Regular Sets, Expressions, Derivatives and Relation Algebra

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

December 14, 2021

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 = {[]}
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *concE*[*elim*]:
assumes *w ∈ A @@ B*
obtains *u v* **where** *u ∈ A v ∈ B w = u@v*
⟨*proof*⟩

lemma *conc-mono*: *A ⊆ C ⟹ B ⊆ D ⟹ A @@ B ⊆ C @@ D*
⟨*proof*⟩

lemma *conc-empty*[*simp*]: **shows** *{} @@ A = {}* **and** *A @@ {} = {}*
⟨*proof*⟩

lemma *conc-epsilon*[*simp*]: **shows** *{[]} @@ A = A* **and** *A @@ {[]} = A*
⟨*proof*⟩

lemma *conc-assoc*: *(A @@ B) @@ C = A @@ (B @@ C)*
⟨*proof*⟩

lemma *conc-Un-distrib*:
shows A @@ $(B \cup C) = A$ @@ $B \cup A$ @@ C
and $(A \cup B)$ @@ $C = A$ @@ $C \cup B$ @@ C
⟨*proof*⟩

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

lemma *conc-subset-lists*: $A \subseteq$ *lists* $S \Longrightarrow B \subseteq$ *lists* $S \Longrightarrow A$ @@ $B \subseteq$ *lists* S
⟨*proof*⟩

lemma *Nil-in-conc*[*simp*]: $[] \in A$ @@ $B \longleftrightarrow [] \in A \wedge [] \in B$
⟨*proof*⟩

lemma *concI-if-Nil1*: $[] \in A \Longrightarrow xs : B \Longrightarrow xs \in A$ @@ B
⟨*proof*⟩

lemma *conc-Diff-if-Nil1*: $[] \in A \Longrightarrow A$ @@ $B = (A - \{[]\})$ @@ $B \cup B$
⟨*proof*⟩

lemma *concI-if-Nil2*: $[] \in B \Longrightarrow xs : A \Longrightarrow xs \in A$ @@ B
⟨*proof*⟩

lemma *conc-Diff-if-Nil2*: $[] \in B \Longrightarrow A$ @@ $B = A$ @@ $(B - \{[]\}) \cup A$
⟨*proof*⟩

lemma *singleton-in-conc*:
$[x] : A$ @@ $B \longleftrightarrow [x] : A \wedge [] : B \vee [] : A \wedge [x] : B$
⟨*proof*⟩

1.2 A^n

lemma *lang-pow-add*: A ^^ $(n + m) = A$ ^^ n @@ A ^^ m
⟨*proof*⟩

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

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

lemma *conc-pow-comm*:
shows A @@ $(A$ ^^ $n) = (A$ ^^ $n)$ @@ A
⟨*proof*⟩

lemma *length-lang-pow-ub*:

$\forall w \in A.\ length\ w \leq k \Longrightarrow w : A \frown\frown n \Longrightarrow length\ w \leq k*n$
⟨*proof*⟩

lemma *length-lang-pow-lb*:
$\forall w \in A.\ length\ w \geq k \Longrightarrow w : A \frown\frown n \Longrightarrow length\ w \geq k*n$
⟨*proof*⟩

lemma *lang-pow-subset-lists*: $A \subseteq lists\ S \Longrightarrow A \frown\frown n \subseteq lists\ S$
⟨*proof*⟩

1.3 *star*

lemma *star-subset-lists*: $A \subseteq lists\ S \Longrightarrow star\ A \subseteq lists\ S$
⟨*proof*⟩

lemma *star-if-lang-pow*[*simp*]: $w : A \frown\frown n \Longrightarrow w : star\ A$
⟨*proof*⟩

lemma *Nil-in-star*[*iff*]: $[] : star\ A$
⟨*proof*⟩

lemma *star-if-lang*[*simp*]: **assumes** $w : A$ **shows** $w : star\ A$
⟨*proof*⟩

lemma *append-in-starI*[*simp*]:
assumes $u : star\ A$ **and** $v : star\ A$ **shows** $u@v : star\ A$
⟨*proof*⟩

lemma *conc-star-star*: $star\ A\ @@\ star\ A = star\ A$
⟨*proof*⟩

lemma *conc-star-comm*:
shows $A\ @@\ star\ A = star\ A\ @@\ A$
⟨*proof*⟩

lemma *star-induct*[*consumes 1*, *case-names Nil append*, *induct set*: *star*]:
assumes $w : star\ A$
and $P\ []$
and *step*: $!!u\ v.\ u : A \Longrightarrow v : star\ A \Longrightarrow P\ v \Longrightarrow P\ (u@v)$
shows $P\ w$
⟨*proof*⟩

