Regular Sets, Expressions, Derivatives and Relation Algebra

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

May 26, 2024

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

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 @@ $(B \cup C) = A$ @@ $B \cup A$ @@ C
and $(A \cup B)$ @@ $C = A$ @@ $C \cup B$ @@ C
by *auto*

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

lemma *conc-subset-lists*: $A \subseteq$ *lists* $S \Longrightarrow B \subseteq$ *lists* $S \Longrightarrow A$ @@ $B \subseteq$ *lists* S
by(*fastforce simp*: *conc-def in-lists-conv-set*)

lemma *Nil-in-conc*[*simp*]: $[] \in A$ @@ $B \longleftrightarrow [] \in A \wedge [] \in B$
by (*metis append-is-Nil-conv concE concI*)

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

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

lemma *concI-if-Nil2*: $[] \in B \Longrightarrow xs : A \Longrightarrow xs \in A$ @@ B
by (*metis append-Nil2 concI*)

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

lemma *singleton-in-conc*:
$[x] : A$ @@ $B \longleftrightarrow [x] : A \wedge [] : B \vee [] : A \wedge [x] : B$
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*: $A \frown\frown (n + m) = A \frown\frown n$ @@ $A \frown\frown m$
by (*induct n*) (*auto simp*: *conc-assoc*)

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

lemma *lang-pow-empty-Suc*[*simp*]: $(\{\}::'a\ lang) \frown\frown Suc\ n = \{\}$
by (*simp add*: *lang-pow-empty*)

lemma *conc-pow-comm*:
shows A @@ $(A \frown\frown n) = (A \frown\frown n)$ @@ A
by (*induct n*) (*simp-all add*: *conc-assoc*[*symmetric*])

lemma *length-lang-pow-ub*:
$\forall w \in A.\ length\ w \leq k \Longrightarrow w : A \frown\frown n \Longrightarrow length\ w \leq k*n$
by(*induct n arbitrary*: *w*) (*fastforce simp*: *conc-def*)+

lemma *length-lang-pow-lb*:
$\forall w \in A.\ length\ w \geq k \Longrightarrow w : A \frown\frown n \Longrightarrow length\ w \geq k*n$
by(*induct n arbitrary*: *w*) (*fastforce simp*: *conc-def*)+

lemma *lang-pow-subset-lists*: $A \subseteq lists\ S \Longrightarrow A \frown\frown n \subseteq lists\ S$
by(*induct n*)(*auto simp*: *conc-subset-lists*)

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

1.3 *star*

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

lemma *star-if-lang-pow*[*simp*]: $w : A \frown\frown n \Longrightarrow w : star\ A$
by (*auto simp*: *star-def*)

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

lemma *star-if-lang*[*simp*]: **assumes** $w : A$ **shows** $w : star\ A$
proof (*rule star-if-lang-pow*)
show $w : A \frown\frown 1$ **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** $u : A \frown\frown m$ **by** (*auto simp*: *star-def*)
moreover
from ‹$v : star\ A$› **obtain** *n* **where** $v : A \frown\frown n$ **by** (*auto simp*: *star-def*)
ultimately have $u@v : A \frown\frown (m+n)$ **by** (*simp add*: *lang-pow-add*)
thus *?thesis* **by** *simp*
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 ⟹ v : *star* A ⟹ P v ⟹ P (u@v)
shows P w
proof −
{ fix n **have** w : A ^^ n ⟹ 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 ∪ {[]} (**is** *?L* = *?R*)
proof
show *?L* ⊆ *?R* **by** (*rule*, *erule star-induct*) *auto*
qed *auto*

lemma *concat-in-star*: *set ws* ⊆ A ⟹ *concat ws* : *star* A
by (*induct ws*) *simp-all*

lemma *in-star-iff-concat*:
w ∈ *star* A = (∃ *ws*. *set ws* ⊆ A ∧ w = *concat ws*)
(**is** - = (∃ *ws*. *?R* w *ws*))
proof
assume w : *star* A **thus** ∃ *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* ⊆ A ∧ v = *concat ws* **by** *blast*
with *append* **have** *?R* (u@v) (u#*ws*) **by** *auto*
thus *?case* **..**
qed
next

assume $\exists$ *us. ?R w us* **thus** *w* : *star A*
by (*auto simp*: *concat-in-star*)
qed

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

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

lemma *star-unfold-left-Nil*: *star A* = (*A* − {[]}) @@ (*star A*) ∪ {[]}
by (*metis insert-Diff-single star-insert-eps star-unfold-left*)

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

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

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

1.4 Left-Quotients of languages

definition *Deriv* :: *'a* ⇒ *'a lang* ⇒ *'a lang*
where *Deriv x A* = { *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-union* **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 *Derivs* [] *A* = *A*
and *Derivs* (*c* # *s*) *A* = *Derivs s* (*Deriv c A*)
and *Derivs* (*s1* @ *s2*) *A* = *Derivs s2* (*Derivs s1 A*)
unfolding *Derivs-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 *Derivs-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* (λ*xs*. *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*]:
$A \parallel \{[]\} = A$
$\{[]\} \parallel B = B$
unfolding *Shuffle-def* **by** *auto*

