

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	2
1.1	$(@@)$	3
1.2	A^n	4
1.3	<i>star</i>	5
1.4	Left-Quotients of languages	6
1.5	Shuffle product	8
1.6	Arden's Lemma	8
2	Regular expressions	9
2.1	Term ordering	10
3	Normalizing Derivative	11
3.1	Normalizing operations	11

4	Deciding Regular Expression Equivalence	13
4.1	Bisimulation between languages and regular expressions . . .	13
4.2	Closure computation	14
4.3	Bisimulation-free proof of closure computation	14
4.4	The overall procedure	15
5	Regular Expressions as Homogeneous Binary Relations	15
6	Proving Relation (In)equalities via Regular Expressions	16
7	Basic constructions on regular expressions	17
7.1	Reverse language	17
7.2	Substituting characters in a language	17
7.3	Subword language	20
7.4	Fragment language	21
7.5	Various regular expression constructions	22
8	Derivatives of regular expressions	24
8.1	Brzozowski's derivatives of regular expressions	24
8.2	Antimirov's partial derivatives	25
8.3	Relating left-quotients and partial derivatives	26
8.4	Relating derivatives and partial derivatives	26
8.5	Finiteness property of partial derivatives	26
9	Deciding Regular Expression Equivalence (2)	29
9.1	Closure computation	30
9.2	The overall procedure	31
9.3	Termination and Completeness	31
10	Extended Regular Expressions	32
11	Deciding Equivalence of Extended Regular Expressions	33
11.1	Term ordering	33
11.2	Normalizing operations	34
11.3	Derivative	35
11.4	Bisimulation between languages and regular expressions . . .	36
11.5	Closure computation	36
11.6	The overall procedure	37

1 Regular sets

```
theory Regular-Set
imports Main
begin
```

type-synonym 'a lang = 'a list set

definition conc :: 'a lang \Rightarrow 'a lang \Rightarrow 'a lang (**infixr** @@ 75) **where**
 $A @@ B = \{xs@ys \mid xs\ ys. xs:A \ \& \ ys:B\}$

checks the code preprocessor for set comprehensions

export-code conc **checking** SML

overloading lang-pow == compow :: nat \Rightarrow 'a lang \Rightarrow 'a lang

begin

primrec lang-pow :: nat \Rightarrow 'a lang \Rightarrow 'a lang **where**

lang-pow 0 A = $\{\}\mid$

lang-pow (Suc n) A = A @@ (lang-pow n A)

end

for code generation

definition lang-pow :: nat \Rightarrow 'a lang \Rightarrow '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 = $\{\}\mid$

\langle proof \rangle

hide-const (open) lang-pow

definition star :: 'a lang \Rightarrow 'a lang **where**

star A = $(\bigcup n. A \overset{\sim}{\sim} n)$

1.1 (@@)

lemma concI[simp,intro]: $u : A \Longrightarrow v : B \Longrightarrow u@v : A @@ B$

\langle proof \rangle

lemma concE[elim]:

assumes $w \in A @@ B$

obtains $u\ v$ **where** $u \in A\ v \in B\ w = u@v$

\langle proof \rangle

lemma conc-mono: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A @@ B \subseteq C @@ D$

\langle proof \rangle

lemma conc-empty[simp]: **shows** $\{\} @@ A = \{\}$ **and** $A @@ \{\} = \{\}$

\langle proof \rangle

lemma conc-epsilon[simp]: **shows** $\{\}\mid @@ A = A$ **and** $A @@ \{\}\mid = A$

\langle proof \rangle

lemma conc-assoc: $(A @@ B) @@ C = A @@ (B @@ C)$

\langle proof \rangle

lemma *conc-Un-distrib*:

shows $A \text{ @@ } (B \cup C) = A \text{ @@ } B \cup A \text{ @@ } C$

and $(A \cup B) \text{ @@ } C = A \text{ @@ } C \cup B \text{ @@ } C$

<proof>

lemma *conc-UNION-distrib*:

shows $A \text{ @@ } \bigcup (M \text{ ' } I) = \bigcup ((\%i. A \text{ @@ } M \ i) \text{ ' } I)$

and $\bigcup (M \text{ ' } I) \text{ @@ } A = \bigcup ((\%i. M \ i \text{ @@ } A) \text{ ' } I)$

<proof>

lemma *conc-subset-lists*: $A \subseteq \text{lists } S \implies B \subseteq \text{lists } S \implies A \text{ @@ } B \subseteq \text{lists } S$

<proof>

lemma *Nil-in-conc[simp]*: $[] \in A \text{ @@ } B \longleftrightarrow [] \in A \wedge [] \in B$

<proof>

lemma *concI-if-Nil1*: $[] \in A \implies xs : B \implies xs \in A \text{ @@ } B$

<proof>

lemma *conc-Diff-if-Nil1*: $[] \in A \implies A \text{ @@ } B = (A - \{[]\}) \text{ @@ } B \cup B$

<proof>

lemma *concI-if-Nil2*: $[] \in B \implies xs : A \implies xs \in A \text{ @@ } B$

<proof>

lemma *conc-Diff-if-Nil2*: $[] \in B \implies A \text{ @@ } B = A \text{ @@ } (B - \{[]\}) \cup A$

<proof>

lemma *singleton-in-conc*:

$[x] : A \text{ @@ } B \longleftrightarrow [x] : A \wedge [] : B \vee [] : A \wedge [x] : B$

<proof>

1.2 A^n

lemma *lang-pow-add*: $A \text{ ^^ } (n + m) = A \text{ ^^ } n \text{ @@ } A \text{ ^^ } m$

<proof>

lemma *lang-pow-empty*: $\{\} \text{ ^^ } n = (\text{if } n = 0 \text{ then } \{[]\} \text{ else } \{\})$

<proof>

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

<proof>

lemma *conc-pow-comm*:

shows $A \text{ @@ } (A \text{ ^^ } n) = (A \text{ ^^ } n) \text{ @@ } A$

<proof>

lemma *length-lang-pow-ub*:

$\forall w \in A. \text{length } w \leq k \implies w : A^{\sim n} \implies \text{length } w \leq k*n$
(proof)

lemma *length-lang-pow-lb*:

$\forall w \in A. \text{length } w \geq k \implies w : A^{\sim n} \implies \text{length } w \geq k*n$
(proof)

lemma *lang-pow-subset-lists*: $A \subseteq \text{lists } S \implies A^{\sim n} \subseteq \text{lists } S$
(proof)

lemma *empty-pow-add*:

assumes $[] \in A \ s \in A^{\sim n}$
shows $s \in A^{\sim (n + m)}$
(proof)