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

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

lemma *star-idemp*[*simp*]: $star\ (star\ A) = star\ A$

⟨*proof*⟩

lemma *star-unfold-left*: *star A* = *A* @@ *star A* ∪ {[]} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *concat-in-star*: *set ws* ⊆ *A* ⟹ *concat ws* : *star A*
⟨*proof*⟩

lemma *in-star-iff-concat*:
w ∈ *star A* = (∃ *ws*. *set ws* ⊆ *A* ∧ *w* = *concat ws*)
(**is** - = (∃ *ws*. *?R w ws*))
⟨*proof*⟩

lemma *star-conv-concat*: *star A* = {*concat ws*|*ws*. *set ws* ⊆ *A*}
⟨*proof*⟩

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

lemma *star-unfold-left-Nil*: *star A* = (*A* − {[]}) @@ (*star A*) ∪ {[]}
⟨*proof*⟩

lemma *star-Diff-Nil-fold*: (*A* − {[]}) @@ *star A* = *star A* − {[]}
⟨*proof*⟩

lemma *star-decom*:
assumes *a*: *x* ∈ *star A x* ≠ []
shows ∃ *a b*. *x* = *a* @ *b* ∧ *a* ≠ [] ∧ *a* ∈ *A* ∧ *b* ∈ *star A*
⟨*proof*⟩

1.4 Left-Quotients of languages

definition *Deriv* :: *'a* ⇒ *'a lang* ⇒ *'a lang*
where *Deriv x A* = { *xs*. *x*#*xs* ∈ *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*))
⟨*proof*⟩

lemma *Der-conc* [*simp*]:
shows *Deriv c* (*A* @@ *B*) = (*Deriv c A*) @@ *B* ∪ (*if* [] ∈ *A then Deriv c B else* {})
⟨*proof*⟩

lemma *Deriv-star* [*simp*]:
shows *Deriv c* (*star A*) = (*Deriv c A*) @@ *star A*
⟨*proof*⟩

lemma *Deriv-diff*[*simp*]:
shows *Deriv c* (*A* − *B*) = *Deriv c A* − *Deriv c B*
⟨*proof*⟩

lemma *Deriv-lists*[*simp*]: *c* : *S* ⟹ *Deriv c* (*lists S*) = *lists S*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *in-fold-Deriv*: *v* ∈ *fold Deriv w L* ⟷ *w* @ *v* ∈ *L*
⟨*proof*⟩

lemma *Derivs-alt-def* [*code*]: *Derivs w L* = *fold Deriv w L*
⟨*proof*⟩

lemma *Deriv-code* [*code*]:
Deriv x A = *tl* ‘ *Set.filter* (λ*xs*. *case xs of x′* # - ⇒ *x* = *x′* | - ⇒ *False*) *A*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *shuffle-subset-lists*:
assumes *A* ⊆ *lists S B* ⊆ *lists S*
shows *A* ⧢ *B* ⊆ *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 $A \parallel \bigcup(M \mathbin{`} I) = \bigcup((\%i.\ A \parallel M\ i) \mathbin{`} I)$
and $\bigcup(M \mathbin{`} I) \parallel A = \bigcup((\%i.\ M\ i \parallel A) \mathbin{`} I)$
⟨*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*: $X = A \mathbin{@@} X \cup B$
shows $X = (A \mathbin{^^} Suc\ n) \mathbin{@@} X \cup (\bigcup m{\leq}n.\ (A \mathbin{^^} m) \mathbin{@@} B)$
⟨*proof*⟩

lemma *Arden*:
assumes $[] \notin A$
shows $X = A \mathbin{@@} X \cup B \longleftrightarrow X = star\ A \mathbin{@@} B$
⟨*proof*⟩

lemma *reversed-arden-helper*:
assumes *eq*: $X = X \mathbin{@@} A \cup B$
shows $X = X \mathbin{@@} (A \mathbin{^^} Suc\ n) \cup (\bigcup m{\leq}n.\ B \mathbin{@@} (A \mathbin{^^} m))$
⟨*proof*⟩

theorem *reversed-Arden*:
assumes *nemp*: $[] \notin A$
shows $X = X \mathbin{@@} A \cup B \longleftrightarrow X = B \mathbin{@@} star\ A$
⟨*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* ≡ (∃ *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*
⟨*proof*⟩

