case* **by** (*auto simp del: Times_PlusR*)
qed

lemma *Star_Star[simp]*: $\text{Star (Star } r) = \text{Star } r$

by (*subst Star_unfoldL, coinduction arbitrary: r rule: language_coinduct_upto_regular*) *auto*

lemma *Times_Star[simp]*: $\text{Times (Star } r) \ (\text{Star } r) = \text{Star } r$

proof (*coinduction arbitrary: r rule: language_coinduct_upto_regular*)

case *Lang*
have $*$: $\bigwedge r \ s. \text{Plus (Times } r \ s) \ r = \text{Times } r \ (\text{Plus } s \ \text{One})$ **by** *simp*
show *?case* **by** (*auto simp del: Times_PlusR Plus_ACI simp: Times_PlusR[symmetric] **)

qed

instantiation *language* :: (type) {semiring_1, order}
begin

lemma *Zero_One[simp]*: $\text{Zero} \neq \text{One}$
by (metis *One.simps(1) Zero.simps(1)*)

definition *zero_language* = *Zero*

definition *one_language* = *One*

definition *plus_language* = *Plus*

definition *times_language* = *Times*

definition *less_eq_language* $r\ s = (\text{Plus } r\ s = s)$

definition *less_language* $r\ s = (\text{Plus } r\ s = s \wedge r \neq s)$

lemmas *language_defs* = *zero_language_def one_language_def plus_language_def times_language_def less_eq_language_def less_language_def*

instance proof *intro_classes*

fix $x\ y\ z :: 'a\ \text{language}$ **assume** $x \leq y\ y \leq z$

then show $x \leq z$ **unfolding** *language_defs* **by** (metis *Plus_assoc*)

next

fix $x\ y\ z :: 'a\ \text{language}$

show $x + y + z = x + (y + z)$ **unfolding** *language_defs* **by** (rule *Plus_assoc*)

qed (auto *simp: language_defs*)

end

lemma *o_mono[dest]*: $r \leq s \implies \mathbf{o}\ r \implies \mathbf{o}\ s$

unfolding *less_eq_language_def* **by** (auto *dest: arg_cong[of _ _ o]*)

lemma *d_mono[dest]*: $r \leq s \implies \mathbf{d}\ r\ a \leq \mathbf{d}\ s\ a$

unfolding *less_eq_language_def* **by** (metis *Plus.simps(2)*)

For reasoning about (\leq) , we prove a coinduction principle and generalize it to support up-to reasoning.

theorem *language_simulation_coinduction[consumes 1, case_names Lang, coinduct pred]*:

assumes $R\ L\ K$

and $(\bigwedge L\ K.\ R\ L\ K \implies \mathbf{o}\ L \leq \mathbf{o}\ K \wedge (\forall x.\ R\ (\mathbf{d}\ L\ x) (\mathbf{d}\ K\ x)))$

shows $L \leq K$

using $\langle R\ L\ K \rangle$ **unfolding** *less_eq_language_def*

by (coinduction *arbitrary: L K*) (auto *dest!: assms(2)*)

lemma *le_PlusL[intro!, simp]*: $r \leq \text{Plus } r\ s$

by (coinduction *arbitrary: r s*) *auto*

lemma *le_PlusR[intro!, simp]*: $s \leq \text{Plus } r\ s$

by (coinduction *arbitrary: r s*) *auto*

inductive *Plus_Times_pre_cong* **for** R **where**

pre_Less[intro, simp]: $x \leq y \implies \text{Plus_Times_pre_cong } R\ x\ y$

| *pre_Trans[intro]*: $[\text{Plus_Times_pre_cong } R\ x\ y; \text{Plus_Times_pre_cong } R\ y\ z] \implies \text{Plus_Times_pre_cong } R\ x\ z$

| *pre_Base[intro, simp]*: $R\ x\ y \implies \text{Plus_Times_pre_cong } R\ x\ y$

| *pre_Plus[intro!, simp]*: $[\text{Plus_Times_pre_cong } R\ x\ y; \text{Plus_Times_pre_cong } R\ x'\ y'] \implies$

