Regular Expression Equivalence

Tobias Nipkow Dmitriy Traytel

February 6, 2026

Abstract

We formalize a unified framework for verified decision procedures for regular expression equivalence. Five recently published formalizations of such decision procedures (three based on derivatives, two on marked regular expressions) can be obtained as instances of the framework. We discover that the two approaches based on marked regular expressions, which were previously thought to be the same, are different, and one seems to produce uniformly smaller automata. The common framework makes it possible to compare the performance of the different decision procedures in a meaningful way.

The formalization is also described in a submitted paper draft [1].

Contents

1 Regular Expressions Equivalence Framework **2**
1.1 The overall procedure 4

2 Finiteness of Derivatives Modulo ACI **6**
2.1 ACI normalization 6
2.2 Atoms 8
2.3 Language 9
2.4 Finiteness of ACI-Equivalent Derivatives 9
2.5 Deriving preserves ACI-equivalence 12
2.6 Alternative ACI defintions 13

3 Connection Between Derivatives and Partial Derivatives **17**

4 Framework Instantiations using (Partial) Derivatives **23**
4.1 Brzozowski Derivatives Modulo ACI 23
4.2 Brzozowski Derivatives Modulo ACI Operating on the Quotient Type 24
4.3 Brzozowski Derivatives Modulo ACI++ (Only Soundness) . . 25
4.4 Partial Derivatives 25
4.5 Languages as States 26

5 Framework Instantiations using Marked Regular Expressions 27
5.1 Marked Regular Expressions 27
5.2 Mark Before Atom 29
5.3 Mark After Atom 30

6 Linear Time Optimization for "Mark After Atom" 33

7 Linear Time Optimization for "Mark Before Atom" (for a Fixed Alphabet) 36

8 Various Algorithms for Regular Expression Equivalence 41

1 Regular Expressions Equivalence Framework

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

lemma set-add-atoms: set (add-atoms r as) = atoms r ∪ set as
by (induction r arbitrary: as) auto

