A Codatatype of Formal Languages

Dmitriy Traytel

May 21, 2019

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* ⇒ *'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* ⇒ *'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*
  ⟨*proof*⟩

theorem *Plus_ZeroR*[*simp*]: *Plus r Zero* = *r*
  ⟨*proof*⟩

theorem *Plus_assoc*: *Plus* (*Plus r s*) *t* = *Plus r* (*Plus s t*)
⟨*proof*⟩

theorem *Plus_comm*: *Plus r s* = *Plus s r*
⟨*proof*⟩

lemma *Plus_rotate*: *Plus r* (*Plus s t*) = *Plus s* (*Plus r t*)
⟨*proof*⟩

theorem *Plus_idem*: *Plus r r* = *r*
⟨*proof*⟩

lemma *Plus_idem_assoc*: *Plus r* (*Plus r s*) = *Plus r s*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Plus_OneR*[*simp*]: 𝔬 *r* ⟹ *Plus r One* = *r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *TimesLR_ZeroR*[*simp*]: *TimesLR r Zero* = *Zero*
⟨*proof*⟩

lemma *TimesLR_PlusL*[*simp*]: *TimesLR* (*Plus r s*) *t* = *Plus* (*TimesLR r t*) (*TimesLR s t*)
⟨*proof*⟩

lemma *TimesLR_PlusR*[*simp*]: *TimesLR r* (*Plus s t*) = *Plus* (*TimesLR r s*) (*TimesLR r t*)
⟨*proof*⟩

lemma *Times_Plus_Zero*[*simp*]: *Times_Plus Zero* = *Zero*
⟨*proof*⟩

lemma *Times_Plus_Plus*[*simp*]: *Times_Plus* (*Plus r s*) = *Plus* (*Times_Plus r*) (*Times_Plus s*)
⟨*proof*⟩

lemma *Times_Plus_TimesLR_One*[*simp*]: *Times_Plus* (*TimesLR r One*) = *r*
⟨*proof*⟩

lemma *Times_Plus_TimesLR_PlusL*[*simp*]:
Times_Plus (*TimesLR* (*Plus r s*) *t*) = *Plus* (*Times_Plus* (*TimesLR r t*)) (*Times_Plus* (*TimesLR s t*))
⟨*proof*⟩

lemma *Times_Plus_TimesLR_PlusR*[*simp*]:
Times_Plus (*TimesLR r* (*Plus s t*)) = *Plus* (*Times_Plus* (*TimesLR r s*)) (*Times_Plus* (*TimesLR r t*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *Times_ZeroL*[*simp*]: *Times Zero r* = *Zero*
⟨*proof*⟩

theorem *Times_ZeroR*[*simp*]: *Times r Zero* = *Zero*
⟨*proof*⟩

theorem *Times_OneL*[*simp*]: *Times One r* = *r*
⟨*proof*⟩

theorem *Times_OneR*[*simp*]: *Times r One* = *r*
⟨*proof*⟩

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*⟩

theorem *Times_PlusL*[*simp*]: *Times* (*Plus r s*) *t* = *Plus* (*Times r t*) (*Times s t*)
⟨*proof*⟩

theorem *Times_PlusR*[*simp*]: *Times r* (*Plus s t*) = *Plus* (*Times r s*) (*Times r t*)
⟨*proof*⟩

theorem *Times_assoc*[*simp*]: *Times* (*Times r s*) *t* = *Times r* (*Times s t*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *StarLR_Plus*[*simp*]: *StarLR* (*Plus r s*) *t* = *Plus* (*StarLR r t*) (*StarLR s t*)
⟨*proof*⟩

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

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

definition *Star* :: *'a language* ⇒ *'a language* **where**
Star r = *StarLR One r*

lemma 𝔬_*Star*[*simp*]: 𝔬 (*Star r*)
⟨*proof*⟩

lemma 𝔡_*Star*[*simp*]: 𝔡 (*Star r*) = (λ*a*. *Times* (𝔡 *r a*) (*Star r*))
⟨*proof*⟩

lemma *Star_Zero*[*simp*]: *Star Zero* = *One*
⟨*proof*⟩

lemma *Star_One*[*simp*]: *Star One* = *One*
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *ShuffleLR_ZeroR*[*simp*]: *ShuffleLR r Zero* = *Zero*
⟨*proof*⟩

lemma *ShuffleLR_PlusL*[*simp*]: *ShuffleLR* (*Plus r s*) *t* = *Plus* (*ShuffleLR r t*) (*ShuffleLR s t*)
⟨*proof*⟩

lemma *ShuffleLR_PlusR*[*simp*]: *ShuffleLR r* (*Plus s t*) = *Plus* (*ShuffleLR r s*) (*ShuffleLR r t*)