$\text{Plus_Times_pre_cong } R\ (\text{Plus } x\ x') (\text{Plus } y\ y')$

| *pre_Times[intro!, simp]*: $[\text{Plus_Times_pre_cong } R\ x\ y; \text{Plus_Times_pre_cong } R\ x'\ y'] \implies$

$Plus_Times_pre_cong\ R\ (Times\ x\ x')\ (Times\ y\ y')$

theorem *language_simulation_coinduction_upto_Plus_Times*[consumes 1, case_names Lang, coinduct pred]:

assumes $R: R\ L\ K$

and hyp: $(\bigwedge L\ K. R\ L\ K \implies \mathfrak{o}\ L \leq \mathfrak{o}\ K \wedge (\forall x. Plus_Times_pre_cong\ R\ (\mathfrak{d}\ L\ x)\ (\mathfrak{d}\ K\ x)))$

shows $L \leq K$

proof (coinduct rule: *language_simulation_coinduction*[of *Plus_Times_pre_cong* R])

fix $L\ K$ **assume** $Plus_Times_pre_cong\ R\ L\ K$

then show $\mathfrak{o}\ L \leq \mathfrak{o}\ K \wedge (\forall x. Plus_Times_pre_cong\ R\ (\mathfrak{d}\ L\ x)\ (\mathfrak{d}\ K\ x))$

by (*induct* rule: *Plus_Times_pre_cong.induct*) (*auto* *dest*: *hyp*)

qed (*intro* *pre_Base* R)

lemma *ge_One[simp]*: $One \leq r \iff \mathfrak{o}\ r$

unfolding *less_eq_language_def* **by** (*metis* *One.sel*(1) *Plus.sel*(1) *Plus_OneL*)

lemma *Plus_mono*: $\llbracket r1 \leq s1; r2 \leq s2 \rrbracket \implies Plus\ r1\ r2 \leq Plus\ s1\ s2$

by (*coinduction* *arbitrary*: $r1\ r2\ s1\ s2$) (*auto* *elim!*: $\mathfrak{d_mono}$)

lemma *Plus_upper*: $\llbracket r1 \leq s; r2 \leq s \rrbracket \implies Plus\ r1\ r2 \leq s$

by (*coinduction* *arbitrary*: $r1\ r2\ s$) *auto*

lemma *Inter_mono*: $\llbracket r1 \leq s1; r2 \leq s2 \rrbracket \implies Inter\ r1\ r2 \leq Inter\ s1\ s2$

by (*coinduction* *arbitrary*: $r1\ r2\ s1\ s2$) (*force* *elim!*: $\mathfrak{d_mono}$)

lemma *Times_mono*: $\llbracket r1 \leq s1; r2 \leq s2 \rrbracket \implies Times\ r1\ r2 \leq Times\ s1\ s2$

by (*coinduction* *arbitrary*: $r1\ r2\ s1\ s2$) (*auto* 0 3 *elim!*: $\mathfrak{d_mono}$)

We prove the missing axioms of Kleene Algebras about *Star*, as well as monotonicity properties and three standard interesting rules: bisimulation, sliding, and denesting.

theorem *le_StarL*: $Plus\ One\ (Times\ r\ (Star\ r)) \leq Star\ r$

by *coinduction* *auto*

theorem *le_StarR*: $Plus\ One\ (Times\ (Star\ r)\ r) \leq Star\ r$

by (*rule* *order_eq_refl*[OF *Star_unfoldR*[*symmetric*]])

lemma *le_TimesL*[*intro*, *simp*]: $\mathfrak{o}\ s \implies r \leq Times\ r\ s$

by (*coinduction* *arbitrary*: r) *auto*

lemma *le_TimesR*[*intro*, *simp*]: $\mathfrak{o}\ r \implies s \leq Times\ r\ s$

by *coinduction* *auto*

lemma *Plus_le_iff*: $Plus\ r\ s \leq t \iff r \leq t \wedge s \leq t$

unfolding *less_eq_language_def*

by (*metis* *Plus_assoc* *Plus_idem_assoc* *Plus_rotate*)

lemma *Plus_Times_pre_cong_mono*:

$L' \leq L \implies K \leq K' \implies Plus_Times_pre_cong\ R\ L\ K \implies Plus_Times_pre_cong\ R\ L'\ K'$

by (*metis* *pre_Trans* *pre_Less*)

theorem *ardenL*: $Plus\ r\ (Times\ s\ x) \leq x \implies Times\ (Star\ s)\ r \leq x$

proof (*coinduction* *arbitrary*: $r\ s\ x$)

case *Lang*

then show *?case*

by (*subst* *Plus_Times_pre_cong_mono*[OF *order_refl* $\mathfrak{d_mono}$ [OF *Lang*]]) *auto*

qed

theorem *ardenR*: $Plus\ r\ (Times\ x\ s) \leq x \implies Times\ r\ (Star\ s) \leq x$

proof (*coinduction* *arbitrary*: $r\ s\ x$)


```

case Lang
then have Plus_Times_pre_cong ( $\lambda L K. \exists r s. L = \text{Times } r (\text{Star } s) \wedge \text{Plus } r (\text{Times } K s) \leq K$ )
  ( $\text{d } (\text{Times } r (\text{Star } s)) a$ ) ( $\text{d } x a$ ) for  $a$  using  $\text{d\_mono}[OF \text{ Lang, of } a]$ 
  by ( $\text{auto } 0 \ 4 \ \text{simp del: Times\_PlusL simp: Times\_PlusL[symmetric] Plus\_le\_iff split: if\_splits}$ )
with Lang show ?case
  by auto
qed

lemma le_Star[intro!, simp]:  $s \leq \text{Star } s$ 
  by coinduction auto

lemma Star_mono:  $r \leq s \implies \text{Star } r \leq \text{Star } s$ 
  by coinduction auto

lemma Not_antimono:  $r \leq s \implies \text{Not } s \leq \text{Not } r$ 
  by (coinduction arbitrary:  $r$   $s$ ) auto

lemma Not_Plus[simp]:  $\text{Not } (\text{Plus } r s) = \text{Inter } (\text{Not } r) (\text{Not } s)$ 
  by (coinduction arbitrary:  $r$   $s$ ) auto

lemma Not_Inter[simp]:  $\text{Not } (\text{Inter } r s) = \text{Plus } (\text{Not } r) (\text{Not } s)$ 
  by (coinduction arbitrary:  $r$   $s$ ) auto

lemma Inter_assoc[simp]:  $\text{Inter } (\text{Inter } r s) t = \text{Inter } r (\text{Inter } s t)$ 
  by (coinduction arbitrary:  $r$   $s$   $t$ ) auto

lemma Inter_comm:  $\text{Inter } r s = \text{Inter } s r$ 
  by (coinduction arbitrary:  $r$   $s$ ) auto

lemma Inter_idem[simp]:  $\text{Inter } r r = r$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Inter_ZeroL[simp]:  $\text{Inter } \text{Zero } r = \text{Zero}$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Inter_ZeroR[simp]:  $\text{Inter } r \text{Zero} = \text{Zero}$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Inter_FullL[simp]:  $\text{Inter } \text{Full } r = r$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Inter_FullR[simp]:  $\text{Inter } r \text{Full} = r$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Plus_FullL[simp]:  $\text{Plus } \text{Full } r = \text{Full}$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Plus_FullR[simp]:  $\text{Plus } r \text{Full} = \text{Full}$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Not_Not[simp]:  $\text{Not } (\text{Not } r) = r$ 
  by (coinduction arbitrary:  $r$ ) auto

lemma Not_Zero[simp]:  $\text{Not } \text{Zero} = \text{Full}$ 
  by coinduction simp

lemma Not_Full[simp]:  $\text{Not } \text{Full} = \text{Zero}$ 
  by coinduction simp