lemma rtrancl-fold-product:
shows {((r,s),(f a r,f a s))| r s a. a : A}^* =
      {((r,s),(fold f w r,fold f w s))| r s w. w : lists A} (is ?L = ?R)
proof−
  { fix r s r' s'
    have ((r,s),(r',s')) : ?L ⟹ ((r,s),(r',s')) : ?R
    proof(induction rule: converse-rtrancl-induct2)
      case refl show ?case by(force intro!: fold-simps(1)[symmetric])
    next
      case step thus ?case by(force intro!: fold-simps(2)[symmetric])
    qed
  } moreover
  { fix r s r' s'
    { fix w have ∀ x∈set w. x ∈ A ⟹ ((r, s), fold f w r, fold f w s) :?L
      proof(induction w rule: rev-induct)
        case Nil show ?case by simp
      next
        case snoc thus ?case by (auto elim!: rtrancl-into-rtrancl)
      qed
    }
```

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

lemma *rtrancl-fold-product1*:
shows $\{(r,s).\ \exists\, a \in A.\ s = f\ a\ r\}^{\frown *} =$
$\{(r,fold\ f\ w\ r) \mid r\ w.\ w : lists\ A\}$ (**is** *?L = ?R*)
proof−
{ fix *r s*
have $(r,s) : ?L \Longrightarrow (r,s) : ?R$
proof(*induction rule*: *converse-rtrancl-induct*)
case *base* **show** *?case* **by**(*force intro*!: *fold-simps*(*1*)[*symmetric*])
next
case *step* **thus** *?case* **by**(*force intro*!: *fold-simps*(*2*)[*symmetric*])
qed
} moreover
{ fix *r s*
{ fix *w* **have** $\forall\, x{\in}set\ w.\ x \in A \Longrightarrow (r,\ fold\ f\ w\ r) : ?L$
proof(*induction w rule*: *rev-induct*)
case *Nil* **show** *?case* **by** *simp*
next
case *snoc* **thus** *?case* **by** (*auto elim*!: *rtrancl-into-rtrancl*)
qed
}
hence $(r,s) : ?R \Longrightarrow (r,s) : ?L$ **by** *auto*
} ultimately show *?thesis* **by** (*auto simp*: *in-lists-conv-set*) *blast*
qed

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

locale *rexp-DA* =
fixes *init* :: *'a rexp* ⇒ *'s*
fixes *delta* :: *'a* ⇒ *'s* ⇒ *'s*
fixes *final* :: *'s* ⇒ *bool*
fixes *L* :: *'s* ⇒ *'a lang*
assumes *L-init*: $L\ (init\ r) = lang\ r$
assumes *L-delta*: $L(delta\ a\ s) = Deriv\ a\ (L\ s)$
assumes *final-iff-Nil*: $final\ s \longleftrightarrow [] \in L\ s$
begin

lemma *L-deltas*: $L\ (fold\ delta\ w\ s) = fold\ Deriv\ w\ (L\ s)$

by (*induction w arbitrary*: *s*) (*auto simp add*: *L-delta*)

definition *closure* :: *'a list* ⇒ *'s* ∗ *'s* ⇒ ((*'s* ∗ *'s*) *list* ∗ (*'s* ∗ *'s*) *set*) *option*
where
closure as = *rtrancl-while* (λ(*p*,*q*). *final p* = *final q*)
(λ(*p*,*q*). *map* (λ*a*. (*delta a p*, *delta a q*)) *as*)

theorem *closure-sound-complete*:
assumes *result*: *closure as* (*init r*,*init s*) = *Some*(*ws*,*R*)
and *atoms*: *set as* = *atoms r* ∪ *atoms s*
shows *ws* = [] ⟷ *lang r* = *lang s*
proof −
have *leq*: (*lang r* = *lang s*) =
(∀ (*r'*,*s'*) ∈ {((*r0*,*s0*),(*delta a r0*,*delta a s0*))| *r0 s0 a*. *a* : *set as*}^∗ `` {(*init r*,*init s*)}.
final r' = *final s'*)
by (*simp add*: *atoms rtrancl-fold-product Ball-def lang-eq-ext-Nil-fold-Deriv imp-ex*
L-deltas L-init final-iff-Nil del: *Un-iff*)
have {(*x*,*y*). *y* ∈ *set* ((λ(*p*,*q*). *map* (λ*a*. (*delta a p*, *delta a q*)) *as*) *x*)} =
{((*r*,*s*), *delta a r*, *delta a s*) |*r s a*. *a* ∈ *set as*}
by *auto*
with *atoms rtrancl-while-Some*[*OF result*[*unfolded closure-def*]]
show *?thesis* **by** (*auto simp add*: *leq Ball-def split*: *if-splits*)
qed

1.1 The overall procedure

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

lemma *soundness*:
assumes *check-eqv r s* **shows** *lang r* = *lang s*
proof −
let *?as* = *add-atoms r* (*add-atoms s* [])
obtain *R* **where** *1*: *closure ?as* (*init r*, *init s*) = *Some*([],*R*)
using *assms* **by** (*auto simp*: *check-eqv-def Let-def split*:*option.splits list.splits*)
from *closure-sound-complete*[*OF this*]
show *lang r* = *lang s* **by** (*simp add*: *set-add-atoms*)
qed

Auxiliary functions:

definition *reachable* :: *'a list* ⇒ *'a rexp* ⇒ *'s set* **where**
reachable as s =
snd(*the*(*rtrancl-while* (λ-. *True*) (λ*s*. *map* (λ*a*. *delta a s*) *as*) (*init s*)))

definition *automaton* :: *'a list* ⇒ *'a rexp* ⇒ ((*'s* ∗ *'a*) ∗ *'s*) *set* **where**
automaton as s =
snd (*the*
(*let i* = *init s*;
start = (([*i*], {*i*}), {});
test = λ((*ws*, *Z*), *A*). *ws* ≠ [];
step = λ((*ws*, *Z*), *A*).
(*let s* = *hd ws*;
new-edges = *map* (λ*a*. ((*s*, *a*), *delta a s*)) *as*;
new = *remdups* (*filter* (λ*ss*. *ss* ∉ *Z*) (*map snd new-edges*))
in ((*new* @ *tl ws*, *set new* ∪ *Z*), *set new-edges* ∪ *A*))
in while-option test step start))

definition *match* :: *'a rexp* ⇒ *'a list* ⇒ *bool* **where**
match s w = *final* (*fold delta w* (*init s*))

lemma *match-correct*: *match s w* ⟷ *w* ∈ *lang s*
unfolding *match-def*
by (*induct w arbitrary*: *s*)
(*auto simp*: *L-init L-delta in-fold-Deriv final-iff-Nil L-deltas Deriv-def*)

end

locale *rexp-DFA* = *rexp-DA* +
assumes *fin*: *finite* {*fold delta w* (*init s*) | *w*. *True*}
begin

lemma *finite-rtrancl-delta-Image*:
finite ({((*r*,*s*),(*delta a r*,*delta a s*))| *r s a*. *a* : *A*}^∗ `` {(*init r*, *init s*)})
unfolding *rtrancl-fold-product Image-singleton*
by (*auto intro*: *finite-subset*[*OF* - *finite-cartesian-product*[*OF fin fin*]])

