Regular Sets, Expressions, Derivatives and Relation Algebra

Alexander Krauss, Tobias Nipkow, Chunhan Wu, Xingyuan Zhang and Christian Urban

September 13, 2023

Abstract

This is a library of constructions on regular expressions and languages. It provides the operations of concatenation, Kleene star and left-quotients of languages. A theory of derivatives and partial derivatives is provided. Arden’s lemma and finiteness of partial derivatives is established. A simple regular expression matcher based on Brozowski’s derivatives is proved to be correct. An executable equivalence checker for regular expressions is verified; it does not need automata but works directly on regular expressions. By mapping regular expressions to binary relations, an automatic and complete proof method for (in)equalities of binary relations over union, concatenation and (reflexive) transitive closure is obtained.

For an exposition of the equivalence checker for regular and relation algebraic expressions see the paper by Krauss and Nipkow [3].

Extended regular expressions with complement and intersection are also defined and an equivalence checker is provided.

Contents

1 Regular sets 2
   1.1 (@@) .......................................................... 3
   1.2 A^n ............................................................ 4
   1.3 star ............................................................ 5
   1.4 Left-Quotients of languages ................................. 7
   1.5 Shuffle product ............................................. 9
   1.6 Arden’s Lemma ............................................. 10

2 Regular expressions 12
   2.1 Term ordering .............................................. 14

3 Normalizing Derivative 16
   3.1 Normalizing operations ................................. 16
4 Deciding Regular Expression Equivalence
   4.1 Bisimulation between languages and regular expressions . . . 18
   4.2 Closure computation . . . . . . . . . . . . . . . . . . . . . . 19
   4.3 Bisimulation-free proof of closure computation . . . . . . . 20
   4.4 The overall procedure . . . . . . . . . . . . . . . . . . . . . . 21

5 Regular Expressions as Homogeneous Binary Relations 22

6 Proving Relation (In)equalities via Regular Expressions 23

7 Basic constructions on regular expressions 25
   7.1 Reverse language . . . . . . . . . . . . . . . . . . . . . . . . 25
   7.2 Substituting characters in a language . . . . . . . . . . . . . . 25
   7.3 Subword language . . . . . . . . . . . . . . . . . . . . . . . . 28
   7.4 Fragment language . . . . . . . . . . . . . . . . . . . . . . . . 29
   7.5 Various regular expression constructions . . . . . . . . . . . . 30

8 Derivatives of regular expressions 32
   8.1 Brzozowski’s derivatives of regular expressions . . . . . . . . 33
   8.2 Antimirov’s partial derivatives . . . . . . . . . . . . . . . . 33
   8.3 Relating left-quotients and partial derivatives . . . . . . . . . 35
   8.4 Relating derivatives and partial derivatives . . . . . . . . . . 35
   8.5 Finiteness property of partial derivatives . . . . . . . . . . . . 35

9 Deciding Regular Expression Equivalence (2) 40
   9.1 Closure computation . . . . . . . . . . . . . . . . . . . . . . . 42
   9.2 The overall procedure . . . . . . . . . . . . . . . . . . . . . . . 44
   9.3 Termination and Completeness . . . . . . . . . . . . . . . . . . 45

10 Extended Regular Expressions 47

11 Deciding Equivalence of Extended Regular Expressions 48
   11.1 Term ordering . . . . . . . . . . . . . . . . . . . . . . . . . . 48
   11.2 Normalizing operations . . . . . . . . . . . . . . . . . . . . . 49
   11.3 Derivative . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
   11.4 Bisimulation between languages and regular expressions . . 50
   11.5 Closure computation . . . . . . . . . . . . . . . . . . . . . . . 52
   11.6 The overall procedure . . . . . . . . . . . . . . . . . . . . . . . 53

1 Regular sets

theory Regular-Set
imports Main
begin
type-synonym 'a lang = 'a list set

definition conc :: 'a lang ⇒ 'a lang ⇒ 'a lang (infixr @@ 75) where
A @@ B = {xs@ys | xs ys. xs:A & ys:B}
    checks the code preprocessor for set comprehensions
export-code conc checking SML

overloading lang-pow == compow :: nat ⇒ 'a lang ⇒ 'a lang
begin
primrec lang-pow :: nat ⇒ 'a lang ⇒ 'a lang where
  lang-pow 0 A = {[]}
  lang-pow (Suc n) A = A @@ (lang-pow n A)
end

for code generation

definition lang-pow :: nat ⇒ 'a lang ⇒ 'a lang where
  lang-pow-code-def
    code-abbrev: lang-pow = compow

lemma [code]:
  lang-pow (Suc n) A = A @@ (lang-pow n A)
  lang-pow 0 A = {[]}
  by (simp-all add: lang-pow-code-def)

hide-const (open) lang-pow

definition star :: 'a lang ⇒ 'a lang where
  star A = (∪n. A ^^ n)

1.1 (@@)

lemma concI[simp,intro]: u : A ⇒ v : B ⇒ u@v : A @@ B
  by (auto simp add: conc-def)

lemma concE[elim]:
  assumes w ∈ A @@ B
  obtains u v where u ∈ A v ∈ B w = u@v
  using assms by (auto simp: conc-def)

lemma conc-mono: A ⊆ C ⇒ B ⊆ D ⇒ A @@ B ⊆ C @@ D
  by (auto simp: conc-def)

lemma conc-empty[simp]: shows {} @@ A = {} and A @@ {} ={}
  by auto

lemma conc-epsilon[simp]: shows {[]} @@ A = A and A @@ {[]} = A
  by (simp-all add:conc-def)

lemma conc-assoc: (A @@ B) @@ C = A @@ (B @@ C)
  by (auto elim!: concE) (simp only: append-assoc[symmetric] concI)
lemma conc-Un-distrib:
  shows $A \odot (B \cup C) = A \odot B \cup A \odot C$
and $(A \cup B) \odot C = A \odot C \cup B \odot C$
by auto

lemma conc-UNION-distrib:
  shows $A \odot \bigcup (M \cdot I) = \bigcup ((\%i. A \odot M \cdot i) \cdot I)$
and $\bigcup (M \cdot I) \odot A = \bigcup ((\%i. M \cdot i \odot A) \cdot I)$
by auto

lemma conc-subset-lists:
  $A \subseteq \text{lists } S \implies B \subseteq \text{lists } S \implies A \odot B \subseteq \text{lists } S$
by (fastforce simp: conc-def in-lists-conv-set)

lemma Nil-in-conc[simp]: $[] : A \odot B \iff [] \in A \land [] \in B$
by (metis append-is-Nil-conv concE concI)

lemma concI-if-Nil1: $[] \in A \implies xs : B \implies xs \in A \odot B$
by (metis append-Nil concI)

lemma conc-Diff-if-Nil1: $[] \in A \implies A \odot B = (A - \{[]\}) \odot B \cup B$
by (fastforce elim: concI-if-Nil1)

lemma concI-if-Nil2: $[] \in B \implies xs : A \implies xs \in A \odot B$
by (metis append-Nil2 concI)

lemma conc-Diff-if-Nil2: $[] \in B \implies A \odot B = A \odot (B - \{[]\}) \cup A$
by (fastforce elim: concI-if-Nil2)

lemma singleton-in-conc:
  $[x] : A \odot B \iff [x] : A \land [] : B \lor [] : A \land [x] : B$
by (fastforce simp: Cons-eq-append-conv append-eq-Cons-conv
             conc-Diff-if-Nil1 conc-Diff-if-Nil2)

1.2 $A^n$

lemma lang-pow-add: $A \odot (n + m) = A \odot n @ A \odot m$
by (induct n) (auto simp: conc-assoc)

lemma lang-pow-empty: $\{\} \odot n = (if n = 0 then \{\} else \{\})$
by (induct n) auto

lemma lang-pow-empty-Suc[simp]: $\{\cdot:\text{a lang}\} \odot \text{Suc } n = \{\}$
by (simp add: lang-pow-empty)

lemma conc-pow-comm:
  shows $A \odot (A \odot n) = (A \odot n) \odot A$
by (induct n) (simp-all add: conc-assoc[symmetric])
lemma length-lang-pow-ub:
\[\forall w \in A.\ length\ w \leq k \implies w : A \leq n \implies length\ w \leq k \ast n\]
by (induct n arbitrary: w) (fastforce simp: conc-def)+

lemma length-lang-pow-lb:
\[\forall w \in A.\ length\ w \geq k \implies w : A \leq n \implies length\ w \geq k \ast n\]
by (induct n arbitrary: w) (fastforce simp: conc-def)+

lemma lang-pow-subset-lists: \(A \subseteq lists\ S\ \implies A \leq n \subseteq lists\ S\)
by (induct n) (auto simp: conc-subset-lists)

lemma empty-pow-add:
assumes \(\in A\ s \in A \leq n\)
shows \(s \in A \leq (n + m)\)
using assms
apply (induct m arbitrary: n)
apply (auto simp add: concI-if-Nil1)
done

1.3 star

lemma star-subset-lists: \(A \subseteq lists\ S\ \implies star\ A \subseteq lists\ S\)
unfolding star-def by (blast dest: lang-pow-subset-lists)

lemma star-if-lang-pow[simp]: \(w : A \leq n \implies w : star\ A\)
by (auto simp: star-def)

lemma Nil-in-star[iff]: \([] : star\ A\)
proof (rule star-if-lang-pow)
  show \([] : A \leq 0\) by simp
qed

lemma star-if-lang[simp]: assumes \(w : A\) shows \(w : star\ A\)
proof (rule star-if-lang-pow)
  show \(w : A \leq 1\) using \(w : A\) by simp
qed

lemma append-in-starI[simp]:
assumes \(u : star\ A\ and \ v : star\ A\)
shows \(u \@ v : star\ A\)
proof –
  from \(u : star\ A\) obtain \(m \ where \ u : A \leq m\)
   using \(auto\ simp: star-def\)
  more over
  from \(v : star\ A\) obtain \(n \ where \ v : A \leq n\)
   using \(auto\ simp: star-def\)
  ultimately have \(w \@ v : A \leq (m + n)\)
   using \(simp\ add: lang-pow-add\)
  thus \(?thesis\)
qed

lemma conc-star-star: \(star\ A \@ star\ A = star\ A\)
by (auto simp: conc-def)
lemma conc-star-comm:
  shows A @@ star A = star A @@ A
unfolding star-def conc-pow-comm conc-UNION-distrib
  by simp

lemma star-induct[consumes 1, case-names Nil append, induct set: star]:
  assumes w : star A
  and P []
  and step: !!u v. u : A \implies v : star A \implies P v \implies P (u@v)
  shows P w
  proof –
    { fix n have w : A ^^ n \implies P w
      by (induct n arbitrary: w) (auto intro: P [] step star-if-lang-pow) }
    with [w : star A] show P w by (auto simp: star-def)
  qed

lemma star-empty[simp]: star {} = {[]} by (auto elim: star-induct)

lemma star-epsilon[simp]: star {[]} = {[]} by (auto elim: star-induct)

lemma star-idemp[simp]: star (star A) = star A by (auto elim: star-induct)

lemma star-unfold-left: star A = A @@ star A \cup {[]}
  (is ?L = ?R)
  proof
    show ?L \subseteq ?R by (rule, erule star-induct) auto
  qed auto

lemma concat-in-star: set ws \subseteq A \implies concat ws : star A
  by (induct ws) simp-all

lemma in-star-iff-concat:
  w \in star A = (\exists ws. set ws \subseteq A \land w = concat ws)
  (is - = (\exists ws. ?R w ws))
  proof
    assume w : star A thus \exists ws. ?R w ws
    proof induct
      case Nil have ?R [] [] by simp
      thus ?case ..
    next
      case (append u v)
      then obtain ws where set ws \subseteq A \land v = concat ws by blast
      with append have ?R (u@v) (u#ws) by auto
      thus ?case ..
    qed
  next
assume $\exists w. \ ?R w$ thus $w : \text{star } A$
by (auto simp: concat-in-star)
qed

lemma \text{star-conv-concat: star } A = \{ \text{concat } ws|ws. \text{ set } ws \subseteq A \}
by (fastforce simp: in-star-iff-concat)

lemma \text{star-insert-eps[simp]: star } (\text{insert } [] A) = \text{star}(A)
proof -
  \{ \text{ fix } us \\
  \quad \text{ have set } us \subseteq \text{ insert } [] A \implies \exists vs. \text{ concat } us = \text{concat } vs \land \text{ set } vs \subseteq A \\
  \quad \quad \text{(is } ?P \implies \exists vs. ?Q vs) \\
  \quad \text{ proof } \\
  \quad \quad \text{ let } ?vs = \text{ filter } (\%u. \ u \neq []) us \\
  \quad \quad \text{ show } ?P \implies ?Q ?vs \text{ by (induct us) auto} \\
  \quad \text{ qed } \\
  \} \text{ thus } ?\text{thesis by (auto simp: star-conv-concat) } \\
qed

lemma \text{star-unfold-left-Nil: star } A = (A - \{[]\}) @\@ \text{star } A \\
\cup \{[]\} 
by (metis insert-Diff-single star-insert-eps star-unfold-left)

lemma \text{star-Diff-Nil-fold: } (A - \{[]\}) @\@ \text{star } A = \text{star } A - \{[]\}
proof -
  \{ \text{ have } [] \notin (A - \{[]\}) @\@ \text{star } A \text{ by simp} \\
  \quad \text{ thus } ?\text{thesis using star-unfold-left-Nil by simp } \\
  \text{ qed } \\
  \}

lemma \text{star-decom:} \\
\text{ assumes } a: x \in \text{star } A \ x \neq [] \\
\text{ shows } \exists a \ b. \ x = a @ b \land a \neq [] \land a \in A \land b \in \text{star } A \\
\text{ using } a \text{ by (induct rule: star-induct) (blast)+ }

lemma \text{star-pow:} \\
\text{ assumes } s \in \text{star } A \\
\text{ shows } \exists n. \ s \in A ^^ n \\
\text{ using assms } \\
\text{ apply(induct) } \\
\text{ apply(rule-tac } x=0 \text{ in exI) } \\
\text{ apply(auto) } \\
\text{ apply(rule-tac } x=Suc \ n \text{ in exI) } \\
\text{ apply(auto) } \\
\text{ done }

1.4 Left-Quotients of languages

\text{definition Deriv :: 'a } \Rightarrow 'a \text{ lang } \Rightarrow 'a \text{ lang } \\
\text{ where } \text{Deriv } x \ A = \{ \text{xs. } x@xs \in A \}
**definition** Derivs :: 'a list ⇒ 'a lang ⇒ 'a lang
**where** Derivs xs A = { ys. xs @ ys ∈ A }

**abbreviation**
Derivss :: 'a list ⇒ 'a lang set ⇒ 'a lang
**where**
Derivss s As ≡ ∪ (Derivs s ' As)

**lemma** Deriv-empty [simp]:
Deriv a {} = {}
**and** Deriv-epsilon [simp]:
Deriv a {} = {}
**and** Deriv-char [simp]:
Deriv a {b} = (if a = b then {} else {})
**and** Deriv-union [simp]:
Deriv a (A ∪ B) = Deriv a A ∪ Deriv a B
**and** Deriv-inter [simp]:
Deriv a (A ∩ B) = Deriv a A ∩ Deriv a B
**and** Deriv-compl [simp]:
Deriv a (− A) = − Deriv a A
**and** Deriv-Union [simp]:
Deriv a (Union M) = Union(Deriv a ' M)
**and** Deriv-UN [simp]:
Deriv a (UN x:I. S x) = (UN x:I. Deriv a (S x))
**by** (auto simp: Deriv-def)

**lemma** Der-conc [simp]:
**shows** Deriv c (A @@ B) = (Deriv c A) @@ B ∪ (if [] ∈ A then Deriv c B else {})
**unfolding** Deriv-def conc-def
**by** (auto simp add: Cons-eq-append-conv)

