A Codatatype of Formal Languages

Dmitriy Traytel

December 14, 2021

1 Introduction

We define formal languages as a codataype 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* (𝔬: *bool*) (𝔡: $'a \Rightarrow 'a$ *language*)

This codatatype is isormorphic 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**
  𝔬 *Zero* = *False*
| 𝔡 *Zero* = (λ_. *Zero*)

primcorec *One* :: $'a$ *language* **where**
  𝔬 *One* = *True*
| 𝔡 *One* = (λ_. *Zero*)

primcorec *Atom* :: $'a \Rightarrow 'a$ *language* **where**
  𝔬 (*Atom a*) = *False*
| 𝔡 (*Atom a*) = (λ*b*. *if a* = *b then One else Zero*)

primcorec *Plus* :: $'a$ *language* ⇒ $'a$ *language* ⇒ $'a$ *language* **where**
  𝔬 (*Plus r s*) = (𝔬 *r* ∨ 𝔬 *s*)
| 𝔡 (*Plus r s*) = (λ*a*. *Plus* (𝔡 *r a*) (𝔡 *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*]: 𝔬 *r* ⟹ *Plus One r* = *r*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Plus_OneR*[*simp*]: 𝔬 *r* ⟹ *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* ⇒ *'a language* ⇒ (*'a* × *bool*) *language* **where**
𝔬 (*TimesLR r s*) = (𝔬 *r* ∧ 𝔬 *s*)
| 𝔡 (*TimesLR r s*) = (λ(*a*, *b*).
if b then TimesLR (𝔡 *r a*) *s else if* 𝔬 *r then TimesLR* (𝔡 *s a*) *One else Zero*)

primcorec *Times_Plus* :: (*'a* × *bool*) *language* ⇒ *'a language* **where**
𝔬 (*Times_Plus r*) = 𝔬 *r*
| 𝔡 (*Times_Plus r*) = (λ*a*. *Times_Plus* (*Plus* (𝔡 *r* (*a*, *True*)) (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* ⇒ *'a language* ⇒ *'a language* **where**
Times *r* *s* = *Times_Plus* (*TimesLR* *r* *s*)

lemma 𝔬*_Times*[*simp*]:
𝔬 (*Times* *r* *s*) = (𝔬 *r* ∧ 𝔬 *s*)
unfolding *Times_def* **by** *simp*

lemma 𝔡*_Times*[*simp*]:
𝔡 (*Times* *r* *s*) = (λ*a*. *if* 𝔬 *r* *then* *Plus* (*Times* (𝔡 *r* *a*) *s*) (𝔡 *s* *a*) *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* ⟹ *Plus_cong* *R* *x* *y*
| *Base*[*intro*]: *R* *x* *y* ⟹ *Plus_cong* *R* *x* *y*
| *Sym*: *Plus_cong* *R* *x* *y* ⟹ *Plus_cong* *R* *y* *x*
| *Trans*[*intro*]: *Plus_cong* *R* *x* *y* ⟹ *Plus_cong* *R* *y* *z* ⟹ *Plus_cong* *R* *x* *z*
| *Plus*[*intro*]: ⟦*Plus_cong* *R* *x* *y*; *Plus_cong* *R* *x′* *y′*⟧ ⟹ *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*:
(⋀*L* *K*. *R* *L* *K* ⟹ 𝔬 *L* = 𝔬 *K* ∧ *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* ∧ *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* ⇒ *'a language* ⇒ *'a language* **where**
  𝔬 (*StarLR r s*) = 𝔬 *r*
| 𝔡 (*StarLR r s*) = (λ*a*. *StarLR* (𝔡 (*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* (𝔡 *r a*) (*Plus* (*Times* (𝔡 *r a*) *s*) (𝔡 *s a*))
    and *R* = *Times* (*Plus* (𝔡 *r a*) (*Plus* (*Times* (𝔡 *r a*) *s*) (𝔡 *s a*))) *s*
    have *Plus L* (*Plus R* (𝔡 *s a*)) = *Plus* (*Plus L* (𝔡 *s a*)) *R* **by** (*metis Plus_assoc Plus_comm*)
    also have *Plus L* (𝔡 *s a*) = *L* **unfolding** *L_def* **by** *simp*
    finally have *Plus L* (*Plus R* (𝔡 *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* ⇒ *'a language* **where**
  Star r = *StarLR One r*

