A Codatatype of Formal Languages

Dmitriy Traytel

September 19, 2015

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.

\[
\text{codatatype 'a language} = \text{Lang (o: bool) (d: 'a ⇒ 'a language)}
\]

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

\[
\text{primcorec Zero :: 'a language where}
\]
\[
\text{o Zero} = \text{False}
\]
\[
\text{d Zero} = (\lambda_. \text{Zero})
\]

\[
\text{primcorec One :: 'a language where}
\]
\[
\text{o One} = \text{True}
\]
\[
\text{d One} = (\Lambda_. \text{Zero})
\]

\[
\text{primcorec Atom :: 'a ⇒ 'a language where}
\]
\[
\text{o (Atom a)} = \text{False}
\]
\[
\text{d (Atom a)} = (\lambda b. \text{if a = b then One else Zero})
\]

\[
\text{primcorec Plus :: 'a language ⇒ 'a language ⇒ 'a language where}
\]
\[
\text{o (Plus r s)} = (\text{o r} \lor \text{o s})
\]
\[
\text{d (Plus r s)} = (\lambda a. \text{Plus (d r a) (d s a)})
\]

\text{theorem Plus_ZeroL[simp]: Plus Zero r = r}
\text{by (coinduction arbitrary: r) simp}

\text{theorem Plus_ZeroR[simp]: Plus r Zero = r}
\text{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_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 intro: exI [of Plus])

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 :: ('a × bool) language ⇒ 'a language where
o (Times r) = o r
| d (Times r) = (λa. Times (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 TimesLR_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_PlusZero [simp]: Times_Plus Zero = Zero
by coinduction simp

lemma Times_PlusOneL [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 o_Times[simp]:
o (Times r s) = (λa. if o r then Plus (Times (o r a) s) (o s a) else Times (o r a) s)

unfolding Times_def by (rule ext, coinduction arbitrary: r s) 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 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_coinductuptoPlus[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 op = (Plus_cong R) (o L) (o 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 op = (Plus_cong R) (o L) (o K) using hyp
  by (induct rule: Plus_cong.induct) (auto simp: rel_fun_def intro: Sym)
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_coinductuptoPlus) auto

theorem Times_PlusR[simp]: Times r (Plus s t) = Plus (Times r s) (Times r t)
by (coinduction arbitrary: r s rule: language_coinductuptoPlus) fastforce

theorem Times_assoc[simp]: Times (Times r s) t = Times r (Times s t)
Similarly to \textit{Times}, iteration is not primitively corecursive (guardedness by \textit{Times} is required). We apply a similar trick to obtain its definition.

\begin{verbatim}
primcorec StarLR :: 'a language \Rightarrow 'a language where
\sigma (StarLR r s) = \sigma r
| \delta (StarLR r s) = (\lambda a. StarLR (\delta (Times r (Plus One s)) a) s)
\end{verbatim}

\begin{verbatim}
lemma StarLR_Zero[simp]: StarLR Zero r = Zero
by coinduction auto
\end{verbatim}

\begin{verbatim}
lemma StarLR_Plus[simp]: StarLR (Plus r s) t = Plus (StarLR r t) (StarLR s t)
by (coinduction arbitrary: r s t rule: language_coinduct_upto_Plus)
\end{verbatim}

\begin{verbatim}
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
def L ≡ Plus (\delta r a) (Plus (Times (\delta r a) s) (\delta s a))
and R ≡ Times (\delta r a) (Plus (Times (\delta r a) s) (\delta s a)) s
have Plus L (Plus R (\delta s a)) = Plus (Plus L (\delta s a)) R by (metis Plus_assoc Plus_comm)
also have Plus L (\delta s a) = L unfolding L_def by simp
finally have Plus L (Plus R (\delta s a)) = Plus L R .
}
then show ?case by (auto simp del: StarLR_Plus_Plus assoc Times_PlusL)
qed
\end{verbatim}