**lemma** Deriv-star [simp]:
**shows** Deriv c (star A) = (Deriv c A) @@ star A
**proof** −
  **have** Deriv c (star A) = Deriv c ({{}} ∪ A @@ star A)
  **by** (metis star-unfold-left sup.commute)
  **also have** ... = Deriv c (A @@ star A)
  **unfolding** Deriv-inter **by** (simp)
  **also have** ... = (Deriv c A) @@ (star A) ∪ (if [] ∈ A then Deriv c (star A) else {})
  **by** simp
  **also have** ... = (Deriv c A) @@ star A
  **unfolding** conc-def Deriv-def
  **using** star-decom **by** (force simp add: Cons-eq-append-conv)
  **finally show** Deriv c (star A) = (Deriv c A) @@ star A .
  **qed**

**lemma** Deriv-diff [simp]:
**shows** Deriv c (A − B) = Deriv c A − Deriv c B
**by**(auto simp add: Deriv-def)

**lemma** Deriv-lists [simp]:
c : S ⇒ Deriv c (lists S) = lists S
**by**(auto simp add: Deriv-def)

**lemma** Derivs-simps [simp]:
shows Deriv [] A = A
and Deriv (c # s) A = Deriv s (Deriv c A)
and Deriv (s1 @ s2) A = Deriv s2 (Deriv s1 A)

unfolding Deriv-def Deriv-def by auto

lemma in-fold-Deriv: v ∈ fold Deriv w L ←→ w @ v ∈ L
by (induct w arbitrary: L) (simp-all add: Deriv-def)

lemma Deriv-alt-def [code]: Derivs w L = fold Deriv w L
by (induct w arbitrary: L) simp-all

lemma Deriv-code [code]:
  Deriv x A = tl ' Set. filter (λxs. case xs of x' # - ⇒ x = x' | - ⇒ False) A
by (auto simp: Deriv-def Set.filter-def image-iff tl-def split: list.splits)

1.5 Shuffle product

definition Shuffle (infixr ∥ 80) where
  Shuffle A B = ∪ {shuffles xs ys | xs ys. xs ∈ A ∧ ys ∈ B}

lemma Deriv-Shuffle[simp]:
  Deriv a (A ∥ B) = Deriv a A ∥ B ∪ A ∥ Deriv a B
unfolding Shuffle-def Deriv-def by (fastforce simp: Cons-in-shuffles-iff neq-Nil-conv)

lemma shuffle-subset-lists:
  assumes A ⊆ lists S B ⊆ lists S
  shows A ∥ B ⊆ lists S
unfolding Shuffle-def proof safe
  fix x and zs xs ys :: 'a list
  assume zs: zs ∈ shuffles xs ys x ∈ set zs and xs ∈ A ys ∈ B
  with assms have xs ∈ lists S ys ∈ lists S by auto
  with zs show x ∈ S by (induct xs ys arbitrary: zs rule: shuffles.induct) auto
qed

lemma Nil-in-Shuffle[simp]: [] ∈ A ∥ B ←→ [] ∈ A ∧ [] ∈ B
unfolding Shuffle-def by force

lemma shuffle-Un-distrib:
  shows A ∥ (B ∪ C) = A ∥ B ∪ A ∥ C
  and A ∥ (B ∪ C) = A ∥ B ∪ A ∥ C
unfolding Shuffle-def by fast+

lemma shuffle-UNION-distrib:
  shows A ∥ (M ' I) = ∪((%i. A ∥ M i) ' I)
  and (M ' I) ∥ A = ∪((%i. M i ∥ A) ' I)
unfolding Shuffle-def by fast+

lemma Shuffle-empty[simp]:
  A ∥ {} = {}
\{\} \parallel B = \{\}

unfolding Shuffle-def by auto

lemma Shuffle-eps[simp]:
\begin{align*}
\{\} \parallel \{\} &= A \\
\{\} \parallel B &= B
\end{align*}

unfolding Shuffle-def by auto

1.6 Arden’s Lemma

lemma arden-helper:
assumes eq: \[ X = A \uplus X \cup B \]
shows \[ X = (A \uplus\uplus\uplus\uplus Suc n) \uplus X \cup (\bigcup_{m \leq n} (A \uplus\uplus m) \uplus B) \]
proof (induct n)
next
\begin{itemize}
\item case (Suc n)
    \begin{itemize}
    \item have ih: \[ X = (A \uplus\uplus Suc n) \uplus X \cup (\bigcup_{m \leq n} (A \uplus\uplus m) \uplus B) \]
        by fact
    \item also have \ldots: \[ (A \uplus\uplus Suc (Suc n)) \uplus X \cup (\bigcup_{m \leq n} (A \uplus\uplus m) \uplus B) \]
        by (simp add: conc-UNION-distrib conc-assoc[symmetric] conc-pow-comm)
    \item finally show \[ X = (A \uplus\uplus Suc (Suc n)) \uplus X \cup (\bigcup_{m \leq n} (A \uplus\uplus m) \uplus B) \]
    qed
\end{itemize}
\end{itemize}
lemma Arden:
assumes \[ \{\} \not\in A \]
shows \[ X = A \uplus X \cup B \iff X = \text{star } A \uplus B \]
proof
assume eq: \[ X = A \uplus X \cup B \]
{ fix \( w \) assume \( w : X \) }
let \?n = size \( w \)
from \[ \{\} \not\in A \]
have \[ \forall u \in A. \text{ length } u \geq 1 \]
by (metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq)

hence \[ \forall u \in A \uplus (\uplus n+1), \text{ length } u \geq ?n+1 \]
by (metis length-lang-pow-lb nat-mult-1)

hence \[ \forall u \in A \uplus (\uplus n+1) \uplus X. \text{ length } u \geq ?n+1 \]
by (auto simp only: conc-def length-append)

hence \[ \not\in A \uplus (\uplus n+1) \uplus X \]
by auto

hence \[ w : \text{star } A \uplus B \]
using \( \{w : X\} \)
using arden-helper[OF eq, where \( n=?n \)]
by (auto simp add: star-def conc-UNION-distrib)
}
}
{ fix w assume w : star A @@ B 
  hence ∃ n. w ∈ A ^ n @@ B by(auto simp: conc-def star-def) 
  hence w : X using arden-helper[OF eq] by blast 
 } ultimately show X = star A @@ B by blast

next 
assume eq: X = star A @@ B 
have star A = A @@ star A ∪ {[]} 
  by (rule star-unfold-left) 
then have star A @@ B = (A @@ star A ∪ {[]}) @@ B 
  by metis 
also have ... = (A @@ star A) @@ B ∪ B 
  unfolding conc-Un-distrib by simp 
also have ... = A @@ (star A @@ B) ∪ B 
  by (simp only: conc-assoc) 
finally show X = A @@ X ∪ B 
  using eq by blast 
qed

lemma reversed-arden-helper:
  assumes eq: X = X @@ A ∪ B 
  shows X = X @@ (A ^ Suc n) ∪ (∪ m ≤ n. B @@ (A ^ m)) 
proof (induct n) 
  case 0 
  show X = X @@ (A ^ Suc 0) ∪ (∪ m ≤ 0. B @@ (A ^ m)) 
    using eq by simp 
  next 
  case (Suc n) 
  have ih: X = X @@ (A ^ Suc n) ∪ (∪ m ≤ n. B @@ (A ^ m)) by fact 
  also have ... = X @@ (A ^ Suc (Suc n)) ∪ (B @@ (A ^ Suc n)) ∪ (∪ m ≤ n. 
    B @@ (A ^ m)) 
    by (simp add: conc-Un-distrib conc-assoc) 
  also have ... = X @@ (A ^ Suc (Suc n)) ∪ (B @@ (A ^ Suc n) ∪ (∪ m ≤ Suc n. B @@ (A ^ m)) 
    by (auto simp add: atMostSuc) 
  finally show X = X @@ (A ^ Suc (Suc n)) ∪ (∪ m ≤ Suc n. B @@ (A ^ m)) 
  qed

theorem reversed-Arden:
  assumes nemp: [] ∉ A 
  shows X = X @@ A ∪ B ↔ X = B @@ star A 
proof 
  assume eq: X = X @@ A ∪ B 
  { fix w assume w : X 
    let ?n = size w 
    from [] ∉ A; have ∀ u ∈ A. length u ≥ 1 
      by (metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq) 
  qed
hence \( \forall u \in A \sim \( ?n+1 \), length u \geq ?n+1 \)
by (metis length-lang-pow-lb nat-mult-1)
hence \( \forall u \in X \preceq A \sim \( ?n+1 \), length u \geq ?n+1 \)
by (auto simp only: conc-def length-append)
hence \( w \notin X \preceq A \sim \( ?n+1 \) \) by auto
hence \( w : B \preceq star A \) using \( w : X \) using reversed-arden-helper[OF \( eq \), where \( n=\?n \)]
by (auto simp add: star-def conc-UNION-distrib)

moreover
\{
fix \( w \) assume \( w : B \preceq star A \)
hence \( \exists n. w \in B \preceq A \sim \( n \) \) by (auto simp: conc-def star-def)
hence \( w : X \) using reversed-arden-helper[OF \( eq \)] by blast
\}

ultimately show \( X = B \preceq star A \) by blast

next
assume \( eq: X = B \preceq star A \)
have \( star A = \{[]\} \cup star A \preceq A \)
  unfolding conc-star-comm[symmetric]
  by (metis Un-commute star-unfold-left)
then have \( B \preceq star A = B \preceq (\{[]\} \cup star A \preceq A) \)
  by metis
also have \( \ldots = B \cup B \preceq (star A \preceq A) \)
  unfolding conc-Un-distrib by simp
also have \( \ldots = B \cup (B \preceq star A) \preceq A \)
  by (simp only: conc-assoc)
finally show \( X = X \preceq A \cup B \)
  using \( eq \) by blast
qed

end

2 Regular expressions

theory Regular-Exp
imports Regular-Set
begin

datatype \( (\text{atoms}: \text{'a}) \) \text{rexp} =
  \text{is-Zero} Zero |
  \text{is-One} One |
  \text{Atom} 'a |
  \text{Plus} ('a \text{ rexp}) ('a \text{ rexp}) |
  \text{Times} ('a \text{ rexp}) ('a \text{ rexp}) |
  \text{Star} ('a \text{ rexp})

primrec \text{lang} :: '\a \text{ rexp} => '\a \text{ lang} where
  \text{lang} \text{ Zero} = \{\} |
  \text{lang} \text{ One} = \{
\} |
  \text{lang} \text{ (Atom \ a) = \{[a]\} |
  \text{lang} \text{ (Plus \ r \ s) = (lang \ r) Un (lang \ s) |
\[ \text{lang} \left( \text{Times} \ r \ s \right) = \text{conc} \left( \text{lang} \ r \right) \left( \text{lang} \ s \right) | \]
\[ \text{lang} \left( \text{Star} \ r \right) = \text{star} \left( \text{lang} \ r \right) \]

**abbreviation** (input) regular-lang where regular-lang \( A \equiv (\exists r, \text{lang} \ r = A) \)

**primrec** nullable :: 'a rexp ⇒ bool where
nullable Zero = False | nullable One = True | nullable (Atom c) = False | nullable (Plus r1 r2) = (nullable r1 ∨ nullable r2) | nullable (Times r1 r2) = (nullable r1 ∧ nullable r2) | nullable (Star r) = True

**lemma** nullable-iff [code-abbrev]: nullable \( r \) ←→ \( \square \in \text{lang} \ r \)
by (induct \( r \)) (auto simp add: conc-def split: if-splits)

**primrec** rexp-empty where
rexp-empty Zero ←→ True | rexp-empty One ←→ False | rexp-empty (Atom a) ←→ False | rexp-empty (Plus r s) ←→ rexp-empty \( r \) ∧ rexp-empty \( s \) | rexp-empty (Times r s) ←→ rexp-empty \( r \) ∨ rexp-empty \( s \) | rexp-empty (Star r) ←→ False

**lemma** rexp-empty-iff [code-abbrev]: rexp-empty \( r \) ←→ \( \text{lang} \ r = \{ \} \)
by (induction \( r \)) auto

Composition on rhs usually complicates matters:

**lemma** map-map-rexp:
map-rexp \( f \) (map-rexp \( g \) \( r \)) = map-rexp (λ\( r \). \( f \) (\( g \) \( r \))) \( r \)
**unfolding** rexp.map-comp o-def ..

**lemma** map-rexp-ident[zip]: map-rexp (λ\( x \). \( x \)) = (λ\( r \). \( r \))
**unfolding** id-def[symmetric] fun-eq-iff rexp.map-id id-apply by (intro allI refl)

**lemma** atoms-lang: \( w : \text{lang} \ r \implies \text{set} \ w \subseteq \text{atoms} \ r \)
**proof** (induction \( r \) arbitrary: \( w \))
- **case** Times thus ?case by fastforce
**next**
- **case** Star thus ?case by (fastforce simp add: star-conv-concat)
**qed** auto

**lemma** lang-eq-ext: (lang \( r \) = lang \( s \)) = 
(∀ \( w \in \text{lists} (\text{atoms} \ r \cup \text{atoms} \ s)\), \( w \in \text{lang} \ r \) ←→ \( w \in \text{lang} \ s \))
by (auto simp: atoms-lang[unfolded subset-iff])

**lemma** lang-eq-ext-Nil-fold-Deriv:
fixes \( r \ s \)
defines $B \equiv \{(\text{fold Deriv w (lang r), fold Deriv w (lang s)}): \ w. \ w \in \text{lists (atoms r } \cup \text{ atoms s)}\}$

shows lang r = lang s $\iff$ $(\forall (K, L) \in B. \ K \subseteq K \iff \emptyset \subseteq L)$

unfolding lang-eq-ext $B$-def by (subst (1 2) in-fold-Deriv[if \emptyset, simplified, symmetric]) auto

2.1 Term ordering

instantiation $\text{rexp :: (order) \{order\}}$

begin

fun $\text{le-rexp :: (\text{\textquotesingle a::order}) \text{rexp \Rightarrow (\text{\textquotesingle a::order}) \text{rexp \Rightarrow bool}}}$

where

$\text{le-rexp Zero - = True}$

$\mid \text{le-rexp - Zero = False}$

$\mid \text{le-rexp One - = True}$

$\mid \text{le-rexp - One = False}$

$\mid \text{le-rexp (Atom a) (Atom b) = (a <= b)}$

$\mid \text{le-rexp (Atom -) - = True}$

$\mid \text{le-rexp - (Atom -) = False}$

$\mid \text{le-rexp (Star r) (Star s) = le-rexp r s}$

$\mid \text{le-rexp (Star -) - = True}$

$\mid \text{le-rexp - (Star -) = False}$

$\mid \text{le-rexp (Plus r r') (Plus s s') =}$

$(\text{if } r = s \text{ then le-rexp r' s' else le-rexp r s})$

$\mid \text{le-rexp (Plus - -) - = True}$

$\mid \text{le-rexp - (Plus - -) = False}$

$\mid \text{le-rexp (Times r r') (Times s s') =}$

$(\text{if } r = s \text{ then le-rexp r' s' else le-rexp r s})$

definition $\text{less-eq-rexp}$ where $r \leq s \equiv \text{le-rexp r s}$

definition $\text{less-rexp}$ where $r < s \equiv \text{le-rexp r s \land r \neq s}$

lemma $\text{le-rexp-Zero: le-rexp r Zero \Rightarrow r = Zero}$

by (induction r) auto

lemma $\text{le-rexp-refl: le-rexp r r}$

by (induction r) auto

lemma $\text{le-rexp-antisym: [le-rexp r s; le-rexp s r] \Rightarrow r = s}$

by (induction r s rule: le-rexp.induct) (auto dest: le-rexp-Zero)

lemma $\text{le-rexp-trans: [le-rexp r s; le-rexp s t] \Rightarrow le-rexp r t}$

proof (induction r s arbitrary; t rule: le-rexp.induct)