lemma *termination*: ∃ *st*. *closure as* (*init r*,*init s*) = *Some st* (**is** ∃ -. *closure as ?i* = -)
unfolding *closure-def* **proof** (*rule rtrancl-while-finite-Some*)
show *finite* ({(*x*, *st*). *st* ∈ *set* ((λ(*p*,*q*). *map* (λ*a*. (*delta a p*, *delta a q*)) *as*) *x*)}^∗ `` {*?i*})
by (*rule finite-subset*[*OF Image-mono*[*OF rtrancl-mono*] *finite-rtrancl-delta-Image*])
auto
qed

lemma *completeness*:
assumes *lang r* = *lang s* **shows** *check-eqv r s*
proof −
let *?as* = *add-atoms r* (*add-atoms s* [])
obtain *ws R* **where** *1*: *closure ?as* (*init r*, *init s*) = *Some*(*ws*,*R*)
using *termination* **by** *fastforce*
with *closure-sound-complete*[*OF this*] *assms*

show *check-eqv r s* **by** (*simp add*: *check-eqv-def set-add-atoms*)
qed

lemma *finite-rtrancl-delta-Image1*:
finite ({(*r*,*s*). ∃ *a* ∈ *A*. *s* = *delta a r*}^* `` {*init r*})
unfolding *rtrancl-fold-product1* **by** (*auto intro*: *finite-subset*[*OF - fin*])

lemma *reachable*: *reachable as r* = {*fold delta w* (*init r*) | *w*. *w* ∈ *lists* (*set as*)}
and *finite-reachable*: *finite* (*reachable as r*)
proof −
obtain *wsZ* **where** ∗: *rtrancl-while* (λ-. *True*) (λ*s*. *map* (λ*a*. *delta a s*) *as*) (*init r*) = *Some wsZ*
by (*atomize-elim*,*intro rtrancl-while-finite-Some Image-mono rtrancl-mono finite-subset*[*OF - finite-rtrancl-delta-Image1*[*of set as r*]]) *auto*
then show *reachable as r* = {*fold delta w* (*init r*) | *w*. *w* ∈ *lists* (*set as*)}
unfolding *reachable-def* **by** (*cases wsZ*)
(*auto dest*!: *rtrancl-while-Some split*: *if-splits simp*: *rtrancl-fold-product1 image-iff*)
then show *finite* (*reachable as r*) **by** (*auto intro*: *finite-subset*[*OF - fin*])
qed

end

2 Finiteness of Derivatives Modulo ACI

Lifting constructors to lists

fun *rexp-of-list* **where**
rexp-of-list OP N [] = *N*
| *rexp-of-list OP N* [*r*] = *r*
| *rexp-of-list OP N* (*r* # *rs*) = *OP r* (*rexp-of-list OP N rs*)

abbreviation *PLUS* ≡ *rexp-of-list Plus Zero*
abbreviation *TIMES* ≡ *rexp-of-list Times One*

lemma *list-singleton-induct* [*case-names nil single cons*]:
assumes *P* [] **and** ⋀*x*. *P* [*x*] **and** ⋀*x y xs*. *P* (*y* # *xs*) ⟹ *P* (*x* # (*y* # *xs*))
shows *P xs*
using *assms* **by** *induction-schema* (*pat-completeness*, *lexicographic-order*)

2.1 ACI normalization

fun *toplevel-summands* :: *'a rexp* ⇒ *'a rexp set* **where**
toplevel-summands (*Plus r s*) = *toplevel-summands r* ∪ *toplevel-summands s*
| *toplevel-summands r* = {*r*}

abbreviation *flatten LISTOP X* ≡ *LISTOP* (*sorted-list-of-set X*)

lemma *toplevel-summands-nonempty*[*simp*]:
toplevel-summands r ≠ {}
by (*induct r*) *auto*

lemma *toplevel-summands-finite*[*simp*]:
finite (*toplevel-summands r*)
by (*induct r*) *auto*

primrec *ACI-norm* :: ('*a*::*linorder*) *rexp* ⇒ '*a rexp* (‹«-»›) **where**
«*Zero*» = *Zero*
| «*One*» = *One*
| «*Atom a*» = *Atom a*
| «*Plus r s*» = *flatten PLUS* (*toplevel-summands* (*Plus* «*r*» «*s*»))
| «*Times r s*» = *Times* «*r*» «*s*»
| «*Star r*» = *Star* «*r*»

lemma *Plus-toplevel-summands*: *Plus r s* ∈ *toplevel-summands t* ⟹ *False*
by (*induction t*) *auto*

lemma *toplevel-summands-not-Plus*[*simp*]:
(∀ *r s*. *x* ≠ *Plus r s*) ⟹ *toplevel-summands x* = {*x*}
by (*induction x*) *auto*

lemma *toplevel-summands-PLUS-strong*:
⟦*xs* ≠ []; *list-all* (λ*x*. ¬(∃ *r s*. *x* = *Plus r s*)) *xs*⟧ ⟹ *toplevel-summands* (*PLUS xs*) = *set xs*
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *toplevel-summands-flatten*:
⟦*X* ≠ {}; *finite X*; ∀ *x* ∈ *X*. ¬(∃ *r s*. *x* = *Plus r s*)⟧ ⟹ *toplevel-summands* (*flatten PLUS X*) = *X*
using *toplevel-summands-PLUS-strong*[*of sorted-list-of-set X*]
unfolding *list-all-iff* **by** *fastforce*

lemma *ACI-norm-Plus*: «*r*» = *Plus s t* ⟹ ∃ *s t*. *r* = *Plus s t*
by (*induction r*) *auto*

lemma *toplevel-summands-flatten-ACI-norm-image*:
toplevel-summands (*flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r*)) = *ACI-norm* ‘ *toplevel-summands r*
by (*intro toplevel-summands-flatten*) (*auto dest*!: *ACI-norm-Plus intro*: *Plus-toplevel-summands*)

lemma *toplevel-summands-flatten-ACI-norm-image-Union*:
toplevel-summands (*flatten PLUS* (*ACI-norm* ‘ *toplevel-summands r* ∪ *ACI-norm* ‘ *toplevel-summands s*)) =
ACI-norm ‘ *toplevel-summands r* ∪ *ACI-norm* ‘ *toplevel-summands s*
by (*intro toplevel-summands-flatten*) (*auto dest*!: *ACI-norm-Plus*[*OF sym*] *intro*: *Plus-toplevel-summands*)

lemma *toplevel-summands-ACI-norm*:
toplevel-summands «r» = ACI-norm ‘ toplevel-summands r
by (*induction r*) (*auto simp*: *toplevel-summands-flatten-ACI-norm-image-Union*)

lemma *ACI-norm-flatten*:
«r» = flatten PLUS (ACI-norm ‘ toplevel-summands r)
by (*induction r*) (*auto simp*: *image-Un toplevel-summands-flatten-ACI-norm-image*)

theorem *ACI-norm-idem*[*simp*]: *««r»» = «r»*
proof (*induct r*)
case (*Plus r s*)
have *««Plus r s»» = «flatten PLUS (toplevel-summands «r» ∪ toplevel-summands «s»)»*
(**is** - = *«flatten PLUS ?U»*) **by** *simp*
also have *... = flatten PLUS (ACI-norm ‘ toplevel-summands (flatten PLUS ?U))*