1.3 star

lemma *star-subset-lists*: $A \subseteq \text{lists } S \implies \text{star } A \subseteq \text{lists } S$
(proof)

lemma *star-if-lang-pow[simp]*: $w : A^{\sim n} \implies w : \text{star } A$
(proof)

lemma *Nil-in-star[iff]*: $[] : \text{star } A$
(proof)

lemma *star-if-lang[simp]*: **assumes** $w : A$ **shows** $w : \text{star } A$
(proof)

lemma *append-in-starI[simp]*:

assumes $u : \text{star } A$ **and** $v : \text{star } A$ **shows** $u@v : \text{star } A$
(proof)

lemma *conc-star-star*: $\text{star } A @@ \text{star } A = \text{star } A$
(proof)

lemma *conc-star-comm*:

shows $A @@ \text{star } A = \text{star } A @@ A$
(proof)

lemma *star-induct[consumes 1, case-names Nil append, induct set: star]*:

assumes $w : \text{star } A$

and $P []$

and *step*: $!!u v. u : A \implies v : \text{star } A \implies P v \implies P (u@v)$

shows $P w$

(proof)

lemma *star-empty[simp]*: $\text{star } \{\} = \{\}$

(proof)

lemma *star-epsilon[simp]*: $star \{\epsilon\} = \{\epsilon\}$
 ⟨proof⟩

lemma *star-idemp[simp]*: $star (star A) = star A$
 ⟨proof⟩

lemma *star-unfold-left*: $star A = A @@ star A \cup \{\epsilon\}$ (**is** $?L = ?R$)
 ⟨proof⟩

lemma *concat-in-star*: $set ws \subseteq A \implies concat ws : star A$
 ⟨proof⟩

lemma *in-star-iff-concat*:
 $w \in star A = (\exists ws. set ws \subseteq A \wedge w = concat ws)$
 (**is** $- = (\exists ws. ?R w ws)$)
 ⟨proof⟩

lemma *star-conv-concat*: $star A = \{concat ws \mid ws. set ws \subseteq A\}$
 ⟨proof⟩

lemma *star-insert-eps[simp]*: $star (insert \epsilon A) = star(A)$
 ⟨proof⟩

lemma *star-unfold-left-Nil*: $star A = (A - \{\epsilon\}) @@ (star A) \cup \{\epsilon\}$
 ⟨proof⟩

lemma *star-Diff-Nil-fold*: $(A - \{\epsilon\}) @@ star A = star A - \{\epsilon\}$
 ⟨proof⟩

lemma *star-decom*:
assumes $a: x \in star A \ x \neq \epsilon$
shows $\exists a b. x = a @ b \wedge a \neq \epsilon \wedge a \in A \wedge b \in star A$
 ⟨proof⟩

lemma *star-pow*:
assumes $s \in star A$
shows $\exists n. s \in A \overset{\sim}{\sim} n$
 ⟨proof⟩

1.4 Left-Quotients of languages

definition *Deriv* :: $'a \Rightarrow 'a \ lang \Rightarrow 'a \ lang$
where $Deriv x A = \{ xs. x \# xs \in A \}$

definition *Derivs* :: $'a \ list \Rightarrow 'a \ lang \Rightarrow 'a \ lang$
where $Derivs xs A = \{ ys. xs @ ys \in A \}$

abbreviation

$Derivs :: 'a list \Rightarrow 'a lang set \Rightarrow 'a lang$
where
 $Derivs s As \equiv \bigcup (Derivs s ' As)$

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]: $Deriv a (A \cup B) = Deriv a A \cup Deriv a B$
and *Deriv-inter*[simp]: $Deriv a (A \cap B) = Deriv a A \cap Deriv a B$
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]: $Deriv a (UN x:I. S x) = (UN x:I. Deriv a (S x))$
 $\langle proof \rangle$

lemma *Der-conc* [simp]:
shows $Deriv c (A @@ B) = (Deriv c A) @@ B \cup (if\ [] \in A\ then\ Deriv c B\ else\ \{\})$
 $\langle proof \rangle$

lemma *Deriv-star* [simp]:
shows $Deriv c (star A) = (Deriv c A) @@ star A$
 $\langle proof \rangle$

lemma *Deriv-diff*[simp]:
shows $Deriv c (A - B) = Deriv c A - Deriv c B$
 $\langle proof \rangle$

lemma *Deriv-lists*[simp]: $c : S \Longrightarrow Deriv c (lists S) = lists S$
 $\langle proof \rangle$

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

lemma *in-fold-Deriv*: $v \in fold Deriv w L \longleftrightarrow w @ v \in L$
 $\langle proof \rangle$

lemma *Derivs-alt-def* [code]: $Derivs w L = fold Deriv w L$
 $\langle proof \rangle$

lemma *Deriv-code* [code]:
 $Deriv x A = tl ' Set.filter (\lambda xs. case xs of x' \# - \Rightarrow x = x' \mid - \Rightarrow False) A$
 $\langle proof \rangle$

1.5 Shuffle product

definition *Shuffle* (infix \parallel 80) **where**

$$\text{Shuffle } A \ B = \bigcup \{ \text{shuffles } xs \ ys \mid xs \ ys. \ xs \in A \wedge \ ys \in B \}$$

lemma *Deriv-Shuffle*[simp]:

$$\text{Deriv } a \ (A \parallel B) = \text{Deriv } a \ A \parallel B \cup A \parallel \text{Deriv } a \ B$$

<proof>

lemma *shuffle-subset-lists*:

assumes $A \subseteq \text{lists } S \ B \subseteq \text{lists } S$

shows $A \parallel B \subseteq \text{lists } S$

<proof>

lemma *Nil-in-Shuffle*[simp]: $\ [] \in A \parallel B \longleftrightarrow \ [] \in A \wedge \ [] \in B$

<proof>

lemma *shuffle-Un-distrib*:

shows $A \parallel (B \cup C) = A \parallel B \cup A \parallel C$

and $A \parallel (B \cup C) = A \parallel B \cup A \parallel C$

<proof>

lemma *shuffle-UNION-distrib*:

shows $A \parallel \bigcup (M \ ' \ I) = \bigcup ((\%i. A \parallel M \ i) \ ' \ I)$

and $\bigcup (M \ ' \ I) \parallel A = \bigcup ((\%i. M \ i \parallel A) \ ' \ I)$

<proof>

lemma *Shuffle-empty*[simp]:

$$A \parallel \{\} = \{\}$$

$$\{\} \parallel B = \{\}$$

<proof>

lemma *Shuffle-eps*[simp]:

$$A \parallel \{\ [] \} = A$$

$$\{\ [] \} \parallel B = B$$

<proof>

1.6 Arden's Lemma

lemma *arden-helper*:

assumes $eq: X = A \ @\@ \ X \cup B$

shows $X = (A \ \sim \text{Suc } n) \ @\@ \ X \cup (\bigcup_{m \leq n}. (A \ \sim m) \ @\@ \ B)$

<proof>

lemma *Arden*:

assumes $\ [] \notin A$

shows $X = A \ @\@ \ X \cup B \longleftrightarrow X = \text{star } A \ @\@ \ B$

<proof>

lemma *reversed-arden-helper*:
assumes *eq*: $X = X \text{ @@ } A \cup B$
shows $X = X \text{ @@ } (A \text{ ~~~ } \text{Suc } n) \cup (\bigcup_{m \leq n}. B \text{ @@ } (A \text{ ~~~ } m))$
<proof>

theorem *reversed-Arden*:
assumes *nemp*: $\square \notin A$
shows $X = X \text{ @@ } A \cup B \longleftrightarrow X = B \text{ @@ } \text{star } A$
<proof>

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* $\equiv (\exists r. \text{lang } r = A)$

primrec *nullable* :: 'a *rexp* \Rightarrow *bool* **where**
nullable Zero = *False* |
nullable One = *True* |
nullable (Atom c) = *False* |
nullable (Plus r1 r2) = (*nullable r1* \vee *nullable r2*) |
nullable (Times r1 r2) = (*nullable r1* \wedge *nullable r2*) |
nullable (Star r) = *True*

lemma *nullable-iff* [*code-abbrev*]: *nullable r* $\longleftrightarrow \square \in \text{lang } r$
<proof>

primrec *rexp-empty* **where**
rexp-empty Zero \longleftrightarrow *True*

```

| rexp-empty One  $\longleftrightarrow$  False
| rexp-empty (Atom a)  $\longleftrightarrow$  False
| rexp-empty (Plus r s)  $\longleftrightarrow$  rexp-empty r  $\wedge$  rexp-empty s
| rexp-empty (Times r s)  $\longleftrightarrow$  rexp-empty r  $\vee$  rexp-empty s
| rexp-empty (Star r)  $\longleftrightarrow$  False

```

lemma *rexp-empty-iff* [code-abbrev]: $\text{rexp-empty } r \longleftrightarrow \text{lang } r = \{\}$
 <proof>