fix v t assume le-rexp (Atom v) t thus le-rexp One t by (cases t) auto

next

fix s1 s2 t assume le-rexp (Plus s1 s2) t thus le-rexp One t by (cases t) auto
next  fix  s1 s2 t  assume  le-rexp (Times s1 s2) t  thus  le-rexp One t  by  (cases t)  auto
next  fix  s t  assume  le-rexp (Star s) t  thus  le-rexp One t  by  (cases t)  auto
next  fix  v u t  assume  le-rexp (Atom v) (Atom u)  le-rexp (Atom u) t  
  thus  le-rexp (Atom v) t  by  (cases t)  auto
next  fix  v s1 s2 t  assume  le-rexp (Plus s1 s2) t  thus  le-rexp (Atom v) t  by  (cases t)  auto
next  fix  v s1 s2 t  assume  le-rexp (Times s1 s2) t  thus  le-rexp (Atom v) t  by  (cases t)  auto
next  fix  v s1 s2 t  assume  le-rexp (Star s) t  thus  le-rexp (Atom v) t  by  (cases t)  auto
next  fix  v s t  assume  le-rexp (Star s) t  thus  le-rexp (Atom v) t  by  (cases t)  auto
next  fix  v s t  assume  le-rexp (Star s) t  thus  le-rexp (Atom v) t  by  (cases t)  auto
next  fix  r s t  assume  IH: \( \forall t. \text{le-rexp } r s \implies \text{le-rexp } s t \implies \text{le-rexp } r t \)
  and  le-rexp (Star r) (Star s)  le-rexp (Star s) t  
  thus  le-rexp (Star r) t  by  (cases t)  auto
next  fix  r s1 s2 t  assume  le-rexp (Plus s1 s2) t  thus  le-rexp (Star r) t  by  (cases t)  auto
next  fix  r s1 s2 t  assume  le-rexp (Times s1 s2) t  thus  le-rexp (Star r) t  by  (cases t)  auto
next  fix  r1 r2 s1 s2 t  
  assume  \( \forall t. r1 = s1 \implies \text{le-rexp } r2 s2 \implies \text{le-rexp } s2 t \implies \text{le-rexp } r2 t \)
  \( \forall t. r1 \neq s1 \implies \text{le-rexp } r1 s1 \implies \text{le-rexp } s1 t \implies \text{le-rexp } r1 t \)
  le-rexp (Plus r1 r2) (Plus s1 s2)  le-rexp (Plus s1 s2) t  
  thus  le-rexp (Plus r1 r2) t  by  (cases t)  (auto split: if-split-asm intro: le-rexp-antisym)
next  fix  r1 r2 s1 s2 t  assume  le-rexp (Times s1 s2) t  thus  le-rexp (Plus r1 r2) t  by  (cases t)  auto
next  fix  r1 r2 s1 s2 t  
  assume  \( \forall t. r1 = s1 \implies \text{le-rexp } r2 s2 \implies \text{le-rexp } s2 t \implies \text{le-rexp } r2 t \)
  \( \forall t. r1 \neq s1 \implies \text{le-rexp } r1 s1 \implies \text{le-rexp } s1 t \implies \text{le-rexp } r1 t \)
  le-rexp (Times r1 r2) (Times s1 s2)  le-rexp (Times s1 s2) t  
  thus  le-rexp (Times r1 r2) t  by  (cases t)  (auto split: if-split-asm intro: le-rexp-antisym)
qed  auto

instance proof

qed  (auto simp add: less-eq-rexp-def less-rexp-def
  intro: le-rexp-refl le-rexp-antisym le-rexp-trans)

end
instantiation rexp :: (linorder) {linorder}
begin

lemma le-rexp-total: le-rexp (r :: 'a :: linorder rexp) s ∨ le-rexp s r
by (induction r s rule: le-rexp.induct) auto

instance proof
qed (unfold less-eq-rexp-def less-rexp-def, rule le-rexp-total)
end

end

3 Normalizing Derivative

theory NDerivative
imports
    Regular-Exp
begin

3.1 Normalizing operations

associativity, commutativity, idempotence, zero

fun nPlus :: 'a::order rexp ⇒ 'a rexp ⇒ 'a rexp
where
    nPlus Zero r = r
| nPlus r Zero = r
| nPlus (Plus r s) t = nPlus r (nPlus s t)
| nPlus r (Plus s t) =
    (if r = s then (Plus s t)
    else if le-rexp r s then Plus r (Plus s t)
    else Plus s (nPlus r t))
| nPlus r s =
    (if r = s then r
    else if le-rexp r s then Plus r s
    else Plus s r)

lemma lang-nPlus[simp]: lang (nPlus r s) = lang (Plus r s)
by (induction r s rule: nPlus.induct) auto
    associativity, zero, one

fun nTimes :: 'a::order rexp ⇒ 'a rexp ⇒ 'a rexp
where
    nTimes Zero - = Zero
| nTimes - Zero = Zero
| nTimes One r = r
| nTimes r One = r
| nTimes (Times r s) t = Times r (nTimes s t)
lemma lang-nTimes[simp]: \( \text{lang} (\text{nTimes } r s) = \text{lang} (\text{Times } r s) \)
by (induction \( r \) \( s \) rule: nTimes.induct) (auto simp: conc-assoc)

primrec norm :: \('a::order rexp \Rightarrow 'a rexp\)
where
  norm \( \text{Zero} \) = \( \text{Zero} \)
| norm \( \text{One} \) = \( \text{One} \)
| norm \((\text{Atom } a)\) = \( \text{Atom } a \)
| norm \((\text{Plus } r s)\) = \( \text{nPlus} (\text{norm } r) (\text{norm } s) \)
| norm \((\text{Times } r s)\) = \( \text{nTimes} (\text{norm } r) (\text{norm } s) \)
| norm \((\text{Star } r)\) = \( \text{Star} (\text{norm } r) \)

lemma lang-norm[simp]: \( \text{lang} (\text{norm } r) = \text{lang } r \)
by (induct \( r \)) auto

primrec nderiv :: \('a::order \Rightarrow 'a rexp \Rightarrow 'a rexp\)
where
  nderiv - \( \text{Zero} \) = \( \text{Zero} \)
| nderiv - \( \text{One} \) = \( \text{Zero} \)
| nderiv \( a \) \((\text{Atom } b)\) = (if \( a = b \) then \( \text{One} \) else \( \text{Zero} \))
| nderiv \( a \) \((\text{Plus } r s)\) = \( \text{nPlus} (\text{nderiv } a \text{ } r) (\text{nderiv } a \text{ } s) \)
| nderiv \( a \) \((\text{Times } r s)\) =
  (let \( r's = \text{nTimes} (\text{nderiv } a \text{ } r) (\text{nderiv } a \text{ } s) \) in if \( \text{nullable } r \) then \( \text{nPlus } r's (\text{nderiv } a \text{ } s) \) else \( r's \))
| nderiv \( a \) \((\text{Star } r)\) = \( \text{nTimes} (\text{nderiv } a \text{ } r) (\text{Star } r) \)

lemma lang-nderiv: \( \text{lang} (\text{nderiv } a \text{ } r) = \text{Deriv } a (\text{lang } r) \)
by (induction \( r \)) (auto simp: Let-def nullable-iff)

lemma deriv-no-occurrence:
\( x \notin \text{atoms } r \implies \text{nderiv } x \text{ } r = \text{Zero} \)
by (induction \( r \)) auto

lemma atoms-nPlus[simp]: \( \text{atoms} (\text{nPlus } r s) = \text{atoms } r \cup \text{atoms } s \)
by (induction \( r \) \( s \) rule: nPlus.induct) auto

lemma atoms-nTimes: \( \text{atoms} (\text{nTimes } r s) \subseteq \text{atoms } r \cup \text{atoms } s \)
by (induction \( r \) \( s \) rule: nTimes.induct) auto

lemma atoms-norm: \( \text{atoms} (\text{norm } r) \subseteq \text{atoms } r \)
by (induction \( r \)) (auto dest!:subsetD[OF atoms-nTimes])

lemma atoms-nderiv: \( \text{atoms} (\text{nderiv } a \text{ } r) \subseteq \text{atoms } r \)
by (induction \( r \)) (auto simp: Let-def dest!:subsetD[OF atoms-nTimes])

end
4 Deciding Regular Expression Equivalence

theory Equivalence-Checking
imports
  NDerivative
  HOL-Library, While-Combinator
begin

4.1 Bisimulation between languages and regular expressions

coinductive bisimilar :: 'a lang ⇒ 'a lang ⇒ bool where
([] ∈ K 「 [] ∈ L)
⇒ (∀x. bisimilar (Deriv x K) (Deriv x L))
⇒ bisimilar K L

lemma equal-if-bisimilar:
assumes bisimilar K L
shows K = L
proof (rule set-eqI)
  fix w
  from ‹bisimilar K L› show w ∈ K 「 w ∈ L
  proof (induct w arbitrary: K L)
    case Nil thus ?case by (auto elim: bisimilar_cases)
  next
    case (Cons a w K L)
    from ‹bisimilar K L› have bisimilar (Deriv a K) (Deriv a L)
    by (auto elim: bisimilar_cases)
    then have w ∈ Deriv a K 「 w ∈ Deriv a L by (rule Cons(1))
    thus ?case by (auto simp: Deriv-def)
  qed
qed

lemma language-coinduct:
fixes R (infixl ∼ 50)
assumes K ∼ L
assumes ∫ K L. K ∼ L ⇒ ([] ∈ K 「 [] ∈ L)
assumes ∫ K L x. K ∼ L ⇒ Deriv x K ∼ Deriv x L
shows K = L
apply (rule equal-if-bisimilar)
apply (rule bisimilar.coinduct[of R, OF ‹K ∼ L›])
apply (auto simp: assms)
done

type-synonym 'a rexp-pair = 'a rexp * 'a rexp

definition is-bisimulation :: 'a::order list ⇒ 'a rexp-pair set ⇒ bool
where
is-bisimulation as R =
(∀(r,s)∈ R. (atoms r ∪ atoms s ⊆ set as) ∧ (nullable r 「 nullable s) ∧
(∀a∈set as. (nderiv a r, nderiv a s) ∈ R))
lemma bisim-lang-eq:
assumes bisim: is-bisimulation as ps
assumes (r, s) ∈ ps
shows lang r = lang s
proof
  define ps' where ps' = insert (Zero, Zero) ps
from bisim have bisim': is-bisimulation as ps'
    by (auto simp: ps'-def is-bisimulation-def)
let ?R = λK L. (∃(r,s)∈ps'. K = lang r ∧ L = lang s)
show ?thesis
proof (rule language-coinduct [where R=?R])
  from (r, s) ∈ ps
  have (r, s) ∈ ps' by (auto simp: ps'-def)
  thus ?thesis
next
  fix K L assume ?R K L
  then obtain r s where rs: (r, s) ∈ ps'
    and KL: K = lang r L = lang s by auto
  with bisim' have nullable r ⟷ nullable s
    by (auto simp: is-bisimulation-def)
  thus [] ∈ K ⟷ [] ∈ L by (auto simp: nullable-iff KL)
fix a
show ?R (Deriv a K) (Deriv a L)
proof cases
  assume a ∈ set as
  with rs bisim'
  have (nderiv a r, nderiv a s) ∈ ps'
    by (auto simp: is-bisimulation-def)
  thus ?thesis by (force simp: KL lang-nderiv)
next
  assume a /∈ set as
  with bisim' rs
  have a /∈ atoms r a /∈ atoms s by (auto simp: is-bisimulation-def)
  then have nderiv a r = Zero nderiv a s = Zero
    by (auto intro: deriv-no-occurrence)
  then have Deriv a K = lang Zero
    Deriv a L = lang Zero
  unfolding KL lang-nderiv[symmetric] by auto
  thus ?thesis by (auto simp: ps'-def)
qed
qed

4.2 Closure computation

definition closure ::
  'a:order list ⇒ 'a rexp-pair ⇒ ('a rexp-pairs * 'a rexp-pair set) option
where
closure as = rtrancl-while (%(r,s). nullable r = nullable s)
%(r,s). map (λa. (nderiv a r, nderiv a s)) as)

definition pre-bisim :: 'a::order list ⇒ 'a rexp ⇒
'a rexp-pairs * 'a rexp-pair set ⇒ bool
where
pre-bisim as r s = (λ(ws,R).
  (r,s) ∈ R ∧ set ws ⊆ R ∧
  (∀(r,s)∈ R. atoms r ∪ atoms s ⊆ set as) ∧
  (∀(r,s)∈ R − set ws. (nullable r ↔ nullable s)) ∧
  (∀a∈set as. (nderiv a r, nderiv a s) ∈ R)))

definition Ball-def
theorem closure-sound:
assumes result: closure as (r,s) = Some([],R)
and atoms: atoms r ∪ atoms s ⊆ set as
shows lang r = lang s
proof –
  let ?test = While-Combinator.rtrancl-while-test (%(r,s). nullable r = nullable s)
  let ?step = While-Combinator.rtrancl-while-step (%(r,s). map (λa. (nderiv a r, nderiv a s)) as)
  { fix st assume inv: pre-bisim as r s st and test: ?test st
    have pre-bisim as r s (?step st)
    proof (cases st)
      fix ws R assume st = (ws, R)
      with test obtain r s t where st: st = ((r, s) ≠ t, R) and nullable r = nullable s
      by (cases ws) auto
      unfolding st rtrancl-while-test.simps rtrancl-while-step.simps pre-bisim-def
      Ball-def
      by simp-all blast+
    qed
  }
moreover
  from atoms
  have pre-bisim as r s ([[(r,s)],[{(r,s)}]]) by (simp add: pre-bisim-def)
ultimately have pre-bisim-ps: pre-bisim as r s ([],R)
  by (rule while-option-rule[OF - result[unfolded closure-def rtrancl-while-def]])
then have is-bisimulation as R (r, s) ∈ R
  by (auto simp: pre-bisim-def is-bisimulation-def)
thus lang r = lang s by (rule bisim-lang-eq)
qed

4.3 Bisimulation-free proof of closure computation

The equivalence check can be viewed as the product construction of two automata. The state space is the reflexive transitive closure of the pair of next-state functions, i.e. derivatives.

lemma rtrancl-nderiv-nderivs: defines nderivs == foldl (%r a. nderiv a r)
shows \{ ((r, s), (\text{nderiv a r, nderv a s})) | r s a : A \} \Rightarrow \{ ((r, s), (\text{nderiv r w, nderv s w})) | r s w : \text{lists A} \} (\text{is } ?L = ?R)

proof –

note [simp] = nderv-def

\{ fix r s r' s' 
  have ((r, s), (r', s')) : ?L \Rightarrow ((r, s), (r', s')) : ?R 
  proof(induction rule: converse-rtrancl-induct2) 
    case refl show ?case by (force intro: foldl.simps(1)[symmetric]) 
  next 
    case step thus ?case by (force intro: foldl.simps(2)[symmetric]) 
  qed 
\} moreover 

\{ fix r s r' s' 
  \{ fix w have \forall x \in \text{set w}. x \in A \Rightarrow ((r, s), \text{nderiv r w, nderv s w}) : ?L 
    proof(induction w rule: rev-induct) 
      case Nil show ?case by simp 
    next 
      case snoc thus ?case by (auto elim: rtrancl-into-rtrancl) 
    qed 
  \} 

hence ((r, s), (r', s')) : ?R \Rightarrow ((r, s), (r', s')) : ?L by auto 
\}
next ultimately show ?thesis by (auto simp add: in-lists-conv-set) blast 
qed 

lemma nullable-nderivs:

nullable (foldl (%r a. \text{nderiv a r}) r w) = (w : lang r) 
by (induct w arbitrary: r) (simp-all add: nullable-iff lang-nderiv Deriv-def)

theorem closure-sound-complete:

assumes result: closure as (r, s) = Some(ws, R) 
and atoms: set as = atms r \cup atms s 
shows ws = [] \iff lang r = lang s 
proof –

have leq: (lang r = lang s) = 
  (\forall (r', s') \in \{ ((r0, s0), (\text{nderiv a r0, nderv a s0})) | r0 s0 a : set as \} \Rightarrow \{ (r, s) \}. 
  nullable r' = nullable s') 
  by(simp add: atms rtrancl-nderiv-nderivs Ball-def lang-eq-ext imp-ex nullable-nderivs del: Un-iff) 

have \{ (x, y). y \in \text{set} ((\lambda (p, q). \text{map} (\lambda a. (\text{nderiv a p, nderv a q})) as) x) \} = 
  \{ ((r, s), \text{nderiv a r, nderv a s}) | r s a. a \in set as \} 
  by auto 

with atms rtrancl-while-Some[OF result[unfolded closure-def]] 
show ?thesis by (auto simp add: leq Ball-def split: if-splits) 
qed 