by (*simp only*: *ACI-norm-flatten*)
also have *toplevel-summands (flatten PLUS ?U) = ?U*
by (*intro toplevel-summands-flatten*) (*auto intro*: *Plus-toplevel-summands*)
also have *flatten PLUS (ACI-norm ‘ ?U) = flatten PLUS (toplevel-summands «r» ∪ toplevel-summands «s»)*
by (*simp only*: *image-Un toplevel-summands-ACI-norm*[*symmetric*] *Plus*)
finally show *?case* **by** *simp*
qed *auto*

2.2 Atoms

lemma *atoms-toplevel-summands*:
atoms s = (⋃ r∈toplevel-summands s. atoms r)
by (*induct s*) *auto*

lemma *wf-PLUS*: *atoms (PLUS xs) ⊆ Σ ⟷ (∀ r ∈ set xs. atoms r ⊆ Σ)*
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *atoms-PLUS*: *atoms (PLUS xs) = (⋃ r ∈ set xs. atoms r)*
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *atoms-flatten-PLUS*:
finite X ⟹ atoms (flatten PLUS X) = (⋃ r ∈ X. atoms r)
using *wf-PLUS*[*of sorted-list-of-set X*] **by** *auto*

theorem *atoms-ACI-norm*: *atoms «r» = atoms r*
proof (*induct r*)
case (*Plus r1 r2*) **thus** *?case*
using *atoms-toplevel-summands*[*of «r1»*] *atoms-toplevel-summands*[*of «r2»*]
by(*simp add*: *atoms-flatten-PLUS ball-Un Un-commute*)
qed *auto*

2.3 Language

lemma *toplevel-summands-lang*: $r \in$ *toplevel-summands* $s \Longrightarrow$ *lang* $r \subseteq$ *lang* s
by (*induct s*) *auto*

lemma *toplevel-summands-lang-UN*:
lang $s = (\bigcup r \in$ *toplevel-summands* s. *lang* $r)$
by (*induct s*) *auto*

lemma *toplevel-summands-in-lang*:
$w \in$ *lang* $s = (\exists\, r \in$ *toplevel-summands* s. $w \in$ *lang* $r)$
by (*induct s*) *auto*

lemma *lang-PLUS*: *lang* (*PLUS xs*) $= (\bigcup r \in$ *set xs*. *lang* $r)$
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-PLUS-map*[*simp*]:
lang (*PLUS* (*map f xs*)) $= (\bigcup a \in$ *set xs*. *lang* (*f a*))
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *lang-flatten-PLUS*[*simp*]:
finite $X \Longrightarrow$ *lang* (*flatten PLUS X*) $= (\bigcup r \in X$. *lang* $r)$
using *lang-PLUS*[*of sorted-list-of-set X*] **by** *fastforce*

theorem *lang-ACI-norm*[*simp*]: *lang* «*r*» = *lang r*
proof (*induct r*)
case (*Plus r1 r2*)
moreover
from *Plus*[*symmetric*] **have** *lang* (*Plus r1 r2*) $\subseteq$ *lang* «*Plus r1 r2*»
using *toplevel-summands-in-lang*[*of* - «*r1*»] *toplevel-summands-in-lang*[*of* - «*r2*»]
by *auto*
ultimately show *?case* **by** (*fastforce dest*!: *toplevel-summands-lang*)
qed *auto*

2.4 Finiteness of ACI-Equivalent Derivatives

lemma *toplevel-summands-deriv*:
toplevel-summands (*deriv as r*) $= (\bigcup s \in$ *toplevel-summands* r. *toplevel-summands* (*deriv as s*))
by (*induction r*) (*auto simp*: *Let-def*)

lemma *derivs-Zero*[*simp*]: *derivs xs Zero* = *Zero*
by (*induction xs*) *auto*

lemma *derivs-One*: *derivs xs One* $\in$ {*Zero*, *One*}
by (*induction xs*) *auto*

lemma *derivs-Atom*: *derivs xs* (*Atom as*) $\in$ {*Zero*, *One*, *Atom as*}
proof (*induction xs*)

case *Cons* **thus** *?case* **by** (*auto intro*: *insertE*[*OF derivs-One*])
qed *simp*

lemma *derivs-Plus*: *derivs xs* (*Plus r s*) = *Plus* (*derivs xs r*) (*derivs xs s*)
by (*induction xs arbitrary*: *r s*) *auto*

lemma *derivs-PLUS*: *derivs xs* (*PLUS ys*) = *PLUS* (*map* (*derivs xs*) *ys*)
by (*induction ys rule*: *list-singleton-induct*) (*auto simp*: *derivs-Plus*)

lemma *toplevel-summands-derivs-Times*: *toplevel-summands* (*derivs xs* (*Times r s*)) ⊆
{*Times* (*derivs xs r*) *s*} ∪
{*r'*. ∃ *ys zs*. *r'* ∈ *toplevel-summands* (*derivs ys s*) ∧ *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
proof (*induction xs arbitrary*: *r s*)
case (*Cons x xs*)
thus *?case* **by** (*auto simp*: *Let-def derivs-Plus*) (*fastforce intro*: *exI*[*of* - *x*#*xs*])+
qed *simp*

lemma *toplevel-summands-derivs-Star-nonempty*:
xs ≠ [] ⟹ *toplevel-summands* (*derivs xs* (*Star r*)) ⊆
{*Times* (*derivs ys r*) (*Star r*) | *ys*. ∃ *zs*. *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
proof (*induction xs rule*: *length-induct*)
case (*1 xs*)
then obtain *y ys* **where** *xs* = *y* # *ys* **by** (*cases xs*) *auto*
thus *?case* **using** *spec*[*OF 1*(*1*)]
by (*auto dest*!: *subsetD*[*OF toplevel-summands-derivs-Times*] *intro*: *exI*[*of* - *y*#*ys*])
(*auto elim*!: *impE dest*!: *meta-spec subsetD*)
qed

lemma *toplevel-summands-derivs-Star*:
toplevel-summands (*derivs xs* (*Star r*)) ⊆
{*Star r*} ∪ {*Times* (*derivs ys r*) (*Star r*) | *ys*. ∃ *zs*. *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
by (*cases xs* = []) (*auto dest*!: *toplevel-summands-derivs-Star-nonempty*)

lemma *toplevel-summands-PLUS*:
xs ≠ [] ⟹ *toplevel-summands* (*PLUS* (*map f xs*)) = (⋃ *r* ∈ *set xs*. *toplevel-summands* (*f r*))
by (*induction xs rule*: *list-singleton-induct*) *simp-all*

lemma *ACI-norm-toplevel-summands-Zero*: *toplevel-summands r* ⊆ {*Zero*} ⟹ «*r*» = *Zero*
by (*subst ACI-norm-flatten*) (*auto dest*: *subset-singletonD*)

lemma *finite-ACI-norm-toplevel-summands*:
finite {*f* «*s*» |*s*. *toplevel-summands s* ⊆ *B*} **if** *finite B*
proof −
have ∗: {*f* «*s*» | *s*.
toplevel-summands s ⊆ *B*} ⊆ (*f* ∘ *flatten PLUS* ∘ (‘) *ACI-norm*) ‘ *Pow B*