1.6 Arden's Lemma

lemma *arden-helper*:
assumes *eq*: $X = A$ @@ $X \cup B$
shows $X = (A \frown\frown Suc\ n)$ @@ $X \cup (\bigcup m{\leq}n.\ (A \frown\frown m)$ @@ $B)$
proof (*induct n*)
case *0*
show $X = (A \frown\frown Suc\ 0)$ @@ $X \cup (\bigcup m{\leq}0.\ (A \frown\frown m)$ @@ $B)$
using *eq* **by** *simp*
next
case (*Suc n*)
have *ih*: $X = (A \frown\frown Suc\ n)$ @@ $X \cup (\bigcup m{\leq}n.\ (A \frown\frown m)$ @@ $B)$ **by** *fact*
also have $\ldots = (A \frown\frown Suc\ n)$ @@ $(A$ @@ $X \cup B) \cup (\bigcup m{\leq}n.\ (A \frown\frown m)$ @@ $B)$
using *eq* **by** *simp*
also have $\ldots = (A \frown\frown Suc\ (Suc\ n))$ @@ $X \cup ((A \frown\frown Suc\ n)$ @@ $B) \cup (\bigcup m{\leq}n.$
$(A \frown\frown m)$ @@ $B)$
by (*simp add*: *conc-Un-distrib conc-assoc*[*symmetric*] *conc-pow-comm*)
also have $\ldots = (A \frown\frown Suc\ (Suc\ n))$ @@ $X \cup (\bigcup m{\leq}Suc\ n.\ (A \frown\frown m)$ @@ $B)$
by (*auto simp add*: *atMost-Suc*)
finally show $X = (A \frown\frown Suc\ (Suc\ n))$ @@ $X \cup (\bigcup m{\leq}Suc\ n.\ (A \frown\frown m)$ @@ $B)$
.
qed

lemma *Arden*:
assumes $[] \notin A$
shows $X = A$ @@ $X \cup B \longleftrightarrow X =$ *star* A @@ B
proof
assume *eq*: $X = A$ @@ $X \cup B$
{ fix w **assume** $w : X$
let *?n* = *size w*
from ‹$[] \notin A$› **have** $\forall\, u \in A.$ *length* $u \geq 1$
by (*metis Suc-eq-plus1 add-leD2 le-0-eq length-0-conv not-less-eq-eq*)
hence $\forall\, u \in A \frown\frown (?n{+}1).$ *length* $u \geq$ *?n+1*
by (*metis length-lang-pow-lb nat-mult-1*)
hence $\forall\, u \in A \frown\frown (?n{+}1) @@ X.$ *length* $u \geq$ *?n+1*
by(*auto simp only*: *conc-def length-append*)
hence $w \notin A \frown\frown (?n{+}1) @@ X$ **by** *auto*
hence w : *star* A @@ B **using** ‹$w : X$› **using** *arden-helper*[*OF eq*, **where** *n=?n*]
by (*auto simp add*: *star-def conc-UNION-distrib*)
} moreover

```
  { 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 ∪ B) @@ (A ^^ Suc n) ∪ (⋃ m≤n. B @@ (A ^^ m))
using eq by simp
  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)) ∪ (⋃ m≤Suc n. B @@ (A ^^ m))
    by (auto simp add: atMost-Suc)
  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)
```

hence $\forall u \in A\widehat{\ }\widehat{\ }(?n+1).\ length\ u \geq ?n+1$
by (*metis length-lang-pow-lb nat-mult-1*)
hence $\forall u \in X\ @@\ A\widehat{\ }\widehat{\ }(?n+1).\ length\ u \geq ?n+1$
by(*auto simp only*: *conc-def length-append*)
hence $w \notin X\ @@\ A\widehat{\ }\widehat{\ }(?n+1)$ **by** *auto*
hence $w : B\ @@\ star\ A$ **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** $w : B\ @@\ star\ A$
hence $\exists n.\ w \in B\ @@\ A\widehat{\ }\widehat{\ }n$ **by** (*auto simp*: *conc-def star-def*)
hence $w : X$ **using** *reversed-arden-helper*[*OF eq*] **by** *blast*
} ultimately show $X = B\ @@\ star\ A$ **by** *blast*
next
assume *eq*: $X = B\ @@\ star\ A$
have $star\ A = \{[]\} \cup star\ A\ @@\ A$
unfolding *conc-star-comm*[*symmetric*]
by(*metis Un-commute star-unfold-left*)
then have $B\ @@\ star\ A = B\ @@\ (\{[]\} \cup star\ A\ @@\ A)$
by *metis*
also have $\ldots = B \cup B\ @@\ (star\ A\ @@\ A)$
unfolding *conc-Un-distrib* **by** *simp*
also have $\ldots = B \cup (B\ @@\ star\ A)\ @@\ A$
by (*simp only*: *conc-assoc*)
finally show $X = X\ @@\ A \cup B$
using *eq* **by** *blast*
qed

end

2 Regular expressions

theory *Regular-Exp*
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*)

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*)

abbreviation (*input*) *regular-lang* **where** *regular-lang A* ≡ (∃ *r*. *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* ⟷ [] ∈ *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* ⟷ *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*[*simp*]: *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* : *lang r* ⟹ *set w* ⊆ *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* ∈ *lists*(*atoms r* ∪ *atoms s*). *w* ∈ *lang r* ⟷ *w* ∈ *lang s*)
by (*auto simp*: *atoms-lang*[*unfolded subset-iff*])

lemma *lang-eq-ext-Nil-fold-Deriv*:
fixes *r s*

defines $\mathfrak{B} \equiv \{(fold\ Deriv\ w\ (lang\ r),\ fold\ Deriv\ w\ (lang\ s))|\ w.\ w{\in}lists\ (atoms\ r \cup atoms\ s)\}$
shows $lang\ r = lang\ s \longleftrightarrow (\forall (K,\ L) \in \mathfrak{B}.\ [] \in K \longleftrightarrow [] \in L)$
unfolding *lang-eq-ext* $\mathfrak{B}$*-def* **by** (*subst* (*1 2*) *in-fold-Deriv*[*of* [], *simplified*, *symmetric*]) *auto*