lemma 𝔬*_Star*[*simp*]: 𝔬 (*Star r*)
  unfolding *Star_def* **by** *simp*

lemma 𝔡*_Star*[*simp*]: 𝔡 (*Star r*) = (λ*a*. *Times* (𝔡 *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* ⇒ *'a language* ⇒ *'a language* **where**
  𝔬 (*Inter r s*) = (𝔬 *r* ∧ 𝔬 *s*)
| 𝔡 (*Inter r s*) = (λ*a*. *Inter* (𝔡 *r a*) (𝔡 *s a*))

primcorec *Not* :: *'a language* ⇒ *'a language* **where**
  𝔬 (*Not r*) = (¬ 𝔬 *r*)
| 𝔡 (*Not r*) = (λ*a*. *Not* (𝔡 *r a*))

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

```
  𝔬 Full = True
| 𝔡 Full = (λ_. 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 language ⇒ 'a language ⇒ ('a × bool) language where
  𝔬 (ShuffleLR r s) = (𝔬 r ∧ 𝔬 s)
| 𝔡 (ShuffleLR r s) = (λ(a, b). if b then ShuffleLR (𝔡 r a) s else ShuffleLR r (𝔡 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 language ⇒ 'a language ⇒ 'a language where
  Shuffle r s = Times_Plus (ShuffleLR r s)

lemma 𝔬_Shuffle[simp]:
  𝔬 (Shuffle r s) = (𝔬 r ∧ 𝔬 s)
  unfolding Shuffle_def by simp

lemma 𝔡_Shuffle[simp]:
  𝔡 (Shuffle r s) = (λa. Plus (Shuffle (𝔡 r a) s) (Shuffle r (𝔡 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*]: *Shuffle r* (*Plus s t*) = *Plus* (*Shuffle r s*) (*Shuffle r 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*]: *Shuffle* (*Shuffle r s*) *t* = *Shuffle r* (*Shuffle s t*)
by (*coinduction arbitrary*: *r s t rule*: *language_coinduct_upto_Plus*) *fastforce*

theorem *Shuffle_comm*[*simp*]: *Shuffle r s* = *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* ⟹ *regular_cong R x y*
| *Sym*[*intro*]: *regular_cong R x y* ⟹ *regular_cong R y x*
| *Trans*[*intro*]: ⟦*regular_cong R x y*; *regular_cong R y z*⟧ ⟹ *regular_cong R x z*
| *Base*[*intro*]: *R x y* ⟹ *regular_cong R x y*
| *Plus*[*intro*, *simp*]: ⟦*regular_cong R x y*; *regular_cong R x′ y′*⟧ ⟹
regular_cong R (*Plus x x′*) (*Plus y y′*)
| *Times*[*intro*, *simp*]: ⟦*regular_cong R x y*; *regular_cong R x′ y′*⟧ ⟹
regular_cong R (*Times x x′*) (*Times y y′*)
| *Star*[*intro*, *simp*]: ⟦*regular_cong R x y*⟧ ⟹
regular_cong R (*Star x*) (*Star y*)
| *Inter*[*intro*, *simp*]: ⟦*regular_cong R x y*; *regular_cong R x′ y′*⟧ ⟹
regular_cong R (*Inter x x′*) (*Inter y y′*)
| *Not*[*intro*, *simp*]: ⟦*regular_cong R x y*⟧ ⟹
regular_cong R (*Not x*) (*Not y*)
| *Shuffle*[*intro*, *simp*]: ⟦*regular_cong R x y*; *regular_cong R x′ y′*⟧ ⟹
regular_cong R (*Shuffle x x′*) (*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*:
(⋀*L K*. *R L K* ⟹ 𝔬 *L* = 𝔬 *K* ∧ *rel_fun* (=) (*regular_cong R*) (𝔡 *L*) (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 𝔬 *L* = 𝔬 *K* ∧ *rel_fun* (=) (*regular_cong R*) (𝔡 *L*) (𝔡 *K*)
by (*induct rule*: *regular_cong.induct*) (*auto dest*: *hyp simp*: *rel_fun_def*)
qed (*intro Base R*)

lemma *Star_unfoldR*: *Star r* = *Plus One* (*Times* (*Star r*) *r*)
proof (*coinduction arbitrary*: *r rule*: *language_coinduct_upto_regular*)
case *Lang*