```

lemma bisimulation:
assumes $Times\ r\ s = Times\ s\ t$
shows $Times\ (Star\ r)\ s = Times\ s\ (Star\ t)$
proof (rule antisym[OF ardenL[OF Plus_upper[OF le_TimesL]] ardenR[OF Plus_upper[OF le_TimesR]]])
show $Times\ r\ (Times\ s\ (Star\ t)) \leq Times\ s\ (Star\ t)$
by (rule order_trans[OF -
 $Times_mono$ [OF order_refl ord_le_eq_trans[OF le_PlusR Star_unfoldL[symmetric]]]])
(simp only: assms Times_assoc[symmetric])
next
show $Times\ (Times\ (Star\ r)\ s)\ t \leq Times\ (Star\ r)\ s$
by (rule order_trans[OF -
 $Times_mono$ [OF ord_le_eq_trans[OF le_PlusR Star_unfoldR[symmetric]] order_refl]])
(simp only: assms Times_assoc)
qed simp_all

lemma sliding: $Times\ (Star\ (Times\ r\ s))\ r = Times\ r\ (Star\ (Times\ s\ r))$
proof (rule antisym[OF ardenL[OF Plus_upper[OF le_TimesL]] ardenR[OF Plus_upper[OF le_TimesR]]])
show $Times\ (Times\ r\ s)\ (Times\ r\ (Star\ (Times\ s\ r))) \leq Times\ r\ (Star\ (Times\ s\ r))$
by (rule order_trans[OF -
 $Times_mono$ [OF order_refl ord_le_eq_trans[OF le_PlusR Star_unfoldL[symmetric]]]]) simp
next
show $Times\ (Times\ (Star\ (Times\ r\ s))\ r)\ (Times\ s\ r) \leq Times\ (Star\ (Times\ r\ s))\ r$
by (rule order_trans[OF -
 $Times_mono$ [OF ord_le_eq_trans[OF le_PlusR Star_unfoldR[symmetric]] order_refl]]) simp
qed simp_all

lemma denesting: $Star\ (Plus\ r\ s) = Times\ (Star\ r)\ (Star\ (Times\ s\ (Star\ r)))$
proof (rule antisym[OF ord_eq_le_trans[OF Times_OneR[symmetric] ardenL[OF Plus_upper]]
ardenR[OF Plus_upper[OF Star_mono[OF le_PlusL]]]])
show $Times\ (Plus\ r\ s)\ (Times\ (Star\ r)\ (Star\ (Times\ s\ (Star\ r))))$
 $\leq Times\ (Star\ r)\ (Star\ (Times\ s\ (Star\ r)))$ (is $Times\ _ ?L \leq ?R$)
unfolding Times_PlusL
by (rule Plus_upper,
metis Star_unfoldL Times_assoc Times_mono le_PlusR order_refl,
metis Star_unfoldL Times_assoc o_Star le_PlusR le_TimesR order_trans)
next
show $Times\ (Star\ (Plus\ r\ s))\ (Times\ s\ (Star\ r)) \leq Star\ (Plus\ r\ s)$
by (metis Plus_comm Star_unfoldL Times_PlusR Times_assoc ardenR bisimulation le_PlusR)
qed simp

It is useful to lift binary operators *Plus* and *Times* to *n*-ary operators (that take a list as input).

definition PLUS :: 'a language list \Rightarrow 'a language **where**
 $PLUS\ xs \equiv foldr\ Plus\ xs\ Zero$

lemma o_foldr_Plus: $o\ (foldr\ Plus\ xs\ s) = (\exists x \in set\ (s\ \# \ xs).\ o\ x)$
by (induct xs arbitrary: s) auto

lemma d_foldr_Plus: $d\ (foldr\ Plus\ xs\ s)\ a = foldr\ Plus\ (map\ (\lambda r.\ d\ r\ a)\ xs)\ (d\ s\ a)$
by (induct xs arbitrary: s) simp_all

lemma o_PLUS[simp]: $o\ (PLUS\ xs) = (\exists x \in set\ xs.\ o\ x)$
unfolding PLUS_def o_foldr_Plus **by** simp

lemma d_PLUS[simp]: $d\ (PLUS\ xs)\ a = PLUS\ (map\ (\lambda r.\ d\ r\ a)\ xs)$
unfolding PLUS_def d_foldr_Plus **by** simp

definition TIMES :: 'a language list \Rightarrow 'a language **where**

$TIMES\ xs \equiv foldr\ Times\ xs\ One$

lemma o_foldr_Times : $o\ (foldr\ Times\ xs\ s) = (\forall x \in set\ (s\ \# \ xs).\ o\ x)$
by $(induct\ xs)\ (auto\ simp:\ PLUS_def)$