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* = {}
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *map-rexp-ident*[*simp*]: *map-rexp* (λ*x*. *x*) = (λ*r*. *r*)
⟨*proof*⟩

lemma *atoms-lang*: *w* : *lang r* ⟹ *set w* ⊆ *atoms r*
⟨*proof*⟩

lemma *lang-eq-ext*: (*lang r* = *lang s*) =
(∀ *w* ∈ *lists*(*atoms r* ∪ *atoms s*). *w* ∈ *lang r* ⟷ *w* ∈ *lang s*)
⟨*proof*⟩

lemma *lang-eq-ext-Nil-fold-Deriv*:
fixes *r s*
defines 𝔅 ≡ {(*fold Deriv w* (*lang r*), *fold Deriv w* (*lang s*))| *w*. *w*∈*lists* (*atoms r* ∪ *atoms s*)}
shows *lang r* = *lang s* ⟷ (∀ (*K*, *L*) ∈ 𝔅. [] ∈ *K* ⟷ [] ∈ *L*)
⟨*proof*⟩

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* ≤ *s* ≡ *le-rexp r s*
definition *less-rexp* **where** *r* < *s* ≡ *le-rexp r s* ∧ *r* ≠ *s*

lemma *le-rexp-Zero*: *le-rexp r Zero* ⟹ *r* = *Zero*

⟨*proof*⟩

lemma *le-rexp-refl*: *le-rexp r r*
⟨*proof*⟩

lemma *le-rexp-antisym*: ⟦*le-rexp r s*; *le-rexp s r*⟧ ⟹ *r* = *s*
⟨*proof*⟩

lemma *le-rexp-trans*: ⟦*le-rexp r s*; *le-rexp s t*⟧ ⟹ *le-rexp r t*
⟨*proof*⟩

instance ⟨*proof*⟩

end

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

lemma *le-rexp-total*: *le-rexp* (*r* :: *'a* :: *linorder rexp*) *s* ∨ *le-rexp s r*
⟨*proof*⟩