2.1 Term ordering

instantiation *rexp* :: (*order*) {*order*}
begin

fun *le-rexp* :: ('*a*::*order*) *rexp* ⇒ ('*a*::*order*) *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* (*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*)

definition *less-eq-rexp* **where** $r \leq s \equiv$ *le-rexp r s*
definition *less-rexp* **where** $r < s \equiv$ *le-rexp r s* $\wedge\ r \neq s$

lemma *le-rexp-Zero*: *le-rexp r Zero* ⟹ *r = Zero*
by (*induction r*) *auto*

lemma *le-rexp-refl*: *le-rexp r r*
by (*induction r*) *auto*

lemma *le-rexp-antisym*: ⟦*le-rexp r s*; *le-rexp s r*⟧ ⟹ *r = s*
by (*induction r s rule*: *le-rexp.induct*) (*auto dest*: *le-rexp-Zero*)

lemma *le-rexp-trans*: ⟦*le-rexp r s*; *le-rexp s t*⟧ ⟹ *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 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*: $\bigwedge$*t. le-rexp r s* $\Longrightarrow$ *le-rexp s t* $\Longrightarrow$ *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 $\bigwedge$*t. r1 = s1* $\Longrightarrow$ *le-rexp r2 s2* $\Longrightarrow$ *le-rexp s2 t* $\Longrightarrow$ *le-rexp r2 t*
$\bigwedge$*t. r1* $\neq$ *s1* $\Longrightarrow$ *le-rexp r1 s1* $\Longrightarrow$ *le-rexp s1 t* $\Longrightarrow$ *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 $\bigwedge$*t. r1 = s1* $\Longrightarrow$ *le-rexp r2 s2* $\Longrightarrow$ *le-rexp s2 t* $\Longrightarrow$ *le-rexp r2 t*
$\bigwedge$*t. r1* $\neq$ *s1* $\Longrightarrow$ *le-rexp r1 s1* $\Longrightarrow$ *le-rexp s1 t* $\Longrightarrow$ *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*)

| *nTimes r s = Times r s*

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

primrec *norm* :: *'a*::*order rexp* ⇒ *'a 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*) = *nTimes* (*norm r*) (*norm s*)
| *norm* (*Star r*) = *Star* (*norm r*)

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

primrec *nderiv* :: *'a*::*order* ⇒ *'a rexp* ⇒ *'a 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*)

lemma *lang-nderiv*: *lang* (*nderiv a r*) = *Deriv a* (*lang r*)
by (*induction r*) (*auto simp*: *Let-def nullable-iff*)

lemma *deriv-no-occurrence*:
  x ∉ *atoms r* ⟹ *nderiv x r* = *Zero*
by (*induction r*) *auto*

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

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

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

lemma *atoms-nderiv*: *atoms* (*nderiv a r*) ⊆ *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*
type-synonym *'a rexp-pairs* = *'a rexp-pair list*

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 $(r, s) \in ps$
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 $?R = \lambda K\ L.\ (\exists (r,s) \in ps'.\ K = lang\ r \wedge L = lang\ s)$
  show *?thesis*
  proof (*rule language-coinduct*[**where** $R=?R$])
    from ‹$(r, s) \in ps$›
    have $(r, s) \in ps'$ **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*: $(r, s) \in ps'$
      and *KL*: *K* = *lang r L* = *lang s* **by** *auto*
    with $bisim'$ **have** *nullable r* ⟷ *nullable s*
      by (*auto simp*: *is-bisimulation-def*)
    thus $[] \in K \longleftrightarrow [] \in L$ **by** (*auto simp*: *nullable-iff KL*)
    fix *a*
    show *?R* (*Deriv a K*) (*Deriv a L*)
    proof *cases*
      assume $a \in set\ as$
      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 $a \notin set\ as$
      with $bisim'$ *rs*
      have $a \notin atoms\ r\ a \notin atoms\ s$ **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
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 ⇒
 '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)))

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
     with inv show ?thesis using atoms-nderiv[of - r] atoms-nderiv[of - s]
       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),(nderiv a r,nderiv a s))| r s a. a : A}^* =
      {((r,s),(nderivs r w,nderivs s w))| r s w. w : lists A} (is ?L = ?R)
proof−
  note [simp] = nderivs-def
  { fix r s r' s'
    have ((r,s),(r',s')) : ?L ⟹ ((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 ∀x∈set w. x ∈ A ⟹ ((r, s), nderivs r w, nderivs 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 ⟹ ((r,s),(r',s')) : ?L by auto
  } ultimately show ?thesis by (auto simp: in-lists-conv-set) blast
qed

lemma nullable-nderivs:
  nullable (foldl (%r a. 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 = atoms r ∪ atoms s
shows ws = [] ⟷ lang r = lang s
proof −
  have leq: (lang r = lang s) =
  (∀(r',s') ∈ {((r0,s0),(nderiv a r0,nderiv a s0))| r0 s0 a. a : set as}^* `` {(r,s)}.
    nullable r' = nullable s')
      by(simp add: atoms rtrancl-nderiv-nderivs Ball-def lang-eq-ext imp-ex nullable-nderivs
          del:Un-iff)
  have {(x,y). y ∈ set ((λ(p,q). map (λa. (nderiv a p, nderiv a q)) as) x)} =
    {((r,s), nderiv a r, nderiv a s) |r s a. a ∈ set as}
    by auto
  with atoms 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 ⇒ 'a list ⇒ '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* ∪ *set as*
by (*induct r arbitrary*: *as*) *auto*