4.4 The overall procedure

primrec add-atoms :: 'a rexp \Rightarrow 'a list \Rightarrow 'a list
where
  add-atoms Zero = id
| add-atoms One = id
| add-atoms (Atom a) = List.insert a
| add-atoms (Plus r s) = add-atoms s o add-atoms r
| add-atoms (Times r s) = add-atoms s o add-atoms r
| add-atoms (Star r) = add-atoms r

lemma set-add-atoms: set (add-atoms r as) = atoms r \cup set as
by (induct r arbitrary: as) auto

definition check-eqv :: nat rexp \Rightarrow nat rexp \Rightarrow bool where
check-eqv r s =
  (let nr = norm r; ns = norm s; as = add-atoms nr (add-atoms ns [])
in case closure as (nr, ns) of
  Some([],-) \Rightarrow True | \_ \Rightarrow False)

lemma soundness:
assumes check-eqv r s shows lang r = lang s
proof −
  let \?nr = norm r let \?ns = norm s
  let \?as = add-atoms \?nr (add-atoms \?ns [])
  obtain R where 1: closure \?as (\?nr,\?ns) = Some([],R)
    using assms by (auto simp: check-eqv-def Let-def split:option.splits list.splits)
  then have lang (norm r) = lang (norm s)
    by (rule closure-sound) (auto simp: set-add-atoms dest!: subsetD[OF atoms-norm])
  thus lang r = lang s by simp
qed

Test:
lemma check-eqv (Plus One (Times (Atom 0) (Star (Atom 0)))) (Star (Atom 0))
by eval

end

5 Regular Expressions as Homogeneous Binary Relations

theory Relation-Interpretation
imports Regular-Exp
begin

primrec rel :: ('a \Rightarrow ('b * 'b) set) \Rightarrow 'a rexp \Rightarrow ('b * 'b) set
where
  rel v Zero = \{\} |
| rel v One = Id |
| rel v (Atom a) = v a |

\[22\]
\[ \text{rel } v \ (\text{Plus } r \ s) = \text{rel } v \ r \cup \text{rel } v \ s \]
\[ \text{rel } v \ (\text{Times } r \ s) = \text{rel } v \ r \ O \text{rel } v \ s \]
\[ \text{rel } v \ (\text{Star } r) = (\text{rel } v \ r)^* \]

primrec word-rel :: ('a ⇒ ('b * 'b) set) ⇒ 'a list ⇒ ('b * 'b) set

where

\[ \text{word-rel } v \ [] = \text{Id} \]
\[ \text{word-rel } v \ (a\#as) = v\ a \ O \text{word-rel } v \ as \]

lemma word-rel-append:

\[ \text{word-rel } v \ w \ O \text{word-rel } v \ w' = \text{word-rel } v \ (w \ @ \ w') \]

by (rule sym) (induct w, auto)

lemma rel-word-rel: \(\text{rel } v \ r = (\bigcup w \in \text{lang } r. \text{word-rel } v \ w)\)

proof (induct r)

\[ \text{case Times thus } ?\text{case} \]

by (auto simp: rel-def word-rel-append conc-def relcomp-UNION-distrib relcomp-UNION-distrib2)

next

\[ \text{case (Star } r) \]

\{ \text{fix } n \]

\[ \text{have } (\text{rel } v \ r)^\sim n = (\bigcup w \in \text{lang } r^\sim n. \text{word-rel } v \ w) \]

proof (induct n)

\[ \text{case 0 show } ?\text{case by simp} \]

next

\[ \text{case (Suc } n) \text{ thus } ?\text{case} \]

unfolding relpow.simps relpow-commute[symmetric]

by (auto simp add: Star conc-def word-rel-append relcomp-UNION-distrib relcomp-UNION-distrib2)

qed \}

thus ?case unfolding rel.simps

by (force simp: rtrancl-power star-def)

qed auto

Soundness:

lemma soundness:

\(\text{lang } r = \text{lang } s \implies \text{rel } v \ r = \text{rel } v \ s\)

unfolding rel-word-rel by auto

end

6 Proving Relation (In)equalities via Regular Expressions

theory Regexp-Method

imports Equivalence-Checking Relation-Interpretation

begin
primrec rel-of-regexp :: ('a * 'a) set list ⇒ nat rexp ⇒ ('a * 'a) set where
rel-of-regexp vs Zero = {} |
rel-of-regexp vs One = Id |
rel-of-regexp vs (Atom i) = vs i |
rel-of-regexp vs (Plus r s) = rel-of-regexp vs r \cup rel-of-regexp vs s |
rel-of-regexp vs (Times r s) = rel-of-regexp vs r O rel-of-regexp vs s |
rel-of-regexp vs (Star r) = (rel-of-regexp vs r) ^

lemma rel-of-regexp-rel: rel-of-regexp vs r = rel (λi. vs ! i) r
by (induct r) auto

primrec rel-eq where
rel-eq (r, s) vs = (rel-of-regexp vs r = rel-of-regexp vs s)

lemma rel-eqI: check-eqv r s ⇒ rel-eq (r, s) vs
unfolding rel-eq.simps rel-of-regexp-rel
by (rule Relation-Interpretation.soundness)
(rule Equivalence-Checking.soundness)

lemmas regexp-reify = rel-of-regexp.simps rel-eq.simps
lemmas regexp-unfold = trancl-unfold-left subset-Un-eq

ML «
local
fun check-eqv (ct, b) = Thm.mk-binop @{cterm Pure.eq :: bool ⇒ bool ⇒ prop} ct (if b then @{cterm True} else @{cterm False});

val (_, check-eqv-oracle) = Context.>>> (Context.map-theory-result
(Thm.add-oracle (@{binding check-eqv}, check-eqv)));

in

val regexp-conv =
@{computation-conv bool terms: check-eqv datatypes: nat rexp}
(fn - => fn b => fn ct => check-eqv-oracle (ct, b))

end »

method-setup regexp = «
Scan.succeed (fn ctxt =>
SIMPLE-METHOD' (TRY o cresolve-tac ctxt @{thms regexp-unfold})
THEN' (Subgoal.FOCUS-PARAMS (fn {context = ctxt', ...} =>
TRY (Local-Defs.unfold-tac ctxt' @{thms regexp-unfold})
THEN Reification.tac ctxt' @{thms regexp-reify} NONE 1
THEN resolve-tac ctxt' @{thms rel-eqI}) 1

24
THEN CONVERSION (HOLogic.Trueprop-conv (regexp-conv ctxt')) THEN resolve-tac ctxt' [TrueI 1] ctxt))
> \{decide relation equalities via regular expressions\}

hide-const (open) le-rexp nPlus nTimes norm nullable bisimilar is-bisimulation closure
pre-bisim add-atoms check-eqv rel word-rel rel-eq

Example:

lemma \((r \cup s^+)^* = (r \cup s)^*\)
  by regexp

end

7 Basic constructions on regular expressions

theory Regexp-Constructions
imports
  Main
  HOL-Library.Sublist
  Regular-Exp
begin

7.1 Reverse language

lemma rev-conc [simp]: rev ' (A @@ B) = rev ' B @@ rev ' A
  unfolding conc_def image_def by force

lemma rev-compower [simp]: rev ' (A ^^ n) = (rev ' A) ^^ n
  by (induction n) (simp_all add: conc_pow_comm)

lemma rev-star [simp]: rev ' star A = star (rev ' A)
  by (simp add: star_def image_UN)

7.2 Substituting characters in a language

definition subst-word :: ('a => 'b list) => 'a list => 'b list where
  subst-word f xs = concat (map f xs)

lemma subst-word-Nil [simp]: subst-word f [] = []
  by (simp add: subst-word_def)

lemma subst-word-singleton [simp]: subst-word f [x] = f x
  by (simp add: subst-word_def)

lemma subst-word-append [simp]: subst-word f (xs @ ys) = subst-word f xs @ subst-word f ys
  by (simp add: subst-word_def)
lemma subst-word-Cons [simp]: subst-word \( f \) \((x \# xs)\) = \( f \ x \ @ \) subst-word \( f \) \(xs\)
by (simp add: subst-word-def)

lemma subst-word-conc [simp]: subst-word \( f \) \((A @@ B)\) = subst-word \( f \) \((A @@ subst-word \( f \) \(B\))
unfolding conc-def image-def by force

lemma subst-word-compower [simp]: subst-word \( f \) \((A \ ^^ n)\) = subst-word \( f \) \((A) \ ^^ n\)
by (induction \( n \)) simp-all

lemma subst-word-star [simp]: subst-word \( f \) \((\mathit{star} \( A \))\) = \(\mathit{star} \ (\mathit{subst-word} \( f \) \(A \))\)
by (simp add: star-def image-UN)

Suffix language
definition Suffixes :: 'a list set ⇒ 'a list set where
Suffixes \( A \) = \{w. ∃q. q @ w ∈ A\}

lemma Suffixes-altdef [code]: Suffixes \( A \) = (⋃w∈A. set (suffixes w))
unfolding Suffixes-def set-suffixes-eq suffix-def by blast

lemma Nil-in-Suffixes-iff [simp]: [] ∈ Suffixes \( A \) ←→ \( A \neq \{\}\)
by (auto simp: Suffixes-def)

lemma Suffixes-empty [simp]: Suffixes \( \{\}\) = \{\}
by (auto simp: Suffixes-def)

lemma Suffixes-empty-iff [simp]: Suffixes \( A \) = \( \{\}\) ←→ \( A = \{\}\)
by (auto simp: Suffixes-altdef)

lemma Suffixes-singleton [simp]: Suffixes \( \{xs\} \) = set (suffixes \( xs\))
by (auto simp: Suffixes-altdef)

lemma Suffixes-insert: Suffixes \((\text{insert} \( xs \ A \))\) = set (suffixes \( xs\)) ∪ Suffixes \( A \)
by (simp add: Suffixes-altdef)

lemma Suffixes-conc [simp]: \( A \neq \{\}\) → Suffixes \((A @@ B)\) = Suffixes \( B \) ∪ (Suffixes \( A @@ B\))
unfolding Suffixes-altdef conc-def by (force simp: suffix-append)

lemma Suffixes-union [simp]: Suffixes \((A \cup B)\) = Suffixes \( A \) ∪ Suffixes \( B \)
by (auto simp: Suffixes-def)

lemma Suffixes-UNION [simp]: Suffixes \((\bigcup f \ ^\ ' A)\) = \(\bigcup (\lambda x. Suffixes \( f \ x\)) \ ^\ ' A\)
by (auto simp: Suffixes-def)

lemma Suffixes-compower:
assumes \( A \neq \{\}\)
shows Suffixes \((A \ ^^ n)\) = insert [] (Suffixes \( A @@ (\bigcup k<n. A \ ^^ k)\)))
proof (induction n)
  case (Suc n)
  from Suc have Suffixes (A ^^ Suc n) =
    insert [] (Suffixes A @@ ((\k<n. A ^^ k) \cup A ^^ n))
    by (simp-all add: assms conc-Un-distrib)
  also have ((\k<n. A ^^ k) \cup A ^^ n) = (\k\in insert n {.<n}. A ^^ k) by blast
  also have insert n {.<n} = {.<Suc n} by auto
  finally show ?case .
qed simp-all

lemma Suffixes-star [simp]:
  assumes A \\{\}
  shows Suffixes (star A) = Suffixes A @@ star A
proof
  have star A = (\n. A ^^ n) unfolding star-def ..
  also have Suffixes .. = (\x. Suffixes (A ^^ x)) by simp
  also have .. = (\n. insert [] (Suffixes A @@ (\k<n. A ^^ k)))
    unfolding Suffixes-compower by auto
  also have .. = insert [] (Suffixes A @@ (\n. (\k<n. A ^^ k)))
    by (simp-all add: conc-UNION-distrib)
  also have (\n. (\k<n. A ^^ k)) = (\n. A ^^ n) by auto
  also have .. = star A unfolding star-def ..
  also have insert [] (Suffixes A @@ star A) = Suffixes A @@ star A
    unfolding star-def ..
  also have insert [] (Suffixes A @@ star A) = Suffixes A @@ star A
    unfolding star-def ..
  finally show \?thesis .
qed

Prefix language

definition Prefixes :: 'a list set where
  Prefixes A = \{ w. \exists q. w @ q \in A \}

lemma Prefixes-altdef [code]: Prefixes A = (\w\in A. set (prefixes w))
unfolding Prefixes-def set-prefixes-eq prefix-def by blast

lemma Nil-in-Prefixes-iff [simp]: [] \\in Prefixes A \iff A \\{\}
by (auto simp: Prefixes-def)

lemma Prefixes-empty [simp]: Prefixes {} = {}
by (auto simp: Prefixes-def)

lemma Prefixes-empty-iff [simp]: Prefixes A = {} \iff A = {}
by (auto simp: Prefixes-altdef)

lemma Prefixes-singleton [simp]: Prefixes \{xs\} = set (prefixes xs)
by (auto simp: Prefixes-altdef)

lemma Prefixes-insert: Prefixes (insert xs A) = set (prefixes xs) \cup Prefixes A
by (simp add: Prefixes-altdef)
lemma Prefixes-conc [simp]: \( B \neq \{\} \implies \text{Prefixes} (A @@ B) = \text{Prefixes} A \cup (A \@\@ \text{Prefixes} B) \)
  unfolding Prefixes-altdef conc-def by (force simp: prefix-append)

lemma Prefixes-union [simp]: \( \text{Prefixes} (A \cup B) = \text{Prefixes} A \cup \text{Prefixes} B \)
  by (auto simp: Prefixes-def)

lemma Prefixes-UNION [simp]: \( \text{Prefixes} (\bigcup (\lambda x. \text{Prefixes} (f x)) \ A) \)
  by (auto simp: Prefixes-def)

lemma Prefixes-rev: \( \text{Prefixes} (\text{rev} A) = \text{rev} \text{Prefixes} A \)
  by (auto simp: Prefixes-altdef prefixes-rev Suffixes-altdef)

lemma Prefixes-compower:
  assumes \( A \neq \{\} \)
  shows \( \text{Prefixes} (A \sim n) = \text{insert} [\] (\bigcup k<n. A \sim k) @@ \text{Prefixes} A \)
proof –
  have \( A \sim n = \text{rev} ((\text{rev} A) \sim n) \) by (simp add: image-image)
  also have \( \text{Prefixes} \ldots = \text{insert} [\] (\bigcup k<n. A \sim k) @@ \text{Prefixes} A \)
  unfolding Prefixes-rev
  by (subst Suffixes-compower) (simp-all add: image-UN image-image Suffixes-rev assms)
finally show \(?thesis\).
qed

lemma Prefixes-star [simp]:
  assumes \( A \neq \{\} \)
  shows \( \text{Prefixes} (\text{star} A) = \text{star} A \@@ \text{Prefixes} A \)
proof –
  have \( \text{star} A = \text{rev} \text{star} (\text{rev} A) \) by (simp add: image-image)
  also have \( \text{Prefixes} \ldots = \text{star} A \@@ \text{Prefixes} A \)
  unfolding Prefixes-rev
  by (subst Suffixes-star) (simp-all add: assms image-image Suffixes-rev)
finally show \(?thesis\).
qed

7.3 Subword language

The language of all sub-words, i.e. all words that are a contiguous sublist of a word in the original language.