\begin{verbatim}
definition Star :: 'a language \Rightarrow 'a language where
Star r = StarLR One r
\end{verbatim}

\begin{verbatim}
lemma o_Star[simp]: o (Star r)
unfolding Star_def by simp
\end{verbatim}

\begin{verbatim}
lemma \delta_Star[simp]: \delta (Star r) = (\lambda a. Times (\delta r a) (Star r))
unfolding Star_def by (rule ext, coinduction arbitrary: r)
(auto simp add: Star_def StarLR_Times[ symmetric])
\end{verbatim}

\begin{verbatim}
lemma Star_Zero[simp]: Star Zero = One
by coinduction auto
\end{verbatim}

\begin{verbatim}
lemma Star_One[simp]: Star One = One
by coinduction auto
\end{verbatim}

\begin{verbatim}
lemma Star_unfoldL: Star r = Plus One (Times r (Star r))
by coinduction auto
\end{verbatim}

\begin{verbatim}
primcorec Inter :: 'a language \Rightarrow 'a language where
\sigma (Inter r s) = (\sigma r \land \sigma s)
| \delta (Inter r s) = (\lambda a. Inter (\delta r a) (\delta s a))
\end{verbatim}

\begin{verbatim}
primcorec Not :: 'a language \Rightarrow 'a language where
\sigma (Not r) = (\sim \sigma r)
| \delta (Not r) = (\lambda a. Not (\delta r a))
\end{verbatim}
\textbf{primcorec} Full :: 'a language \((\Sigma^*)\) where
\begin{align*}
\circ \text{ Full} &= \text{True} \\
\circ \text{ Full} &= (\lambda x. \text{Full})
\end{align*}
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.

\textbf{primcorec} ShuffleLR :: 'a language \Rightarrow 'a language \Rightarrow ('a \times \text{bool}) language where
\begin{align*}
\circ \text{ (ShuffleLR } r s) &= (\circ r \land \circ s) \\
\circ \text{ (ShuffleLR } r s) &= (\lambda (a, b). \text{ if } b \text{ then ShuffleLR } (\circ r a) s \text{ else ShuffleLR } r (\circ s a))
\end{align*}

\textbf{primcorec} Shuffle_Plus :: ('a \times \text{bool}) language \Rightarrow 'a language where
\begin{align*}
\circ \text{ (Shuffle_Plus } r) &= \circ r \\
\circ \text{ (Shuffle_Plus } r) &= (\lambda a. \text{ Shuffle_Plus } (\circ r (a, \text{True})) (\circ r (a, \text{False})))
\end{align*}

\textbf{lemma} ShuffleLR_ZeroL[simp]: ShuffleLR Zero r = Zero
by (coinduction arbitrary: r) auto

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

\textbf{lemma} ShuffleLR_PlusL[simp]: ShuffleLR (Plus r s) t = Plus (ShuffleLR r t) (ShuffleLR s t)
by (coinduction arbitrary: r s t) auto

\textbf{lemma} ShuffleLR_PlusR[simp]: ShuffleLR r (Plus s t) = Plus (ShuffleLR r s) (ShuffleLR r t)
by (coinduction arbitrary: r s t) auto

\textbf{lemma} Shuffle_Plus_Zero[simp]: Shuffle_Plus Zero = Zero
by coinduction simp

\textbf{lemma} Shuffle_Plus_Plus[simp]: Shuffle_Plus (Plus r s) = Plus (Shuffle_Plus r) (Shuffle_Plus s)
proof (coinduction arbitrary: r s)
\begin{align*}
\text{case } &\text{ (Lang } r s) \\
\text{ then show } &\text{ ?case unfolding Shuffle_Plus.sel Plus.sel} \\
&\text{ by (intro conjI[OF refl]} (metis Plus_comm Plus_rotate)
\end{align*}
qed

\textbf{lemma} Shuffle_Plus_ShuffleLR_One[simp]: Shuffle_Plus (ShuffleLR r One) = r
by (coinduction arbitrary: r) simp