definition *check-eqv* :: *nat rexp* ⇒ *nat rexp* ⇒ *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([],-) ⇒ *True* | - ⇒ *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* ⇒ (′*b* ∗ ′*b*) *set*) ⇒ ′*a rexp* ⇒ (′*b* ∗ ′*b*) *set*
where
rel v Zero = {} |
rel v One = *Id* |
rel v (*Atom a*) = *v a* |

```
  rel v (Plus r s) = rel v r ∪ rel v s |
  rel v (Times r s) = rel v r O rel v s |
  rel v (Star r) = (rel v r)^*

primrec word-rel :: ('a ⇒ ('b * 'b) set) ⇒ 'a list ⇒ ('b * 'b) set
where
  word-rel v [] = Id
| word-rel v (a#as) = v a O word-rel v as

lemma word-rel-append:
  word-rel v w O word-rel v w' = word-rel v (w @ w')
by (rule sym) (induct w, auto)

lemma rel-word-rel: rel v r = (⋃w∈lang r. word-rel v w)
proof (induct r)
  case Times thus ?case
    by (auto simp: rel-def word-rel-append conc-def relcomp-UNION-distrib rel-
comp-UNION-distrib2)
next
  case (Star r)
  { fix n
    have (rel v r) ^^ n = (⋃w ∈ lang r ^^ n. word-rel v w)
    proof (induct n)
      case 0 show ?case by simp
    next
      case (Suc n) thus ?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:
 lang r = lang s ⟹ rel v r = 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* $\Rightarrow$ *nat rexp* $\Rightarrow$ $('a * 'a)$ *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*)$\widehat{}*$

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* $\Longrightarrow$ *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* $\Rightarrow$ *bool* $\Rightarrow$ *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 eresolve-tac ctxt* @{*thms rev-subsetD*})
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*

THEN CONVERSION (*HOLogic.Trueprop-conv* (*regexp-conv ctxt'*)) *1*
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 $(r \cup s\hat{}+)\hat{}* = (r \cup s)\hat{}*$
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* ‘ (*star A*) = *star* (*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* ≠ {}
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* (*insert xs A*) = *set* (*suffixes xs*) ∪ *Suffixes A*
by (*simp add*: *Suffixes-altdef*)

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

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

lemma *Suffixes-UNION* [*simp*]: *Suffixes* (⋃ (*f* ‘ *A*)) = ⋃ ((λ*x*. *Suffixes* (*f x*)) ‘ *A*)
by (*auto simp*: *Suffixes-def*)

lemma *Suffixes-compower*:
assumes *A* ≠ {}
shows *Suffixes* (*A* ^^ *n*) = *insert* [] (*Suffixes A* @@ (⋃ *k*<*n*. *A* ^^ *k*))

proof (*induction n*)
case (*Suc n*)
from *Suc* **have** *Suffixes* (*A* ⁀⁀ *Suc n*) =
insert [] (*Suffixes A* @@ ((⋃*k*<*n*. *A* ⁀⁀ *k*) ∪ *A* ⁀⁀ *n*))
by (*simp-all add*: *assms conc-Un-distrib*)
also have (⋃*k*<*n*. *A* ⁀⁀ *k*) ∪ *A* ⁀⁀ *n* = (⋃*k*∈*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*)))
using *assms* **by** (*subst Suffixes-compower*) *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*
using *assms* **by** *auto*
finally show *?thesis* **.**
qed

Prefix language

definition *Prefixes* :: *'a list set* ⇒ *'a list set* **where**
Prefixes A = {*w*. ∃ *q*. *w* @ *q* ∈ *A*}

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

lemma *Nil-in-Prefixes-iff* [*simp*]: [] ∈ *Prefixes A* ⟷ *A* ≠ {}
by (*auto simp*: *Prefixes-def*)

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

lemma *Prefixes-empty-iff* [*simp*]: *Prefixes A* = {} ⟷ *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*) ∪ *Prefixes A*
by (*simp add*: *Prefixes-altdef*)

lemma *Prefixes-conc* [*simp*]: $B \neq \{\} \Longrightarrow$ *Prefixes* $(A$ @@ $B) =$ *Prefixes* $A \cup (A$ @@ *Prefixes* $B)$
unfolding *Prefixes-altdef conc-def* **by** (*force simp*: *prefix-append*)

lemma *Prefixes-union* [*simp*]: *Prefixes* $(A \cup B) =$ *Prefixes* $A \cup$ *Prefixes* B
by (*auto simp*: *Prefixes-def*)

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

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

lemma *Suffixes-rev*: *Suffixes* $(rev \text{ ` } A) = rev \text{ ` }$ *Prefixes* A
by (*auto simp*: *Prefixes-altdef suffixes-rev Suffixes-altdef*)

lemma *Prefixes-compower*:
assumes $A \neq \{\}$
shows *Prefixes* $(A \frown n) =$ *insert* [] $((\bigcup k{<}n.\ A \frown k)$ @@ *Prefixes* $A)$
proof −
have $A \frown n = rev \text{ ` } ((rev \text{ ` } A) \frown n)$ **by** (*simp add*: *image-image*)
also have *Prefixes* ... = *insert* [] $((\bigcup k{<}n.\ A \frown k)$ @@ *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 *Prefixes* $(star\ A) =$ *star* A @@ *Prefixes* A
proof −
have *star* $A = rev \text{ ` } star\ (rev \text{ ` } A)$ **by** (*simp add*: *image-image*)
also have *Prefixes* ... = *star* A @@ *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* ⇒ *'a list set* **where**
Sublists $A = \{w.\ \exists\, q{\in}A.$ *sublist* $w\ q\}$

lemma *Sublists-altdef* [*code*]: *Sublists* $A = (\bigcup w{\in}A.$ *set* $(sublists\ w))$

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*)) = ⋃((λ*x*. *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*)) = ⋃((λ*x*. *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 ∗(*3*) **obtain** *xs1 xs2* **where** *xs* = *xs1* @ *xs2 subseq xs1 ys subseq xs2 zs*
by (*rule subseq-appendE*)
with ∗(*1*,*2*) **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*

| *rexp-rev* (*Atom x*) = *Atom x*
| *rexp-rev* (*Plus r s*) = *Plus* (*rexp-rev r*) (*rexp-rev s*)
| *rexp-rev* (*Times r s*) = *Times* (*rexp-rev s*) (*rexp-rev r*)
| *rexp-rev* (*Star r*) = *Star* (*rexp-rev r*)

lemma *lang-rexp-rev* [*simp*]: *lang* (*rexp-rev r*) = *rev* ‘ *lang r*
by (*induction r*) (*simp-all add*: *image-Un*)

The obvious construction for a singleton-language regular expression:

fun *rexp-of-word* **where**
rexp-of-word [] = *One*
| *rexp-of-word* [*x*] = *Atom x*
| *rexp-of-word* (*x*#*xs*) = *Times* (*Atom x*) (*rexp-of-word xs*)

lemma *lang-rexp-of-word* [*simp*]: *lang* (*rexp-of-word xs*) = {*xs*}
by (*induction xs rule*: *rexp-of-word.induct*) (*simp-all add*: *conc-def*)

lemma *size-rexp-of-word* [*simp*]: *size* (*rexp-of-word xs*) = *Suc* (*2* ∗ (*length xs* − *1*))
by (*induction xs rule*: *rexp-of-word.induct*) *auto*

Character substitution in a regular expression:

primrec *rexp-subst* **where**
rexp-subst f Zero = *Zero*
| *rexp-subst f One* = *One*
| *rexp-subst f* (*Atom x*) = *rexp-of-word* (*f x*)
| *rexp-subst f* (*Plus r s*) = *Plus* (*rexp-subst f r*) (*rexp-subst f s*)
| *rexp-subst f* (*Times r s*) = *Times* (*rexp-subst f r*) (*rexp-subst f s*)
| *rexp-subst f* (*Star r*) = *Star* (*rexp-subst f r*)

lemma *lang-rexp-subst*: *lang* (*rexp-subst f r*) = *subst-word f* ‘ *lang r*
by (*induction r*) (*simp-all add*: *image-Un*)

Suffix language of a regular expression:

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

theorem *lang-suffix-rexp* [*simp*]:
lang (*suffix-rexp r*) = *Suffixes* (*lang r*)
by (*induction r*) (*auto simp*: *rexp-empty-iff*)

Prefix language of a regular expression:

primrec *prefix-rexp* :: *'a rexp* ⇒ *'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
```