⟨*proof*⟩

lemma *Shuffle_Plus_ShuffleLR_One*[*simp*]: *Times_Plus* (*ShuffleLR r One*) = *r*
⟨*proof*⟩

lemma *Shuffle_Plus_ShuffleLR_PlusL*[*simp*]:
Times_Plus (*ShuffleLR* (*Plus r s*) *t*) = *Plus* (*Times_Plus* (*ShuffleLR r t*)) (*Times_Plus* (*ShuffleLR s t*))
⟨*proof*⟩

lemma *Shuffle_Plus_ShuffleLR_PlusR*[*simp*]:
Times_Plus (*ShuffleLR r* (*Plus s t*)) = *Plus* (*Times_Plus* (*ShuffleLR r s*)) (*Times_Plus* (*ShuffleLR r t*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma 𝔡_*Shuffle*[*simp*]:
𝔡 (*Shuffle r s*) = (λ*a*. *Plus* (*Shuffle* (𝔡 *r a*) *s*) (*Shuffle r* (𝔡 *s a*)))
⟨*proof*⟩

theorem *Shuffle_ZeroL*[*simp*]: *Shuffle Zero r* = *Zero*
⟨*proof*⟩

theorem *Shuffle_ZeroR*[*simp*]: *Shuffle r Zero* = *Zero*
⟨*proof*⟩

theorem *Shuffle_OneL*[*simp*]: *Shuffle One r* = *r*
⟨*proof*⟩

theorem *Shuffle_OneR*[*simp*]: *Shuffle r One* = *r*
⟨*proof*⟩

theorem *Shuffle_PlusL*[*simp*]: *Shuffle* (*Plus r s*) *t* = *Plus* (*Shuffle r t*) (*Shuffle s t*)
⟨*proof*⟩

theorem *Shuffle_PlusR*[*simp*]: *Shuffle r* (*Plus s t*) = *Plus* (*Shuffle r s*) (*Shuffle r t*)
⟨*proof*⟩

theorem *Shuffle_assoc*[*simp*]: *Shuffle* (*Shuffle r s*) *t* = *Shuffle r* (*Shuffle s t*)
⟨*proof*⟩

theorem *Shuffle_comm*[*simp*]: *Shuffle r s* = *Shuffle s r*
⟨*proof*⟩

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*⟩

lemma *Star_unfoldR*: *Star r* = *Plus One* (*Times* (*Star r*) *r*)
⟨*proof*⟩

lemma *Star_Star*[*simp*]: *Star* (*Star r*) = *Star r*
  ⟨*proof*⟩

lemma *Times_Star*[*simp*]: *Times* (*Star r*) (*Star r*) = *Star r*
⟨*proof*⟩

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

lemma *Zero_One*[*simp*]: *Zero* ≠ *One*
  ⟨*proof*⟩

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* ∧ *r* ≠ *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*⟩

end

lemma 𝔬*_mono*[*dest*]: *r* ≤ *s* ⟹ 𝔬 *r* ⟹ 𝔬 *s*
  ⟨*proof*⟩

lemma 𝔡*_mono*[*dest*]: *r* ≤ *s* ⟹ 𝔡 *r a* ≤ 𝔡 *s a*
  ⟨*proof*⟩

For reasoning about (≤), 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 (⋀*L K*. *R L K* ⟹ 𝔬 *L* ≤ 𝔬 *K* ∧ (∀*x*. *R* (𝔡 *L x*) (𝔡 *K x*)))
  shows *L* ≤ *K*
  ⟨*proof*⟩