\textbf{lemma} Shuffle_Plus_ShuffleLR_PlusL[simp]:
\begin{align*}
\text{Shuffle_Plus } (\text{ShuffleLR } (\text{Plus } r s) t) &= \text{Plus } (\text{Shuffle_Plus } (\text{ShuffleLR } r t) (\text{Shuffle_Plus } (\text{ShuffleLR } s t)) \\
\text{by (coinduction arbitrary: r s t) auto}
\end{align*}

\textbf{lemma} Shuffle_Plus_ShuffleLR_PlusR[simp]:
\begin{align*}
\text{Shuffle_Plus } (\text{ShuffleLR } r (\text{Plus } s t)) &= \text{Plus } (\text{Shuffle_Plus } (\text{ShuffleLR } r s) (\text{Shuffle_Plus } (\text{ShuffleLR } r t)) \\
\text{by (coinduction arbitrary: r s t) auto}
\end{align*}

\textbf{definition} Shuffle :: 'a language \Rightarrow 'a language \Rightarrow 'a language where
\begin{align*}
\text{Shuffle } r s &= \text{Shuffle_Plus } (\text{ShuffleLR } r s)
\end{align*}

\textbf{lemma} \circ \text{ Shuffle}[simp]:
\begin{align*}
\circ \text{ (Shuffle } r s) &= (\circ r \land \circ s) \\
\text{unfolding } &\text{Shuffle_def by simp}
\end{align*}

\textbf{lemma} \circ \text{ Shuffle}[simp]:
\begin{align*}
\circ \text{ (Shuffle } r s) &= (\lambda a. \text{ Plus } (\circ r a s) (\circ s a)))
\end{align*}
unfolding \texttt{Shuffle\_def} by (rule ext, coinduction arbitrary; \( r s \)) auto

\textbf{theorem} \texttt{Shuffle\_ZeroL[simp]}: \( \text{Shuffle} \ Zero \ r = \text{Zero} \)
\textbf{by} (coinduction arbitrary; \( r \) rule: \texttt{language\_coinduct\_upto\_Plus}) (auto 0 4)

\textbf{theorem} \texttt{Shuffle\_ZeroR[simp]}: \( \text{Shuffle} \ r \ \text{Zero} = \text{Zero} \)
\textbf{by} (coinduction arbitrary; \( r \) rule: \texttt{language\_coinduct\_upto\_Plus}) (auto 0 4)

\textbf{theorem} \texttt{Shuffle\_OneL[simp]}: \( \text{Shuffle} \ One \ r = r \)
\textbf{by} (coinduction arbitrary; \( r \) simp)

\textbf{theorem} \texttt{Shuffle\_OneR[simp]}: \( \text{Shuffle} \ r \ One = r \)
\textbf{by} (coinduction arbitrary; \( r \) simp)

\textbf{theorem} \texttt{Shuffle\_PlusL[simp]}: \( \text{Shuffle} \ (\text{Plus} \ r \ s \ t) = \text{Plus} \ (\text{Shuffle} \ r \ t) \ (\text{Shuffle} \ s \ t) \)
\textbf{by} (coinduction arbitrary; \( r \ s \ t \) rule: \texttt{language\_coinduct\_upto\_Plus})
\textbf{(force intro! \texttt{Trans} \texttt{OF} \texttt{Plus} \texttt{OF} \texttt{Base} \texttt{Base} \texttt{Refl})}

\textbf{theorem} \texttt{Shuffle\_PlusR[simp]}: \( \text{Shuffle} \ r \ (\text{Plus} \ s \ t) = \text{Plus} \ (\text{Shuffle} \ r \ s) \ (\text{Shuffle} \ r \ t) \)
\textbf{by} (coinduction arbitrary; \( r \ s \ t \) rule: \texttt{language\_coinduct\_upto\_Plus})
\textbf{(force intro! \texttt{Trans} \texttt{OF} \texttt{Plus} \texttt{OF} \texttt{Base} \texttt{Base} \texttt{Refl})}