imports *Regular-Exp*
begin

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
Timess rs r ≡ (⋃ *r'* ∈ *rs*. {*Times r' r*})

lemma *Timess-eq-image*:
Timess rs r = (λ*r'*. *Times r' r*) ‘ *rs*
by *auto*

primrec
pderiv :: *'a* ⇒ *'a rexp* ⇒ *'a rexp set*
where
pderiv c Zero = {}
| *pderiv c One* = {}
| *pderiv c* (*Atom c'*) = (*if c* = *c' then* {*One*} *else* {})
| *pderiv c* (*Plus r1 r2*) = (*pderiv c r1*) ∪ (*pderiv c r2*)
| *pderiv c* (*Times r1 r2*) =
(*if nullable r1 then Timess* (*pderiv c r1*) *r2* ∪ *pderiv c r2 else Timess* (*pderiv c r1*) *r2*)
| *pderiv c* (*Star r*) = *Timess* (*pderiv c r*) (*Star r*)

primrec
pderivs :: *'a list* ⇒ *'a rexp* ⇒ (*'a rexp*) *set*
where
pderivs [] *r* = {*r*}
| *pderivs* (*c* # *s*) *r* = ⋃ (*pderivs s* ‘ *pderiv c r*)

abbreviation
pderiv-set :: *'a* ⇒ *'a rexp set* ⇒ *'a rexp set*
where
pderiv-set c rs ≡ ⋃ (*pderiv c* ‘ *rs*)

abbreviation
pderivs-set :: *'a list* ⇒ *'a rexp set* ⇒ *'a rexp set*
where
pderivs-set s rs ≡ ⋃ (*pderivs s* ‘ *rs*)

lemma *pderivs-append*:
pderivs (*s1* @ *s2*) *r* = ⋃ (*pderivs s2* ‘ *pderivs s1 r*)
by (*induct s1 arbitrary*: *r*) (*simp-all*)

lemma *pderivs-snoc*:
shows *pderivs* (*s* @ [*c*]) *r* = *pderiv-set c* (*pderivs s r*)
by (*simp add*: *pderivs-append*)

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

lemma *pderivs-Atom*:

shows *pderivs s* (*Atom c*) ⊆ {*Atom c*, *One*}
by (*induct s*) (*simp-all*)

8.3 Relating left-quotients and partial derivatives

lemma *Deriv-pderiv*:
shows *Deriv c* (*lang r*) = ⋃ (*lang ' pderiv c r*)
by (*induct r*) (*auto simp add*: *nullable-iff conc-UNION-distrib*)

lemma *Derivs-pderivs*:
shows *Derivs s* (*lang r*) = ⋃ (*lang ' pderivs s r*)
proof (*induct s arbitrary*: *r*)
case (*Cons c s*)
have *ih*: ⋀*r*. *Derivs s* (*lang r*) = ⋃ (*lang ' pderivs s r*) **by** *fact*
have *Derivs* (*c # s*) (*lang r*) = *Derivs s* (*Deriv c* (*lang r*)) **by** *simp*
also have ... = *Derivs s* (⋃ (*lang ' pderiv c r*)) **by** (*simp add*: *Deriv-pderiv*)
also have ... = *Derivss s* (*lang ' pderiv c r*))
by (*auto simp add*: *Derivs-def*)
also have ... = ⋃ (*lang ' pderivs-set s* (*pderiv c r*)))
using *ih* **by** *auto*
also have ... = ⋃ (*lang ' pderivs* (*c # s*) *r*)) **by** *simp*
finally show *Derivs* (*c # s*) (*lang r*) = ⋃ (*lang ' pderivs* (*c # s*) *r*) **.**
qed (*simp add*: *Derivs-def*)

8.4 Relating derivatives and partial derivatives

lemma *deriv-pderiv*:
shows ⋃ (*lang ' pderiv c r*)) = *lang* (*deriv c r*)
unfolding *lang-deriv Deriv-pderiv* **by** *simp*

lemma *derivs-pderivs*:
shows ⋃ (*lang ' pderivs s r*)) = *lang* (*derivs s r*)
unfolding *lang-derivs Derivs-pderivs* **by** *simp*