{ fix *a* **have** *Plus* (*Times* (𝔡 *r a*) (*Times* (*Star r*) *r*)) (𝔡 *r a*) =
Times (𝔡 *r a*) (*Plus One* (*Times* (*Star r*) *r*)) **by** *simp*
}
then show *?case* **by** (*auto simp del*: *Times_PlusR*)
qed

lemma *Star_Star*[*simp*]: *Star* (*Star r*) = *Star r*
by (*subst Star_unfoldL*, *coinduction arbitrary*: *r rule*: *language_coinduct_upto_regular*) *auto*

lemma *Times_Star*[*simp*]: *Times* (*Star r*) (*Star r*) = *Star r*
proof (*coinduction arbitrary*: *r rule*: *language_coinduct_upto_regular*)

case *Lang*
have $*$: $\bigwedge r\ s.\ Plus\ (Times\ r\ s)\ r = Times\ r\ (Plus\ s\ One)$ **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*]: $Zero \neq One$
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 $less\_eq\_language\ r\ s = (Plus\ r\ s = s)$
definition $less\_language\ r\ s = (Plus\ r\ s = s \wedge r \neq s)$

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 $x\ y\ z :: 'a\ language$ **assume** $x \leq y\ y \leq z$
then show $x \leq z$ **unfolding** *language_defs* **by** (*metis Plus_assoc*)
next
fix $x\ y\ z :: 'a\ language$
show $x + y + z = x + (y + z)$ **unfolding** *language_defs* **by** (*rule Plus_assoc*)
qed (*auto simp*: *language_defs*)

end

lemma $\mathfrak{o}$*_mono*[*dest*]: $r \leq s \Longrightarrow \mathfrak{o}\ r \Longrightarrow \mathfrak{o}\ s$
unfolding *less_eq_language_def* **by** (*auto dest*: *arg_cong*[*of* _ _ $\mathfrak{o}$])

lemma $\mathfrak{d}$*_mono*[*dest*]: $r \leq s \Longrightarrow \mathfrak{d}\ r\ a \leq \mathfrak{d}\ s\ a$
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 $(\bigwedge L\ K.\ R\ L\ K \Longrightarrow \mathfrak{o}\ L \leq \mathfrak{o}\ K \wedge (\forall x.\ R\ (\mathfrak{d}\ L\ x)\ (\mathfrak{d}\ K\ x)))$
shows $L \leq K$
using ‹*R L K*› **unfolding** *less_eq_language_def*
by (*coinduction arbitrary*: *L K*) (*auto dest*!: *assms*(*2*))

lemma *le_PlusL*[*intro*!, *simp*]: $r \leq Plus\ r\ s$
by (*coinduction arbitrary*: *r s*) *auto*

lemma *le_PlusR*[*intro*!, *simp*]: $s \leq Plus\ r\ s$
by (*coinduction arbitrary*: *r s*) *auto*

inductive *Plus_Times_pre_cong* **for** *R* **where**
pre_Less[*intro*, *simp*]: $x \leq y \Longrightarrow Plus\_Times\_pre\_cong\ R\ x\ y$
| *pre_Trans*[*intro*]: $[\![Plus\_Times\_pre\_cong\ R\ x\ y; Plus\_Times\_pre\_cong\ R\ y\ z]\!] \Longrightarrow Plus\_Times\_pre\_cong\ R\ x\ z$
| *pre_Base*[*intro*, *simp*]: $R\ x\ y \Longrightarrow Plus\_Times\_pre\_cong\ R\ x\ y$

| *pre_Plus*[*intro*!, *simp*]: ⟦*Plus_Times_pre_cong R x y*; *Plus_Times_pre_cong R x′ y′*⟧ ⟹
Plus_Times_pre_cong R (*Plus x x′*) (*Plus y y′*)
| *pre_Times*[*intro*!, *simp*]: ⟦*Plus_Times_pre_cong R x y*; *Plus_Times_pre_cong R x′ y′*⟧ ⟹
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*: (⋀*L K*. *R L K* ⟹ 𝔬 *L* ≤ 𝔬 *K* ∧ (∀ *x*. *Plus_Times_pre_cong R* (𝔡 *L x*) (𝔡 *K x*)))
shows *L* ≤ *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 𝔬 *L* ≤ 𝔬 *K* ∧ (∀ *x*. *Plus_Times_pre_cong R* (𝔡 *L x*) (𝔡 *K x*))