\textbf{theorem} \texttt{Shuffle\_assoc[simp]}: \( \text{Shuffle} \ (\text{Shuffle} \ r \ s) t = \text{Shuffle} \ r \ (\text{Shuffle} \ s \ t) \)
\textbf{by} (coinduction arbitrary; \( r \ s \ t \) rule: \texttt{language\_coinduct\_upto\_Plus}) fastforce

We generalize coinduction up-to \texttt{Plus} to coinduction up-to all previously defined concepts.

\textbf{inductive} \texttt{regular\_cong where}
\texttt{Refl[intro]}: \( x = y \Rightarrow \text{regular\_cong} \ R \ x \ y \)
\texttt{Sym[intro]}: \( \text{regular\_cong} \ R \ x \ y \Rightarrow \text{regular\_cong} \ R \ y \ x \)
\texttt{Trans[intro]}: \[ \text{regular\_cong} \ R \ x \ y; \text{regular\_cong} \ R \ y \ z \] \Rightarrow \text{regular\_cong} \ R \ x \ z \)
\texttt{Base[intro]}: \( R \ x \ y \Rightarrow \text{regular\_cong} \ R \ x \ y \)
\texttt{Plus[intro]}: \[ \text{regular\_cong} \ R \ x \ y; \text{regular\_cong} \ R \ x' \ y' \] \Rightarrow \text{regular\_cong} \ R \ (\text{Plus} \ x \ x') \ (\text{Plus} \ y \ y') \)
\texttt{Times[intro]}: \[ \text{regular\_cong} \ R \ x \ y; \text{regular\_cong} \ R \ x' \ y' \] \Rightarrow \text{regular\_cong} \ R \ (\text{Times} \ x \ x') \ (\text{Times} \ y \ y') \)
\texttt{Star[intro]}: \( \text{regular\_cong} \ R \ x \ y \Rightarrow \text{regular\_cong} \ R \ (\text{Star} \ x) \ (\text{Star} \ y) \)
\texttt{Inter[intro]}: \[ \text{regular\_cong} \ R \ x \ y; \text{regular\_cong} \ R \ x' \ y' \] \Rightarrow \text{regular\_cong} \ R \ (\text{Inter} \ x \ x') \ (\text{Inter} \ y \ y') \)
\texttt{Not[intro]}: \( \text{regular\_cong} \ R \ x \ y \Rightarrow \text{regular\_cong} \ R \ (\text{Not} \ x) \ (\text{Not} \ y) \)
\texttt{Shuffle[intro]}: \[ \text{regular\_cong} \ R \ x \ y; \text{regular\_cong} \ R \ x' \ y' \] \Rightarrow \text{regular\_cong} \ R \ (\text{Shuffle} \ x \ x') \ (\text{Shuffle} \ y \ y') \)

\textbf{lemma} \texttt{language\_coinduct\_upto\_regular[unfolded rel\_fun\_def, simplified, case\_names \textbf{Lang}, consumes 1]}:
\textbf{assumes} \( R: \ R \ L \ K \ \text{and} \ \text{hyp}: \ (\forall \ L \ K. \ R \ L \ K \Rightarrow \circ \ L \ = \circ \ K \ \land \ \text{rel\_fun \ op} = \ (\text{regular\_cong} \ R) \ (\circ \ L) \ (\circ \ K)) \)
\textbf{shows} \( L = K \)
\textbf{proof} (coinduct rule: \texttt{language\_coinduct[of regular\_cong \ R]})
\textbf{fix} \( L \ K \) \textbf{assume} \( \text{regular\_cong} \ R \ L \ K \)
\textbf{then show} \( \circ \ L \ = \circ \ K \ \land \ \text{rel\_fun \ op} = \ (\text{regular\_cong} \ R) \ (\circ \ L) \ (\circ \ K) \) \textbf{using} \text{hyp}
\textbf{by} (induct rule: \texttt{regular\_cong.induct}) (auto simp: \texttt{rel\_fun\_def} \texttt{Plus} \texttt{Times} \texttt{Shuffle})
\textbf{qed} (intro \texttt{Base \ R})