primrec $tails$ **where**

$tails\ [] = [[]]$
 $| tails\ (x\ \# \ xs) = (x\ \# \ xs)\ \# \ tails\ xs$

lemma $tails_snoc[simp]$: $tails\ (xs\ @\ [x]) = map\ (\lambda ys.\ ys\ @\ [x])\ (tails\ xs)\ @\ [[]]$
by $(induct\ xs)\ auto$

lemma $length_tails[simp]$: $length\ (tails\ xs) = Suc\ (length\ xs)$
by $(induct\ xs)\ auto$

lemma δ_foldr_Times : $\delta\ (foldr\ Times\ xs\ s)\ a =$
 $(let\ n = length\ (takeWhile\ o\ xs)$
 $in\ PLUS\ (map\ (\lambda zs.\ TIMES\ (\delta\ (hd\ zs)\ a\ \# \ tl\ zs))\ (take\ (Suc\ n)\ (tails\ (xs\ @\ [s])))))$
by $(induct\ xs)\ (auto\ simp:\ TIMES_def\ PLUS_def\ Let_def\ foldr_map\ o_def)$

lemma $o_TIMES[simp]$: $o\ (TIMES\ xs) = (\forall x \in set\ xs.\ o\ x)$
unfolding $TIMES_def\ o_foldr_Times$ **by** $simp$

lemma $TIMES_snoc_One[simp]$: $TIMES\ (xs\ @\ [One]) = TIMES\ xs$
by $(induct\ xs)\ (auto\ simp:\ TIMES_def)$

lemma $\delta_TIMES[simp]$: $\delta\ (TIMES\ xs)\ a = (let\ n = length\ (takeWhile\ o\ xs)$
 $in\ PLUS\ (map\ (\lambda zs.\ TIMES\ (\delta\ (hd\ zs)\ a\ \# \ tl\ zs))\ (take\ (Suc\ n)\ (tails\ (xs\ @\ [One])))))$
unfolding $TIMES_def\ \delta_foldr_Times$ **by** $simp$

3 Word-theoretic Semantics of Languages

We show our *language* codatatype being isomorphic to the standard language representation as a set of lists.

primrec $in_language$:: $'a\ language \Rightarrow 'a\ list \Rightarrow bool$ **where**
 $in_language\ L\ [] = o\ L$
 $| in_language\ L\ (x\ \# \ xs) = in_language\ (\delta\ L\ x)\ xs$

primcorec $to_language$:: $'a\ list\ set \Rightarrow 'a\ language$ **where**
 $o\ (to_language\ L) = (\[] \in L)$
 $| \delta\ (to_language\ L) = (\lambda a.\ to_language\ \{w.\ a\ \# \ w \in L\})$

lemma $in_language_to_language[simp]$: $Collect\ (in_language\ (to_language\ L)) = L$
proof $(rule\ set_eqI,\ unfold\ mem_Collect_eq)$
fix w **show** $in_language\ (to_language\ L)\ w = (w \in L)$ **by** $(induct\ w\ arbitrary:\ L)\ auto$
qed

lemma $to_language_in_language[simp]$: $to_language\ (Collect\ (in_language\ L)) = L$
by $(coinduction\ arbitrary:\ L)\ auto$

lemma $in_language_bij$: $bij\ (Collect\ o\ in_language)$

proof $(rule\ bijI',\ unfold\ o_apply,\ safe)$
fix $L\ R :: 'a\ language$ **assume** $Collect\ (in_language\ L) = Collect\ (in_language\ R)$
then show $L = R$ **unfolding** $set_eq_iff\ mem_Collect_eq$
by $(coinduction\ arbitrary:\ L\ R)\ (metis\ in_language.simps)$
next
fix $L :: 'a\ list\ set$

have $L = \text{Collect } (\text{in_language } (\text{to_language } L))$ **by** *simp*
then show $\exists K. L = \text{Collect } (\text{in_language } K)$ **by** *blast*
qed

lemma *to_language_bij*: *bij to_language*
by (*rule o_bij*[of *Collect o in_language*]) (*simp_all add: fun_eq_iff*)