instance ⟨*proof*⟩

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 ∉ atoms r ⟹ nderiv x r = Zero
⟨proof⟩
```

lemma *atoms-nPlus*[*simp*]: *atoms* (*nPlus r s*) = *atoms r* ∪ *atoms s*
⟨*proof*⟩

lemma *atoms-nTimes*: *atoms* (*nTimes r s*) ⊆ *atoms r* ∪ *atoms s*
⟨*proof*⟩

lemma *atoms-norm*: *atoms* (*norm r*) ⊆ *atoms r*
⟨*proof*⟩

lemma *atoms-nderiv*: *atoms* (*nderiv a r*) ⊆ *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* ⇒ *'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*⟩

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*
⟨*proof*⟩

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*⟩

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* (λ*a*. (*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*⟩

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*⟩

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* ∪ *atoms s*
shows *ws* = [] ⟷ *lang r* = *lang s*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

Test:

lemma *check-eqv* (*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* ⇒ (*'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′*)
⟨*proof*⟩

lemma *rel-word-rel*: *rel v r* = (⋃ *w*∈*lang r*. *word-rel v w*)
⟨*proof*⟩

Soundness:

lemma *soundness*:
lang r = *lang s* ⟹ *rel v r* = *rel v s*
⟨*proof*⟩

end

6 Proving Relation (In)equalities via Regular Expressions

theory *Regexp-Method*
imports *Equivalence-Checking Relation-Interpretation*
begin

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

lemma *rel-of-regexp-rel*: *rel-of-regexp vs r* = *rel* (λ*i*. *vs* ! *i*) *r*
⟨*proof*⟩

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

lemma *rel-eqI*: *check-eqv r s* ⟹ *rel-eq* (*r*, *s*) *vs*
⟨*proof*⟩

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

⟨*ML*⟩

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 $(r \cup s\widehat{}+)\widehat{}* = (r \cup s)\widehat{}*$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *rev-compower* [*simp*]: *rev* ‘ $(A \frown\frown n) = (rev$ ‘ $A) \frown\frown n$
⟨*proof*⟩

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

7.2 Substituting characters in a language

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

lemma *subst-word-Nil* [*simp*]: *subst-word f* [] = []
⟨*proof*⟩

lemma *subst-word-singleton* [*simp*]: *subst-word f* [*x*] = *f x*
⟨*proof*⟩

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

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

lemma *subst-word-conc* [*simp*]: *subst-word f* ‘ $(A$ @@ $B) =$ *subst-word f* ‘ A @@ *subst-word f* ‘ B
⟨*proof*⟩

lemma *subst-word-compower* [*simp*]: *subst-word f ‘ (A ^^ n) = (subst-word f ‘ A) ^^ n*
⟨*proof*⟩

lemma *subst-word-star* [*simp*]: *subst-word f ‘ (star A) = star (subst-word f ‘ A)*
⟨*proof*⟩

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))*
⟨*proof*⟩

lemma *Nil-in-Suffixes-iff* [*simp*]: *[] ∈ Suffixes A ⟷ A ≠ {}*
⟨*proof*⟩

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

lemma *Suffixes-empty-iff* [*simp*]: *Suffixes A = {} ⟷ A = {}*
⟨*proof*⟩

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

lemma *Suffixes-insert*: *Suffixes (insert xs A) = set (suffixes xs) ∪ Suffixes A*
⟨*proof*⟩

lemma *Suffixes-conc* [*simp*]: *A ≠ {} ⟹ Suffixes (A @@ B) = Suffixes B ∪ (Suffixes A @@ B)*
⟨*proof*⟩

lemma *Suffixes-union* [*simp*]: *Suffixes (A ∪ B) = Suffixes A ∪ Suffixes B*
⟨*proof*⟩

lemma *Suffixes-UNION* [*simp*]: *Suffixes (⋃ (f ‘ A)) = ⋃ ((λx. Suffixes (f x)) ‘ A)*
⟨*proof*⟩

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

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

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*))