definition Sublists :: 'a list set \( \Rightarrow \) 'a list set where
  \( \text{Sublists} A = \{ w. \exists q\in A. \text{sublist} w q\} \)

lemma Sublists-altdef [code]: \( \text{Sublists} A = (\bigcup w\in A. \text{set} \text{sublists} w) \)
by (auto simp: Sublists-def)

lemma Sublists-empty [simp]: Sublists {} = {}
  by (auto simp: Sublists-def)

lemma Sublists-singleton [simp]: Sublists {w} = set (sublists w)
  by (auto simp: Sublists-altdef)

lemma Sublists-insert: Sublists (insert w A) = set (sublists w) ∪ Sublists A
  by (auto simp: Sublists-altdef)

lemma Sublists-Un [simp]: Sublists (A ∪ B) = Sublists A ∪ Sublists B
  by (auto simp: Sublists-altdef)

lemma Sublists-UN [simp]: Sublists (∪ f ' A) = (∪ (λx. Sublists (f x)) ' A)
  by (auto simp: Sublists-altdef)

lemma Sublists-conv-Prefixes: Sublists A = Prefixes (Suffixes A)
  by (auto simp: Sublists-def Prefixes-def Suffixes-def sublist-def)

lemma Sublists-conv-Suffixes: Sublists A = Suffixes (Prefixes A)
  by (auto simp: Sublists-def Prefixes-def Suffixes-def sublist-def)

lemma Sublists-conc [simp]:
  assumes A ≠ {} B ≠ {} shows Sublists (A @@ B) = Sublists A ∪ Sublists B ∪ Suffixes A @@ Prefixes B
  using assms unfolding Sublists-conv-Suffixes by auto

lemma star-not-empty [simp]: star A ≠ {}
  by auto

lemma Sublists-star:
  A ≠ {} ⇒ Sublists (star A) = Sublists A ⊆ Suffixes A @@ star A @@ Prefixes A
  by (simp add: Sublists-conv-Prefixes)

lemma Prefixes-subset-Sublists: Prefixes A ⊆ Sublists A
  unfolding Prefixes-def Sublists-def by auto

lemma Suffixes-subset-Sublists: Suffixes A ⊆ Sublists A
  unfolding Suffixes-def Sublists-def by auto

7.4 Fragment language

The following is the fragment language of a given language, i.e. the set of all words that are (not necessarily contiguous) sub-sequences of a word in the original language.

definition Subseqs where Subseqs A = (⋃ w∈A. set (subseqs w))
lemma Subseqs-empty [simp]: Subseqs {} = {}
  by (simp add: Subseqs-def)

lemma Subseqs-insert [simp]: Subseqs (insert xs A) = set (subseqs xs) ∪ Subseqs A
  by (simp add: Subseqs-def)

lemma Subseqs-singleton [simp]: Subseqs {xs} = set (subseqs xs)
  by simp

lemma Subseqs-Un [simp]: Subseqs (A ∪ B) = Subseqs A ∪ Subseqs B
  by (simp add: Subseqs-def)

lemma Subseqs-UNION [simp]: Subseqs (⋃(f ' A)) = ⋃((λx. Subseqs (f x)) ' A)
  by (simp add: Subseqs-def)

lemma Subseqs-conc [simp]: Subseqs (A @@ B) = Subseqs A @@ Subseqs B
proof safe
  fix xs assume xs ∈ Subseqs (A @@ B)
  then obtain ys zs where *: ys ∈ A zs ∈ B subseq xs (ys @@ zs)
    by (auto simp: Subseqs-def conc-def)
  from * obtain xs1 xs2 where xs = xs1 @@ xs2 subseq xs1 ys subseq xs2 zs
    by (rule subseq-appendE)
  with * show xs ∈ Subseqs A @@ Subseqs B by (auto simp: Subseqs-def set-subseqs-eq)
next
  fix xs assume xs ∈ Subseqs A @@ Subseqs B
  then obtain xs1 xs2 ys zs
    where xs = xs1 @@ xs2 subseq xs1 ys subseq xs2 zs ys ∈ A zs ∈ B
    by (auto simp: conc-def Subseqs-def)
  thus xs ∈ Subseqs (A @@ B) by (force simp: Subseqs-def conc-def intro: list-emb-append-mono)
qed

lemma Subseqs-compower [simp]: Subseqs (A ^^ n) = Subseqs A ^^ n
  by (induction n) simp-all

lemma Subseqs-star [simp]: Subseqs (star A) = star (Subseqs A)
  by (simp add: star-def)

lemma Sublists-subset-Subseqs: Sublists A ⊆ Subseqs A
  by (auto simp: Sublists-def Subseqs-def dest!: sublist-imp-subseq)

7.5 Various regular expression constructions

A construction for language reversal of a regular expression:

primrec rexp-rev where
  rexp-rev Zero = Zero
| rexp-rev One = One
\[ \text{rexp-rev (Atom } x \text{)} = \text{Atom } x \]
\[ \text{rexp-rev (Plus } r \ s \text{)} = \text{Plus (rexp-rev } r \text{) (rexp-rev } s \text{)} \]
\[ \text{rexp-rev (Times } r \ s \text{)} = \text{Times (rexp-rev } s \text{) (rexp-rev } r \text{)} \]
\[ \text{rexp-rev (Star } r \text{)} = \text{Star (rexp-rev } r \text{)} \]

**Lemma** lang-rexp-rev [simp]: \( \text{lang (rexp-rev } r \text{)} = \text{rev } \text{lang } r \)

by (induction \( r \)) (simp-all add: image-Un)

The obvious construction for a singleton-language regular expression:

**Fun** rexp-of-word where
\[
\text{rexp-of-word } [] = \text{One} \\
\text{rexp-of-word } [x] = \text{Atom } x \\
\text{rexp-of-word } (x \# xs) = \text{Times (Atom } x \text{) (rexp-of-word } xs \text{)}
\]

**Lemma** lang-rexp-of-word [simp]: \( \text{lang (rexp-of-word } xs \text{)} = \{xs\} \)

by (induction \( xs \)) rule: rexp-of-word.induct (simp-all add: conc-def)

**Lemma** size-rexp-of-word [simp]: \( \text{size (rexp-of-word } xs \text{)} = \text{Suc (2 } \times (\text{length } xs - 1) \text{)} \)

by (induction \( xs \)) rule: rexp-of-word.induct auto

Character substitution in a regular expression:

**Primrec** rexp-subst where
\[
\text{rexp-subst } f \ \text{Zero} = \text{Zero} \\
\text{rexp-subst } f \ \text{One} = \text{One} \\
\text{rexp-subst } f \ (\text{Atom } x) = \text{rexp-of-word } (f \ x) \\
\text{rexp-subst } f \ (\text{Plus } r \ s) = \text{Plus (rexp-subst } f \ r \text{) (rexp-subst } f \ s \text{)} \\
\text{rexp-subst } f \ (\text{Times } r \ s) = \text{Times (rexp-subst } f \ r \text{) (rexp-subst } f \ s \text{)} \\
\text{rexp-subst } f \ (\text{Star } r) = \text{Star (rexp-subst } f \ r \text{)}
\]

**Lemma** lang-rexp-subst: \( \text{lang (rexp-subst } f \ r \text{)} = \text{subst-word } f \ ' \text{lang } r \)

by (induction \( r \)) (simp-all add: image-Un)

Suffix language of a regular expression:

**Primrec** suffix-rexp :: \( \text{'a rexp } \Rightarrow \text{'a rexp where} \)
\[
\text{suffix-rexp } \text{Zero} = \text{Zero} \\
\text{suffix-rexp } \text{One} = \text{One} \\
\text{suffix-rexp } (\text{Atom } a) = \text{Plus (Atom } a \text{) One} \\
\text{suffix-rexp } (\text{Plus } r \ s) = \text{Plus (suffix-rexp } r \text{) (suffix-rexp } s \text{)} \\
\text{suffix-rexp } (\text{Times } r \ s) = \\
\quad \begin{cases} 
\text{Zero} & \text{if rexp-empty } r \\
\text{Plus (Times (suffix-rexp } r \text{) } s \text{)} & \text{if rexp-empty } r \text{ else}
\end{cases}
\]
\[
\text{suffix-rexp } (\text{Star } r) = \\
\quad \begin{cases} 
\text{One} & \text{if rexp-empty } r \\
\text{Times (suffix-rexp } r \text{) (Star } r) & \text{else}
\end{cases}
\]

**Theorem** lang-suffix-rexp [simp]: \( \text{lang (suffix-rexp } r \text{)} = \text{Suffixes (lang } r \text{)} \)

by (induction \( r \)) (auto simp: rexp-empty-iff)

Prefix language of a regular expression:

**Primrec** prefix-rexp :: \( \text{'a rexp } \Rightarrow \text{'a rexp where} \)
prefix-rexp Zero = Zero
| prefix-rexp One = One
| prefix-rexp (Atom a) = Plus (Atom a) One
| prefix-rexp (Plus r s) = Plus (prefix-rexp r) (prefix-rexp s)
| prefix-rexp (Times r s) =
  (if rexp-empty s then Zero else Plus (Times r (prefix-rexp s)) (prefix-rexp r))
| prefix-rexp (Star r) =
  (if rexp-empty r then One else Times (Star r) (prefix-rexp r))

theorem lang-prefix-rexp [simp]:
  lang (prefix-rexp r) = Prefixes (lang r)
by (induction r) (auto simp: rexp-empty-iff)

Sub-word language of a regular expression

primrec sublist-rexp :: 'a rexp ⇒ 'a rexp where
  sublist-rexp Zero = Zero
| sublist-rexp One = One
| sublist-rexp (Atom a) = Plus (Atom a) One
| sublist-rexp (Plus r s) = Plus (sublist-rexp r) (sublist-rexp s)
| sublist-rexp (Times r s) =
  (if rexp-empty r ∨ rexp-empty s then Zero else
   Plus (sublist-rexp r) (Plus (sublist-rexp s) (Times (suffix-rexp r) (prefix-rexp s))))
| sublist-rexp (Star r) =
  (if rexp-empty r then One else
   Plus (sublist-rexp r) (Times (suffix-rexp r) (Times (Star r) (prefix-rexp r))))

theorem lang-sublist-rexp [simp]:
  lang (sublist-rexp r) = Sublists (lang r)
by (induction r) (auto simp: rexp-empty-iff Sublists-star)

Fragment language of a regular expression:

primrec subseqs-rexp :: 'a rexp ⇒ 'a rexp where
  subseqs-rexp Zero = Zero
| subseqs-rexp One = One
| subseqs-rexp (Atom x) = Plus (Atom x) One
| subseqs-rexp (Plus r s) = Plus (subseqs-rexp r) (subseqs-rexp s)
| subseqs-rexp (Times r s) = Times (subseqs-rexp r) (subseqs-rexp s)
| subseqs-rexp (Star r) = Star (subseqs-rexp r)

lemma lang-subseqs-rexp [simp]: lang (subseqs-rexp r) = Subseqs (lang r)
by (induction r) auto

Subword language of a regular expression

end

8 Derivatives of regular expressions

theory Derivatives
This theory is based on work by Brozowski [2] and Antimirov [1].

8.1 Brzozowski’s derivatives of regular expressions

fun deriv :: 'a ⇒ 'a rexp ⇒ 'a rexp
where
  deriv c (Zero) = Zero
| deriv c (One) = Zero
| deriv c (Atom c') = (if c = c' then One else Zero)
| deriv c (Plus r1 r2) = Plus (deriv c r1) (deriv c r2)
| deriv c (Times r1 r2) = (if nullable r1 then Plus (Times (deriv c r1) r2) (deriv c r2) else Times (deriv c r1) r2)
| deriv c (Star r) = Times (deriv c r) (Star r)

fun derivs :: 'a list ⇒ 'a rexp ⇒ 'a rexp
where
  derivs [] r = r
| derivs (c # s) r = derivs s (deriv c r)

lemma atoms-deriv-subset: atoms (deriv x r) ⊆ atoms r
by (induction r) (auto)

lemma atoms-derivs-subset: atoms (derivs w r) ⊆ atoms r
by (induction w arbitrary: r) (auto dest: atoms-deriv-subset THEN subsetD)

lemma lang-deriv: lang (deriv c r) = Deriv c (lang r)
by (induct r) (simp-all add: nullable-iff)

lemma lang-derivs: lang (derivs s r) = Derivs s (lang r)
by (induct s arbitrary: r) (simp-all add: lang-deriv)

A regular expression matcher:

definition matcher :: 'a rexp ⇒ 'a list ⇒ bool where
matcher r s = nullable (derivs s r)

lemma matcher-correctness: matcher r s ⇔ s ∈ lang r
by (induct s arbitrary: r)
  (simp-all add: nullable-iff lang-deriv matcher-def Deriv-def)

8.2 Antimirov’s partial derivatives

abbreviation
Times rs r ≡ (∪r' ∈ rs. {Times r' r})
lemma Timess-eq-image:
\[ \text{Timess } r \equiv (\lambda r'. \text{Times } r' ) \cdot rs \]
by auto

primrec
\[ \text{pderiv } :: \ 'a \Rightarrow 'a \text{ rexp } \Rightarrow 'a \text{ rexp set} \]
where
\[ \text{pderiv } c \text{ Zero} = \{\} \]
\[ \text{pderiv } c \text{ One} = \{\} \]
\[ \text{pderiv } c \text{ (Atom } c') = (\text{if } c = c' \text{ then } \{\text{One}\} \text{ else } \{\}) \]
\[ \text{pderiv } c \text{ (Plus } r1 \text{ r2)} = (\text{pderiv } c \text{ r1}) \cup (\text{pderiv } c \text{ r2}) \]
\[ \text{pderiv } c \text{ (Times } r1 \text{ r2)} = \]
\[ \text{(if } \text{nullable } r1 \text{ then Timess } (\text{pderiv } c \text{ r1}) \text{ r2} \cup \text{pderiv } c \text{ r2} \text{ else Timess } (\text{pderiv } c \text{ r1}) \text{ r2}) \]
\[ \text{pderiv } c \text{ (Star } r) = \text{Timess } (\text{pderiv } c \text{ r}) \text{ (Star } r) \]

primrec
\[ \text{pderivs } :: \ 'a \text{ list } \Rightarrow 'a \text{ rexp } \Rightarrow (\text{'}a \text{ rexp}) \text{ set} \]
where
\[ \text{pderivs } [] \cdot r = \{r\} \]
\[ \text{pderivs } (c \ # \ s) \cdot r = \bigcup (\text{pderivs } s \cdot \text{pderiv } c \cdot r) \]

abbreviation
\[ \text{pderiv-set } :: \ 'a \Rightarrow 'a \text{ rexp set } \Rightarrow 'a \text{ rexp set} \]
where
\[ \text{pderiv-set } c \text{ rs} \equiv \bigcup (\text{pderiv } c \cdot \text{rs}) \]

abbreviation
\[ \text{pderivs-set } :: \ 'a \text{ list } \Rightarrow 'a \text{ rexp set } \Rightarrow 'a \text{ rexp set} \]
where
\[ \text{pderivs-set } s \text{ rs} \equiv \bigcup (\text{pderivs } s \cdot \text{rs}) \]

lemma pderivs-append:
\[ \text{pderivs } (s1 \ @ \ s2) \cdot r = \bigcup (\text{pderivs } s2 \cdot \text{pderiv } s1 \cdot r) \]
by (induct \(s1\) arbitrary: \(r\)) (simp-all)

lemma pderivs-snoc:
\[ \text{shows } \text{pderivs } (s \ @ \ [c]) \cdot r = \text{pderiv-set } c \cdot (\text{pderivs } s \cdot r) \]
by (simp add: pderivs-append)

lemma pderivs-simps [simp]:
\[ \text{shows } \text{pderivs } s \cdot \text{Zero} = (\text{if } s = [] \text{ then } \{\text{Zero}\} \text{ else } \{\}) \]
and \(\text{pderivs } s \cdot \text{One} = (\text{if } s = [] \text{ then } \{\text{One}\} \text{ else } \{\})\)
and \(\text{pderivs } s \cdot (\text{Plus } r1 \text{ r2}) = (\text{if } s = [] \text{ then } (\text{Plus } r1 \text{ r2}) \text{ else } (\text{pderivs } s \cdot r1) \cup (\text{pderivs } s \cdot r2))\)
by (induct \(s\)) (simp-all)

lemma pderivs-Atom:
\[ \text{shows } \mathbin{\text{pderivs } s (\text{Atom } c)} \subseteq \{\text{Atom } c, \text{One}\} \]

by \((\text{induct } s) \ (\text{simp-all})\)

8.3 Relating left-quotients and partial derivatives

\textbf{lemma} \(\text{Deriv-pderiv:}\)

\begin{itemize}
  \item \textbf{shows} \(\text{Deriv } c (\text{lang } r) = \bigcup (\text{lang } ' \text{ pderiv } c r)\)
  \item \textbf{by} \((\text{induct } r) \ (\text{auto simp add: nullable-iff conc-UNION-distrib})\)
\end{itemize}