4 Coinductively Defined Operations Are Standard

lemma *to_language_empty*[*simp*]: *to_language {} = Zero*
by (*coinduction*) *auto*

lemma *in_language_Zero*[*simp*]: $\neg \text{in_language } \text{Zero } xs$
by (*induct xs*) *auto*

lemma *in_language_One*[*simp*]: *in_language One xs \implies xs = []*
by (*cases xs*) *auto*

lemma *in_language_Atom*[*simp*]: *in_language (Atom a) xs \implies xs = [a]*
by (*cases xs*) (*auto split: if_splits*)

lemma *to_language_eps*[*simp*]: *to_language {} = One*
by (*rule bij_is_inj*[OF *in_language_bij*, THEN *injD*]) *auto*

lemma *to_language_singleton*[*simp*]: *to_language {[a]} = (Atom a)*
by (*rule bij_is_inj*[OF *in_language_bij*, THEN *injD*]) *auto*

lemma *to_language_Un*[*simp*]: *to_language (A \cup B) = Plus (to_language A) (to_language B)*
by (*coinduction arbitrary: A B*) (*auto simp: Collect_disj_eq*)

lemma *to_language_Int*[*simp*]: *to_language (A \cap B) = Inter (to_language A) (to_language B)*
by (*coinduction arbitrary: A B*) (*auto simp: Collect_conj_eq*)

lemma *to_language_Neg*[*simp*]: *to_language (\neg A) = Not (to_language A)*
by (*coinduction arbitrary: A*) (*auto simp: Collect_neg_eq*)

lemma *to_language_Diff*[*simp*]: *to_language (A $-$ B) = Inter (to_language A) (Not (to_language B))*
by (*auto simp: Diff_eq*)

lemma *to_language_conc*[*simp*]: *to_language (A @@ B) = Times (to_language A) (to_language B)*
by (*coinduction arbitrary: A B rule: language_coinduct_upto_Plus*)
(auto simp: Deriv_def[symmetric])

lemma *to_language_star*[*simp*]: *to_language (star A) = Star (to_language A)*
by (*coinduction arbitrary: A rule: language_coinduct_upto_regular*)
(auto simp: Deriv_def[symmetric])

lemma *to_language_shuffle*[*simp*]: *to_language (A || B) = Shuffle (to_language A) (to_language B)*
by (*coinduction arbitrary: A B rule: language_coinduct_upto_Plus*)
(force simp: Deriv_def[symmetric])

5 Word Problem for Context-Free Grammars

6 Context Free Languages

A context-free grammar consists of a list of productions for every nonterminal and an initial nonterminal. The productions are required to be in weak Greibach normal form, i.e. each right hand side of a production must either be empty or start with a terminal.

abbreviation *wgreibach* $\alpha \equiv (\text{case } \alpha \text{ of } (\text{Inr } N \# _) \Rightarrow \text{False} \mid _ \Rightarrow \text{True})$

record $(t, 'n)$ *cfg* =
init :: $'n$:: *finite*
prod :: $'n \Rightarrow (t + 'n)$ *list fset*

context

fixes G :: $(t, 'n$:: *finite*) *cfg*

begin

inductive *in_cfl* **where**

in_cfl [] []
 $\mid \text{in_cfl } \alpha \ w \Longrightarrow \text{in_cfl } (\text{Inl } a \# \alpha) (a \# w)$
 $\mid \text{fBex } (\text{prod } G \ N) (\lambda\beta. \text{in_cfl } (\beta \ @ \ \alpha) \ w) \Longrightarrow \text{in_cfl } (\text{Inr } N \# \alpha) \ w$

abbreviation *lang_trad* **where**

lang_trad $\equiv \{w. \text{in_cfl } [\text{Inr } (\text{init } G)] \ w\}$

fun \circ_P **where**

\circ_P [] = *True*
 $\mid \circ_P (\text{Inl } _ \# _) = \text{False}$
 $\mid \circ_P (\text{Inr } N \# \alpha) = (\text{[] } \mid \in \mid \text{prod } G \ N \wedge \circ_P \ \alpha)$