lemma *le_PlusL*[*intro*!, *simp*]: $r \leq \mathit{Plus}\ r\ s$
⟨*proof*⟩

lemma *le_PlusR*[*intro*!, *simp*]: $s \leq \mathit{Plus}\ r\ s$
⟨*proof*⟩

inductive *Plus_Times_pre_cong* **for** R **where**
pre_Less[*intro*, *simp*]: $x \leq y \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ x\ y$
| *pre_Trans*[*intro*]: $[\![\mathit{Plus\_Times\_pre\_cong}\ R\ x\ y;\ \mathit{Plus\_Times\_pre\_cong}\ R\ y\ z]\!] \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ x\ z$
| *pre_Base*[*intro*, *simp*]: $R\ x\ y \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ x\ y$
| *pre_Plus*[*intro*!, *simp*]: $[\![\mathit{Plus\_Times\_pre\_cong}\ R\ x\ y;\ \mathit{Plus\_Times\_pre\_cong}\ R\ x'\ y']\!] \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ (\mathit{Plus}\ x\ x')\ (\mathit{Plus}\ y\ y')$
| *pre_Times*[*intro*!, *simp*]: $[\![\mathit{Plus\_Times\_pre\_cong}\ R\ x\ y;\ \mathit{Plus\_Times\_pre\_cong}\ R\ x'\ y']\!] \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ (\mathit{Times}\ x\ x')\ (\mathit{Times}\ y\ y')$

theorem *language_simulation_coinduction_upto_Plus_Times*[*consumes 1*, *case_names Lang*, *coinduct pred*]:
assumes R: $R\ L\ K$
and *hyp*: $(\bigwedge L\ K.\ R\ L\ K \Longrightarrow \mathfrak{o}\ L \leq \mathfrak{o}\ K \wedge (\forall x.\ \mathit{Plus\_Times\_pre\_cong}\ R\ (\mathfrak{d}\ L\ x)\ (\mathfrak{d}\ K\ x)))$
shows $L \leq K$
⟨*proof*⟩

lemma *ge_One*[*simp*]: $\mathit{One} \leq r \longleftrightarrow \mathfrak{o}\ r$
⟨*proof*⟩

lemma *Plus_mono*: $[\![r1 \leq s1;\ r2 \leq s2]\!] \Longrightarrow \mathit{Plus}\ r1\ r2 \leq \mathit{Plus}\ s1\ s2$
⟨*proof*⟩

lemma *Plus_upper*: $[\![r1 \leq s;\ r2 \leq s]\!] \Longrightarrow \mathit{Plus}\ r1\ r2 \leq s$
⟨*proof*⟩

lemma *Inter_mono*: $[\![r1 \leq s1;\ r2 \leq s2]\!] \Longrightarrow \mathit{Inter}\ r1\ r2 \leq \mathit{Inter}\ s1\ s2$
⟨*proof*⟩

lemma *Times_mono*: $[\![r1 \leq s1;\ r2 \leq s2]\!] \Longrightarrow \mathit{Times}\ r1\ r2 \leq \mathit{Times}\ s1\ s2$
⟨*proof*⟩

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*: $\mathit{Plus}\ \mathit{One}\ (\mathit{Times}\ r\ (\mathit{Star}\ r)) \leq \mathit{Star}\ r$
⟨*proof*⟩

theorem *le_StarR*: $\mathit{Plus}\ \mathit{One}\ (\mathit{Times}\ (\mathit{Star}\ r)\ r) \leq \mathit{Star}\ r$
⟨*proof*⟩

lemma *le_TimesL*[*intro*, *simp*]: $\mathfrak{o}\ s \Longrightarrow r \leq \mathit{Times}\ r\ s$
⟨*proof*⟩

lemma *le_TimesR*[*intro*, *simp*]: $\mathfrak{o}\ r \Longrightarrow s \leq \mathit{Times}\ r\ s$
⟨*proof*⟩

lemma *Plus_le_iff*: $\mathit{Plus}\ r\ s \leq t \longleftrightarrow r \leq t \wedge s \leq t$
⟨*proof*⟩

lemma *Plus_Times_pre_cong_mono*:
$L' \leq L \Longrightarrow K \leq K' \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ L\ K \Longrightarrow \mathit{Plus\_Times\_pre\_cong}\ R\ L'\ K'$
⟨*proof*⟩

theorem *ardenL*: $Plus\ r\ (Times\ s\ x) \leq x \Longrightarrow Times\ (Star\ s)\ r \leq x$
⟨*proof*⟩

theorem *ardenR*: $Plus\ r\ (Times\ x\ s) \leq x \Longrightarrow Times\ r\ (Star\ s) \leq x$
⟨*proof*⟩

lemma *le_Star*[*intro*!, *simp*]: $s \leq Star\ s$
⟨*proof*⟩

lemma *Star_mono*: $r \leq s \Longrightarrow Star\ r \leq Star\ s$
⟨*proof*⟩

lemma *Not_antimono*: $r \leq s \Longrightarrow Not\ s \leq Not\ r$
⟨*proof*⟩

lemma *Not_Plus*[*simp*]: $Not\ (Plus\ r\ s) = Inter\ (Not\ r)\ (Not\ s)$
⟨*proof*⟩