8.5 Finiteness property of partial derivatives

definition
pderivs-lang :: *'a lang* ⇒ *'a rexp* ⇒ *'a rexp set*
where
pderivs-lang A r ≡ ⋃*x* ∈ *A*. *pderivs x r*

lemma *pderivs-lang-subsetI*:
assumes ⋀*s*. *s* ∈ *A* ⟹ *pderivs s r* ⊆ *C*
shows *pderivs-lang A r* ⊆ *C*
using *assms* **unfolding** *pderivs-lang-def* **by** (*rule UN-least*)

lemma *pderivs-lang-union*:
shows *pderivs-lang* (*A* ∪ *B*) *r* = (*pderivs-lang A r* ∪ *pderivs-lang B r*)
by (*simp add*: *pderivs-lang-def*)

lemma *pderivs-lang-subset*:
shows $A \subseteq B \Longrightarrow$ *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 $\equiv \{v.\ v \neq [] \wedge (\exists u.\ u \mathbin{@} v = s)\}$

lemma *PSuf-snoc*:
shows *PSuf* $(s \mathbin{@} [c]) =$ (*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*. $f\ (v \mathbin{@} [c]))$
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 (*simp add*: *PSuf-Union pderivs-snoc*)

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

lemma *pderivs-lang-Times-aux1*:
assumes *a*: *s* ∈ *UNIV1*
shows *pderivs-lang* (*PSuf s*) *r* ⊆ *pderivs-lang UNIV1 r*
using *a* **unfolding** *UNIV1-def PSuf-def pderivs-lang-def* **by** *auto*

lemma *pderivs-lang-Times-aux2*:
assumes *a*: *s* ∈ *UNIV1*
shows *Timess* (*pderivs s r1*) *r2* ⊆ *Timess* (*pderivs-lang UNIV1 r1*) *r2*
using *a* **unfolding** *pderivs-lang-def* **by** *auto*

lemma *pderivs-lang-Times*:

shows *pderivs-lang UNIV1* (*Times r1 r2*) ⊆ *Timess* (*pderivs-lang UNIV1 r1*) *r2* ∪ *pderivs-lang UNIV1 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* ≠ []
shows *pderivs s* (*Star r*) ⊆ *Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*)
using *a*
proof (*induct s rule*: *rev-induct*)
case (*snoc c s*)
have *ih*: *s* ≠ [] ⟹ *pderivs s* (*Star r*) ⊆ *Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*) **by** *fact*
{ assume *asm*: *s* ≠ []
have *pderivs* (*s* @ [*c*]) (*Star r*) = *pderiv-set c* (*pderivs s* (*Star r*)) **by** (*simp add*: *pderivs-snoc*)
also have ... ⊆ *pderiv-set c* (*Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*))
using *ih*[*OF asm*] **by** *fast*
also have ... ⊆ *Timess* (*pderiv-set c* (*pderivs-lang* (*PSuf s*) *r*)) (*Star r*) ∪ *pderiv c* (*Star r*)
by (*auto split*: *if-splits*)
also have ... ⊆ *Timess* (*pderivs-lang* (*PSuf* (*s* @ [*c*])) *r*) (*Star r*) ∪ (*Timess* (*pderiv c r*) (*Star r*))
by (*simp only*: *PSuf-snoc pderivs-lang-snoc pderivs-lang-union*)
(*auto simp add*: *pderivs-lang-def*)
also have ... = *Timess* (*pderivs-lang* (*PSuf* (*s* @ [*c*])) *r*) (*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*) ⊆ *Timess* (*pderivs-lang UNIV1 r*) (*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*)

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*)

lemma *finite-pderivs-lang*:
  shows *finite* (*pderivs-lang A r*)
by (*metis finite-pderivs-lang-UNIV pderivs-lang-subset rev-finite-subset subset-UNIV*)

The following relationship between the alphabetic width of regular expressions (called *awidth* below) and the number of partial derivatives was proved by Antimirov [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* ∪

pderivs-lang UNIV1 r2) **by** *simp*
also have ... ≤ *card* (*pderivs-lang UNIV1 r1*) + *card* (*pderivs-lang UNIV1 r2*)
by(*simp add*: *card-Un-le*)
also have ... ≤ *awidth* (*Plus r1 r2*) **using** *Plus.IH* **by** *simp*
finally show *?case* **.**
next
case (*Times r1 r2*)
have *card* (*pderivs-lang UNIV1* (*Times r1 r2*)) ≤ *card* (*Timess* (*pderivs-lang UNIV1 r1*) *r2* ∪ *pderivs-lang UNIV1 r2*)
by (*simp add*: *card-mono finite-pderivs-lang pderivs-lang-Times*)
also have ... ≤ *card* (*Timess* (*pderivs-lang UNIV1 r1*) *r2*) + *card* (*pderivs-lang UNIV1 r2*)
by (*simp add*: *card-Un-le*)
also have ... ≤ *card* (*pderivs-lang UNIV1 r1*) + *card* (*pderivs-lang UNIV1 r2*)
by (*simp add*: *card-Timess-pderivs-lang-le*)
also have ... ≤ *awidth* (*Times r1 r2*) **using** *Times.IH* **by** *simp*
finally show *?case* **.**
next
case (*Star r*)
have *card* (*pderivs-lang UNIV1* (*Star r*)) ≤ *card* (*Timess* (*pderivs-lang UNIV1 r*) (*Star r*))
by (*simp add*: *card-mono finite-pderivs-lang pderivs-lang-Star*)
also have ... ≤ *card* (*pderivs-lang UNIV1 r*) **by** (*rule card-Timess-pderivs-lang-le*)
also have ... ≤ *awidth* (*Star r*) **by** (*simp add*: *Star.IH*)
finally show *?case* **.**
qed (*auto*)

Antimirov's Theorem 3.4:

theorem *card-pderivs-lang-UNIV-le-awidth*: *card* (*pderivs-lang UNIV r*) ≤ *awidth r + 1*
proof −
have *card* (*insert r* (*pderivs-lang UNIV1 r*)) ≤ *Suc* (*card* (*pderivs-lang UNIV1 r*))
by(*auto simp*: *card-insert-if*[*OF finite-pderivs-lang-UNIV1*])
also have ... ≤ *Suc* (*awidth r*) **by**(*simp add*: *card-pderivs-lang-UNIV1-le-awidth*)
finally show *?thesis* **by**(*simp add*: *pderivs-lang-UNIV*)
qed