by (*induct rule*: *Plus_Times_pre_cong.induct*) (*auto dest*: *hyp*)
qed (*intro pre_Base R*)

lemma *ge_One*[*simp*]: *One* ≤ *r* ⟷ 𝔬 *r*
unfolding *less_eq_language_def* **by** (*metis One.sel*(*1*) *Plus.sel*(*1*) *Plus_OneL*)

lemma *Plus_mono*: ⟦*r1* ≤ *s1*; *r2* ≤ *s2*⟧ ⟹ *Plus r1 r2* ≤ *Plus s1 s2*
by (*coinduction arbitrary*: *r1 r2 s1 s2*) (*auto elim*!: 𝔡*_mono*)

lemma *Plus_upper*: ⟦*r1* ≤ *s*; *r2* ≤ *s*⟧ ⟹ *Plus r1 r2* ≤ *s*
by (*coinduction arbitrary*: *r1 r2 s*) *auto*

lemma *Inter_mono*: ⟦*r1* ≤ *s1*; *r2* ≤ *s2*⟧ ⟹ *Inter r1 r2* ≤ *Inter s1 s2*
by (*coinduction arbitrary*: *r1 r2 s1 s2*) (*force elim*!: 𝔡*_mono*)

lemma *Times_mono*: ⟦*r1* ≤ *s1*; *r2* ≤ *s2*⟧ ⟹ *Times r1 r2* ≤ *Times s1 s2*
by (*coinduction arbitrary*: *r1 r2 s1 s2*) (*auto 0 3 elim*!: 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*)) ≤ *Star r*
by *coinduction auto*

theorem *le_StarR*: *Plus One* (*Times* (*Star r*) *r*) ≤ *Star r*
by (*rule order_eq_refl*[*OF Star_unfoldR*[*symmetric*]])

lemma *le_TimesL*[*intro*, *simp*]: 𝔬 *s* ⟹ *r* ≤ *Times r s*
by (*coinduction arbitrary*: *r*) *auto*

lemma *le_TimesR*[*intro*, *simp*]: 𝔬 *r* ⟹ *s* ≤ *Times r s*
by *coinduction auto*

lemma *Plus_le_iff*: *Plus r s* ≤ *t* ⟷ *r* ≤ *t* ∧ *s* ≤ *t*
unfolding *less_eq_language_def*
by (*metis Plus_assoc Plus_idem_assoc Plus_rotate*)

lemma *Plus_Times_pre_cong_mono*:
L′ ≤ *L* ⟹ *K* ≤ *K′* ⟹ *Plus_Times_pre_cong R L K* ⟹ *Plus_Times_pre_cong R L′ K′*
by (*metis pre_Trans pre_Less*)

theorem *ardenL*: *Plus r* (*Times s x*) ≤ *x* ⟹ *Times* (*Star s*) *r* ≤ *x*
proof (*coinduction arbitrary*: *r s x*)
case *Lang*
then show *?case*
by (*subst Plus_Times_pre_cong_mono*[*OF order_refl* 𝔡*_mono*[*OF Lang*]]) *auto*

qed

theorem *ardenR*: *Plus r* (*Times x s*) ≤ *x* ⟹ *Times r* (*Star s*) ≤ *x*
proof (*coinduction arbitrary*: *r s x*)
case *Lang*
then have *Plus_Times_pre_cong* (λ*L K*. ∃ *r s*. *L* = *Times r* (*Star s*) ∧ *Plus r* (*Times K s*) ≤ *K*)
(𝔡 (*Times r* (*Star s*)) *a*) (𝔡 *x a*) **for** *a* **using** 𝔡*_mono*[*OF Lang*, *of a*]
by (*auto 0 4 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* ≤ *Star s*
by *coinduction auto*

lemma *Star_mono*: *r* ≤ *s* ⟹ *Star r* ≤ *Star s*
by *coinduction auto*

lemma *Not_antimono*: *r* ≤ *s* ⟹ *Not s* ≤ *Not r*
by (*coinduction arbitrary*: *r s*) *auto*

lemma *Not_Plus*[*simp*]: *Not* (*Plus r s*) = *Inter* (*Not r*) (*Not s*)
by (*coinduction arbitrary*: *r s*) *auto*

lemma *Not_Inter*[*simp*]: *Not* (*Inter r s*) = *Plus* (*Not r*) (*Not s*)
by (*coinduction arbitrary*: *r s*) *auto*

lemma *Inter_assoc*[*simp*]: *Inter* (*Inter r s*) *t* = *Inter r* (*Inter s t*)
by (*coinduction arbitrary*: *r s t*) *auto*

lemma *Inter_comm*: *Inter r s* = *Inter s r*
by (*coinduction arbitrary*: *r s*) *auto*