```
    by (subst ACI-norm-flatten) auto
  with that show ?thesis
    by (rule finite-surj [OF iffD2 [OF finite-Pow-iff]])
qed

theorem finite-derivs: finite {«derivs xs r» | xs . True}
proof (induct r)
  case Zero show ?case by simp
next
  case One show ?case
   by (rule finite-surj[of {Zero, One}]) (blast intro: insertE[OF derivs-One])+
next
  case (Atom as) show ?case
     by (rule finite-surj[of {Zero, One, Atom as}]) (blast intro: insertE[OF de-
rivs-Atom])+
next
  case (Plus r s)
 show ?case by (auto simp: derivs-Plus intro!: finite-surj[OF finite-cartesian-product[OF
Plus]])
next
  case (Times r s)
  hence finite (⋃ (toplevel-summands ‘ {«derivs xs s» | xs . True})) by auto
  moreover have {«r′» |r′. ∃ ys. r′ ∈ toplevel-summands (derivs ys s)} =
    {r′. ∃ ys. r′ ∈ toplevel-summands «derivs ys s»}
    by (auto simp: toplevel-summands-ACI-norm)
  ultimately have fin: finite {«r′» |r′. ∃ ys. r′ ∈ toplevel-summands (derivs ys s)}
    by (fastforce intro: finite-subset[of - ⋃ (toplevel-summands ‘ {«derivs xs s» |
xs . True})])
 let ?X = λxs. {Times (derivs ys r) s | ys. True} ∪ {r′. r′ ∈ (⋃ ys. toplevel-summands
(derivs ys s))}
  show ?case
  proof (simp only: ACI-norm-flatten,
      rule finite-surj[of {X. ∃ xs. X ⊆ ACI-norm ‘ ?X xs} - flatten PLUS])
    show finite {X. ∃ xs. X ⊆ ACI-norm ‘ ?X xs}
       using fin by (fastforce simp: image-Un elim: finite-subset[rotated] intro:
finite-surj[OF Times(1), of - λr. Times r «s»])
 qed (fastforce dest!: subsetD[OF toplevel-summands-derivs-Times] intro!: imageI)
next
  case (Star r)
  let ?f = λf r′. Times r′ (Star (f r))
  let ?X = {Star r} ∪ ?f id ‘ {r′. r′ ∈ {derivs ys r|ys. True}}
  show ?case
  proof (simp only: ACI-norm-flatten,
      rule finite-surj[of {X. X ⊆ ACI-norm ‘ ?X} - flatten PLUS])
    have ∗: ⋀X. ACI-norm ‘ ?f (λx. x) ‘ X = ?f ACI-norm ‘ ACI-norm ‘ X by
(auto simp: image-def)
    show finite {X. X ⊆ ACI-norm ‘ ?X}
      by (rule finite-Collect-subsets)
        (auto simp: ∗ intro!: finite-imageI[of - ?f ACI-norm] intro: finite-subset[OF
```

- *Star*])
qed (*fastforce dest*!: *subsetD*[*OF toplevel-summands-derivs-Star*] *intro*!: *imageI*)
qed

2.5 Deriving preserves ACI-equivalence

lemma *ACI-norm-PLUS*:
list-all2 (λ*r s*. «*r*» = «*s*») *xs ys* ⟹ «*PLUS xs*» = «*PLUS ys*»
proof (*induct rule*: *list-all2-induct*)
case (*Cons x xs y ys*)
hence *length xs* = *length ys* **by** (*elim list-all2-lengthD*)
thus *?case* **using** *Cons* **by** (*induct xs ys rule*: *list-induct2*) *auto*
qed *simp*

lemma *toplevel-summands-ACI-norm-deriv*:
(⋃ *a*∈*toplevel-summands r*. *toplevel-summands* «*deriv as* «*a*»») = *toplevel-summands* «*deriv as* «*r*»»
proof (*induct r*)
case (*Plus r1 r2*) **thus** *?case*
unfolding *toplevel-summands.simps toplevel-summands-ACI-norm*
toplevel-summands-deriv[*of as* «*Plus r1 r2*»] *image-Un Union-Un-distrib*
by (*simp add*: *image-UN*)
qed (*auto simp*: *Let-def*)

lemma *toplevel-summands-nullable*:
nullable s = (∃ *r*∈*toplevel-summands s*. *nullable r*)
by (*induction s*) *auto*

lemma *nullable-PLUS*:
nullable (*PLUS xs*) = (∃ *r* ∈ *set xs*. *nullable r*)
by (*induction xs rule*: *list-singleton-induct*) *auto*

theorem *ACI-norm-nullable*: *nullable* «*r*» = *nullable r*
proof (*induction r*)
case (*Plus r1 r2*) **thus** *?case* **using** *toplevel-summands-nullable*
by (*auto simp*: *nullable-PLUS*)
qed *auto*

theorem *ACI-norm-deriv*: «*deriv as* «*r*»» = «*deriv as r*»
proof (*induction r arbitrary*: *as*)
case (*Plus r1 r2*) **thus** *?case*
unfolding *deriv.simps ACI-norm-flatten*[*of deriv as* «*Plus r1 r2*»]
toplevel-summands-deriv[*of as* «*Plus r1 r2*»] *image-Un image-UN*
by (*auto simp*: *toplevel-summands-ACI-norm toplevel-summands-flatten-ACI-norm-image-Union*)
(*auto simp*: *toplevel-summands-ACI-norm*[*symmetric*] *toplevel-summands-ACI-norm-deriv*)
qed (*simp-all add*: *ACI-norm-nullable*)

corollary *deriv-preserves*: «*r*» = «*s*» ⟹ «*deriv as r*» = «*deriv as s*»

by (*rule box-equals*[*OF - ACI-norm-deriv ACI-norm-deriv*]) (*erule arg-cong*)