Composition on rhs usually complicates matters:

lemma *map-map-rexp*:
 $\text{map-rexp } f (\text{map-rexp } g r) = \text{map-rexp } (\lambda r. f (g r)) r$
 <proof>

lemma *map-rexp-ident*[simp]: $\text{map-rexp } (\lambda x. x) = (\lambda r. r)$
 <proof>

lemma *atoms-lang*: $w : \text{lang } r \implies \text{set } w \subseteq \text{atoms } r$
 <proof>

lemma *lang-eq-ext*: $(\text{lang } r = \text{lang } s) =$
 $(\forall w \in \text{lists}(\text{atoms } r \cup \text{atoms } s). w \in \text{lang } r \longleftrightarrow w \in \text{lang } s)$
 <proof>

lemma *lang-eq-ext-Nil-fold-Deriv*:

fixes $r s$
defines $\mathfrak{B} \equiv \{(\text{fold Deriv } w (\text{lang } r), \text{fold Deriv } w (\text{lang } s)) \mid w. w \in \text{lists } (\text{atoms } r \cup \text{atoms } s)\}$
shows $\text{lang } r = \text{lang } s \longleftrightarrow (\forall (K, L) \in \mathfrak{B}. [] \in K \longleftrightarrow [] \in L)$
 <proof>

2.1 Term ordering

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

fun *le-rexp* :: ('a::order) rexp \Rightarrow ('a::order) rexp \Rightarrow 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** $r \leq s \equiv \text{le-rexp } r \ s$

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

lemma *le-rexp-Zero*: $\text{le-rexp } r \ \text{Zero} \implies r = \text{Zero}$
 $\langle \text{proof} \rangle$

lemma *le-rexp-refl*: $\text{le-rexp } r \ r$
 $\langle \text{proof} \rangle$

lemma *le-rexp-antisym*: $\llbracket \text{le-rexp } r \ s; \text{le-rexp } s \ r \rrbracket \implies r = s$
 $\langle \text{proof} \rangle$

lemma *le-rexp-trans*: $\llbracket \text{le-rexp } r \ s; \text{le-rexp } s \ t \rrbracket \implies \text{le-rexp } r \ t$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

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

lemma *le-rexp-total*: $\text{le-rexp } (r :: 'a :: \text{linorder } \text{rexp}) \ s \vee \text{le-rexp } s \ r$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

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*)

⟨*proof*⟩

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*)

⟨*proof*⟩

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*

⟨*proof*⟩

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: $lang (nderiv a r) = Deriv a (lang r)$
 ⟨proof⟩

lemma deriv-no-occurrence:
 $x \notin atoms r \implies nderiv x r = Zero$
 ⟨proof⟩

lemma atoms-nPlus[simp]: $atoms (nPlus r s) = atoms r \cup atoms s$
 ⟨proof⟩

lemma atoms-nTimes: $atoms (nTimes r s) \subseteq atoms r \cup atoms s$
 ⟨proof⟩

lemma atoms-norm: $atoms (norm r) \subseteq atoms r$
 ⟨proof⟩

lemma atoms-nderiv: $atoms (nderiv a r) \subseteq atoms r$
 ⟨proof⟩

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 \Rightarrow 'a lang \Rightarrow bool **where**
 ($\square \in K \longleftrightarrow \square \in L$)
 $\implies (\bigwedge x. bisimilar (Deriv x K) (Deriv x L))$
 $\implies bisimilar K L$

lemma *equal-if-bisimilar:*
assumes *bisimilar K L* **shows** $K = L$
 ⟨proof⟩

lemma *language-coinduct:*
fixes R (**infixl** \sim 50)
assumes $K \sim L$
assumes $\bigwedge K L. K \sim L \implies (\square \in K \longleftrightarrow \square \in L)$
assumes $\bigwedge K L x. K \sim L \implies Deriv x K \sim Deriv x L$

shows $K = L$
 $\langle proof \rangle$

type-synonym $'a \text{ rexp-pair} = 'a \text{ rexp} * 'a \text{ rexp}$
type-synonym $'a \text{ rexp-pairs} = 'a \text{ rexp-pair list}$

definition $is\text{-bisimulation} :: 'a::order \text{ list} \Rightarrow 'a \text{ rexp-pair set} \Rightarrow bool$
where

$is\text{-bisimulation as } R =$
 $(\forall (r,s) \in R. (atoms\ r \cup atoms\ s \subseteq set\ as) \wedge (nullable\ r \longleftrightarrow nullable\ s) \wedge$
 $(\forall a \in set\ as. (nderiv\ a\ r, nderiv\ a\ s) \in R))$

lemma $bisim\text{-lang}\text{-eq}$:
assumes $bisim$: $is\text{-bisimulation as } ps$
assumes $(r, s) \in ps$
shows $lang\ r = lang\ s$
 $\langle proof \rangle$

4.2 Closure computation

definition $closure ::$
 $'a::order \text{ list} \Rightarrow 'a \text{ rexp-pair} \Rightarrow ('a \text{ rexp-pairs} * 'a \text{ rexp-pair set}) \text{ option}$

where
 $closure\ as = rtrancl\text{-while } (\% (r,s). nullable\ r = nullable\ s)$
 $(\% (r,s). map (\lambda a. (nderiv\ a\ r, nderiv\ a\ s)) as)$

definition $pre\text{-bisim} :: 'a::order \text{ list} \Rightarrow 'a \text{ rexp} \Rightarrow 'a \text{ rexp} \Rightarrow$
 $'a \text{ rexp-pairs} * 'a \text{ rexp-pair set} \Rightarrow bool$

where
 $pre\text{-bisim as } r\ s = (\lambda (ws,R).$
 $(r,s) \in R \wedge set\ ws \subseteq R \wedge$
 $(\forall (r,s) \in R. atoms\ r \cup atoms\ s \subseteq set\ as) \wedge$
 $(\forall (r,s) \in R - set\ ws. (nullable\ r \longleftrightarrow nullable\ s) \wedge$
 $(\forall a \in set\ as. (nderiv\ a\ r, nderiv\ a\ s) \in R))$

theorem $closure\text{-sound}$:
assumes $result$: $closure\ as (r,s) = Some([],R)$
and $atoms$: $atoms\ r \cup atoms\ s \subseteq set\ as$
shows $lang\ r = lang\ s$
 $\langle proof \rangle$

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 $rtrancl\text{-nderiv}\text{-nderivs}$: **defines** $nderivs == foldl (\% r\ a. nderiv\ a\ r)$
shows $\{((r,s),(nderiv\ a\ r, nderiv\ a\ s)) \mid r\ s\ a. a : A\}^{\hat{*}} =$
 $\{((r,s),(nderivs\ r\ w, nderivs\ s\ w)) \mid r\ s\ w. w : lists\ A\}$ (**is** $?L = ?R$)

<proof>

lemma *nullable-nderivs*:

nullable (foldl (%r a. nderiv a r) r w) = (w : lang r)