\textbf{lemma} \texttt{Star\_unfoldR}: \( \text{Star} \ r = \text{Plus} \ One \ (\text{Times} \ \text{Star} \ r) \ r \)
\textbf{proof} (coinduction arbitrary; \( r \) rule: \texttt{language\_coinduct\_upto\_regular})
\textbf{case} \texttt{Lang}
\{ \textbf{fix} \( a \) \textbf{have} \( \text{Plus} \ (\text{Times} \ (\circ \ r \ a) \ (\text{Times} \ \text{Star} \ r) \ r) \) \( (\circ \ r \ a) = \)

\textbf{6}
lemma Star_star[simp]: Star (Star r) = Star r
  by (subst Star, 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 *: ∀r s. Plus (Times r s) r = Times r (Plus s One) by simp
  show ?case by (auto simp del: TimesPlusR PlusACI simp: TimesPlusR[symmetric] *)
qed

instantiation language :: (type) {semiring_1, order}
begin
lemma zero_one[simp]: Zero ≠ 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 ∧ 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 intro_classes
  fix x y z :: 'a language
  assume x ≤ y y ≤ z
  then show x ≤ 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

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 (rule order_eq_refl[OF Star_unfoldL[symmetric]])

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

theorem ardenL: Plus r (Times s x) ≤ x ⇒ Times (Star s) r ≤ x
unfolding language_defs
proof (coinduction arbitrary: r s x rule: language coinduct upto regular)
case Lang
  hence o r ⇒ o x by (metis Plus.sel(1))
moreover
  { fix a
    let ?R = (λL K. ∃r s. L = Plus (Times (Star s) r) K ∧ Plus r (Plus (Times s K) K) = K)
have regular_cong ?R (Plus x (Times (Star s) r)) x 
  using Lang[unfolded Plus_assoc by (auto simp only: Plus_comm)
hence regular_cong ?R
  (Plus (Times (d s a) (Plus x (Times (Star s) r))) (Plus (d r a) (d x a)))
  (Plus (d r a) (Plus (Times (d s a) x) (d x a)))
  by (auto simp del: Times_PlusR)
also have (Plus (Times (d s a) (Plus x (Times (Star s) r))) (Plus (d r a) (d x a)))
  (Plus (d r a) (Plus (Times (d s a) x) (d x a)))
  by (subst (4) Lang[symmetric]) auto
finally have regular_cong ?R
  (Plus (Times (d s a) (Times (Star s) r)) (Plus (d r a) (d x a)))
  (Plus (d r a) (Plus (Times (d s a) x) (d x a))).
}
ultimately show ?case by (subst (4) Lang[symmetric]) auto
qed

theorem ardenR: Plus r (Times x s) ≤ x ⇒ Times r (Star s) ≤ x
unfolding language_defs
proof (coinduction arbitrary: r s x rule: language_coinduct upto_regular)
case Lang
  let ?R = (λL K. ∃r s. L = Plus (Times r (Star s)) K ∧ Plus r (Times (Times K s) K) = K)
  have ∃a. a o x ⇒ Plus (d s a) (d x a) = d x a
    by (subst (1 2) Lang[symmetric]) auto
  then have etc:
    Lang[symmetric] (auto simp del: Plus_comm)
  qed
from Lang have a o r ⇒ a x by (metis Plus.sel(1))
moreover
  {fix a assume o x
    have regular_cong ?R (Plus (Times (d s a) (Star s)) (d x a)) (d x a)
      (rule Base excl conj[OF OF refl])
      from o x' show Plus (d s a) (Plus (Times (d x a) s) (d x a)) = d x a
        by (subst (1 3) Lang[symmetric]) auto
    qed
  from Plus[OF Base[of ?R, OF s[of a]] this] have regular_cong ?R
    (Plus (Times (d r a) (Star s)) (Plus (Times (d s a) (Star s)) (d x a))) (d x a)
    by auto
  }
ultimately show ?case by auto
qed