Antimirov's Corollary 3.5:

corollary *card-pderivs-lang-le-awidth*: *card* (*pderivs-lang A r*) ≤ *awidth r + 1*
by(*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)

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*
type-synonym *'a Rexp-pairs = 'a Rexp-pair list*

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) ∧

$(\forall\, a{\in}set\ as.\ (Pderiv\ a\ R,\ Pderiv\ a\ S) \in set\ ps))$

lemma *Bisim-Lang-eq*:
assumes *Bisim*: *is-Bisimulation as ps*
assumes $(R,\ S) \in set\ ps$
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*)
  let $?R = \lambda K\ L.\ (\exists\,(R,S){\in}set\ ps'.\ K = Lang\ R \wedge L = Lang\ S)$
  show *?thesis*
  proof (*rule language-coinduct*[**where** $R{=}?R$])
    from ‹$(R,S) \in set\ ps$›
    have $(R,S) \in set\ ps'$ **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*: $(R,\ S) \in set\ ps'$
      and *KL*: $K = Lang\ R\ L = Lang\ S$ **by** *auto*
    with *Bisim'* **have** $Nullable\ R \longleftrightarrow Nullable\ S$
      by (*auto simp*: *is-Bisimulation-def*)
    thus $[] \in K \longleftrightarrow [] \in L$
      by (*auto simp*: *nullable-iff KL Nullable-def Lang-def*)
    fix *a*
    show $?R\ (Deriv\ a\ K)\ (Deriv\ a\ L)$
    proof *cases*
      assume $a \in set\ as$
      with *rs Bisim'*
      have $(Pderiv\ a\ R,\ Pderiv\ a\ S) \in set\ ps'$
        by (*auto simp*: *is-Bisimulation-def*)
      thus *?thesis* **by** (*fastforce simp*: *KL Deriv-Lang*)
    next
      assume $a \notin set\ as$
      with *Bisim' rs*
      have $a \notin Atoms\ R \cup Atoms\ S$
        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

fun *test* :: *'a Rexp-pairs* ∗ *'a Rexp-pairs* ⇒ *bool* **where**

```
test (ws, ps) = (case ws of [] ⇒  False | (R,S)#- ⇒ Nullable R = Nullable S)

fun step :: 'a list ⇒
  'a Rexp-pairs * 'a Rexp-pairs ⇒ 'a Rexp-pairs * 'a Rexp-pairs
where step as (ws,ps) =
    (let
      (R,S) = hd ws;
      ps' = (R,S) # ps;
      succs = map (λa. (Pderiv a R, Pderiv a S)) as;
      new = filter (λp. p ∉ set ps ∪ set ws) succs
    in (remdups new @ tl ws, ps'))

definition closure ::
  'a list ⇒ 'a Rexp-pairs * 'a Rexp-pairs
   ⇒ ('a Rexp-pairs * 'a Rexp-pairs) option where
closure as = while-option test (step as)

definition pre-Bisim :: 'a list ⇒ 'a rexp set ⇒ 'a rexp set ⇒
 'a Rexp-pairs * 'a Rexp-pairs ⇒ bool
where
pre-Bisim as R S = (λ(ws,ps).
 ((R,S) ∈ set ws ∪ set ps) ∧
 (∀(R,S)∈ set ws ∪ set ps. Atoms R ∪ Atoms S ⊆ set as) ∧
 (∀(R,S)∈ set ps. (Nullable R ⟷ Nullable S) ∧
   (∀a∈set as. (Pderiv a R, Pderiv a S) ∈ set ps ∪ set ws)))

lemma step-set-eq: ⟦ test (ws,ps); step as (ws,ps) = (ws',ps') ⟧
  ⟹ set ws' ∪ set ps' =
     set ws ∪ set ps
     ∪ (⋃a∈set as. {(Pderiv a (fst(hd ws)), Pderiv a (snd(hd ws)))})
by(auto split: list.splits)

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

```
as)) @ wstl
      by auto
    have ∀(R',S')∈set wstl ∪ set ps'. Atoms R' ∪ Atoms S' ⊆ set as
      using prems(4) by auto
    moreover
    have ∀a∈set as. Atoms(Pderiv a R) ∪ Atoms(Pderiv a S) ⊆ 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
 }
 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) ∈ set ps
  by (auto simp: pre-Bisim-def is-Bisimulation-def)
 thus Lang R = Lang S by (rule Bisim-Lang-eq)
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 = add-atoms r (add-atoms s [])
 obtain ps where 1: closure ?as ([({r},{s})],[]) = 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 w* **where** *s* : *R r'* : *pderivs w s*
  by(*auto simp*: *PDERIVS-def pderivs-lang-def*)
 hence *r* ∈ *pderivs* (*w* @ [*a*]) *s* **using** ‹*r* : *pderiv a r'*› **by**(*auto simp add*:*pderivs-snoc*)
 thus *r* : *PDERIVS R* **using** ‹*s* : *R*› **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* (λ*a*. (*Pderiv a R*, *Pderiv a S*)) *as*
  let *?new* = *filter* (λ*p*. *p* ∉ *set bs* ∪ *set ws*) *?succs*
  let *?ws'* = *remdups ?new* @ *ws'*
  have ∗: *?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*)

```
    by(metis assms finite-Diff finite-PDERIVS finite-cartesian-product finite-Pow-iff)
     have ?m2(step as s) < ?m2 s using ‹?Inv s› psubset-card-mono[OF ‹fi-
nite(?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
  } note step = this
  show ∃ 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 ∀(R,S)∈set(fst p). ∃ w. R = pderivs w r ∧ 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 R S ws′ where [simp]: ws = (R,S)#ws′
     by(auto split: prod.splits list.splits)
   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′
   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 ∧ Pderiv x S = pderivs w s
       by(rule-tac x=w@[x] in exI)(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 ccontr)
  assume ¬ ?C
  then obtain R S ws bs
   where cl: closure as ([({r},{s})],[]) = Some((R,S)#ws,bs)
   using closure-Some[of {r} {s}, simplified]
```