<proof>

theorem *closure-sound-complete*:

assumes *result*: *closure as (r,s) = Some(ws,R)*

and *atoms*: *set as = atoms r \cup atoms s*

shows *ws = [] \longleftrightarrow lang r = lang s*

<proof>

4.4 The overall procedure

primrec *add-atoms* :: *'a rexp \Rightarrow 'a list \Rightarrow 'a list*

where

add-atoms Zero = id

| *add-atoms One = id*

| *add-atoms (Atom a) = List.insert a*

| *add-atoms (Plus r s) = add-atoms s o add-atoms r*

| *add-atoms (Times r s) = add-atoms s o add-atoms r*

| *add-atoms (Star r) = add-atoms r*

lemma *set-add-atoms*: *set (add-atoms r as) = atoms r \cup set as*

<proof>

definition *check-equiv* :: *nat rexp \Rightarrow nat rexp \Rightarrow bool* **where**

check-equiv r s =

(let nr = norm r; ns = norm s; as = add-atoms nr (add-atoms ns [])

in case closure as (nr, ns) of

Some([],-) \Rightarrow True | - \Rightarrow False)

lemma *soundness*:

assumes *check-equiv r s* **shows** *lang r = lang s*

<proof>

Test:

lemma *check-equiv (Plus One (Times (Atom 0) (Star(Atom 0)))) (Star(Atom 0))*

<proof>

end

5 Regular Expressions as Homogeneous Binary Relations

theory *Relation-Interpretation*

imports *Regular-Exp*

begin

primrec $rel :: ('a \Rightarrow ('b * 'b) set) \Rightarrow 'a\ rexp \Rightarrow ('b * 'b) set$

where

$rel\ v\ Zero = \{\} \mid$
 $rel\ v\ One = Id \mid$
 $rel\ v\ (Atom\ a) = v\ a \mid$
 $rel\ v\ (Plus\ r\ s) = rel\ v\ r \cup rel\ v\ s \mid$
 $rel\ v\ (Times\ r\ s) = rel\ v\ r\ O\ rel\ v\ s \mid$
 $rel\ v\ (Star\ r) = (rel\ v\ r)^{\hat{*}}$

primrec $word-rel :: ('a \Rightarrow ('b * 'b) set) \Rightarrow 'a\ list \Rightarrow ('b * 'b) set$

where

$word-rel\ v\ [] = Id$
 $\mid\ word-rel\ v\ (a\#\ as) = v\ a\ O\ word-rel\ v\ as$

lemma $word-rel-append:$

$word-rel\ v\ w\ O\ word-rel\ v\ w' = word-rel\ v\ (w\ @\ w')$
 $\langle proof \rangle$

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

$\langle proof \rangle$

Soundness:

lemma $soundness:$

$lang\ r = lang\ s \implies rel\ v\ r = rel\ v\ s$
 $\langle proof \rangle$

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**

$rel-of-regexp\ vs\ Zero = \{\} \mid$
 $rel-of-regexp\ vs\ One = Id \mid$
 $rel-of-regexp\ vs\ (Atom\ i) = vs\ !\ i \mid$
 $rel-of-regexp\ vs\ (Plus\ r\ s) = rel-of-regexp\ vs\ r \cup rel-of-regexp\ vs\ s \mid$
 $rel-of-regexp\ vs\ (Times\ r\ s) = rel-of-regexp\ vs\ r\ O\ rel-of-regexp\ vs\ s \mid$
 $rel-of-regexp\ vs\ (Star\ r) = (rel-of-regexp\ vs\ r)^{\hat{*}}$

lemma $rel-of-regexp-rel: rel-of-regexp\ vs\ r = rel\ (\lambda i. vs\ !\ i)\ r$

$\langle proof \rangle$

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

lemma *rel-eqI*: *check-eqv* *r* *s* \implies *rel-eq* (*r*, *s*) *vs*
 <*proof*>

lemmas *regex-reify* = *rel-of-regex.simps* *rel-eq.simps*

lemmas *regex-unfold* = *trancl-unfold-left* *subset-Un-eq*

<*ML*>

hide-const (**open**) *le-regex* *nPlus* *nTimes* *norm* *nullable* *bisimilar* *is-bisimulation*
closure

pre-bisim *add-atoms* *check-eqv* *rel* *word-rel* *rel-eq*

Example:

lemma $(r \cup s^+)^* = (r \cup s)^*$
 <*proof*>

end

7 Basic constructions on regular expressions

theory *Regex-Constructions*

imports

Main

HOL-Library.Sublist

Regular-Exp

begin

7.1 Reverse language

lemma *rev-conc* [*simp*]: $\text{rev } ' (A \ @\@ B) = \text{rev } ' B \ @\@ \text{rev } ' A$
 <*proof*>

lemma *rev-compact* [*simp*]: $\text{rev } ' (A \ \sim n) = (\text{rev } ' A) \ \sim n$
 <*proof*>

lemma *rev-star* [*simp*]: $\text{rev } ' \text{star } A = \text{star } (\text{rev } ' A)$
 <*proof*>

7.2 Substituting characters in a language

definition *subst-word* :: $('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list}$ **where**
subst-word *f* *xs* = *concat* (*map* *f* *xs*)

lemma *subst-word-Nil* [*simp*]: *subst-word* *f* [] = []
 <*proof*>

lemma *subst-word-singleton* [simp]: $\text{subst-word } f [x] = f x$
 ⟨proof⟩

lemma *subst-word-append* [simp]: $\text{subst-word } f (xs @ ys) = \text{subst-word } f xs @ \text{subst-word } f ys$
 ⟨proof⟩

lemma *subst-word-Cons* [simp]: $\text{subst-word } f (x \# xs) = f x @ \text{subst-word } f xs$
 ⟨proof⟩

lemma *subst-word-conc* [simp]: $\text{subst-word } f ' (A @@@ B) = \text{subst-word } f ' A @@@ \text{subst-word } f ' B$
 ⟨proof⟩

lemma *subst-word-compower* [simp]: $\text{subst-word } f ' (A \overset{\sim}{\sim} n) = (\text{subst-word } f ' A) \overset{\sim}{\sim} n$
 ⟨proof⟩

lemma *subst-word-star* [simp]: $\text{subst-word } f ' (\text{star } A) = \text{star } (\text{subst-word } f ' A)$
 ⟨proof⟩

Suffix language

definition *Suffixes* :: 'a list set \Rightarrow 'a list set **where**
 $\text{Suffixes } A = \{w. \exists q. q @ w \in A\}$

lemma *Suffixes-altdef* [code]: $\text{Suffixes } A = (\bigcup w \in A. \text{set } (\text{suffixes } w))$
 ⟨proof⟩

lemma *Nil-in-Suffixes-iff* [simp]: $[] \in \text{Suffixes } A \longleftrightarrow A \neq \{\}$
 ⟨proof⟩

lemma *Suffixes-empty* [simp]: $\text{Suffixes } \{\} = \{\}$
 ⟨proof⟩

lemma *Suffixes-empty-iff* [simp]: $\text{Suffixes } A = \{\} \longleftrightarrow A = \{\}$
 ⟨proof⟩

lemma *Suffixes-singleton* [simp]: $\text{Suffixes } \{xs\} = \text{set } (\text{suffixes } xs)$
 ⟨proof⟩

lemma *Suffixes-insert*: $\text{Suffixes } (\text{insert } xs A) = \text{set } (\text{suffixes } xs) \cup \text{Suffixes } A$
 ⟨proof⟩

lemma *Suffixes-conc* [simp]: $A \neq \{\} \implies \text{Suffixes } (A @@@ B) = \text{Suffixes } B \cup (\text{Suffixes } A @@@ B)$
 ⟨proof⟩

lemma *Suffixes-union* [simp]: $\text{Suffixes } (A \cup B) = \text{Suffixes } A \cup \text{Suffixes } B$
 ⟨proof⟩

lemma *Suffixes-UNION* [simp]: $Suffixes (\bigcup (f \text{ ` } A)) = \bigcup ((\lambda x. Suffixes (f x)) \text{ ` } A)$
 ⟨proof⟩

lemma *Suffixes-compower*:
 assumes $A \neq \{\}$
 shows $Suffixes (A \text{ } \sim n) = insert [] (Suffixes A \text{ } @@ (\bigcup_{k < n}. A \text{ } \sim k))$
 ⟨proof⟩

lemma *Suffixes-star* [simp]:
 assumes $A \neq \{\}$
 shows $Suffixes (star A) = Suffixes A \text{ } @@ star A$
 ⟨proof⟩