lemma *Not_Inter*[*simp*]: $Not\ (Inter\ r\ s) = Plus\ (Not\ r)\ (Not\ s)$
⟨*proof*⟩

lemma *Inter_assoc*[*simp*]: $Inter\ (Inter\ r\ s)\ t = Inter\ r\ (Inter\ s\ t)$
⟨*proof*⟩

lemma *Inter_comm*: $Inter\ r\ s = Inter\ s\ r$
⟨*proof*⟩

lemma *Inter_idem*[*simp*]: $Inter\ r\ r = r$
⟨*proof*⟩

lemma *Inter_ZeroL*[*simp*]: $Inter\ Zero\ r = Zero$
⟨*proof*⟩

lemma *Inter_ZeroR*[*simp*]: $Inter\ r\ Zero = Zero$
⟨*proof*⟩

lemma *Inter_FullL*[*simp*]: $Inter\ Full\ r = r$
⟨*proof*⟩

lemma *Inter_FullR*[*simp*]: $Inter\ r\ Full = r$
⟨*proof*⟩

lemma *Plus_FullL*[*simp*]: $Plus\ Full\ r = Full$
⟨*proof*⟩

lemma *Plus_FullR*[*simp*]: $Plus\ r\ Full = Full$
⟨*proof*⟩

lemma *Not_Not*[*simp*]: $Not\ (Not\ r) = r$
⟨*proof*⟩

lemma *Not_Zero*[*simp*]: $Not\ Zero = Full$
⟨*proof*⟩

lemma *Not_Full*[*simp*]: $Not\ Full = Zero$
⟨*proof*⟩

lemma *bisimulation*:
assumes *Times r s* = *Times s t*
shows *Times* (*Star r*) *s* = *Times s* (*Star t*)
⟨*proof*⟩

lemma *sliding*: *Times* (*Star* (*Times r s*)) *r* = *Times r* (*Star* (*Times s r*))
⟨*proof*⟩

lemma *denesting*: *Star* (*Plus r s*) = *Times* (*Star r*) (*Star* (*Times s* (*Star r*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma 𝔡_*foldr_Plus*: 𝔡 (*foldr Plus xs s*) *a* = *foldr Plus* (*map* (λ*r*. 𝔡 *r a*) *xs*) (𝔡 *s a*)
⟨*proof*⟩

lemma 𝔬_*PLUS*[*simp*]: 𝔬 (*PLUS xs*) = (∃ *x*∈*set xs*. 𝔬 *x*)
⟨*proof*⟩

lemma 𝔡_*PLUS*[*simp*]: 𝔡 (*PLUS xs*) *a* = *PLUS* (*map* (λ*r*. 𝔡 *r a*) *xs*)
⟨*proof*⟩

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

lemma 𝔬_*foldr_Times*: 𝔬 (*foldr Times xs s*) = (∀ *x*∈*set* (*s* # *xs*). 𝔬 *x*)
⟨*proof*⟩

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*) @ [[]]
⟨*proof*⟩

lemma *length_tails*[*simp*]: *length* (*tails xs*) = *Suc* (*length xs*)
⟨*proof*⟩

lemma 𝔡_*foldr_Times*: 𝔡 (*foldr Times xs s*) *a* =
(*let n* = *length* (*takeWhile* 𝔬 *xs*)
in PLUS (*map* (λ*zs*. *TIMES* (𝔡 (*hd zs*) *a* # *tl zs*)) (*take* (*Suc n*) (*tails* (*xs* @ [*s*])))))
⟨*proof*⟩

lemma 𝔬_*TIMES*[*simp*]: 𝔬 (*TIMES xs*) = (∀ *x*∈*set xs*. 𝔬 *x*)
⟨*proof*⟩

lemma *TIMES_snoc_One*[*simp*]: *TIMES* (*xs* @ [*One*]) = *TIMES xs*
⟨*proof*⟩

lemma 𝔡_*TIMES*[*simp*]: 𝔡 (*TIMES xs*) *a* = (*let n* = *length* (*takeWhile* 𝔬 *xs*)
in PLUS (*map* (λ*zs*. *TIMES* (𝔡 (*hd zs*) *a* # *tl zs*)) (*take* (*Suc n*) (*tails* (*xs* @ [*One*])))))
⟨*proof*⟩