lemma *derivs-snoc*[*simp*]: *derivs* (*xs* @ [*x*]) *r* = (*deriv x* (*derivs xs r*))
by (*induction xs arbitrary*: *r*) *auto*

theorem *ACI-norm-derivs*: «*derivs xs* «*r*»» = «*derivs xs r*»
proof (*induction xs arbitrary*: *r rule*: *rev-induct*)
case (*snoc x xs*) **thus** *?case*
using *ACI-norm-deriv*[*of x derivs xs r*] *ACI-norm-deriv*[*of x derivs xs* «*r*»] **by**
auto
qed *simp*

2.6 Alternative ACI defintions

Not necessary but conceptually nicer (and seems also to be faster?!)

fun *ACI-nPlus* :: *'a*::*linorder rexp* ⇒ *'a rexp* ⇒ *'a rexp*
where
ACI-nPlus (*Plus r1 r2*) *s* = *ACI-nPlus r1* (*ACI-nPlus r2 s*)
| *ACI-nPlus r* (*Plus s1 s2*) =
(*if r* = *s1 then Plus s1 s2*
else if r < *s1 then Plus r* (*Plus s1 s2*)
else Plus s1 (*ACI-nPlus r s2*))
| *ACI-nPlus r s* =
(*if r* = *s then r*
else if r < *s then Plus r s*
else Plus s r)

primrec *ACI-norm-alt* **where**
ACI-norm-alt Zero = *Zero*
| *ACI-norm-alt One* = *One*
| *ACI-norm-alt* (*Atom a*) = *Atom a*
| *ACI-norm-alt* (*Plus r s*) = *ACI-nPlus* (*ACI-norm-alt r*) (*ACI-norm-alt s*)
| *ACI-norm-alt* (*Times r s*) = *Times* (*ACI-norm-alt r*) (*ACI-norm-alt s*)
| *ACI-norm-alt* (*Star r*) = *Star* (*ACI-norm-alt r*)

lemma *toplevel-summands-ACI-nPlus*:
toplevel-summands (*ACI-nPlus r s*) = *toplevel-summands* (*Plus r s*)
by (*induct r s rule*: *ACI-nPlus.induct*) *auto*

lemma *toplevel-summands-ACI-norm-alt*:
toplevel-summands (*ACI-norm-alt r*) = *ACI-norm-alt* ‘ *toplevel-summands r*
by (*induct r*) (*auto simp*: *toplevel-summands-ACI-nPlus*)

lemma *ACI-norm-alt-Plus*:
ACI-norm-alt r = *Plus s t* ⟹ ∃ *s t*. *r* = *Plus s t*
by (*induct r*) *auto*

lemma *toplevel-summands-flatten-ACI-norm-alt-image*:
toplevel-summands (*flatten PLUS* (*ACI-norm-alt* ‘ *toplevel-summands r*)) = *ACI-norm-alt*

‘ *toplevel-summands r*
by (*intro toplevel-summands-flatten*) (*auto dest*!: *ACI-norm-alt-Plus intro*: *Plus-toplevel-summands*)

lemma *ACI-norm-ACI-norm-alt*: «*ACI-norm-alt r*» = «*r*»
proof (*induction r*)
case (*Plus r s*) **show** *?case*
using *ACI-norm-flatten* [*of r*] *ACI-norm-flatten* [*of s*]
by (*auto simp add*: *toplevel-summands-ACI-nPlus*)
(*metis ACI-norm-flatten Plus.IH*(*1*) *Plus.IH*(*2*) *image-Un toplevel-summands.simps*(*1*)
toplevel-summands-ACI-nPlus toplevel-summands-ACI-norm)
qed *auto*

lemma *ACI-nPlus-singleton-PLUS*:
⟦*xs* ≠ []; *sorted xs*; *distinct xs*; ∀ *x* ∈ {*x*} ∪ *set xs*. ¬(∃ *r s*. *x* = *Plus r s*)⟧ ⟹
ACI-nPlus x (*PLUS xs*) = (*if x* ∈ *set xs then PLUS xs else PLUS* (*insort x xs*))
proof (*induct xs rule*: *list-singleton-induct*)
case (*single y*)
thus *?case*
by (*cases x y rule*: *linorder-cases*) (*induct x y rule*: *ACI-nPlus.induct*, *auto*)+
next
case (*cons y1 y2 ys*) **thus** *?case* **by** (*cases x*) (*auto*)
qed *simp*

lemma *ACI-nPlus-PLUS*:
⟦*xs1* ≠ []; *xs2* ≠ []; ∀ *x* ∈ *set* (*xs1* @ *xs2*). ¬(∃ *r s*. *x* = *Plus r s*); *sorted xs2*;
distinct xs2⟧⟹
ACI-nPlus (*PLUS xs1*) (*PLUS xs2*) = *flatten PLUS* (*set* (*xs1* @ *xs2*))
proof (*induct xs1 arbitrary*: *xs2 rule*: *list-singleton-induct*)
case (*single x1*)
thus *?case*
apply (*auto intro*!: *trans*[*OF ACI-nPlus-singleton-PLUS*] *simp del*: *sorted-list-of-set-insert-remove*)
apply (*simp only*: *insert-absorb*)
apply (*metis List.finite-set finite-sorted-distinct-unique sorted-list-of-set*)
apply (*rule arg-cong*[*of - - PLUS*])
apply (*metis remdups-id-iff-distinct sorted-list-of-set-sort-remdups sorted-sort-id*)
done
next
case (*cons x11 x12 xs1*) **thus** *?case*
apply (*simp del*: *sorted-list-of-set-insert-remove*)
apply (*rule trans*[*OF ACI-nPlus-singleton-PLUS*])
apply (*auto simp del*: *sorted-list-of-set-insert-remove simp add*: *insert-commute*[*of x11*])
apply (*auto simp only*: *Un-insert-left*[*of x11*, *symmetric*] *insert-absorb*) []
apply (*auto simp only*: *Un-insert-right*[*of - x11*, *symmetric*] *insert-absorb*) []
apply (*auto simp add*: *insert-commute*[*of x12*])
done
qed *simp*

lemma *ACI-nPlus-flatten-PLUS*:

```
  ⟦X1 ≠ {}; X2 ≠ {}; finite X1; finite X2; ∀ x ∈ X1 ∪ X2. ¬(∃ r s. x = Plus r
s)⟧⟹
  ACI-nPlus (flatten PLUS X1) (flatten PLUS X2) = flatten PLUS (X1 ∪ X2)
  by (rule trans[OF ACI-nPlus-PLUS]) auto

lemma ACI-nPlus-ACI-norm [simp]: ACI-nPlus «r» «s» = «Plus r s»
  by (auto simp: image-Un Un-assoc ACI-norm-flatten [of r] ACI-norm-flatten [of
s] ACI-norm-flatten [of Plus r s]
    toplevel-summands-flatten-ACI-norm-image
    intro!: trans [OF ACI-nPlus-flatten-PLUS])
    (metis ACI-norm-Plus Plus-toplevel-summands)+

lemma ACI-norm-alt:
  ACI-norm-alt r = «r»
  by (induct r) auto

declare ACI-norm-alt[symmetric, code]

inductive ACI where
  ACI-refl:       ACI r r |
  ACI-sym:        ACI r s ⟹ ACI s r |
  ACI-trans:      ACI r s ⟹ ACI s t ⟹ ACI r t |
  ACI-Plus-cong:  ⟦ACI r1 s1; ACI r2 s2⟧ ⟹ ACI (Plus r1 r2) (Plus s1 s2) |
  ACI-Times-cong: ⟦ACI r1 s1; ACI r2 s2⟧ ⟹ ACI (Times r1 r2) (Times s1 s2)
|
  ACI-Star-cong:  ACI r s ⟹ ACI (Star r) (Star s) |
  ACI-assoc:      ACI (Plus (Plus r s) t) (Plus r (Plus s t)) |
  ACI-comm:       ACI (Plus r s) (Plus s r) |
  ACI-idem:       ACI (Plus r r) r

lemma ACI-atoms: ACI r s ⟹ atoms r = atoms s
  by (induct rule: ACI.induct) auto

lemma ACI-nullable: ACI r s ⟹ nullable r = nullable s
  by (induct rule: ACI.induct) auto

lemma ACI-lang: ACI r s ⟹ lang r = lang s
  by (induct rule: ACI.induct) auto

lemma ACI-deriv: ACI r s ⟹ ACI (deriv a r) (deriv a s)
proof (induct arbitrary: a rule: ACI.induct)
  case (ACI-Times-cong r1 s1 r2 s2) thus ?case
    by (auto simp: Let-def intro: ACI.intros dest: ACI-nullable)
      (metis ACI.ACI-Times-cong ACI-Plus-cong)
qed (auto intro: ACI.intros)