fun ∂_P **where**

∂_P [] $a = \{\mid\}$
 $\mid \partial_P (\text{Inl } b \# \alpha) \ a = (\text{if } a = b \text{ then } \{\mid\alpha\} \text{ else } \{\mid\})$
 $\mid \partial_P (\text{Inr } N \# \alpha) \ a =$
 $(\lambda\beta. \text{tl } \beta \ @ \ \alpha) \mid \text{' } \text{ffilter } (\lambda\beta. \beta \neq [] \wedge \text{hd } \beta = \text{Inl } a) (\text{prod } G \ N) \mid \cup$
 $(\text{if } [] \mid \in \mid \text{prod } G \ N \text{ then } \partial_P \ \alpha \ a \text{ else } \{\mid\})$

primcorec *subst* :: $(t + 'n)$ *list fset* \Rightarrow $'t$ *language* **where**

subst $P = \text{Lang } (\text{fBex } P \ \circ_P) (\lambda a. \text{subst } (\text{ffUnion } ((\lambda r. \partial_P \ r \ a) \mid \text{' } P)))$

inductive *in_cfls* **where**

fBex $P \ \circ_P \Longrightarrow \text{in_cfls } P$ []
 $\mid \text{in_cfls } (\text{ffUnion } ((\lambda\alpha. \partial_P \ \alpha \ a) \mid \text{' } P)) \ w \Longrightarrow \text{in_cfls } P (a \# w)$

inductive_cases [*elim!*]: *in_cfls* P []

inductive_cases [*elim!*]: *in_cfls* $P (a \# w)$

declare *inj_eq*[*OF* *bij_is_inj*[*OF* *to_language_bij*], *simp*]

lemma *subst_in_cfls*: *subst* $P = \text{to_language } \{w. \text{in_cfls } P \ w\}$

by (*coinduction* *arbitrary*: P) (*auto* *intro*: *in_cfls.intros*)

lemma \circ_P .*in_cfl*: $\circ_P \ \alpha \Longrightarrow \text{in_cfl } \alpha$ []

by (*induct* α *rule*: \circ_P .*induct*) (*auto* *intro!*: *in_cfl.intros* *elim*: *fBexI*[*rotated*])

lemma ∂_P .*in_cfl*: $\beta \mid \in \mid \partial_P \ \alpha \ a \Longrightarrow \text{in_cfl } \beta \ w \Longrightarrow \text{in_cfl } \alpha (a \# w)$

proof (*induct* α *a* *arbitrary*: $\beta \ w$ *rule*: ∂_P .*induct*)

```

case (3 N α a)
then show ?case
  by (auto simp: rev_fBexI neq_Nil_conv split: if_splits
        intro!: in_cfl.intros elim!: rev_fBexI[of - # -])
qed (auto split: if_splits intro: in_cfl.intros)

lemma in_cfls_in_cfl: in_cfls P w  $\implies$  fBex P (λα. in_cfl α w)
by (induct P w rule: in_cfls.induct)
  (auto simp: o_P_in_cfl d_P_in_cfl ffUnion.rep_eq fmember.rep_eq fBex.rep_eq fBall.rep_eq
    intro: in_cfl.intros elim: rev_bexI)

lemma in_cfls_mono: in_cfls P w  $\implies$  P | $\subseteq$ | Q  $\implies$  in_cfls Q w
proof (induct P w arbitrary: Q rule: in_cfls.induct)
  case (2 a P w)
  from 2(3) 2(2)[of ffUnion ((λα. local.d_P α a) |' Q)] show ?case
  by (auto intro!: funion_mono in_cfls.intros)
qed (auto intro!: in_cfls.intros)

end

locale cfg_wgreibach =
  fixes G :: ('t, 'n :: finite) cfg
  assumes weakGreibach:  $\bigwedge N \alpha. \alpha \in | \text{prod } G \ N \implies \text{wgreibach } \alpha$ 
begin

lemma in_cfl_in_cfls: in_cfl G α w  $\implies$  in_cfls G {|\α|} w
proof (induct α w rule: in_cfl.induct)
  case (3 N α w)
  then obtain β where
    β: β | $\in$ | prod G N and
    in_cfl: in_cfl G (β @ α) w and
    in_cfls: in_cfls G {|\β @ α|} w by blast
  then show ?case
proof (cases β)
  case [simp]: Nil
  from β in_cfls show ?thesis
  by (cases w) (auto intro!: in_cfls.intros elim: in_cfls_mono)
next
  case [simp]: (Cons x γ)
  from weakGreibach[OF β] obtain a where [simp]: x = Inl a by (cases x) auto
  with β in_cfls show ?thesis
  apply –
  apply (rule in_cfl.cases[OF in_cfl]; auto)
  apply (force intro: in_cfls.intros(2) elim!: in_cfls_mono)
  done
qed
qed (auto intro!: in_cfls.intros)

abbreviation lang where
  lang  $\equiv$  subst G {|[Inr (init G)]|}

lemma lang_lang_trad: lang = to_language (lang_trad G)
proof –
  have in_cfls G {|[Inr (init G)]|} w  $\longleftrightarrow$  in_cfl G [Inr (init G)] w for w
  by (auto dest: in_cfls.in_cfl in_cfl_in_cfls)
  then show ?thesis
  by (auto simp: subst_in_cfls)
qed