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[intro]: [r1 ≤ s1; r2 ≤ s2] ⇒ Plus r1 r2 ≤ Plus s1 s2
unfolding less_eq_language_def by (metis Plus_assoc Plus_comm)

lemma Plus_upper: [r1 ≤ s; r2 ≤ s] ⇒ Plus r1 r2 ≤ s
  by (metis Plus_mono Plus_idem)

lemma le_PlusL: l ≤ Plus r s
  by (metis Plus_idem_assoc less_eq_language_def)

lemma le_PlusR: s ≤ Plus r s
  by (metis Plus_comm Plus_idem_assoc less_eq_language_def)

lemma Times_mono[intro]: [r1 ≤ s1; r2 ≤ s2] ⇒ Times r1 r2 ≤ Times s1 s2
proof (unfold less_eq_language_def)
  assume s1[symmetric]: Plus r1 s1 = s1 and s2[symmetric]: Plus r2 s2 = s2
have Plus (Times r1 r2) (Times s1 s2) = 
  Plus (Times r1 r2) (Plus (Times r1 r2) (Plus (Times s1 r2) (Times s1 s2)))
by (subst s1, subst s2) auto

also have ... = Plus (Times r1 r2) (Plus (Times s1 r2) (Times s1 s2))
by (metis Plus_idem Plus_assoc)

also have ... = Times s1 s2 by (subst s1, subst s2) auto

finally show Plus (Times r1 r2) (Times s1 s2) = Times s1 s2 .

qed

lemma le_TimesL: o s \Rightarrow r \leq Times r s
by (metis Plus_OneL Times_OneR Times_mono le_PlusL order_refl)

lemma le_TimesR: o r \Rightarrow s \leq Times r s
by (metis Plus_OneR Times_OneL Times_mono le_PlusR order_refl)

lemma le_Star: s \leq Star s
by (subst Star_unfoldL, subst Star_unfoldL) (auto intro: order_trans[OF le_PlusL le_PlusR])

lemma Star_mono:
assumes rs: r \leq s
shows Star r \leq Star s
proof
  have Star r = Plus One (Times (Star r) r) by (rule Star_unfoldR)
  also have ... \leq Plus One (Times (Star r) s) by (blast intro: rs)
  also have Times (Star r) s \leq Star s
  proof (rule ardenL[OF Plus_upper[OF le_Star]])
    have Times r (Star s) \leq Times s (Star s) by (blast intro: rs)
    also have Times s (Star s) \leq Plus One (Times s (Star s)) by (rule le_PlusR)
    finally show Times r (Star s) \leq Star s by (subst 2 Star_unfoldL)
  qed
  finally show ?thesis by auto
qed

lemma Inter_mono: [r1 \leq s1; r2 \leq s2] \Rightarrow Inter r1 r2 \leq Inter s1 s2
unfolding less_eq_language_def proof (coinduction arbitrary: r1 r2 s1 s2)
case Lang
  then have o (Plus r1 s1) = o s1 o (Plus r2 s2) = o s2
  \forall a. d (Plus r1 s1) a = d s1 a \forall a. d (Plus r2 s2) a = d s2 a by simp_all
  then show ?case by fastforce
qed

lemma Not_antimono: r \leq s \Rightarrow Not s \leq Not r
unfolding less_eq_language_def proof (coinduction arbitrary: r s)
case Lang
  then have o (Plus r s) = o s \forall a. d (Plus r s) a = d s a by simp_all
  then show ?case by auto
qed