lemma *Inter_idem*[*simp*]: *Inter r r* = *r*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Inter_ZeroL*[*simp*]: *Inter Zero r* = *Zero*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Inter_ZeroR*[*simp*]: *Inter r Zero* = *Zero*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Inter_FullL*[*simp*]: *Inter Full r* = *r*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Inter_FullR*[*simp*]: *Inter r Full* = *r*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Plus_FullL*[*simp*]: *Plus Full r* = *Full*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Plus_FullR*[*simp*]: *Plus r Full* = *Full*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Not_Not*[*simp*]: *Not* (*Not r*) = *r*
by (*coinduction arbitrary*: *r*) *auto*

lemma *Not_Zero*[*simp*]: *Not Zero* = *Full*

by *coinduction simp*

lemma *Not_Full*[*simp*]: *Not Full* = *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*)) ≤ *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* ≤ *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*))) ≤ *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*) ≤ *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*))))
≤ *Times* (*Star r*) (*Star* (*Times s* (*Star r*))) (**is** *Times _ ?L* ≤ *?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*)) ≤ *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* ⇒ *'a language* **where**
PLUS xs ≡ *foldr Plus xs Zero*

lemma 𝔬*_foldr_Plus*: 𝔬 (*foldr Plus xs s*) = (∃ *x*∈*set* (*s* # *xs*). 𝔬 *x*)
by (*induct xs arbitrary*: *s*) *auto*

lemma 𝔡*_foldr_Plus*: 𝔡 (*foldr Plus xs s*) *a* = *foldr Plus* (*map* (λ*r*. 𝔡 *r a*) *xs*) (𝔡 *s a*)
by (*induct xs arbitrary*: *s*) *simp_all*

lemma 𝔬*_PLUS*[*simp*]: 𝔬 (*PLUS xs*) = (∃ *x*∈*set xs*. 𝔬 *x*)
unfolding *PLUS_def* 𝔬*_foldr_Plus* **by** *simp*

lemma $\mathfrak{d}$*_PLUS*[*simp*]: $\mathfrak{d}$ (*PLUS xs*) *a* = *PLUS* (*map* (λ*r*. $\mathfrak{d}$ *r a*) *xs*)
unfolding *PLUS_def* $\mathfrak{d}$*_foldr_Plus* **by** *simp*

definition *TIMES* :: *'a language list* ⇒ *'a language* **where**
TIMES xs ≡ *foldr Times xs One*

lemma $\mathfrak{o}$*_foldr_Times*: $\mathfrak{o}$ (*foldr Times xs s*) = (∀ *x*∈*set* (*s* # *xs*). $\mathfrak{o}$ *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* (λ*ys*. *ys* @ [*x*]) (*tails xs*) @ [[]]
by (*induct xs*) *auto*

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

lemma $\mathfrak{d}$*_foldr_Times*: $\mathfrak{d}$ (*foldr Times xs s*) *a* =
(*let n* = *length* (*takeWhile* $\mathfrak{o}$ *xs*)
in PLUS (*map* (λ*zs*. *TIMES* ($\mathfrak{d}$ (*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 $\mathfrak{o}$*_TIMES*[*simp*]: $\mathfrak{o}$ (*TIMES xs*) = (∀ *x*∈*set xs*. $\mathfrak{o}$ *x*)
unfolding *TIMES_def* $\mathfrak{o}$*_foldr_Times* **by** *simp*

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

lemma $\mathfrak{d}$*_TIMES*[*simp*]: $\mathfrak{d}$ (*TIMES xs*) *a* = (*let n* = *length* (*takeWhile* $\mathfrak{o}$ *xs*)