\textbf{lemma} \(\text{Derivs-pderivs:}\)

\begin{itemize}
  \item \textbf{shows} \(\text{Derivs } s (\text{lang } r) = \bigcup (\text{lang } ' \text{ pderivs } s r)\)
  \item \textbf{proof} \((\text{induct } s \text{ arbitrary: } r)\)
    \begin{itemize}
      \item \textbf{case} \((\text{Cons } c s)\)
        \begin{itemize}
          \item \textbf{have} \(\text{ih: } \forall r. \text{Deriv } s (\text{lang } r) = \bigcup (\text{lang } ' \text{ pderivs } s r) \text{ by fact}\)
          \item \textbf{have} \(\text{Deriv } c (\text{lang } r) = \text{Derivs } s (\text{Deriv } c (\text{lang } r)) \text{ by simp}\)
          \item \textbf{also have} \(\ldots = \text{Derivs } s (\bigcup (\text{lang } ' \text{ pderiv } c r)) \text{ by (simp add: Deriv-pderiv)}\)
          \item \textbf{also have} \(\ldots = \text{Derivss } s (\text{lang } ' (\text{pderiv } c r))\)
          \item \textbf{by} \((\text{auto simp add: Derivs-def})\)
          \item \textbf{also have} \(\ldots = \bigcup (\text{lang } ' (\text{pderivs-set } s (\text{pderiv } c r)))\)
          \item \textbf{using} \(\text{ih by auto}\)
          \item \textbf{also have} \(\ldots = \bigcup (\text{lang } ' (\text{pderivs } c (\# s) r)) \text{ by simp}\)
          \item \textbf{finally show} \(\text{Deriv } c (\# s) (\text{lang } r) = \bigcup (\text{lang } ' \text{ pderivs } (c (\# s) r))\).\)
        \end{itemize}
    \end{itemize}
  \item \textbf{qed} \((\text{simp add: Derivs-def})\)
\end{itemize}

8.4 Relating derivatives and partial derivatives

\textbf{lemma} \(\text{deriv-pderiv:}\)

\begin{itemize}
  \item \textbf{shows} \(\bigcup (\text{lang } ' (\text{pderiv } c r)) = \text{lang } (\text{deriv } c r)\)
  \item \textbf{unfolding} \(\text{lang-deriv Deriv-pderiv by simp}\)
\end{itemize}

\textbf{lemma} \(\text{derivs-pderivs:}\)

\begin{itemize}
  \item \textbf{shows} \(\bigcup (\text{lang } ' (\text{pderivs } s r)) = \text{lang } (\text{derivs } s r)\)
  \item \textbf{unfolding} \(\text{lang-deriv Derivs-pderivs by simp}\)
\end{itemize}

8.5 Finiteness property of partial derivatives

\textbf{definition} \(\text{pderivs-lang :: 'a lang } \Rightarrow 'a \text{ rexp } \Rightarrow 'a \text{ rexp set}\)

\textbf{where} \(\text{pderivs-lang } A r \equiv \bigcup x \in A. \text{ pderivs } x r\)

\textbf{lemma} \(\text{pderivs-lang-subsetI:}\)

\begin{itemize}
  \item \textbf{assumes} \(\forall s. s \in A \Rightarrow \text{pderivs } s r \subseteq C\)
  \item \textbf{shows} \(\text{pderivs-lang } A r \subseteq C\)
  \item \textbf{using} \(\text{assms unfolding pderivs-lang-def by (rule UN-least)}\)
\end{itemize}

\textbf{lemma} \(\text{pderivs-lang-union:}\)

\begin{itemize}
  \item \textbf{shows} \(\text{pderivs-lang } (A \cup B) r = (\text{pderivs-lang } A r \cup \text{pderivs-lang } B r)\)
  \item \textbf{by} \((\text{simp add: pderivs-lang-def})\)
\end{itemize}
lemma pderivs-lang-subset:
  shows \( A \subseteq B \Rightarrow pderivs-lang A r \subseteq pderivs-lang B r \)
by (auto simp add: pderivs-lang-def)

definition UNIV1 \equiv UNIV - {[]}

lemma pderivs-lang-Zero [simp]:
  shows pderivs-lang UNIV1 Zero = {}
unfolding UNIV1-def pderivs-lang-def by auto

lemma pderivs-lang-One [simp]:
  shows pderivs-lang UNIV1 One = {}
unfolding UNIV1-def pderivs-lang-def by (auto split: if-splits)

lemma pderivs-lang-Atom [simp]:
  shows pderivs-lang UNIV1 (Atom c) = {One}
unfolding UNIV1-def pderivs-lang-def
apply(auto)
apply(frule rev-subsetD)
apply(rule pderivs-Atom)
apply(simp)
apply(case_tac xa)
apply(auto split: if-splits)
done

lemma pderivs-lang-Plus [simp]:
  shows pderivs-lang UNIV1 (Plus r1 r2) = pderivs-lang UNIV1 r1 \cup pderivs-lang UNIV1 r2
unfolding UNIV1-def pderivs-lang-def by auto

  Non-empty suffixes of a string (needed for the cases of Times and Star below)

definition PSuf s \equiv \{v. v \neq [] \land (\exists u. u @ v = s)\}

lemma PSuf-snoc:
  shows PSuf (s @ [c]) = (PSuf s) @@ {[c]} \cup {[c]}
unfolding PSuf-def conc-def by (auto simp add: append-eq-append-conv2 append-eq-Cons-conv)

lemma PSuf-Union:
  shows (\bigcup v \in PSuf s @ @ {[c]}, f v) = (\bigcup v \in PSuf s. f (v @ [c]))
by (auto simp add: conc-def)

lemma pderivs-lang-snoc:
  shows pderivs-lang (PSuf s @ @ {[c]}) r = (pderiv-set c (pderivs-lang (PSuf s) r))
unfolding pderivs-lang-def
by \textit{(simp add: PSuf-Union pderivs-snoc)}

\textbf{lemma} \, pderivs-Times:
\textit{shows} pderivs \, s \, (\text{Times} \, r1 \, r2) \subseteq \text{Timess} \, (\text{pderivs} \, s \, r1) \, r2 \cup (\text{pderivs-lang} \, (\text{PSuf} \, s) \, r2)
\textit{proof} \, (\text{induct} \, s \, \text{rule: rev-induct})
\text{case} \, (\text{snoc} \, c \, s)
\textit{have} \, \textit{ih}: \, pderivs \, s \, (\text{Times} \, r1 \, r2) \subseteq \text{Timess} \, (\text{pderivs} \, s \, r1) \, r2 \cup (\text{pderivs-lang} \, (\text{PSuf} \, s) \, r2)
\textit{by \, fact}
\textit{have} \, \textit{pderiv} \, (s @ [c]) \, (\text{Times} \, r1 \, r2) = \text{pderiv-set} \, c \, (\text{pderiv} \, s \, (\text{Times} \, r1 \, r2))
\textit{by \, (simp \, add: pderivs-snoc)}
\textit{also \, have} \, \ldots \subseteq \text{pderiv-set} \, c \, (\text{Timess} \, (\text{pderivs} \, s \, r1) \, r2 \cup (\text{pderivs-lang} \, (\text{PSuf} \, s) \, r2))
\textit{using} \, \textit{ih \, by \, fastforce}
\textit{also \, have} \, \ldots = \text{pderiv-set} \, c \, (\text{Timess} \, (\text{pderivs} \, s \, r1) \, r2) \cup \text{pderiv-set} \, c \, (\text{pderivs-lang} \, (\text{PSuf} \, s) \, r2)
\textit{by \, (simp)}
\textit{also \, have} \, \ldots = \text{pderiv-set} \, c \, (\text{Timess} \, (\text{pderivs} \, s \, r1) \, r2) \cup \text{pderivs-lang} \, (\text{PSuf} \, s \, @@ \{[c]\}) \, r2
\textit{by \, (simp \, add: pderivs-lang-snoc)}
\textit{also}
\textit{have} \, \ldots \subseteq \text{pderiv-set} \, c \, (\text{Timess} \, (\text{pderivs} \, s \, r1) \, r2) \cup \text{pderiv} \, c \, r2 \cup \text{pderivs-lang} \, (\text{PSuf} \, s \, @@ \{[c]\}) \, r2
\textit{by \, auto}
\textit{also}
\textit{have} \, \ldots \subseteq \text{Timess} \, (\text{pderiv-set} \, c \, (\text{pderivs} \, s \, r1)) \, r2 \cup \text{pderiv} \, c \, r2 \cup \text{pderivs-lang} \, (\text{PSuf} \, s \, @@ \{[c]\}) \, r2
\textit{by \, (auto \, simp \, add: if-splits)}
\textit{also \, have} \, \ldots = \text{Timess} \, (\text{pderiv} \, (s @ [c]) \, r1) \, r2 \cup \text{pderiv} \, c \, r2 \cup \text{pderivs-lang} \, (\text{PSuf} \, s \, @@ \{[c]\}) \, r2
\textit{by \, (simp \, add: pderivs-snoc)}
\textit{also \, have} \, \ldots \subseteq \text{Timess} \, (\text{pderiv} \, (s @ [c]) \, r1) \, r2 \cup \text{pderivs-lang} \, (\text{PSuf} \, (s @ [c])) \, r2
\textit{unfolding} \, \text{pderivs-lang-def} \, \textit{by \, (auto \, simp \, add: PSuf-snoc)}
\textit{finally \, show} \, ?case \, .
\textit{qed \, (simp)}

\textbf{lemma} \, pderivs-lang-Times-aux1:
\textit{assumes} \, a: \, s \in \text{UNIV1}
\textit{shows} \, \text{pderivs-lang} \, (\text{PSuf} \, s) \, r \subseteq \text{pderivs-lang} \, \text{UNIV1} \, r
\textit{using} \, a \, \text{unfolding} \, \text{UNIV1-def \, PSuf-def \, pderivs-lang-def \, by \, auto}

\textbf{lemma} \, pderivs-lang-Times-aux2:
\textit{assumes} \, a: \, s \in \text{UNIV1}
\textit{shows} \, \text{Timess} \, (\text{pderivs} \, s \, r1) \, r2 \subseteq \text{Timess} \, (\text{pderivs-lang} \, \text{UNIV1} \, r1) \, r2
\textit{using} \, a \, \text{unfolding} \, \text{pderivs-lang-def \, by \, auto}

\textbf{lemma} \, pderivs-lang-Times:
shows \( \text{pderivs-lang UNIV1} (\text{Times r1 r2}) \subseteq \text{Times} (\text{pderivs-lang UNIV1 r1}) \text{ r2} \)
apply(rule pderivs-lang-subsetI)
apply(rule subset-trans)
apply(rule pderivs-Times)
using pderivs-lang-Times-aux1 pderivs-lang-Times-aux2
apply auto
apply blast
done

lemma pderivs-Star:
assumes a: \( s \neq [] \)
shows \( \text{pderivs s} (\text{Star r}) \subseteq \text{Times} (\text{pderivs-lang (PSuf s) r}) \text{ (Star r)} \)
using a

proof (induct s rule: rev-induct)
case (snoc c s)
have ih: \( s \neq [] \Rightarrow \text{pderivs s} (\text{Star r}) \subseteq \text{Times} (\text{pderivs-lang (PSuf s) r}) \text{ (Star r)} \) by fact
{ assume asm: \( s \neq [] \)
  have pderiv (s @ [c]) (Star r) = pderiv-set c (pderiv s (Star r)) by (simp add: pderivs-snoc)
  also have \( \ldots \subseteq \text{pderiv-set c} \ (\text{Times} (\text{pderivs-lang (PSuf s) r}) \text{ (Star r)}) \) using ih[OF asm] by fast
  also have \( \ldots \subseteq \text{Times} (\text{pderiv-set c} \ (\text{pderivs-lang (PSuf s) r})) \text{ (Star r)} \cup \text{pderiv c (Star r)} \)
  by (auto split: if-splits)
  also have \( \ldots \subseteq \text{Times} (\text{pderiv-lang (PSuf (s @ [c]) r}) \text{ (Star r)} \cup \text{Times (pderiv c r) (Star r)}) \)
  by (simp only: PSuf-snoc pderivs-lang-snoc pderivs-lang-union)
  (auto simp add: pderivs-lang-def)
  also have \( \ldots = \text{Times} (\text{pderiv-lang (PSuf (s @ [c]) r}) \text{ (Star r)}) \)
  by (auto simp add: PSuf-snoc PSuf-Union pderivs-snoc pderivs-lang-def)
  finally have \?case .
}
moreover
{ assume asm: \( s = [] \)
  then have \?case by (auto simp add: pderivs-lang-def pderivs-snoc PSuf-def)
}
ultimately show \?case by blast
qed (simp)

lemma pderivs-lang-Star:
shows pderivs-lang UNIV1 (Star r) \( \subseteq \text{Times} (\text{pderivs-lang UNIV1 r}) \text{ (Star r)} \)
apply(rule pderivs-lang-subsetI)
apply(rule subset-trans)
apply(rule pderivs-Star)
apply(simp add: UNIV1-def)
apply(simp add: UNIV1-def PSuf-def)
apply(auto simp add: pderivs-lang-def)

38
done

lemma finite-Timess [simp]:
  assumes a: finite A
  shows finite (Timess A r)
using a by auto

lemma finite-pderivs-lang-UNIV1:
  shows finite (pderivs-lang UNIV1 r)
apply (induct r)
apply (simp-all add:
  finite-subset[OF pderivs-lang-Times]
  finite-subset[OF pderivs-lang-Star])
done

lemma pderivs-lang-UNIV:
  shows pderivs-lang UNIV r = pderivs [] r ∪ pderivs-lang UNIV1 r
unfolding UNIV1-def pderivs-lang-def
by blast

lemma finite-pderivs-lang-UNIV:
  shows finite (pderivs-lang UNIV r)
unfolding pderivs-lang-UNIV
by (simp add: finite-pderivs-lang-UNIV1)

The following relationship between the alphabetic width of regular expressions (called \( \text{awidth} \) below) and the number of partial derivatives was proved by Antimirov \cite{1} and formalized by Max Haslbeck.

fun awidth :: 'a rexp ⇒ nat where
awidth Zero = 0 |
awidth One = 0 |
awidth (Atom a) = 1 |
awidth (Plus r1 r2) = awidth r1 + awidth r2 |
awidth (Times r1 r2) = awidth r1 + awidth r2 |
awidth (Star r1) = awidth r1

lemma card-Timess-pderivs-lang-le:
  card (Timess (pderivs-lang A r) s) ≤ card (pderivs-lang A r)
using finite-pderivs-lang unfolding Timess-eq-image by (rule card-image-le)

lemma card-pderivs-lang-UNIV1-le-awidth: card (pderivs-lang UNIV1 r) ≤ awidth r
proof (induction r)
case (Plus r1 r2)
  have card (pderivs-lang UNIV1 (Plus r1 r2)) = card (pderivs-lang UNIV1 r1 ∪
also have \ldots \leq \card{(\text{pderivs-lang UNIV} r_1) + \text{card}(\text{pderivs-lang UNIV} r_2)}
  by\,(\text{simp add: card-Un-le})
also have \ldots \leq \awidth{(\text{Plus} r_1 r_2)}\text{ using Plus.IH by simp}
finally show \textit{?case}.
next
  case (\text{Times} r_1 r_2)
  have \card{(\text{pderivs-lang UNIV} (\text{Times} r_1 r_2))} \leq \card{(\text{Timess} (\text{pderivs-lang UNIV} r_1) r_2)) + \text{card}(\text{pderivs-lang UNIV} r_2)}
    by\,(\text{simp add: card-mono finite-pderivs-lang pderivs-lang-Times})
also have \ldots \leq \card{(\text{Timess} (\text{pderivs-lang UNIV} r_1) r_2)) + \text{card}(\text{pderivs-lang UNIV} r_2)}
    by\,(\text{simp add: card-Un-le})
also have \ldots \leq \awidth{(\text{Times} r_1 r_2)}\text{ using Times.IH by simp}
finally show \textit{?case}.
next
  case (\text{Star} r)
  have \card{(\text{pderivs-lang UNIV} (\text{Star} r))} \leq \card{(\text{Timess} (\text{pderivs-lang UNIV} r) (\text{Star} r))}
    by\,(\text{simp add: card-mono finite-pderivs-lang pderivs-lang-Star})
also have \ldots \leq \text{card}(\text{pderivs-lang UNIV} r)\text{ by\,(rule card-Timess-pderivs-lang-le)}
also have \ldots \leq \awidth{(\text{Star} r)}\text{ by\,(simp add: Star.IH)}
finally show \textit{?case}.
qed\,(auto)

Antimirov’s Theorem 3.4:

\textbf{theorem} card-pderivs-lang-UNIV-le-awidth: \card{(\text{pderivs-lang UNIV} r)} \leq \awidth{r + 1}
proof –
  have \card{(\text{insert} r (\text{pderivs-lang UNIV} r))} \leq \text{Suc} \,(\card{(\text{pderivs-lang UNIV} r})
    by\,(\text{auto simp: card-insert-if[OF finite-pderivs-lang-UNIV1]})
also have \ldots \leq \text{Suc} \,(\awidth{r})\text{ by\,(simp add: card-pderivs-lang-UNIV1-le-awidth)}
finally show \textit{?thesis}\text{ by\,(simp add: pderivs-lang-UNIV)}
qed

Antimirov’s Corollary 3.5:

\textbf{corollary} card-pderivs-lang-le-awidth: \card{(\text{pderivs-lang A} r)} \leq \awidth{r + 1}
by\,(\text{rule order-trans[OF card-mono[OF finite-pderivs-lang-UNIV pderivs-lang-subset[OF subset-UNIV]]\ card-pderivs-lang-UNIV-le-awidth]})

end

9 Deciding Regular Expression Equivalence (2)

\textbf{theory} pEquivalence-Checking
imports Equivalence-Checking Derivatives
begin

Similar to theory Regular-Sets.Equivalence-Checking, but with partial derivatives instead of derivatives.