```

end

The function *in_language* decides the word problem for a given language. Since we can construct the language of a CFG using *cfg_wgreibach.lang* we obtain an executable (but not very efficient) decision procedure for CFGs for free.

abbreviation *a* \equiv *Inl True*
abbreviation *b* \equiv *Inl False*
abbreviation *S* \equiv *Inr ()*

interpretation *palindromes*: *cfg_wgreibach* (*init* = (), *prod* = $\lambda_{-} \{[], [a], [b], [a, S, a], [b, S, b]\}$)
by *unfold_locales auto*

lemma *in_language palindromes.lang []* by *normalization*
lemma *in_language palindromes.lang [True]* by *normalization*
lemma *in_language palindromes.lang [False]* by *normalization*
lemma *in_language palindromes.lang [True, True]* by *normalization*
lemma *in_language palindromes.lang [True, False, True]* by *normalization*
lemma \neg *in_language palindromes.lang [True, False]* by *normalization*
lemma \neg *in_language palindromes.lang [True, False, True, False]* by *normalization*
lemma *in_language palindromes.lang [True, False, True, True, False, True]* by *normalization*
lemma \neg *in_language palindromes.lang [True, False, True, False, False, True]* by *normalization*

interpretation *Dyck*: *cfg_wgreibach* (*init* = (), *prod* = $\lambda_{-} \{[], [a, S, b, S]\}$)
by *unfold_locales auto*

lemma *in_language Dyck.lang []* by *normalization*
lemma \neg *in_language Dyck.lang [True]* by *normalization*
lemma \neg *in_language Dyck.lang [False]* by *normalization*
lemma *in_language Dyck.lang [True, False, True, False]* by *normalization*
lemma *in_language Dyck.lang [True, True, False, False]* by *normalization*
lemma *in_language Dyck.lang [True, False, True, False]* by *normalization*
lemma *in_language Dyck.lang [True, False, True, False, True, True, False, False]* by *normalization*
lemma \neg *in_language Dyck.lang [True, False, True, True, False]* by *normalization*
lemma \neg *in_language Dyck.lang [True, True, False, False, False, True]* by *normalization*

interpretation *abSSa*: *cfg_wgreibach* (*init* = (), *prod* = $\lambda_{-} \{[], [a, b, S, S, a]\}$)
by *unfold_locales auto*

lemma *in_language abSSa.lang []* by *normalization*
lemma \neg *in_language abSSa.lang [True]* by *normalization*
lemma \neg *in_language abSSa.lang [False]* by *normalization*
lemma *in_language abSSa.lang [True, False, True]* by *normalization*
lemma *in_language abSSa.lang [True, False, True, False, True, True, False, True, True]* by *normalization*
lemma *in_language abSSa.lang [True, False, True, False, True, True]* by *normalization*
lemma \neg *in_language abSSa.lang [True, False, True, True, False]* by *normalization*
lemma \neg *in_language abSSa.lang [True, True, False, False, False, True]* by *normalization*