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 [] = 𝔬 *L*
| *in_language L* (*x* # *xs*) = *in_language* (𝔡 *L x*) *xs*

primcorec *to_language* :: *'a list set* ⇒ *'a language* **where**
  𝔬 (*to_language L*) = ([] ∈ *L*)
| 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*⟩

lemma *to_language_in_language*[*simp*]: *to_language* (*Collect* (*in_language L*)) = *L*
  ⟨*proof*⟩

lemma *in_language_bij*: *bij* (*Collect o in_language*)
⟨*proof*⟩

lemma *to_language_bij*: *bij to_language*
  ⟨*proof*⟩

4 Coinductively Defined Operations Are Standard

lemma *to_language_empty*[*simp*]: *to_language* {} = *Zero*
  ⟨*proof*⟩

lemma *in_language_Zero*[*simp*]: ¬ *in_language Zero xs*
  ⟨*proof*⟩

lemma *in_language_One*[*simp*]: *in_language One xs* ⟹ *xs* = []
  ⟨*proof*⟩

lemma *in_language_Atom*[*simp*]: *in_language* (*Atom a*) *xs* ⟹ *xs* = [*a*]
  ⟨*proof*⟩

lemma *to_language_eps*[*simp*]: *to_language* {[]} = *One*
  ⟨*proof*⟩

lemma *to_language_singleton*[*simp*]: *to_language* {[*a*]} = (*Atom a*)
  ⟨*proof*⟩

lemma *to_language_Un*[*simp*]: *to_language* (*A* ∪ *B*) = *Plus* (*to_language A*) (*to_language B*)
  ⟨*proof*⟩

lemma *to_language_Int*[*simp*]: *to_language* (*A* ∩ *B*) = *Inter* (*to_language A*) (*to_language B*)
  ⟨*proof*⟩

lemma *to_language_Neg*[*simp*]: *to_language* (− *A*) = *Not* (*to_language A*)
  ⟨*proof*⟩

lemma *to_language_Diff*[*simp*]: *to_language* (*A* − *B*) = *Inter* (*to_language A*) (*Not* (*to_language B*))
  ⟨*proof*⟩

lemma *to_language_conc*[*simp*]: *to_language* (*A* @@ *B*) = *Times* (*to_language A*) (*to_language B*)
⟨*proof*⟩

lemma *to_language_star*[*simp*]: *to_language* (*star A*) = *Star* (*to_language A*)
⟨*proof*⟩

lemma *to_language_shuffle*[*simp*]: *to_language* (*A* ‖ *B*) = *Shuffle* (*to_language A*) (*to_language B*)
⟨*proof*⟩

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* *α* ≡ (*case α of* (*Inr N* # _) ⇒ *False* | _ ⇒ *True*)

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

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

inductive *in_cfl* **where**
in_cfl [] []
| *in_cfl α w* ⟹ *in_cfl* (*Inl a* # *α*) (*a* # *w*)
| *fBex* (*prod G N*) (λ*β*. *in_cfl* (*β* @ *α*) *w*) ⟹ *in_cfl* (*Inr N* # *α*) *w*

abbreviation *lang_trad* **where**
lang_trad ≡ {*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*) = ([] |∈| *prod G N* ∧ $\mathfrak{o}_P$ *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* =
(λ*β*. *tl β* @ *α*) |‘| *ffilter* (λ*β*. *β* ≠ [] ∧ *hd β* = *Inl a*) (*prod G N*) |∪|
(*if* [] |∈| *prod G N then* $\mathfrak{d}_P$ *α a else* {||})

primcorec *subst* :: ('*t* + '*n*) *list fset* ⇒ '*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 $\mathfrak{o}_P$ ⟹ *in_cfls P* []
| *in_cfls* (*ffUnion* ((λ*α*. $\mathfrak{d}_P$ *α a*) |‘| *P*)) *w* ⟹ *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*}
⟨*proof*⟩

lemma $\mathfrak{o}_P$*_in_cfl*: $\mathfrak{o}_P\ \alpha \Longrightarrow$ *in_cfl* α []
⟨*proof*⟩

lemma $\mathfrak{d}_P$*_in_cfl*: β |∈| $\mathfrak{d}_P\ \alpha\ a \Longrightarrow$ *in_cfl* $\beta\ w \Longrightarrow$ *in_cfl* α (a # w)
⟨*proof*⟩