in PLUS (*map* (λ*zs*. *TIMES* ($\mathfrak{d}$ (*hd zs*) *a* # *tl zs*)) (*take* (*Suc n*) (*tails* (*xs* @ [*One*])))))
unfolding *TIMES_def* $\mathfrak{d}$*_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* ⇒ *'a list* ⇒ *bool* **where**
in_language L [] = $\mathfrak{o}$ *L*
| *in_language L* (*x* # *xs*) = *in_language* ($\mathfrak{d}$ *L x*) *xs*

primcorec *to_language* :: *'a list set* ⇒ *'a language* **where**
$\mathfrak{o}$ (*to_language L*) = ([] ∈ *L*)
| $\mathfrak{d}$ (*to_language L*) = (λ*a*. *to_language* {*w*. *a* # *w* ∈ *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* ∈ *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 $L = Collect\ (in\_language\ (to\_language\ L))$ **by** *simp*
then show $\exists K.\ L = Collect\ (in\_language\ K)$ **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*]: $\neg\ in\_language\ Zero\ xs$
by (*induct xs*) *auto*

lemma *in_language_One*[*simp*]: $in\_language\ One\ xs \Longrightarrow xs = []$
by (*cases xs*) *auto*

lemma *in_language_Atom*[*simp*]: $in\_language\ (Atom\ a)\ xs \Longrightarrow xs = [a]$
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*]: $to\_language\ (-\ A) = Not\ (to\_language\ A)$
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*]: $to\_language\ (A \parallel B) = Shuffle\ (to\_language\ A)\ (to\_language\ B)$
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* $\alpha \equiv$ (*case* α *of* (*Inr* N # _) $\Rightarrow$ *False* | _ $\Rightarrow$ *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 [] []
| *in_cfl* α *w* $\Longrightarrow$ *in_cfl* (*Inl a* # α) (*a* # *w*)
| *fBex* (*prod G N*) ($\lambda\beta$. *in_cfl* (β @ α) *w*) $\Longrightarrow$ *in_cfl* (*Inr N* # α) *w*

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

fun $\mathfrak{o}_P$ **where**
  $\mathfrak{o}_P$ [] = *True*
| $\mathfrak{o}_P$ (*Inl* _ # _) = *False*
| $\mathfrak{o}_P$ (*Inr N* # α) = ([] |$\in$| *prod G N* $\wedge$ $\mathfrak{o}_P$ α)

fun $\mathfrak{d}_P$ **where**
  $\mathfrak{d}_P$ [] *a* = {||}
| $\mathfrak{d}_P$ (*Inl b* # α) *a* = (*if a* = *b then* {|α|} *else* {||})
| $\mathfrak{d}_P$ (*Inr N* # α) *a* =
    ($\lambda\beta$. *tl* β @ α) |‘| *ffilter* ($\lambda\beta$. $\beta \neq$ [] $\wedge$ *hd* β = *Inl a*) (*prod G N*) |∪|
    (*if* [] |$\in$| *prod G N then* $\mathfrak{d}_P$ α *a else* {||})

primcorec *subst* :: ('*t* + '*n*) *list fset* $\Rightarrow$ '*t language* **where**
  subst P = *Lang* (*fBex P* $\mathfrak{o}_P$) (λ*a*. *subst* (*ffUnion* ((λ*r*. $\mathfrak{d}_P$ *r a*) |‘| *P*)))

inductive *in_cfls* **where**
  fBex P $\mathfrak{o}_P$ $\Longrightarrow$ *in_cfls P* []
| *in_cfls* (*ffUnion* (($\lambda\alpha$. $\mathfrak{d}_P$ α *a*) |‘| *P*)) *w* $\Longrightarrow$ *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* = *to_language* {*w*. *in_cfls P w*}
  by (*coinduction arbitrary*: *P*) (*auto intro*: *in_cfls.intros*)

lemma $\mathfrak{o}_P$*_in_cfl*: $\mathfrak{o}_P$ α $\Longrightarrow$ *in_cfl* α []