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*) = *nTimes* (*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*)

$\Longrightarrow (\bigwedge x.\ x{:}S \Longrightarrow bisimilar\ (Deriv\ x\ K)\ (Deriv\ x\ L))$
$\Longrightarrow bisimilar\ K\ L$

lemma *equal-if-bisimilar*:
assumes $K \subseteq lists\ S\ L \subseteq lists\ S\ bisimilar\ K\ L$ **shows** $K = L$
proof (*rule set-eqI*)
  fix w
  from *assms* **show** $w \in K \longleftrightarrow w \in L$
  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 $a : S$
      with ‹*bisimilar K L*› **have** *bisimilar* (*Deriv a K*) (*Deriv a L*)
        by (*auto elim*: *bisimilar.cases*)
      then have $w \in Deriv\ a\ K \longleftrightarrow w \in Deriv\ a\ L$
        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
qed

lemma *language-coinduct*:
fixes R (**infixl** $\sim$ *50*)
assumes $\bigwedge K\ L.\ K \sim L \Longrightarrow K \subseteq lists\ S \wedge L \subseteq lists\ S$
assumes $K \sim L$
assumes $\bigwedge K\ L.\ K \sim L \Longrightarrow ([] \in K \longleftrightarrow [] \in L)$
assumes $\bigwedge K\ L\ x.\ K \sim L \Longrightarrow x : S \Longrightarrow Deriv\ x\ K \sim Deriv\ x\ L$
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 =

$(\forall (r,s) \in set\ ps.\ (atoms\ r \cup atoms\ s \subseteq set\ as) \wedge (nullable\ r \longleftrightarrow nullable\ s) \wedge$
$(\forall a \in set\ as.\ (nderiv\ a\ r,\ nderiv\ a\ s) \in set\ ps))$

lemma *bisim-lang-eq*:
assumes *bisim*: *is-bisimulation as ps*
assumes $(r,\ s) \in set\ ps$
shows *lang* $(set\ as)\ r = lang\ (set\ as)\ s$
proof −
let *?R* $= \lambda K\ L.\ (\exists (r,s) \in set\ ps.\ K = lang\ (set\ as)\ r \wedge L = lang\ (set\ as)\ s)$
show *?thesis*
proof (*rule language-coinduct*[**where** *R*=*?R* **and** *S* = *set as*])
from ‹$(r,\ s) \in set\ ps$› **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** *rs*: $(r,\ s) \in set\ ps$
and *KL*: $K = lang\ (set\ as)\ r\ L = lang\ (set\ as)\ s$ **by** *auto*
with *bisim* **have** *nullable r* $\longleftrightarrow$ *nullable s*
by (*auto simp*: *is-bisimulation-def*)
thus $[] \in K \longleftrightarrow [] \in L$ **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 $K \subseteq lists\ (set\ as) \wedge L \subseteq lists\ (set\ as)$
unfolding *is-bisimulation-def* **by** *blast*

next case, but shared context

fix *a* **assume** $a \in set\ as$
with *rs bisim*
have $(nderiv\ a\ r,\ nderiv\ a\ s) \in set\ ps$
by (*auto simp*: *is-bisimulation-def*)
thus *?R* $(Deriv\ a\ K)\ (Deriv\ a\ L)$ **using** ‹$a \in set\ as$›
by (*force simp*: *KL lang-nderiv*)
qed
qed

11.5 Closure computation

fun *test* :: *rexp-pairs* $*$ *rexp-pairs* $\Rightarrow$ *bool*
where *test* $(ws,\ ps) = (case\ ws\ of\ [] \Rightarrow\ False \mid (p,q)\#\text{-} \Rightarrow nullable\ p = nullable\ q)$

fun *step* :: *nat list* $\Rightarrow$ *rexp-pairs* $*$ *rexp-pairs* $\Rightarrow$ *rexp-pairs* $*$ *rexp-pairs*
where *step as* $(ws,ps) =$
$(let$
$(r,\ s) = hd\ ws;$
$ps' = (r,\ s)\ \#\ ps;$
$succs = map\ (\lambda a.\ (nderiv\ a\ r,\ nderiv\ a\ s))\ as;$
$new = filter\ (\lambda p.\ p \notin set\ ps' \cup set\ ws)\ succs$

```
    in (new @ tl ws, ps'))

definition closure ::
  nat list ⇒ rexp-pairs ∗ rexp-pairs
   ⇒ (rexp-pairs ∗ rexp-pairs) option where
closure as = while-option test (step as)

definition pre-bisim :: nat list ⇒ nat rexp ⇒ nat rexp ⇒
 rexp-pairs ∗ rexp-pairs ⇒ bool
where
pre-bisim as r s = (λ(ws,ps).
 ((r, s) ∈ set ws ∪ set ps) ∧
 (∀(r,s)∈ set ws ∪ set ps. atoms r ∪ atoms s ⊆ set as) ∧
 (∀(r,s)∈ set ps. (nullable r ⟷ nullable s) ∧
   (∀a∈set as. (nderiv a r, nderiv a s) ∈ set ps ∪ set ws)))

theorem closure-sound:
assumes result: closure as ([(r,s)],[]) = Some([],ps)
and atoms: atoms r ∪ atoms s ⊆ set as
shows lang (set as) r = lang (set as) s
proof−
  { fix st have pre-bisim as r s st ⟹ test st ⟹ 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) ∈ 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

```
primrec add-atoms :: nat rexp ⇒ nat list ⇒ nat 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 (Not r) = add-atoms r
| add-atoms (Inter 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 ∪ 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([],-) ⇒ *True* | - ⇒ *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

[1] V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.

[2] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11:481–494, 1964.

[3] A. Krauss and T. Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, 49:95–106, 2012. published online March 2011.