Lifting many notions to sets:

definition Lang R == UN r:R. lang r
definition Nullable R == EX r:R. nullable r
definition Pderiv a R == UN r:R. pderiv a r
definition Atoms R == UN r:R. atoms r

lemma Atoms-pderiv: Atoms(pderiv a r) ⊆ atoms r
apply (induct r)
apply (auto simp: Atoms-def UN-subset-iff)
apply (fastforce)+
done

lemma Atoms-Pderiv: Atoms(Pderiv a R) ⊆ Atoms R
using Atoms-pderiv by (fastforce simp: Atoms-def Pderiv-def)

lemma pderiv-no-occurrence:
  x /∈ atoms r → pderiv x r = {}
by (induct r) auto

lemma Pderiv-no-occurrence:
  x /∈ Atoms R → Pderiv x R = {}
by (auto simp: pderiv-no-occurrence Atoms-def Pderiv-def)

lemma Deriv-Lang: Deriv c (Lang R) = Lang (Pderiv c R)
by (auto simp: Deriv-pderiv Pderiv-def Lang-def)

lemma Nullable-pderiv[simp]: Nullable(pderivs w r) = (w : lang r)
apply (induction w arbitrary: r)
apply (simp add: Nullable-def nullable-iff singleton-iff)
using eqset-imp-iff[OF Deriv-pderiv where 'a = 'a]
apply (simp add: Nullable-def Deriv-def)
done

type-synonym 'a Rexp-pair = 'a rexp set * 'a rexp set

definition is-Bisimulation :: 'a list ⇒ 'a Rexp-pairs ⇒ bool
where
  is-Bisimulation as ps =
  (∀(R,S)∈ set ps. Atoms R ∪ Atoms S ⊆ set as ∧
    (Nullable R ↔ Nullable S) ∧

41
(∀ a∈ set as. (Pderiv a R, Pderiv a S) ∈ set ps)\)

lemma Bisim-Lang-eq:
assumes Bisim: is-Bisimulation as ps
assumes (R, S) ∈ set ps
shows Lang R = Lang S
proof –
define ps’ where ps’ = ({} , {}) # ps
from Bisim have Bisim’: is-Bisimulation as ps’
by (fastforce simp: ps’-def is-Bisimulation-def UN-subset-iff Pderiv-def Atoms-def)
show ?thesis
proof (rule language-coinduct[where R=?R])
  from ⟨R,S⟩ ∈ set ps
  have (R,S) ∈ set ps’ by (auto simp: ps’-def)
  thus ?R (Lang R) (Lang S) by auto
next
fix K L assume ?R K L
then obtain R S where rs: (R, S) ∈ set ps’
  and KL: K = Lang R L = Lang S by auto
with Bisim’ have Nullable R ←→ Nullable S
  by (auto simp: is-Bisimulation-def)
thus [] ∈ K ←→ [] ∈ L
  by (auto simp: nullable-iff KL Nullable-def Lang-def)
fix a
show ?R (Deriv a K) (Deriv a L)
proof cases
  assume a ∈ set as
  with rs Bisim’
  have (Pderiv a R, Pderiv a S) ∈ set ps’
    by (auto simp: is-Bisimulation-def)
  thus ?thesis by (fastforce simp: KL Deriv-Lang)
next
assume a ∉ set as
with Bisim’ rs
have a ∉ Atoms R ∪ Atoms S
  by (fastforce simp: is-Bisimulation-def UN-subset-iff)
then have Pderiv a R = {} Pderiv a S = {}
  by (metis Pderiv-no-occurrence Un-iff)+
then have Deriv a K = Lang {} Deriv a L = Lang {}
  unfolding KL Deriv-Lang by auto
thus ?thesis by (auto simp: ps’-def)
qed
qed
qed

9.1 Closure computation

def fun test :: 'a Rexp-pairs * 'a Rexp-pairs ⇒ bool where
\[\text{test} (\text{ws}, \text{ps}) = (\text{case ws of } [] \Rightarrow \text{False} | (R, S) \# - \Rightarrow \text{Nullable } R = \text{Nullable } S)\]

\textbf{fun step ::} 'a list \Rightarrow
'a Rexp-pairs * 'a Rexp-pairs \Rightarrow 'a Rexp-pairs * 'a Rexp-pairs
\textbf{where} step as (ws, ps) =
(let
  (R, S) = \text{hd ws};
  ps' = (R, S) # ps;
  succs = \text{map} (\lambda (Pderiv a R, Pderiv a S)) as;
  new = \text{filter} (\lambda p. p \notin set ps \cup set ws) succs
in (\text{remdups new} @ tl ws, ps'))

\textbf{definition closure ::}
'a list \Rightarrow 'a Rexp-pairs * 'a Rexp-pairs \Rightarrow 'a Rexp-pairs * 'a Rexp-pairs * option
\textbf{where} closure as = \text{while-option test} (\text{step as})

\textbf{definition pre-Bisim ::} 'a list \Rightarrow 'a rexp set \Rightarrow 'a rexp set \Rightarrow 'a Rexp-pairs * 'a Rexp-pairs \Rightarrow bool
\textbf{where} pre-Bisim as R S = (\lambda (ws, ps),
  ((R, S) \in set ws \cup set ps) \land
  (\forall (R, S) \in set ws \cup set ps. \text{Atoms } R \cup \text{Atoms } S \subseteq set as) \land
  (\forall (R, S) \in set ps. (\text{Nullable } R \leftrightarrow \text{Nullable } S) \land
  (\forall a \in set as. (Pderiv a R \leftrightarrow Pderiv a S) \in set ps \cup set ws)))

\textbf{lemma step-set-eq:} [\text{test} (\text{ws}, \text{ps}); \text{step as} (\text{ws}, \text{ps}) = (\text{ws}', \text{ps}')] \Rightarrow set \text{ws}' \cup set \text{ps}' =
\text{set \text{ws} \cup set \text{ps}}
\cup (\bigcup a \in set as. ((Pderiv a (\text{fst}(\text{hd ws})), Pderiv a (\text{snd}(\text{hd ws})))))
\text{by(auto split: list.splits)}

\textbf{theorem closure-sound:}
\textbf{assumes} result: closure as ([R,S],[]) = Some([], ps)
\textbf{and} atoms: \text{Atoms } R \cup \text{Atoms } S \subseteq set as
\textbf{shows} \text{Lang } R = \text{Lang } S
\textbf{proof} -
\{ \text{fix st}
  \text{have pre-Bisim as R S st \Rightarrow test st \Rightarrow pre-Bisim as R S (step as st)}
\text{unfolding pre-Bisim-def}
\text{proof(split prod.splits, elim case-prodE conjE, intro allI impI conjI, goal-cases)}
\text{case 1 thus ?case by(auto split: list.splits)}
\text{next}
\text{case prems: (2 ws ps ws' ps')}
\text{note prems(2)[simp]}
\text{from (test st) obtain wstl R S where [simp]: ws = (R, S)#wstl}
\text{ by (auto split: list.splits)}
\text{from (step as st = (ws',ps')) obtain P where [simp]: ps' = (R, S) # ps}
\text{ and [simp]: ws' = remdups(filter P (map (\lambda a. (Pderiv a R, Pderiv a S))}

43
as)) @ wstl
  by auto
have \( \forall (R',S') \in \text{set } wstl \cup \text{set } ps' \). Atoms \( R' \cup \text{Atoms } S' \subseteq \text{set as} \)
  using prems(4) by auto
moreover
have \( \forall a \in \text{set as}. \text{Atoms}(\text{Pderiv } a \ R) \cup \text{Atoms}(\text{Pderiv } a \ S) \subseteq \text{set as} \)
  using prems(4) by simp (metis (lifting) Atoms-Pderiv order-trans)
ultimately show \(?case by simp blast\)
next
  case 3 thus \(?case
apply (clarsimp simp: image_iff split: prod.splits list.splits)
by hypsubst-thin metis
qed

9.2 The overall procedure

definition check-eqv :: 'a rexp ⇒ 'a rexp ⇒ bool
where
check-eqv r s = (case closure (add-atoms r (add-atoms s [])) ([(r), (s)], []) of
  Some([], -) ⇒ True | - ⇒ False)

lemma soundness: assumes check-eqv r s shows lang r = lang s
proof
  let \( ?as = \text{add-atoms } r \ (\text{add-atoms } s []) \)
obtain ps where 1: closure \( ?as [(r), (s)], [] = \text{Some}([], ps) \)
  using assms by (auto simp: check-eqv_def split: option.splits list.splits)
then have lang r = lang s
  by (rule closure-sound[of - {r} {s}, simplified Lang_def, simplified])
    (auto simp: set-add-atoms Atoms_def)
thus lang r = lang s by simp
qed

Test:

lemma check-eqv
(Plus One (Times (Atom 0) (Star (Atom 0))))
(Star (Atom (0::nat)))
by eval
9.3 Termination and Completeness

**definition** PDERIVS :: 'a rexp set ⇒ 'a rexp set where PDERIVS R = (UN r:R. pderivs-lang UNIV r)

**lemma** PDERIVS-incr[simp]: R ⊆ PDERIVS R
**apply** (auto simp add: PDERIVS-def pderivs-lang-def)
**by** (metis pderivs.simps(1) insertI1)

**lemma** Pderiv-PDERIVS: assumes R' ⊆ PDERIVS R shows Pderiv a R' ⊆ PDERIVS R
**proof**
fix r assume r: Pderiv a R'
then obtain r' where r': R' r : pderiv a r' by(auto simp: Pderiv-def)
from r': R' r' ⊆ PDERIVS R obtain s where s : R r' : pderiv w s
by(auto simp: PDERIVS-def pderivs-lang-def)
hence r ∈ pderiv (w @ [a]) s using r : pderiv a r' by(auto simp add:pderiv-snoc)
thus r : PDERIVS R using s by(auto simp: PDERIVS-def pderivs-lang-def)
**qed**

**lemma** finite-PDERIVS: finite R ⇒ finite(PDERIVS R)
**by** (simp add: PDERIVS-def finite-PDERIVS-lang-UNIV)

**lemma** closure-Some: assumes finite R0 finite S0 shows ∃ p. closure as ((R0,S0),[])
= Some p
**proof**(unfold closure-def)
let ?Inv = %(ws,bs). distinct ws ∧ (ALL (R,S) : set ws. R ⊆ PDERIVS R0 ∧ S ⊆ PDERIVS S0 ∧ (R,S) \∉ set bs)
let ?m1 = %bs. Pow(PDERIVS R0) × Pow(PDERIVS S0) – set bs
let ?m2 = %ws bs. card(?m1 bs)

have Inv0: ?Inv (([R0,S0]), []) by simp
{ fix s assume test s ?Inv s
obtain ws bs where [simp]: s = (ws,bs) by fastforce
from test s obtain R S ws' where [simp]: ws = (R,S)#ws'
by(auto split: prod.splits list.splits)
let ?bs' = (R,S) # bs
let ?succs = map (λa. (Pderiv a R, Pderiv a S)) as
let ?new = filter (λp. p \∉ set bs ∪ set ws) ?succs
let ?ws' = remdups ?new @ ws'

have s: ?Inv (step as s)
proof –
from ?Inv s have distinct ?ws' by auto
have ALL (R,S) : set ?ws'. R ⊆ PDERIVS R0 ∧ S ⊆ PDERIVS S0 ∧ (R,S) \∉ set ?bs'
using ?Inv s
by(simp add: Ball-def image-iff) (metis Pderiv-PDERIVS)
with ⟨distinct ?ws'⟩ show ?thesis by(simp)
**qed**

have ?m2(step as s) < ?m2 s
**proof** –
have fin: finite(?m1 bs)

45
proof

have \(?m2\) (step as s) < \(?m2\) s using \(?Inv\) s psubset-card_mono[OF \(?finite\){\(?m1\) bs}]
  apply (simp split: prod.split_asm)
  by (metis Diff iff Pow iff SigmaI fin card_gt_0_iff diff_Suc_less emptyE)
then show \(?thesis\) using \(?Inv\) s by simp
qed

note \(*\) and this

show \(\exists p.\) while-option test (step as) \(((\{R0, S0\}), [])\) = Some p
  by (rule measure-while-option-Some [where \(P = \?Inv\) and \(f = \?m2\), \(OF - Inv0\)]) (simp add: step)
qed

theorem closure-Some-Inv: assumes closure as \(((\{r\},\{s\}),[])\) \(= Some\) p
shows \(\forall (R,S)\in set(fst p).\) \(\exists w.\) \(R = pderivs w r \land S = pderivs w s\) (is \(?Inv\) p)
proof
  from assms have 1: while-option test (step as) \(((\{r\},\{s\}),[])\) \(= Some\) p
    by (simp add: closure-def)
  have Inv0: \(?Inv\) \(((\{r\},\{s\}),[])\) by simp (metis pderivs.simps(1))
  { fix p assume \(?Inv\) p test p
    obtain ws bs where [simp]: \(p = (ws,bs)\) by fastforce
      from (test p) obtain Rs ws where [simp]: \(ws = (R,S)\#ws'\)
        by (auto split: prod.splits list.splits)
      let \(?suc\) = map (\(\lambda a. (Pderiv a R, Pderiv a S)\)) as
      let \(?new\) = filter (\(\lambda p. p \notin set bs \cup set ws\) ?suc\)
      let \(?ws'\) = rendups \(?new\) \(?ws'\)
    from \(?Inv\) p obtain w where [simp]: \(R = pderivs w r S = pderivs w s\)
      by auto
    { fix x assume x : set as
      have EX w. Pderiv x R = pderivs w r \land Pderiv x S = pderivs w s
        by (rule_tac \(x=x@[x]\) in ezI) simp add: pderivs-append Pderiv-def 
    }
    with \(?Inv\) p have \(?Inv\) (step as p) by auto
  }
  note Inv-step = this
  show \(?thesis\)
    apply (rule while-option-rule[OF - 1])
    apply (erule (1) Inv-step)
    apply (rule Inv0)
  done
qed

lemma closure-complete: assumes lang r = lang s
shows EX bs. closure as \(((\{r\},\{s\}),[])\) \(= Some([],bs)\) (is \(?C\))
proof (rule contr)
  assume \(\sim \?C\)
  then obtain Rs ws bs
    where \(\chi\): closure as \(((\{r\},\{s\}),[])\) \(= Some((R,S)\#ws,bs)\)
    using closure-Some[of \{r\} \{s\}, simplified]