  by (*induct* α *rule*: $\mathfrak{o}_P$*.induct*) (*auto intro*!: *in_cfl.intros elim*: *fBexI*[*rotated*])

lemma $\mathfrak{d}_P$*_in_cfl*: β |$\in$| $\mathfrak{d}_P$ α *a* $\Longrightarrow$ *in_cfl* β *w* $\Longrightarrow$ *in_cfl* α (*a* # *w*)
proof (*induct* α *a arbitrary*: β *w rule*: $\mathfrak{d}_P$*.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 ⟹ 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 ⟹ P |⊆| Q ⟹ 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!: ffunion_mono in_cfls.intros)
qed (auto intro!: in_cfls.intros)

end

locale cfg_wgreibach =
  fixes G :: ('t, 'n :: finite) cfg
  assumes weakGreibach: ⋀N α. α |∈| prod G N ⟹ wgreibach α
begin

lemma in_cfl_in_cfls: in_cfl G α w ⟹ in_cfls G {|α|} w
proof (induct α w rule: in_cfl.induct)
  case (3 N α w)
  then obtain β where
    β: β |∈| 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 ≡ subst G {|[Inr (init G)]|}

lemma lang_lang_trad: lang = to_language (lang_trad G)
proof −
  have in_cfls G {|[Inr (init G)]|} w ⟷ 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 $\mathfrak{a} \equiv$ *Inl True*
abbreviation $\mathfrak{b} \equiv$ *Inl False*
abbreviation $S \equiv$ *Inr* ()

interpretation *palindromes*: *cfg_wgreibach* (|*init* = (), *prod* = λ_. {|[], [$\mathfrak{a}$], [$\mathfrak{b}$], [$\mathfrak{a}$, S, $\mathfrak{a}$], [$\mathfrak{b}$, S, $\mathfrak{b}$]|}|)
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 ¬ *in_language palindromes.lang* [*True*, *False*] **by** *normalization*
lemma ¬ *in_language palindromes.lang* [*True*, *False*, *True*, *False*] **by** *normalization*
lemma *in_language palindromes.lang* [*True*, *False*, *True*, *True*, *False*, *True*] **by** *normalization*
lemma ¬ *in_language palindromes.lang* [*True*, *False*, *True*, *False*, *False*, *True*] **by** *normalization*

interpretation *Dyck*: *cfg_wgreibach* (|*init* = (), *prod* = λ_. {|[], [$\mathfrak{a}$, S, $\mathfrak{b}$, S]|}|)
by *unfold_locales auto*
lemma *in_language Dyck.lang* [] **by** *normalization*
lemma ¬ *in_language Dyck.lang* [*True*] **by** *normalization*
lemma ¬ *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 ¬ *in_language Dyck.lang* [*True*, *False*, *True*, *True*, *False*] **by** *normalization*
lemma ¬ *in_language Dyck.lang* [*True*, *True*, *False*, *False*, *False*, *True*] **by** *normalization*

interpretation *abSSa*: *cfg_wgreibach* (|*init* = (), *prod* = λ_. {|[], [$\mathfrak{a}$, $\mathfrak{b}$, S, S, $\mathfrak{a}$]|}|)
by *unfold_locales auto*
lemma *in_language abSSa.lang* [] **by** *normalization*
lemma ¬ *in_language abSSa.lang* [*True*] **by** *normalization*
lemma ¬ *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 ¬ *in_language abSSa.lang* [*True*, *False*, *True*, *True*, *False*] **by** *normalization*
lemma ¬ *in_language abSSa.lang* [*True*, *True*, *False*, *False*, *False*, *True*] **by** *normalization*