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 simp

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
    (rule ardenL[OF ardenL[OF Plus_upper[OF le_TimesL]] ardenR[OF Plus_upper[OF le_TimesR]]]
      have Times r (Times s (Star t)) = Times s (Times t (Star t)) (is ?L = ?)
        by (simp only: assms Times_assoc[symmetric])
      also have ... ≤ Times s (Star t) (is _ ≤ ?R)
        by (rule Times_mono[OF order_refl ord_le_eq_trans[OF le_PlusR Star_unfoldL[symmetric]]])
      finally show ?L ≤ ?R.
    next
      have Times (Times (Star r) s) t = Times (Times (Star r) r) s (is ?L = ?)
        by (simp only: assms Times_assoc)
      also have ... ≤ Times (Star t) s (is _ ≤ ?R)
        by (rule Times_mono[OF ord_le_eq_trans[OF le_PlusR Star_unfoldR[symmetric]] order_refl])
      finally show ?L ≤ ?R.
    qed simp_all
  )

lemma sliding: Times (Star (Times r s)) r = Times r (Star (Times s r))
  proof (rule antisym
    (rule ardenL[OF ardenL[OF Plus_upper[OF le_TimesL]] ardenR[OF Plus_upper[OF le_TimesR]]]
      have Times (Times r s) (Times r (Star (Times s r))) =
        Times r (Times (Times s r) (Star (Times s r))) (is ?L = ? R)
        by simp
      also have ... ≤ Times r (Star (Times s r)) (is _ ≤ ?R)
        by (rule Times_mono[OF order_refl ord_le_eq_trans[OF le_PlusR Star_unfoldL[symmetric]]])
      finally show ?L ≤ ?R.
    next
      have Times (Times (Star (Times r s))) r (Times s r) =
It is useful to lift binary operators $\text{Plus}$ and $\times$ to $n$-ary operators (that take a list as input).

**Definition:** $\text{PLUS} :: \text{'a language list} \Rightarrow \text{'a language where}$

$\text{PLUS } xs \equiv \text{foldr Plus } xs \text{ Zero}$

**Lemma:** $\forall \text{foldr Plus } xs \in \text{'a language list.} \forall x \in \text{'a language.}$

$\text{foldr Plus } xs = (\exists \text{set } s \not= xs. x)$

**Lemma:** $\forall \text{foldr Plus } xs \in \text{'a language list.} \forall a \in \text{'a language.}$

$\text{foldr Plus } xs = (\exists \text{set } s \not= xs. a)$

**Unfolding:** $\text{PLUS_def } \forall \text{foldr Plus by simp}$

**Definition:** $\text{TIMES} :: \text{'a language list} \Rightarrow \text{'a language where}$

$\text{TIMES } xs \equiv \text{foldr Times } xs \text{ One}$

**Lemma:** $\forall \text{foldr Times } xs \in \text{'a language list.} \forall x \in \text{'a language.}$

$\text{foldr Times } xs = (\forall \text{set } s \not= xs. x)$

**by (induct xs) (auto simp: PLUS_def)**

**primrec** $\text{tails where}$

- $\text{tails } [] = [[]]$
- $\text{tails } (x \# xs) = (x \# xs) \# \text{tails } xs$

**Lemma:** $\forall \text{tails_snoc } xs \in \text{'a language list.} \forall x \in \text{'a language.}$

$\text{tails_snoc } xs = \text{map } (\lambda y. \text{ys @ } [x]) (\text{tails } xs) \# []$

**by (induct xs) auto**

**Lemma:** $\forall \text{length_tails } xs \in \text{'a language list.}$

$\text{length } (\text{tails } xs) = \text{Suc } (\text{length } xs)$

**by (induct xs) auto**
lemma θ_foldr_Times: θ (foldr Times xs s) a = 
  (let n = length (takeWhile θ xs)
  in PLUS (map (λzs. TIMES (θ (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) = (∀x ∈ set xs. o x)
unfolding TIMES_def o_foldr_Times by simp

lemma θ_TIMES[simp]: θ (TIMES xs) a = (let n = length (takeWhile θ xs)
  in PLUS (map (λzs. TIMES (θ (hd zs) a # tl zs))) (take (Suc n) (tails (xs @ [One])))))
unfolding TIMES_def θ_foldr_Times by simp