Prefix language

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

lemma *Prefixes-altdef* [code]: $Prefixes A = (\bigcup_{w \in A}. set (prefixes w))$
 ⟨proof⟩

lemma *Nil-in-Prefixes-iff* [simp]: $[] \in Prefixes A \iff A \neq \{\}$
 ⟨proof⟩

lemma *Prefixes-empty* [simp]: $Prefixes \{\} = \{\}$
 ⟨proof⟩

lemma *Prefixes-empty-iff* [simp]: $Prefixes A = \{\} \iff A = \{\}$
 ⟨proof⟩

lemma *Prefixes-singleton* [simp]: $Prefixes \{xs\} = set (prefixes xs)$
 ⟨proof⟩

lemma *Prefixes-insert*: $Prefixes (insert xs A) = set (prefixes xs) \cup Prefixes A$
 ⟨proof⟩

lemma *Prefixes-conc* [simp]: $B \neq \{\} \implies Prefixes (A \text{ } @@ B) = Prefixes A \cup (A \text{ } @@ Prefixes B)$
 ⟨proof⟩

lemma *Prefixes-union* [simp]: $Prefixes (A \cup B) = Prefixes A \cup Prefixes B$
 ⟨proof⟩

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

lemma *Prefixes-rev*: $Prefixes (rev \text{ ` } A) = rev \text{ ` } Suffixes A$
 ⟨proof⟩

lemma *Suffixes-rev*: $Suffixes (rev \text{ ' } A) = rev \text{ ' } Prefixes A$
 ⟨proof⟩

lemma *Prefixes-compower*:
assumes $A \neq \{\}$
shows $Prefixes (A \text{ } \sim n) = insert [] ((\bigcup k < n. A \text{ } \sim k) @@ Prefixes A)$
 ⟨proof⟩

lemma *Prefixes-star [simp]*:
assumes $A \neq \{\}$
shows $Prefixes (star A) = star A @@ Prefixes A$
 ⟨proof⟩

7.3 Subword language

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

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

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

lemma *Sublists-empty [simp]*: $Sublists \{\} = \{\}$
 ⟨proof⟩

lemma *Sublists-singleton [simp]*: $Sublists \{w\} = set (sublists w)$
 ⟨proof⟩

lemma *Sublists-insert*: $Sublists (insert w A) = set (sublists w) \cup Sublists A$
 ⟨proof⟩

lemma *Sublists-Un [simp]*: $Sublists (A \cup B) = Sublists A \cup Sublists B$
 ⟨proof⟩

lemma *Sublists-UN [simp]*: $Sublists (\bigcup (f \text{ ' } A)) = \bigcup ((\lambda x. Sublists (f x)) \text{ ' } A)$
 ⟨proof⟩

lemma *Sublists-conv-Prefixes*: $Sublists A = Prefixes (Suffixes A)$
 ⟨proof⟩

lemma *Sublists-conv-Suffixes*: $Sublists A = Suffixes (Prefixes A)$
 ⟨proof⟩

lemma *Sublists-conc [simp]*:
assumes $A \neq \{\}$ $B \neq \{\}$
shows $Sublists (A @@ B) = Sublists A \cup Sublists B \cup Suffixes A @@ Prefixes$

B
<proof>

lemma *star-not-empty* [simp]: $\text{star } A \neq \{\}$
<proof>

lemma *Sublists-star*:
 $A \neq \{\} \implies \text{Sublists } (\text{star } A) = \text{Sublists } A \cup \text{Suffixes } A \text{ @@ } \text{star } A \text{ @@ } \text{Prefixes } A$
<proof>

lemma *Prefixes-subset-Sublists*: $\text{Prefixes } A \subseteq \text{Sublists } A$
<proof>

lemma *Suffixes-subset-Sublists*: $\text{Suffixes } A \subseteq \text{Sublists } A$
<proof>

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 $\text{Subseqs } A = (\bigcup w \in A. \text{set } (\text{subseqs } w))$

lemma *Subseqs-empty* [simp]: $\text{Subseqs } \{\} = \{\}$
<proof>

lemma *Subseqs-insert* [simp]: $\text{Subseqs } (\text{insert } xs \ A) = \text{set } (\text{subseqs } xs) \cup \text{Subseqs } A$
<proof>

lemma *Subseqs-singleton* [simp]: $\text{Subseqs } \{xs\} = \text{set } (\text{subseqs } xs)$
<proof>

lemma *Subseqs-Un* [simp]: $\text{Subseqs } (A \cup B) = \text{Subseqs } A \cup \text{Subseqs } B$
<proof>

lemma *Subseqs-UNION* [simp]: $\text{Subseqs } (\bigcup (f \text{ ' } A)) = \bigcup ((\lambda x. \text{Subseqs } (f \ x)) \text{ ' } A)$
<proof>

lemma *Subseqs-conc* [simp]: $\text{Subseqs } (A \text{ @@ } B) = \text{Subseqs } A \text{ @@ } \text{Subseqs } B$
<proof>

lemma *Subseqs-compower* [simp]: $\text{Subseqs } (A \text{ ^^ } n) = \text{Subseqs } A \text{ ^^ } n$
<proof>

lemma *Subseqs-star* [simp]: $\text{Subseqs } (\text{star } A) = \text{star } (\text{Subseqs } A)$
<proof>

lemma *Sublists-subset-Subseqs*: $Sublists\ A \subseteq Subseqs\ A$
 ⟨proof⟩

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*]: $lang\ (rexp-rev\ r) = rev\ 'lang\ r$
 ⟨proof⟩

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*]: $lang\ (rexp-of-word\ xs) = \{xs\}$
 ⟨proof⟩

lemma *size-rexp-of-word* [*simp*]: $size\ (rexp-of-word\ xs) = Suc\ (2 * (length\ xs - 1))$
 ⟨proof⟩

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*: $lang\ (rexp-subst\ f\ r) = subst-word\ f\ 'lang\ r$
 ⟨proof⟩

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)
⟨proof⟩

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)
⟨proof⟩

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)
⟨proof⟩

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]*: $\text{lang } (\text{subseqs-rexp } r) = \text{Subseqs } (\text{lang } r)$
 ⟨proof⟩

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 Brzowski'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*: $\text{atoms } (\text{deriv } x \ r) \subseteq \text{atoms } r$

⟨proof⟩

lemma *atoms-derivs-subset*: $\text{atoms } (\text{derivs } w \ r) \subseteq \text{atoms } r$

⟨proof⟩

lemma *lang-deriv*: $\text{lang } (\text{deriv } c \ r) = \text{Deriv } c \ (\text{lang } r)$

⟨proof⟩

lemma *lang-derivs*: $\text{lang } (\text{derivs } s \ r) = \text{Derivs } s \ (\text{lang } r)$

⟨proof⟩

A regular expression matcher:

definition $matcher :: 'a rexp \Rightarrow 'a list \Rightarrow bool$ **where**
 $matcher\ r\ s = nullable\ (derivs\ s\ r)$

lemma *matcher-correctness*: $matcher\ r\ s \longleftrightarrow s \in lang\ r$
 $\langle proof \rangle$

8.2 Antimirov's partial derivatives

abbreviation

$Timess\ rs\ r \equiv (\bigcup r' \in rs. \{Times\ r'\ r\})$

lemma *Timess-eq-image*:

$Timess\ rs\ r = (\lambda r'. Times\ r'\ r) \text{ ` } rs$
 $\langle proof \rangle$

primrec

$pderiv :: 'a \Rightarrow 'a rexp \Rightarrow 'a rexp\ set$

where

$pderiv\ c\ Zero = \{\}$
 $| pderiv\ c\ One = \{\}$
 $| pderiv\ c\ (Atom\ c') = (if\ c = c'\ then\ \{One\}\ else\ \{\})$
 $| pderiv\ c\ (Plus\ r1\ r2) = (pderiv\ c\ r1) \cup (pderiv\ c\ r2)$
 $| pderiv\ c\ (Times\ r1\ r2) =$
 $(if\ nullable\ r1\ then\ Timess\ (pderiv\ c\ r1)\ r2 \cup pderiv\ c\ r2\ else\ Timess\ (pderiv$
 $c\ r1)\ r2)$
 $| pderiv\ c\ (Star\ r) = Timess\ (pderiv\ c\ r)\ (Star\ r)$

primrec

$derivs :: 'a list \Rightarrow 'a rexp \Rightarrow ('a rexp)\ set$

where

$derivs\ []\ r = \{r\}$
 $| derivs\ (c\ \# s)\ r = \bigcup (derivs\ s\ \text{` } pderiv\ c\ r)$

abbreviation

$pderiv\ set :: 'a \Rightarrow 'a rexp\ set \Rightarrow 'a rexp\ set$

where

$pderiv\ set\ c\ rs \equiv \bigcup (pderiv\ c\ \text{` } rs)$

abbreviation

$pderivs\ set :: 'a list \Rightarrow 'a rexp\ set \Rightarrow 'a rexp\ set$

where

$pderivs\ set\ s\ rs \equiv \bigcup (pderivs\ s\ \text{` } rs)$

lemma *pderivs-append*:

$pderivs\ (s1\ @\ s2)\ r = \bigcup (pderivs\ s2\ \text{` } pderivs\ s1\ r)$
 $\langle proof \rangle$

lemma *pderivs-snoc*:

shows $pderivs\ (s\ @\ [c])\ r = pderiv\ set\ c\ (pderivs\ s\ r)$

$\langle \text{proof} \rangle$

lemma *pderivs-simps* [*simp*]:

shows $pderivs\ s\ Zero = (if\ s = []\ then\ \{Zero\}\ else\ \{\})$
and $pderivs\ s\ One = (if\ s = []\ then\ \{One\}\ else\ \{\})$
and $pderivs\ s\ (Plus\ r1\ r2) = (if\ s = []\ then\ \{Plus\ r1\ r2\}\ else\ (pderivs\ s\ r1) \cup (pderivs\ s\ r2))$
 $\langle \text{proof} \rangle$

lemma *pderivs-Atom*:

shows $pderivs\ s\ (Atom\ c) \subseteq \{Atom\ c,\ One\}$
 $\langle \text{proof} \rangle$

8.3 Relating left-quotients and partial derivatives

lemma *Deriv-pderiv*:

shows $Deriv\ c\ (lang\ r) = \bigcup (lang\ ' \ pderiv\ c\ r)$
 $\langle \text{proof} \rangle$

lemma *Derivs-pderivs*:

shows $Derivs\ s\ (lang\ r) = \bigcup (lang\ ' \ pderivs\ s\ r)$
 $\langle \text{proof} \rangle$

8.4 Relating derivatives and partial derivatives

lemma *deriv-pderiv*:

shows $\bigcup (lang\ ' \ (pderiv\ c\ r)) = lang\ (deriv\ c\ r)$
 $\langle \text{proof} \rangle$

lemma *derivs-pderivs*:

shows $\bigcup (lang\ ' \ (pderivs\ s\ r)) = lang\ (derivs\ s\ r)$
 $\langle \text{proof} \rangle$

8.5 Finiteness property of partial derivatives

definition

$pderivs\text{-}lang :: 'a\ lang \Rightarrow 'a\ rexp \Rightarrow 'a\ rexp\ set$

where

$pderivs\text{-}lang\ A\ r \equiv \bigcup x \in A. pderivs\ x\ r$

lemma *pderivs-lang-subsetI*:

assumes $\bigwedge s. s \in A \implies pderivs\ s\ r \subseteq C$
shows $pderivs\text{-}lang\ A\ r \subseteq C$
 $\langle \text{proof} \rangle$

lemma *pderivs-lang-union*:

shows $pderivs\text{-}lang\ (A \cup B)\ r = (pderivs\text{-}lang\ A\ r \cup pderivs\text{-}lang\ B\ r)$
 $\langle \text{proof} \rangle$

lemma *pderivs-lang-subset*:

shows $A \subseteq B \implies \text{pderivs-lang } A \ r \subseteq \text{pderivs-lang } B \ r$
 ⟨proof⟩

definition

$$\text{UNIV1} \equiv \text{UNIV} - \{\emptyset\}$$

lemma *pderivs-lang-Zero* [simp]:

shows $\text{pderivs-lang } \text{UNIV1} \ \text{Zero} = \{\}$
 ⟨proof⟩

lemma *pderivs-lang-One* [simp]:

shows $\text{pderivs-lang } \text{UNIV1} \ \text{One} = \{\}$
 ⟨proof⟩

lemma *pderivs-lang-Atom* [simp]:

shows $\text{pderivs-lang } \text{UNIV1} \ (\text{Atom } c) = \{\text{One}\}$
 ⟨proof⟩

lemma *pderivs-lang-Plus* [simp]:

shows $\text{pderivs-lang } \text{UNIV1} \ (\text{Plus } r1 \ r2) = \text{pderivs-lang } \text{UNIV1} \ r1 \cup \text{pderivs-lang } \text{UNIV1} \ r2$
 ⟨proof⟩

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

definition

$$\text{PSuf } s \equiv \{v. v \neq \emptyset \wedge (\exists u. u @ v = s)\}$$

lemma *PSuf-snoc*:

shows $\text{PSuf } (s @ [c]) = (\text{PSuf } s) @ @ \{[c]\} \cup \{[c]\}$
 ⟨proof⟩

lemma *PSuf-Union*:

shows $(\bigcup v \in \text{PSuf } s @ @ \{[c]\}. f \ v) = (\bigcup v \in \text{PSuf } s. f \ (v @ [c]))$
 ⟨proof⟩

lemma *pderivs-lang-snoc*:

shows $\text{pderivs-lang } (\text{PSuf } s @ @ \{[c]\}) \ r = (\text{pderiv-set } c \ (\text{pderivs-lang } (\text{PSuf } s) \ r))$
 ⟨proof⟩

lemma *pderivs-Times*:

shows $\text{pderivs } s \ (\text{Times } r1 \ r2) \subseteq \text{Times} \ (\text{pderivs } s \ r1) \ r2 \cup (\text{pderivs-lang } (\text{PSuf } s) \ r2)$
 ⟨proof⟩

lemma *pderivs-lang-Times-aux1*:

assumes $a: s \in \text{UNIV1}$

shows $\text{pderivs-lang } (\text{PSuf } s) \ r \subseteq \text{pderivs-lang } \text{UNIV1} \ r$

<proof>

lemma *pderivs-lang-Times-aux2*:

assumes *a*: $s \in \text{UNIV1}$

shows $\text{Timess } (\text{pderivs } s \ r1) \ r2 \subseteq \text{Timess } (\text{pderivs-lang UNIV1 } r1) \ r2$

<proof>

lemma *pderivs-lang-Times*:

shows $\text{pderivs-lang UNIV1 } (\text{Times } r1 \ r2) \subseteq \text{Timess } (\text{pderivs-lang UNIV1 } r1) \ r2$
 $\cup \text{pderivs-lang UNIV1 } r2$

<proof>

lemma *pderivs-Star*:

assumes *a*: $s \neq []$

shows $\text{pderivs } s \ (\text{Star } r) \subseteq \text{Timess } (\text{pderivs-lang } (\text{PSuf } s) \ r) \ (\text{Star } r)$

<proof>

lemma *pderivs-lang-Star*:

shows $\text{pderivs-lang UNIV1 } (\text{Star } r) \subseteq \text{Timess } (\text{pderivs-lang UNIV1 } r) \ (\text{Star } r)$