⟨*proof*⟩

lemma *Nil-in-Prefixes-iff* [*simp*]: [] ∈ *Prefixes A* ⟷ *A* ≠ {}
⟨*proof*⟩

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

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

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

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

lemma *Prefixes-conc* [*simp*]: *B* ≠ {} ⟹ *Prefixes* (*A* @@ *B*) = *Prefixes A* ∪ (*A* @@ *Prefixes B*)
⟨*proof*⟩

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

lemma *Prefixes-UNION* [*simp*]: *Prefixes* (⋃ (*f* ‘ *A*)) = ⋃ ((λ*x*. *Prefixes* (*f x*)) ‘ *A*)
⟨*proof*⟩

lemma *Prefixes-rev*: *Prefixes* (*rev* ‘ *A*) = *rev* ‘ *Suffixes A*
⟨*proof*⟩

lemma *Suffixes-rev*: *Suffixes* (*rev* ‘ *A*) = *rev* ‘ *Prefixes A*
⟨*proof*⟩

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

lemma *Prefixes-star* [*simp*]:
assumes *A* ≠ {}
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* ⇒ *'a list set* **where**
Sublists A = {*w*. ∃ *q*∈*A*. *sublist w q*}

lemma *Sublists-altdef* [*code*]: *Sublists A* = (⋃ *w*∈*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*) ∪ *Sublists A*
⟨*proof*⟩

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

lemma *Sublists-UN* [*simp*]: *Sublists* (⋃ (*f* ‘ *A*)) = ⋃ ((λ*x*. *Sublists* (*f x*)) ‘ *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* ≠ {} *B* ≠ {}
shows *Sublists* (*A* @@ *B*) = *Sublists A* ∪ *Sublists B* ∪ *Suffixes A* @@ *Prefixes B*
⟨*proof*⟩

lemma *star-not-empty* [*simp*]: *star A* ≠ {}
⟨*proof*⟩

lemma *Sublists-star*:
A ≠ {} ⟹ *Sublists* (*star A*) = *Sublists A* ∪ *Suffixes A* @@ *star A* @@ *Prefixes A*
⟨*proof*⟩

lemma *Prefixes-subset-Sublists*: *Prefixes A* ⊆ *Sublists A*
⟨*proof*⟩

lemma *Suffixes-subset-Sublists*: *Suffixes A* ⊆ *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** *Subseqs A* = (⋃ *w*∈*A*. *set* (*subseqs w*))

lemma *Subseqs-empty* [*simp*]: *Subseqs* {} = {}
⟨*proof*⟩

lemma *Subseqs-insert* [*simp*]: *Subseqs* (*insert xs A*) = *set* (*subseqs xs*) ∪ *Subseqs A*
⟨*proof*⟩

lemma *Subseqs-singleton* [*simp*]: *Subseqs* {*xs*} = *set* (*subseqs xs*)
⟨*proof*⟩

lemma *Subseqs-Un* [*simp*]: *Subseqs* (*A* ∪ *B*) = *Subseqs A* ∪ *Subseqs B*
⟨*proof*⟩

lemma *Subseqs-UNION* [*simp*]: *Subseqs* (⋃(*f* ‘ *A*)) = ⋃((λ*x*. *Subseqs* (*f x*)) ‘ *A*)
⟨*proof*⟩

lemma *Subseqs-conc* [*simp*]: *Subseqs* (*A* @@ *B*) = *Subseqs A* @@ *Subseqs B*
⟨*proof*⟩

lemma *Subseqs-compower* [*simp*]: *Subseqs* (*A* ^^ *n*) = *Subseqs A* ^^ *n*
⟨*proof*⟩

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

lemma *Sublists-subset-Subseqs*: *Sublists A* ⊆ *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*]: *lang* (*subseqs-rexp r*) = *Subseqs* (*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 Brzozowski's derivatives of regular expressions

primrec
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*)

primrec
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*
⟨*proof*⟩

lemma *atoms-derivs-subset*: *atoms* (*derivs w r*) ⊆ *atoms r*
⟨*proof*⟩

lemma *lang-deriv*: *lang* (*deriv c r*) = *Deriv c* (*lang r*)
⟨*proof*⟩

lemma *lang-derivs*: *lang* (*derivs s r*) = *Derivs s* (*lang r*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pderivs-snoc*:
shows *pderivs* (*s* @ [*c*]) *r* = *pderiv-set c* (*pderivs s r*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *pderivs-Atom*:
shows *pderivs s* (*Atom c*) ⊆ {*Atom c*, *One*}
⟨*proof*⟩

8.3 Relating left-quotients and partial derivatives

lemma *Deriv-pderiv*:

shows $Deriv\ c\ (lang\ r) = \bigcup\ (lang\ `\ pderiv\ c\ r)$
⟨*proof*⟩

lemma *Derivs-pderivs*:
shows $Derivs\ s\ (lang\ r) = \bigcup\ (lang\ `\ pderivs\ s\ r)$
⟨*proof*⟩

8.4 Relating derivatives and partial derivatives

lemma *deriv-pderiv*:
shows $\bigcup\ (lang\ `\ (pderiv\ c\ r)) = lang\ (deriv\ c\ r)$
⟨*proof*⟩

lemma *derivs-pderivs*:
shows $\bigcup\ (lang\ `\ (pderivs\ s\ r)) = lang\ (derivs\ s\ r)$
⟨*proof*⟩

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 $\bigwedge s.\ s \in A \Longrightarrow pderivs\ s\ r \subseteq C$
shows $pderivs\text{-}lang\ A\ r \subseteq C$
⟨*proof*⟩

lemma *pderivs-lang-union*:
shows $pderivs\text{-}lang\ (A \cup B)\ r = (pderivs\text{-}lang\ A\ r \cup pderivs\text{-}lang\ B\ r)$
⟨*proof*⟩

lemma *pderivs-lang-subset*:
shows $A \subseteq B \Longrightarrow pderivs\text{-}lang\ A\ r \subseteq pderivs\text{-}lang\ B\ r$
⟨*proof*⟩

definition
$UNIV1 \equiv UNIV - \{[]\}$

lemma *pderivs-lang-Zero* [*simp*]:
shows $pderivs\text{-}lang\ UNIV1\ Zero = \{\}$
⟨*proof*⟩

lemma *pderivs-lang-One* [*simp*]:
shows $pderivs\text{-}lang\ UNIV1\ One = \{\}$
⟨*proof*⟩

lemma *pderivs-lang-Atom* [*simp*]:
shows $pderivs\text{-}lang\ UNIV1\ (Atom\ c) = \{One\}$

⟨*proof*⟩

lemma *pderivs-lang-Plus* [*simp*]:
shows *pderivs-lang UNIV1* (*Plus r1 r2*) = *pderivs-lang UNIV1 r1* ∪ *pderivs-lang UNIV1 r2*
⟨*proof*⟩

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

definition
PSuf s ≡ {*v*. *v* ≠ [] ∧ (∃ *u*. *u* @ *v* = *s*)}

lemma *PSuf-snoc*:
shows *PSuf* (*s* @ [*c*]) = (*PSuf s*) @@ {[*c*]} ∪ {[*c*]}
⟨*proof*⟩

lemma *PSuf-Union*:
shows (⋃ *v* ∈ *PSuf s* @@ {[*c*]}. *f v*) = (⋃ *v* ∈ *PSuf s*. *f* (*v* @ [*c*]))
⟨*proof*⟩

lemma *pderivs-lang-snoc*:
shows *pderivs-lang* (*PSuf s* @@ {[*c*]}) *r* = (*pderiv-set c* (*pderivs-lang* (*PSuf s*) *r*))
⟨*proof*⟩

lemma *pderivs-Times*:
shows *pderivs s* (*Times r1 r2*) ⊆ *Timess* (*pderivs s r1*) *r2* ∪ (*pderivs-lang* (*PSuf s*) *r2*)
⟨*proof*⟩

lemma *pderivs-lang-Times-aux1*:
assumes *a*: *s* ∈ *UNIV1*
shows *pderivs-lang* (*PSuf s*) *r* ⊆ *pderivs-lang UNIV1 r*
⟨*proof*⟩

lemma *pderivs-lang-Times-aux2*:
assumes *a*: *s* ∈ *UNIV1*
shows *Timess* (*pderivs s r1*) *r2* ⊆ *Timess* (*pderivs-lang UNIV1 r1*) *r2*
⟨*proof*⟩

lemma *pderivs-lang-Times*:
shows *pderivs-lang UNIV1* (*Times r1 r2*) ⊆ *Timess* (*pderivs-lang UNIV1 r1*) *r2* ∪ *pderivs-lang UNIV1 r2*
⟨*proof*⟩

lemma *pderivs-Star*:
assumes *a*: *s* ≠ []
shows *pderivs s* (*Star r*) ⊆ *Timess* (*pderivs-lang* (*PSuf s*) *r*) (*Star r*)
⟨*proof*⟩

lemma *pderivs-lang-Star*:
shows *pderivs-lang UNIV1 (Star r) ⊆ Timess (pderivs-lang UNIV1 r) (Star r)*
⟨*proof*⟩

lemma *finite-Timess* [*simp*]:
assumes *a*: *finite A*
shows *finite (Timess A r)*
⟨*proof*⟩

lemma *finite-pderivs-lang-UNIV1*:
shows *finite (pderivs-lang UNIV1 r)*
⟨*proof*⟩

lemma *pderivs-lang-UNIV*:
shows *pderivs-lang UNIV r = pderivs [] r ∪ pderivs-lang UNIV1 r*
⟨*proof*⟩

lemma *finite-pderivs-lang-UNIV*:
shows *finite (pderivs-lang UNIV r)*
⟨*proof*⟩

lemma *finite-pderivs-lang*:
shows *finite (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 ⇒ 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)
⟨*proof*⟩

lemma *card-pderivs-lang-UNIV1-le-awidth*: *card (pderivs-lang UNIV1 r) ≤ awidth r*
⟨*proof*⟩

Antimirov's Theorem 3.4:

theorem *card-pderivs-lang-UNIV-le-awidth*: *card (pderivs-lang UNIV r) ≤ 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* $\Longrightarrow$ *pderiv x r* = {}
⟨*proof*⟩

lemma *Pderiv-no-occurrence*:
x $\notin$ *Atoms R* $\Longrightarrow$ *Pderiv x R* = {}
⟨*proof*⟩

lemma *Deriv-Lang*: *Deriv c* (*Lang R*) = *Lang* (*Pderiv c R*)
⟨*proof*⟩

lemma *Nullable-pderiv*[*simp*]: *Nullable*(*pderivs w r*) = (*w* : *lang r*)
⟨*proof*⟩