3 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.

locale cfg =
fixes init :: 'n::enum
and prod :: 'n⇒ ('t+ 'n) list list
assumes weakGreibach: ∀N. ∀rhs ∈ set (prod N). case rhs of (Inr N #) ⇒ False | _ ⇒ True

case
fixes init :: 'n::enum
and prod :: 'n⇒ ('t+ 'n) list list
begin

private abbreviation o_n N ≡ ([] ∈ set (prod N))

private fun o where
  o [] = True
| o (Inl _ #) = False
| o (Inr N # xs) = (o_n N ∧ o xs)

private abbreviation o_P P a ≡ fold (op @) (map o_P P) False

private abbreviation θ_n N a ≡ 
  List.map_filter (λxs. case xs of Inl b # xs ⇒ if a = b then Some xs else None | _ ⇒ None) (prod N)

private fun θ where
  θ [] a = []
| θ (Inl b # xs) a = (if a = b then [xs] else [])
| θ (Inr N # xs) a = map (λys. ys @ xs) (θ_n N a) @ (if o_n N then θ_r xs a else [])

private abbreviation θ_P P a ≡ fold (op @) (map (λr. θ_r r a) P) []

primcorec subst :: ('t + 'n) list list ⇒ 't language where
  subst P = Lang (θ_P P) (λr. subst (θ_P P x))

end

abbreviation (in cfg) lang where
  lang ≡ subst prod (prod init)
4 Word-theoretic Semantics of Languages

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

```isar
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
```

```isar
primcorec to_language :: `'a list set ⇒ 'a language
where
  o (to_language L) = (∈ L)
| d (to_language L) = (λa. to_language {w. a # w ∈ L})
```

```isar
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
```

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

```isar
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_eqIff 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 ∃K. L = Collect (in_language K) by blast
qed
```

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

5 Coinductively Defined Operations Are Standard

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

```isar
lemma in_language_Zero[simp]: ¬ in_language Zero xs
by (induct xs) auto
```

```isar
lemma in_language_One[simp]: in_language One xs ⇒ xs = []
by (cases xs) auto
```

```isar
lemma in_language_Atom[simp]: in_language (Atom a) xs ⇒ xs = [a]
by (cases xs) (auto split: if_splits)
```

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

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

```isar
lemma to_language_Un[simp]: to_language (A ∪ B) = Plus (to_language A) (to_language B)
```
by (coinduction arbitrary: A B) (auto simp: Collect_disj_eq)

lemma to_language_Inf[simp]: to_language (A ∩ 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 || B) = Shuffle (to_language A) (to_language B)
  by (coinduction arbitrary: A B rule: language_coinduct_upto_Plus)
  (auto simp: Deriv_def[symmetric])

6 Word Problem for Context-Free Grammars

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

abbreviation a ≡ Inl True
abbreviation b ≡ Inl False
abbreviation S ≡ Inr ()

interpretation palindromes!: cfg () λ_. [[], [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 in_language_palindromes.lang [True, False, False] by normalization
lemma in_language_palindromes.lang [True, False, False, True] by normalization
lemma in_language_palindromes.lang [True, False, False, False] by normalization

interpretation Dyck!: cfg () λ_. [[], [a, S, 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, True, False, False] by normalization
lemma in_language_Dyck.lang [True, False, True, False] by normalization
lemma in_language_Dyck.lang [True, False, False, True] by normalization
lemma in_language_Dyck.lang [True, False, False, False] by normalization
lemma in_language_Dyck.lang [True, False, False, False, True] by normalization
lemma in_language_Dyck.lang [True, False, False, False, False] by normalization

14
interpretation abSSa!: cfg () λ_. [[], [a, b, S, S, 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, True, False, False, False, True] by normalization

end