lemma *in_cfls_in_cfl*: *in_cfls P w* ⟹ *fBex P* (λα. *in_cfl* α *w*)
⟨*proof*⟩

lemma *in_cfls_mono*: *in_cfls P w* ⟹ *P* |⊆| *Q* ⟹ *in_cfls Q w*
⟨*proof*⟩

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*⟩

abbreviation *lang* **where**
lang ≡ *subst G* {|[*Inr* (*init G*)]|}

lemma *lang_lang_trad*: *lang* = *to_language* (*lang_trad G*)
⟨*proof*⟩

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 $\mathfrak{a}$ ≡ *Inl True*
abbreviation $\mathfrak{b}$ ≡ *Inl False*
abbreviation *S* ≡ *Inr* ()

interpretation *palindromes*: *cfg_wgreibach* (|*init* = (), *prod* = λ_. {|[], [$\mathfrak{a}$], [$\mathfrak{b}$], [$\mathfrak{a}$, *S*, $\mathfrak{a}$], [$\mathfrak{b}$, *S*, $\mathfrak{b}$]|}|)
⟨*proof*⟩

lemma *in_language palindromes.lang* [] ⟨*proof*⟩
lemma *in_language palindromes.lang* [*True*] ⟨*proof*⟩
lemma *in_language palindromes.lang* [*False*] ⟨*proof*⟩
lemma *in_language palindromes.lang* [*True*, *True*] ⟨*proof*⟩
lemma *in_language palindromes.lang* [*True*, *False*, *True*] ⟨*proof*⟩
lemma ¬ *in_language palindromes.lang* [*True*, *False*] ⟨*proof*⟩
lemma ¬ *in_language palindromes.lang* [*True*, *False*, *True*, *False*] ⟨*proof*⟩
lemma *in_language palindromes.lang* [*True*, *False*, *True*, *True*, *False*, *True*] ⟨*proof*⟩
lemma ¬ *in_language palindromes.lang* [*True*, *False*, *True*, *False*, *False*, *True*] ⟨*proof*⟩

interpretation *Dyck*: *cfg_wgreibach* (|*init* = (), *prod* = λ_. {|[], [$\mathfrak{a}$, *S*, $\mathfrak{b}$, *S*]|}|)
⟨*proof*⟩
lemma *in_language Dyck.lang* [] ⟨*proof*⟩

lemma ¬ *in_language Dyck.lang* [*True*] ⟨*proof*⟩
lemma ¬ *in_language Dyck.lang* [*False*] ⟨*proof*⟩
lemma *in_language Dyck.lang* [*True*, *False*, *True*, *False*] ⟨*proof*⟩
lemma *in_language Dyck.lang* [*True*, *True*, *False*, *False*] ⟨*proof*⟩
lemma *in_language Dyck.lang* [*True*, *False*, *True*, *False*] ⟨*proof*⟩
lemma *in_language Dyck.lang* [*True*, *False*, *True*, *False*, *True*, *True*, *False*, *False*] ⟨*proof*⟩
lemma ¬ *in_language Dyck.lang* [*True*, *False*, *True*, *True*, *False*] ⟨*proof*⟩
lemma ¬ *in_language Dyck.lang* [*True*, *True*, *False*, *False*, *False*, *True*] ⟨*proof*⟩

interpretation *abSSa*: *cfg_wgreibach* (|*init* = (), *prod* = λ_. {|[], [𝔞, 𝔟, *S*, *S*, 𝔞]|}|)
⟨*proof*⟩
lemma *in_language abSSa.lang* [] ⟨*proof*⟩
lemma ¬ *in_language abSSa.lang* [*True*] ⟨*proof*⟩
lemma ¬ *in_language abSSa.lang* [*False*] ⟨*proof*⟩
lemma *in_language abSSa.lang* [*True*, *False*, *True*] ⟨*proof*⟩
lemma *in_language abSSa.lang* [*True*, *False*, *True*, *False*, *True*, *True*, *False*, *True*, *True*] ⟨*proof*⟩
lemma *in_language abSSa.lang* [*True*, *False*, *True*, *False*, *True*, *True*] ⟨*proof*⟩
lemma ¬ *in_language abSSa.lang* [*True*, *False*, *True*, *True*, *False*] ⟨*proof*⟩
lemma ¬ *in_language abSSa.lang* [*True*, *True*, *False*, *False*, *False*, *True*] ⟨*proof*⟩