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

is-Bisimulation as ps =
(∀ (*R*,*S*)∈ *set ps*. *Atoms R* ∪ *Atoms S* ⊆ *set as* ∧
(*Nullable R* ⟷ *Nullable S*) ∧
(∀ *a*∈*set as*. (*Pderiv a R*, *Pderiv a S*) ∈ *set ps*))

lemma *Bisim-Lang-eq*:
assumes *Bisim*: *is-Bisimulation as ps*
assumes (*R*, *S*) ∈ *set ps*
shows *Lang R* = *Lang S*
⟨*proof*⟩

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* (λ*a*. (*Pderiv a R*, *Pderiv a S*)) *as*;
new = *filter* (λ*p*. *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*)))})
⟨*proof*⟩

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*⟩

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*⟩

Test:

lemma *check-eqv*
(*Plus One* (*Times* (*Atom 0*) (*Star*(*Atom 0*))))
(*Star*(*Atom*(*0*::*nat*)))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *Pderiv-PDERIVS*: **assumes** *R'* ⊆ *PDERIVS R* **shows** *Pderiv a R'* ⊆ *PDERIVS R*
⟨*proof*⟩

lemma *finite-PDERIVS*: *finite R* ⟹ *finite*(*PDERIVS R*)
⟨*proof*⟩

lemma *closure-Some*: **assumes** *finite R0 finite S0* **shows** ∃ *p*. *closure as* ([(*R0*,*S0*)],[])
= *Some p*
⟨*proof*⟩

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*⟩

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* ⟹ *check-eqv 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*) = (*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*
⟨*proof*⟩

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*
⟨*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* ⇒ *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*)
⟨*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 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*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *atoms-nPlus*[*simp*]: *atoms (nPlus r s) = atoms r ∪ atoms s*
⟨*proof*⟩

lemma *atoms-nTimes*: *atoms (nTimes r s) ⊆ atoms r ∪ atoms s*
⟨*proof*⟩

lemma *atoms-nInter*: *atoms (nInter r s) ⊆ atoms r ∪ atoms s*
⟨*proof*⟩

lemma *atoms-norm*: *atoms (norm r) ⊆ atoms r*
⟨*proof*⟩

lemma *atoms-nderiv*: *atoms (nderiv a r) ⊆ atoms r*
⟨*proof*⟩

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)
 ⟹ (⋀x. x:S ⟹ bisimilar (Deriv x K) (Deriv x L))
 ⟹ bisimilar K L

lemma *equal-if-bisimilar*:
assumes *K ⊆ lists S L ⊆ lists S bisimilar K L* **shows** *K = L*
⟨*proof*⟩

lemma *language-coinduct*:
fixes *R* (**infixl** *∼ 50*)
assumes *⋀K L. K ∼ L ⟹ K ⊆ lists S ∧ L ⊆ 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 $K = L$
⟨*proof*⟩

end

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

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

lemma *bisim-lang-eq*:
assumes *bisim*: *is-bisimulation as ps*
assumes (*r*, *s*) ∈ *set ps*
shows *lang* (*set as*) *r* = *lang* (*set as*) *s*
⟨*proof*⟩

11.5 Closure computation

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

fun *step* :: *nat list* ⇒ *rexp-pairs* ∗ *rexp-pairs* ⇒ *rexp-pairs* ∗ *rexp-pairs*
where *step as* (*ws*,*ps*) =
(*let*
(*r*, *s*) = *hd ws*;
ps′ = (*r*, *s*) # *ps*;
succs = *map* (λ*a*. (*nderiv a r*, *nderiv a s*)) *as*;
new = *filter* (λ*p*. *p* ∉ *set ps*′ ∪ *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*) ∧

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

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

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*
⟨*proof*⟩

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*⟩

lemma *check-eqv* [*0*] (*Plus One* (*Times* (*Atom 0*) (*Star*(*Atom 0*)))) (*Star*(*Atom 0*))
⟨*proof*⟩

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*)))))
⟨*proof*⟩

lemma *check-eqv* [*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.