<proof>

lemma *finite-Timess [simp]*:

assumes *a*: *finite A*

shows *finite* ($\text{Timess } A \ r$)

<proof>

lemma *finite-pderivs-lang-UNIV1*:

shows *finite* ($\text{pderivs-lang UNIV1 } r$)

<proof>

lemma *pderivs-lang-UNIV*:

shows $\text{pderivs-lang UNIV } r = \text{pderivs } [] \ r \cup \text{pderivs-lang UNIV1 } r$

<proof>

lemma *finite-pderivs-lang-UNIV*:

shows *finite* ($\text{pderivs-lang UNIV } r$)

<proof>

lemma *finite-pderivs-lang*:

shows *finite* ($\text{pderivs-lang } A \ r$)

<proof>

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* \Rightarrow *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-Times-pderivs-lang-le:*

$card (Times (pderivs-lang\ A\ r)\ s) \leq card (pderivs-lang\ A\ r)$
 <proof>

lemma *card-pderivs-lang-UNIV1-le-awidth:* $card (pderivs-lang\ UNIV1\ r) \leq awidth\ r$

<proof>

Antimirov's Theorem 3.4:

theorem *card-pderivs-lang-UNIV-le-awidth:* $card (pderivs-lang\ UNIV\ r) \leq awidth\ r + 1$

<proof>

Antimirov's Corollary 3.5:

corollary *card-pderivs-lang-le-awidth:* $card (pderivs-lang\ A\ r) \leq awidth\ r + 1$

<proof>

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:* $Atoms(pderiv\ a\ r) \subseteq atoms\ r$

<proof>

lemma *Atoms-Pderiv:* $Atoms(Pderiv\ a\ R) \subseteq Atoms\ R$

<proof>

lemma *pderiv-no-occurrence:*

$x \notin atoms\ r \implies pderiv\ x\ r = \{\}$

<proof>

lemma *Pderiv-no-occurrence*:
 $x \notin \text{Atoms } R \implies \text{Pderiv } x \ R = \{\}$
 $\langle \text{proof} \rangle$

lemma *Deriv-Lang*: $\text{Deriv } c \ (\text{Lang } R) = \text{Lang } (\text{Pderiv } c \ R)$
 $\langle \text{proof} \rangle$

lemma *Nullable-pderiv[simp]*: $\text{Nullable}(\text{pderivs } w \ r) = (w : \text{lang } r)$
 $\langle \text{proof} \rangle$

type-synonym $'a \ \text{Rexp-pair} = 'a \ \text{rexp set} * 'a \ \text{rexp set}$
type-synonym $'a \ \text{Rexp-pairs} = 'a \ \text{Rexp-pair list}$

definition *is-Bisimulation* $:: 'a \ \text{list} \Rightarrow 'a \ \text{Rexp-pairs} \Rightarrow \text{bool}$
where
is-Bisimulation as ps =
 $(\forall (R,S) \in \text{set } ps. \text{Atoms } R \cup \text{Atoms } S \subseteq \text{set } as \wedge$
 $(\text{Nullable } R \longleftrightarrow \text{Nullable } S) \wedge$
 $(\forall a \in \text{set } as. (\text{Pderiv } a \ R, \text{Pderiv } a \ S) \in \text{set } ps))$

lemma *Bisim-Lang-eq*:
assumes *Bisim*: *is-Bisimulation as ps*
assumes $(R, S) \in \text{set } ps$
shows $\text{Lang } R = \text{Lang } S$
 $\langle \text{proof} \rangle$

9.1 Closure computation

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

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

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

definition *pre-Bisim* $:: 'a \ \text{list} \Rightarrow 'a \ \text{rexp set} \Rightarrow 'a \ \text{rexp set} \Rightarrow$

'a Rexp-pairs * 'a Rexp-pairs \Rightarrow bool

where

pre-Bisim as $R S = (\lambda(ws,ps).$

$((R,S) \in \text{set } ws \cup \text{set } ps) \wedge$
 $(\forall (R,S) \in \text{set } ws \cup \text{set } ps. \text{Atoms } R \cup \text{Atoms } S \subseteq \text{set } as) \wedge$
 $(\forall (R,S) \in \text{set } ps. (\text{Nullable } R \longleftrightarrow \text{Nullable } S) \wedge$
 $(\forall a \in \text{set } as. (Pderiv a R, Pderiv a S) \in \text{set } ps \cup \text{set } ws)))$

lemma step-set-eq: $\llbracket \text{test } (ws,ps); \text{step as } (ws,ps) = (ws',ps') \rrbracket$

$\Longrightarrow \text{set } ws' \cup \text{set } ps' =$

$\text{set } ws \cup \text{set } ps$

$\cup (\bigcup a \in \text{set } as. \{(Pderiv a (\text{fst}(\text{hd } ws)), Pderiv a (\text{snd}(\text{hd } ws)))\})$

$\langle \text{proof} \rangle$

theorem closure-sound:

assumes result: closure as $([(R,S)], []) = \text{Some}([], ps)$

and atoms: $\text{Atoms } R \cup \text{Atoms } S \subseteq \text{set } as$

shows $\text{Lang } R = \text{Lang } S$

$\langle \text{proof} \rangle$

9.2 The overall procedure

definition check-eqv :: 'a rexp \Rightarrow 'a rexp \Rightarrow bool

where

check-eqv r s =

$(\text{case closure } (\text{add-atoms } r (\text{add-atoms } s [])) ((\{r\}, \{s\}), []) \text{ of}$
 $\text{Some}([], -) \Rightarrow \text{True} \mid - \Rightarrow \text{False})$

lemma soundness: **assumes** check-eqv r s **shows** lang r = lang s

$\langle \text{proof} \rangle$

Test:

lemma check-eqv

$(\text{Plus One } (\text{Times } (\text{Atom } 0) (\text{Star}(\text{Atom } 0))))$

$(\text{Star}(\text{Atom}(0::\text{nat})))$

$\langle \text{proof} \rangle$

9.3 Termination and Completeness

definition PDERIVS :: 'a rexp set \Rightarrow 'a rexp set **where**

$PDERIVS R = (\text{UN } r:R. \text{pderivs-lang UNIV } r)$

lemma PDERIVS-incr[simp]: $R \subseteq PDERIVS R$

$\langle \text{proof} \rangle$

lemma Pderiv-PDERIVS: **assumes** $R' \subseteq PDERIVS R$ **shows** $Pderiv a R' \subseteq PDERIVS R$

$\langle \text{proof} \rangle$

lemma finite-PDERIVS: finite R \Longrightarrow finite(PDERIVS R)

<proof>

lemma *closure-Some*: **assumes** *finite R0 finite S0* **shows** $\exists p.$ *closure as* $((R0, S0), [])$
= *Some p*
<proof>

theorem *closure-Some-Inv*: **assumes** *closure as* $([({r}, {s})], [])$ = *Some p*
shows $\forall (R, S) \in \text{set}(\text{fst } p).$ $\exists w.$ $R = \text{pderiv } w \ r \wedge S = \text{pderiv } w \ s$ (**is** *?Inv p*)
<proof>

lemma *closure-complete*: **assumes** *lang r = lang s*
shows *EX bs. closure as* $([({r}, {s})], [])$ = *Some([], bs)* (**is** *?C*)
<proof>

corollary *completeness*: *lang r = lang s* \implies *check-equiv r s*
<proof>

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) = $(\text{lang } r) \cup (\text{lang } s)$ |
lang (Times r s) = $\text{conc } (\text{lang } r) (\text{lang } s)$ |
lang (Star r) = $\text{star}(\text{lang } r)$ |
lang (Not r) = $\text{lists } S - \text{lang } r$ |
lang (Inter r s) = $(\text{lang } r \cap \text{lang } s)$

end

lemma *lang-subset-lists*: $\text{atoms } r \subseteq S \implies \text{lang } S \ r \subseteq \text{lists } S$
(*proof*)

primrec *nullable* :: 'a *rexp* \Rightarrow *bool* **where**
nullable *Zero* = *False* |
nullable *One* = *True* |
nullable (*Atom* *c*) = *False* |
nullable (*Plus* *r1* *r2*) = (*nullable* *r1* \vee *nullable* *r2*) |
nullable (*Times* *r1* *r2*) = (*nullable* *r1* \wedge *nullable* *r2*) |
nullable (*Star* *r*) = *True* |
nullable (*Not* *r*) = (\neg (*nullable* *r*)) |
nullable (*Inter* *r* *s*) = (*nullable* *r* \wedge *nullable* *s*)

lemma *nullable-iff*: $\text{nullable } r \longleftrightarrow [] \in \text{lang } S \ r$
(*proof*)