46
by (metis (opaque-lifting, no-types) list.exhaust prod.exhaust)
from assms closure-Some-Inv[OF this]
   while-option-stop[OF cl[unfolded closure-def]]
show False by auto
qed

corollary completeness: lang r = lang s ==> check-eqv r s
by(auto simp add: check-eqv-def dest!: closure-complete
               split: option.split list.split)

end

10 Extended Regular Expressions

theory Regular-Exp2
imports Regular-Set
begin

datatype (atoms: 'a) rexp =
is-Zero: Zero |
is-One: One |
Atom 'a |
Plus ('a rexp) ('a rexp) |
Times ('a rexp) ('a rexp) |
Star ('a rexp) |
Not ('a rexp) |
Inter ('a rexp) ('a rexp)

context
fixes S :: 'a set
begin

primrec lang :: 'a rexp => 'a lang where
lang Zero = {} |
lang One = {[[]]} |
lang (Atom a) = {[a]} |
lang (Plus r s) = (lang r) Un (lang s) |
lang (Times r s) = conc (lang r) (lang s) |
lang (Star r) = star(lang r) |
lang (Not r) = lists S - lang r |
lang (Inter r s) = (lang r Int lang s)

end

lemma lang-subset-lists: atoms r ⊆ S ==> lang S r ⊆ lists S
by(induction r)(auto simp: conc-subset-lists star-subset-lists)

primrec nullable :: 'a rexp => bool where
nullable Zero = False |
nullable One = True |
nullable (Atom c) = False |
nullable (Plus r1 r2) = (nullable r1 ∨ nullable r2) |
nullable (Times r1 r2) = (nullable r1 ∧ nullable r2) |
nullable (Star r) = True |
nullable (Not r) = (∼ (nullable r)) |
nullable (Inter r s) = (nullable r ∧ nullable s)

lemma nullable-iff: nullable r ←→ [] ∈ lang S r
by (induct r) (auto simp add: conc-def split: if-splits)
end

11 Deciding Equivalence of Extended Regular Expressions

theory Equivalence-Checking2
imports Regular-Exp2 HOL−Library. While-Combinator
begin

11.1 Term ordering

fun le-rexp :: nat rexp ⇒ nat rexp ⇒ bool
where
le-rexp Zero - = True
| le-rexp - Zero = False
| le-rexp One - = True
| le-rexp - One = False
| le-rexp (Atom a) (Atom b) = (a <= b)
| le-rexp (Atom -) - = True
| le-rexp - (Atom -) = False
| le-rexp (Star r) (Star s) = le-rexp r s
| le-rexp (Star -) - = True
| le-rexp - (Star -) = False
| le-rexp (Not r) (Not s) = le-rexp r s
| le-rexp (Not -) - = True
| le-rexp - (Not -) = False
| le-rexp (Plus r r') (Plus s s') =
  (if r = s then le-rexp r' s' else le-rexp r s)
| le-rexp (Plus -) - = True
| le-rexp - (Plus -) = False
| le-rexp (Times r r') (Times s s') =
  (if r = s then le-rexp r' s' else le-rexp r s)
| le-rexp (Times -) - = True
| le-rexp - (Times -) = False
| le-rexp (Inter r r') (Inter s s') =
  (if r = s then le-rexp r' s' else le-rexp r s)
11.2 Normalizing operations

associativity, commutativity, idempotence, zero

fun nPlus :: nat rexp ⇒ nat rexp ⇒ nat rexp
where
  nPlus Zero r = r
  nPlus r Zero = r
  nPlus (Plus r s) t = nPlus r (nPlus s t)
  nPlus r (Plus s t) =
    if r = s then (Plus s t)
    else if le-rexp r s then Plus r (Plus s t)
    else Plus s (nPlus r t)
  nPlus r s =
    if r = s then r
    else if le-rexp r s then Plus r s
    else Plus s r

lemma lang-nPlus[simp]: lang S (nPlus r s) = lang S (Plus r s)
by (induct r s rule: nPlus.induct) auto

associativity, zero, one

fun nTimes :: nat rexp ⇒ nat rexp ⇒ nat rexp
where
  nTimes Zero - = Zero
  nTimes - Zero = Zero
  nTimes One r = r
  nTimes r One = r
  nTimes (Times r s) t = Times r (nTimes s t)
  nTimes r s = Times r s

lemma lang-nTimes[simp]: lang S (nTimes r s) = lang S (Times r s)
by (induct r s rule: nTimes.induct) (auto simp: conc-assoc)

more optimisations:

fun nInter :: nat rexp ⇒ nat rexp ⇒ nat rexp
where
  nInter Zero - = Zero
  nInter - Zero = Zero
  nInter r s = Inter r s

lemma lang-nInter[simp]: lang S (nInter r s) = lang S (Inter r s)
by (induct r s rule: nInter.induct) (auto)

primrec norm :: nat rexp ⇒ nat rexp
where
  norm Zero = Zero
  norm One = One
  norm (Atom a) = Atom a
  norm (Plus r s) = nPlus (norm r) (norm s)
norm (Times r s) = aTimes (norm r) (norm s)
norm (Star r) = Star (norm r)
norm (Not r) = Not (norm r)
norm (Inter r1 r2) = nInter (norm r1) (norm r2)

lemma lang-norm[simp]: lang S (norm r) = lang S r
by (induct r) auto

11.3 Derivative

primrec nderiv :: nat ⇒ nat rexp ⇒ nat rexp
where
  nderiv - Zero = Zero
  nderiv - One = Zero
  nderiv a (Atom b) = (if a = b then One else Zero)
  nderiv a (Plus r s) = nPlus (nderiv a r) (nderiv a s)
  nderiv a (Times r s) = (let r’ s = nTimes (nderiv a r) s
    in if nullable r then nPlus r’ s (nderiv a s) else r’ s)
  nderiv a (Star r) = nTimes (nderiv a r) (Star r)
  nderiv a (Not r) = Not (nderiv a r)
  nderiv a (Inter r1 r2) = nInter (nderiv a r1) (nderiv a r2)

lemma lang-nderiv: a:S ⇒ lang S (nderiv a r) = Deriv a (lang S r)
by (induct r) (auto simp: Let-def nullable-iff[where S=S])

lemma atoms-nPlus[simp]: atoms (nPlus r s) = atoms r ∪ atoms s
by (induct r s rule: nPlus.induct) auto

lemma atoms-nTimes: atoms (nTimes r s) ⊆ atoms r ∪ atoms s
by (induct r s rule: nTimes.induct) auto

lemma atoms-nInter: atoms (nInter r s) ⊆ atoms r ∪ atoms s
by (induct r s rule: nInter.induct) auto

lemma atoms-norm: atoms (norm r) ⊆ atoms r
by (induct r) (auto dest!:subsetD[OF atoms-nTimes]subsetD[OF atoms-nInter])

lemma atoms-nderiv: atoms (nderiv a r) ⊆ atoms r
by (induct r) (auto simp: Let-def dest!:subsetD[OF atoms-nTimes]subsetD[OF atoms-nInter])

11.4 Bisimulation between languages and regular expressions

context
fixes S :: 'a set
begin

coinductive bisimilar :: 'a lang ⇒ 'a lang ⇒ bool where
K ⊆ lists S ⇒ L ⊆ lists S
⇒ ([] ∈ K ⇔ [] ∈ L)
\[ \Rightarrow (\forall x. x\in S \Rightarrow \text{bisimilar} \ (\text{Deriv } x \ K) \ (\text{Deriv } x \ L)) \]
\[ \Rightarrow \text{bisimilar } K \ L \]

**Lemma** equal-if-bisimilar:
**Assumes** \( K \subseteq \text{lists } S \ \wedge \ L \subseteq \text{lists } S \) \( \text{bisimilar} \ K \ L \) **Shows** \( K = L \)

**Proof** (rule set-eqI)

fix \( w \)
from assms show \( w \in K \iff w \in L \)
proof (induction \( w \) arbitrary: \( K \ L \))
  case Nil  thus ?case by (auto elim: bisimilar_cases)
next
  case (Cons \( a \) \( w \) \( K \ L \))
  show ?case
  proof
    cases
    assume \( a \in S \)
    with \( \text{bisimilar } K \ L \) have \( \text{bisimilar } (\text{Deriv } a \ K) \ (\text{Deriv } a \ L) \)
    by (auto elim: bisimilar_cases)
    then have \( w \in \text{Deriv } a \ K \iff w \in \text{Deriv } a \ L \)
    by (metis Cons.IH bisimilar_cases)
    thus ?case by (auto simp: Deriv-def)
  next
    assume \( a \notin S \)
    thus ?case using Cons.prems by auto
  qed
  qed

**Lemma** language-coinduct:
**Fixes** \( R \) (infixl \( \sim \) 50)
**Assumes** \( \bigwedge K \ L. \ K \sim L \Rightarrow K \subseteq \text{lists } S \ \wedge \ L \subseteq \text{lists } S \)
**Assumes** \( K \sim L \)
**Assumes** \( \bigwedge K \ L \ x. \ K \sim L \Rightarrow (\ [\in K \iff \ ] \in L) \)
**Assumes** \( \bigwedge K \ L \ x. \ K \sim L \Rightarrow x : S \Rightarrow \text{Deriv } x \ K \sim \text{Deriv } x \ L \)
**Shows** \( K = L \)
apply (rule equal-if-bisimilar)
apply (metis assms(1) assms(2))
apply (metis assms(1) assms(2))
apply (rule bisimilar.coinduct[of \( R \) \( OF \ \{K \sim L\}\)])
apply (auto simp: assms)
done

end

type-synonym rexp-pair = nat rexp * nat rexp

type-synonym rexp-pairs = rexp-pair list

definition is-bisimulation :: nat list \( \Rightarrow \) rexp-pairs \( \Rightarrow \) bool
where
is-bisimulation as \( ps = \)
∀ (r, s) ∈ set ps. (atoms r ∪ atoms s ⊆ set as) ∧ (nullable r ↔ nullable s) ∧
(∀ a ∈ set as. (nderiv a r, nderiv a s) ∈ set ps))

lemma bisim-lang-eq:
assumes bisim: is-bisimulation as ps
assumes (r, s) ∈ set ps
shows lang (set as) r = lang (set as) s
proof –
let ?R = λ K L. (∃ (r, s) ∈ set ps. K = lang (set as) r ∧ L = lang (set as) s)
show ?thesis
proof (rule language-coinduct [where R=?R and S=set as])
from ⟨(r, s) ∈ set ps⟩ show ?R (lang (set as) r) (lang (set as) s)
  by auto
next
fix K L assume ?R K L
then obtain r s where: (r, s) ∈ set ps
  and KL: K = lang (set as) r ∧ L = lang (set as) s by auto
with bisim have nullable r ↔ nullable s
  by (auto simp: is-bisimulation-def)
thus [] ∈ K ↔ [] ∈ L by (auto simp: nullable-iff [where S=set as] KL)
next case, but shared context
from bisim rs KL lang-subset-lists [of - set as]
show K ⊆ lists (set as) ∧ L ⊆ lists (set as)
  unfolding is-bisimulation-def by blast
next case, but shared context
fix a assume a ∈ set as
with rs bisim
have (nderiv a r, nderiv a s) ∈ set ps
  by (auto simp: is-bisimulation-def)
thus ?R (Deriv a K) (Deriv a L) using ⟨a ∈ set as⟩
  by (force simp: KL lang-nderiv)
qed
qed

11.5 Closure computation

fun test :: rexp-pairs * rexp-pairs ⇒ bool
where test (ws, ps) = (case ws of [] ⇒ False | (p,q)#. ⇒ nullable p = nullable q)

fun step :: nat list ⇒ rexp-pairs * rexp-pairs ⇒ rexp-pairs * rexp-pairs
where step as (us,ps) =
  (let
    (r, s) = hd ws;
    ps′ = (r, s) # ps;
    succs = map (λa. (nderiv a r, nderiv a s)) as;
    new = filter (λp. p /∈ set ps′ ∪ set ws) succs
    )
\[\text{in (new @ tl ws, ps')}\]

**Definition** closure ::

\[\text{nat list} \Rightarrow \text{rexp-pairs} \ast \text{rexp-pairs} \Rightarrow (\text{rexp-pairs} \ast \text{rexp-pairs}) \text{ option where}\]

\[\text{closure as} = \text{while-option test (step as)}\]

**Definition** pre-bisim :: \(\text{nat list} \Rightarrow \text{nat rexp} \Rightarrow \text{nat rexp} \Rightarrow \text{rexp-pairs} \ast \text{rexp-pairs} \Rightarrow \text{bool}\)

where

\[\text{pre-bisim as r s} = (\lambda (ws, ps). \left(\forall (r, s) \in \text{set ws} \cup \text{set ps}. \text{atoms r} \cup \text{atoms s} \subseteq \text{set as}\right) \land \left(\forall (r, s) \in \text{set ps}. \text{nullable r} \leftarrow\rightarrow \text{nullable s}\right) \land \left(\forall a \in \text{set as}. \text{nderiv a r, nderiv a s} \in \text{set ps} \cup \text{set ws}\right))\]

**Theorem** closure-sound:

assumes result: closure as ([(r, s)], []) = Some([], ps)

and atoms: \(\text{atoms r} \cup \text{atoms s} \subseteq \text{set as}\)

shows lang (set as) r = lang (set as) s

proof –

\{ fix st have pre-bisim as r s st \implies test st \implies pre-bisim as r s (step as st)

unfolding pre-bisim-def

by (cases st) (auto simp: split-def split: list.splits prod.splits dest!: subsetD[OF atoms-nderiv]) \}

moreover

from atoms

have pre-bisim as r s ([(r, s)], []) by (simp add: pre-bisim-def)

ultimately have pre-bisim-ps: pre-bisim as r s ([], ps)

by (rule while-option-rule[OF - result[unfolded closure-def]])

then have is-bisimulation as ps (r, s) \in \text{set ps}

by (auto simp: pre-bisim-def is-bisimulation-def)

thus lang (set as) r = lang (set as) s by (rule bisim-lang-eq)

qed

11.6 The overall procedure

**Primitive** add-atoms :: \(\text{nat rexp} \Rightarrow \text{nat list} \Rightarrow \text{nat list}\)

where

\[\text{add-atoms Zero} = \text{id}\]

\[\text{add-atoms One} = \text{id}\]

\[\text{add-atoms (Atom a)} = \text{List.insert a}\]

\[\text{add-atoms (Plus r s)} = \text{add-atoms s o add-atoms r}\]

\[\text{add-atoms (Times r s)} = \text{add-atoms s o add-atoms r}\]

\[\text{add-atoms (Not r)} = \text{add-atoms r}\]

\[\text{add-atoms (Inter r s)} = \text{add-atoms s o add-atoms r}\]

\[\text{add-atoms (Star r)} = \text{add-atoms r}\]

**Lemma** set-add-atoms: \(\text{set (add-atoms r as)} = \text{atoms r} \cup \text{set as}\)
by (induct r arbitrary: as) auto

definition check-eqv :: nat list ⇒ nat rexp ⇒ nat rexp ⇒ bool
where
check-eqv as r s ⇐⇒ set(add-atoms r (add-atoms s [])) ⊆ set as ∧
(case closure as ([(norm r, norm s)], []) of
  Some([],-) ⇒ Truc | - ⇒ False
)

lemma soundness:
assumes check-eqv as r s shows lang (set as) r = lang (set as) s
proof –
obtain ps where cl: closure as ([(norm r,norm s)],[]) = Some([],ps)
  and at: atoms r ∪ atoms s ⊆ set as
using assms
by (auto simp: check-eqv-def set-add-atoms split:option.splits list.splits)
hence atoms(norm r) ∪ atoms(norm s) ⊆ set as
using atoms-norm by blast
hence lang (set as) (norm r) = lang (set as) (norm s)
by (rule closure-sound[OF cl])
thus lang (set as) r = lang (set as) s by simp
qed

lemma check-eqv [0] (Plus One (Times (Atom 0) (Star(Atom 0)))) (Star(Atom 0))
by eval

lemma check-eqv [0,1] (Not(Atom 0))
(Plus One (Times (Plus (Atom 1) (Times (Atom 0) (Plus (Atom 0) (Atom 1))))
  (Star(Plus (Atom 0) (Atom 1)))))
by eval

lemma check-eqv [0] (Atom 0) (Inter (Star (Atom 0)) (Atom 0))
by eval

end

References