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* \Rightarrow *nat rexp* \Rightarrow *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* \leq *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)*
 ⟨*proof*⟩

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)*
 ⟨*proof*⟩

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)*
 ⟨*proof*⟩

primrec $norm :: nat \text{ rexp} \Rightarrow nat \text{ rexp}$

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 $lang\text{-}norm[simp]: lang \ S \ (norm \ r) = lang \ S \ r$
(proof)

11.3 Derivative

primrec $nderiv :: nat \Rightarrow nat \text{ rexp} \Rightarrow nat \text{ rexp}$

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 $lang\text{-}nderiv: a:S \Longrightarrow lang \ S \ (nderiv \ a \ r) = Deriv \ a \ (lang \ S \ r)$
(proof)

lemma $atoms\text{-}nPlus[simp]: atoms \ (nPlus \ r \ s) = atoms \ r \cup atoms \ s$
(proof)

lemma $atoms\text{-}nTimes: atoms \ (nTimes \ r \ s) \subseteq atoms \ r \cup atoms \ s$
(proof)

lemma $atoms\text{-}nInter: atoms \ (nInter \ r \ s) \subseteq atoms \ r \cup atoms \ s$
(proof)

lemma $atoms\text{-}norm: atoms \ (norm \ r) \subseteq atoms \ r$
(proof)

lemma $atoms\text{-}nderiv: atoms \ (nderiv \ a \ r) \subseteq atoms \ r$
(proof)

11.4 Bisimulation between languages and regular expressions

context

fixes $S :: 'a \text{ set}$

begin

coinductive *bisimilar* :: $'a \text{ lang} \Rightarrow 'a \text{ lang} \Rightarrow \text{bool}$ **where**

$K \subseteq \text{lists } S \Longrightarrow L \subseteq \text{lists } S$

$\Longrightarrow (\[] \in K \longleftrightarrow \[] \in L)$

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

$\Longrightarrow \text{bisimilar } K L$

lemma *equal-if-bisimilar*:

assumes $K \subseteq \text{lists } S \ L \subseteq \text{lists } S \ \text{bisimilar } K L$ **shows** $K = L$

<proof>

lemma *language-coinduct*:

fixes R (**infixl** \sim 50)

assumes $\bigwedge K L. K \sim L \Longrightarrow K \subseteq \text{lists } S \wedge L \subseteq \text{lists } S$

assumes $K \sim L$

assumes $\bigwedge K L. K \sim L \Longrightarrow (\[] \in K \longleftrightarrow \[] \in L)$

assumes $\bigwedge K L x. K \sim L \Longrightarrow x : S \Longrightarrow \text{Deriv } x K \sim \text{Deriv } x L$

shows $K = L$

<proof>

end

type-synonym *rexp-pair* = $\text{nat rexp} * \text{nat rexp}$

type-synonym *rexp-pairs* = rexp-pair list

definition *is-bisimulation* :: $\text{nat list} \Rightarrow \text{rexp-pairs} \Rightarrow \text{bool}$

where

is-bisimulation as ps =

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

lemma *bisim-lang-eq*:

assumes *bisim*: *is-bisimulation as ps*

assumes $(r, s) \in \text{set } ps$

shows $\text{lang } (\text{set } as) r = \text{lang } (\text{set } as) s$

<proof>

11.5 Closure computation

fun *test* :: $\text{rexp-pairs} * \text{rexp-pairs} \Rightarrow \text{bool}$

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

fun *step* :: $\text{nat list} \Rightarrow \text{rexp-pairs} * \text{rexp-pairs} \Rightarrow \text{rexp-pairs} * \text{rexp-pairs}$

where $step\ as\ (ws,ps) =$
 $(let$
 $\quad (r, s) = hd\ ws;$
 $\quad ps' = (r, s) \# ps;$
 $\quad succs = map\ (\lambda a. (nderiv\ a\ r, nderiv\ a\ s))\ as;$
 $\quad new = filter\ (\lambda p. p \notin set\ ps' \cup set\ ws)\ succs$
 $\quad in\ (new\ @\ tl\ ws,\ ps'))$

definition $closure ::$
 $nat\ list \Rightarrow rexp\ pairs * rexp\ pairs$
 $\Rightarrow (rexp\ pairs * rexp\ pairs)\ option\ \mathbf{where}$
 $closure\ as = while\ option\ test\ (step\ as)$

definition $pre\ bisim :: nat\ list \Rightarrow nat\ rexp \Rightarrow nat\ rexp \Rightarrow$
 $rexp\ pairs * rexp\ pairs \Rightarrow bool$

where
 $pre\ bisim\ as\ r\ s = (\lambda(ws,ps).$
 $\quad ((r, s) \in set\ ws \cup set\ ps) \wedge$
 $\quad (\forall (r,s) \in set\ ws \cup set\ ps. atoms\ r \cup atoms\ s \subseteq set\ as) \wedge$
 $\quad (\forall (r,s) \in set\ ps. (nullable\ r \longleftrightarrow nullable\ s) \wedge$
 $\quad (\forall a \in set\ as. (nderiv\ a\ r, nderiv\ a\ s) \in set\ ps \cup set\ ws)))$

theorem $closure\ sound:$
assumes $result: closure\ as\ ([[(r,s)], []]) = Some([], ps)$
and $atoms: atoms\ r \cup atoms\ s \subseteq set\ as$
shows $lang\ (set\ as)\ r = lang\ (set\ as)\ s$
 $\langle proof \rangle$

11.6 The overall procedure

primrec $add\ atoms :: nat\ rexp \Rightarrow nat\ list \Rightarrow nat\ list$

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 $set\ add\ atoms: set\ (add\ atoms\ r\ as) = atoms\ r \cup set\ as$
 $\langle proof \rangle$

definition $check\ eqv :: nat\ list \Rightarrow nat\ rexp \Rightarrow nat\ rexp \Rightarrow bool$
where
 $check\ eqv\ as\ r\ s \longleftrightarrow set(add\ atoms\ r\ (add\ atoms\ s\ [])) \subseteq set\ as \wedge$
 $\quad (case\ closure\ as\ ([[(norm\ r,\ norm\ s)], []])\ of$
 $\quad\quad Some([], -) \Rightarrow True\ | - \Rightarrow False)$

lemma *soundness*:

assumes *check-equiv as r s* **shows** $\text{lang } (\text{set as}) \ r = \text{lang } (\text{set as}) \ s$
<proof>

lemma *check-equiv [0]* (*Plus One (Times (Atom 0) (Star(Atom 0)))) (Star(Atom 0))*
<proof>

lemma *check-equiv [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))))*)
<proof>

lemma *check-equiv [0]* (*Atom 0*) (*Inter (Star (Atom 0)) (Atom 0)*)
<proof>

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.