lemma ACI-Plus-assocI[intro]:
  ACI (Plus r1 r2) s2 ⟹ ACI (Plus r1 (Plus s1 r2)) (Plus s1 s2)
  ACI (Plus r1 r2) s2 ⟹ ACI (Plus r1 (Plus r2 s1)) (Plus s1 s2)
```

by (*metis ACI-assoc ACI-comm ACI-Plus-cong ACI-refl ACI-trans*)+

lemma *ACI-Plus-idemI*[*intro*]: ⟦*ACI r s1*; *ACI r s2*⟧ ⟹ *ACI r* (*Plus s1 s2*)
by (*metis ACI-Plus-cong ACI-idem ACI-sym ACI-trans*)

lemma *ACI-Plus-idemI'*[*intro*]:
⟦*ACI r1 s1*; *ACI* (*Plus r1 r2*) *s2*⟧ ⟹ *ACI* (*Plus r1 r2*) (*Plus s1 s2*)
by (*rule ACI-trans*[*OF ACI-Plus-cong*[*OF ACI-sym*[*OF ACI-idem*] *ACI-refl*]
ACI-trans[*OF ACI-assoc ACI-trans*[*OF ACI-Plus-cong ACI-refl*]]])

lemma *ACI-ACI-nPlus*: ⟦*ACI r1 s1*; *ACI r2 s2*⟧ ⟹ *ACI* (*ACI-nPlus r1 r2*) (*Plus s1 s2*)
proof (*induct r1 r2 arbitrary*: *s1 s2 rule*: *ACI-nPlus.induct*)
case *1*
from *1*(*2*)[*OF ACI-refl 1*(*1*)[*OF ACI-refl 1*(*4*)]] *1*(*3*) **show** *?case* **by** (*auto intro*: *ACI-comm ACI-trans*)
next
case (*2-1 r1 r2*)
with *ACI-Plus-cong*[*OF ACI-refl 2-1*(*1*)[*OF - - 2-1*(*2*) *ACI-refl*], *of r1*]
show *?case* **by** (*auto intro*: *ACI.intros*)
next
case (*2-2 r1 r2*)
with *ACI-Plus-cong*[*OF ACI-refl 2-2*(*1*)[*OF - - 2-2*(*2*) *ACI-refl*], *of r1*]
show *?case* **by** (*auto intro*: *ACI.intros*)
next
case (*2-3 - r1 r2*)
with *ACI-Plus-cong*[*OF ACI-refl 2-3*(*1*)[*OF - - 2-3*(*2*) *ACI-refl*], *of r1*]
show *?case* **by** (*auto intro*: *ACI.intros*)
next
case (*2-4 - - r1 r2*)
with *ACI-Plus-cong*[*OF ACI-refl 2-4*(*1*)[*OF - - 2-4*(*2*) *ACI-refl*], *of r1*]
show *?case* **by** (*auto intro*: *ACI.intros*)
next
case (*2-5 - r1 r2*)
with *ACI-Plus-cong*[*OF ACI-refl 2-5*(*1*)[*OF - - 2-5*(*2*) *ACI-refl*], *of r1*]
show *?case* **by** (*auto intro*: *ACI.intros*)
qed (*auto intro*: *ACI.intros*)

lemma *ACI-ACI-norm*: *ACI* «*r*» *r*
unfolding *ACI-norm-alt*[*symmetric*]
by (*induct r*) (*auto intro*: *ACI.intros simp*: *ACI-ACI-nPlus*)

lemma *ACI-norm-eqI*: *ACI r s* ⟹ «*r*» = «*s*»
by (*induct rule*: *ACI.induct*) (*auto simp*: *toplevel-summands-ACI-norm ACI-norm-flatten*[*symmetric*]
toplevel-summands-flatten-ACI-norm-image-Union ac-simps)

lemma *ACI-I*: «*r*» = «*s*» ⟹ *ACI r s*
by (*metis ACI-ACI-norm ACI-sym ACI-trans*)

lemma *ACI-decidable*: *ACI r s* = (*«r»* = *«s»*)
by (*metis ACI-I ACI-norm-eqI*)

3 Connection Between Derivatives and Partial Derivatives

lemma *pderiv-not-is-Zero-is-Plus*[*simp*]: ∀ *x* ∈ *pderiv a r*. ¬ *is-Zero x* ∧ ¬ *is-Plus x*
by (*induct r*) *auto*

lemma *finite-pderiv*[*simp*]: *finite* (*pderiv a r*)
by (*induct r*) *auto*

lemma *PLUS-inject*: ⟦∀ *x* ∈ *set xs* ∪ *set ys*. ¬ *is-Zero x* ∧ ¬ *is-Plus x*; *sorted xs*; *sorted ys*⟧ ⟹
(*PLUS xs* = *PLUS ys*) ⟷ *xs* = *ys*
proof (*induct xs arbitrary*: *ys rule*: *list-singleton-induct*)
case *nil* **then show** *?case* **by** (*induct ys rule*: *list-singleton-induct*) *auto*
next
case *single* **then show** *?case* **by** (*induct ys rule*: *list-singleton-induct*) *auto*
next
case *cons* **then show** *?case* **by** (*induct ys rule*: *list-singleton-induct*) *auto*
qed

lemma *sorted-list-of-set-inject*: ⟦*finite R*; *finite S*⟧ ⟹
(*sorted-list-of-set R* = *sorted-list-of-set S*) ⟷ *R* = *S*
proof (*induct R arbitrary*: *S rule*: *finite-linorder-min-induct*)
case *empty* **then show** *?case*
proof (*induct S rule*: *finite-linorder-min-induct*)
case (*insert b S*) **then show** *?case* **by** *simp* (*metis insort-not-Nil*)
qed *simp*
next
case (*insert a R*) **from** *this*(*4*,*1*−*3*) **show** *?case*
proof (*induct S rule*: *finite-linorder-min-induct*)
case (*insert b S*)
show *?case*
proof
assume *sorted-list-of-set* (*insert a R*) = *sorted-list-of-set* (*insert b S*)
with *insert*(*1*,*2*,*4*,*5*) **have** *insort a* (*sorted-list-of-set R*) = *insort b* (*sorted-list-of-set S*)
by *fastforce*
with *insert*(*2*,*5*) **have** *a* # *sorted-list-of-set R* = *b* # *sorted-list-of-set S*
apply (*cases sorted-list-of-set R sorted-list-of-set S rule*: *list.exhaust*[*case-product list.exhaust*])
apply (*auto split*: *if-splits simp add*: *not-le*)
using *insort-not-Nil* **apply** *metis*

using *insert.prems*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.prems*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.prems*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.hyps*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.hyps*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.hyps*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.hyps*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
using *insert.hyps*(*1*) *set-sorted-list-of-set* **apply** *fastforce*
done
with *insert* **show** *insert a R* = *insert b S* **by** *auto*
next
assume *insert a R* = *insert b S*
then show *sorted-list-of-set* (*insert a R*) = *sorted-list-of-set* (*insert b S*) **by** *simp*
qed
qed *simp*
qed

lemma *flatten-PLUS-inject*: ⟦∀ *x* ∈ *R* ∪ *S*. ¬ *is-Zero x* ∧ ¬ *is-Plus x*; *finite R*; *finite S*⟧ ⟹
(*flatten PLUS R* = *flatten PLUS S*) = (*R* = *S*)
by (*rule trans*[*OF PLUS-inject sorted-list-of-set-inject*]) *auto*

primrec *pset* **where**
pset Zero = {}
| *pset One* = {*One*}
| *pset* (*Atom a*) = {*Atom a*}
| *pset* (*Plus r s*) = *pset r* ∪ *pset s*
| *pset* (*Times r s*) = *Timess* (*pset r*) *s*
| *pset* (*Star r*) = {*Star r*}

lemma *pset-not-is-Zero-is-Plus*[*simp*]: ∀ *x* ∈ *pset r*. ¬ *is-Zero x* ∧ ¬ *is-Plus x*
by (*induct r*) *auto*

lemma *finite-pset*[*simp*]: *finite* (*pset r*)
by (*induct r*) *auto*

lemma *pset-deriv*: *pset* (*deriv a r*) = *pderiv a r*
by (*induct r*) *auto*

definition *pnorm* **where**
pnorm = *flatten PLUS o pset*

lemma *pnorm-deriv-eq-iff-pderiv-eq*:
pnorm (*deriv a r*) = *pnorm* (*deriv a s*) ⟷ *pderiv a r* = *pderiv a s*
unfolding *pnorm-def o-apply pset-deriv*
by (*rule flatten-PLUS-inject*) *auto*

fun *pnPlus* :: *'a*::*linorder rexp* ⇒ *'a rexp* ⇒ *'a rexp* **where**

```
  pnPlus Zero r = r
| pnPlus r Zero = r
| pnPlus (Plus r s) t = pnPlus r (pnPlus s t)
| pnPlus 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 (pnPlus r t))
| pnPlus r s =
    (if r = s then r
     else if le-rexp r s then Plus r s
     else Plus s r)

fun pnTimes :: 'a::linorder rexp ⇒ 'a rexp ⇒ 'a rexp where
  pnTimes Zero r = Zero
| pnTimes (Plus r s) t = pnPlus (pnTimes r t) (pnTimes s t)
| pnTimes r s = Times r s

primrec pnorm-alt :: 'a::linorder rexp ⇒ 'a rexp where
  pnorm-alt Zero = Zero
| pnorm-alt One = One
| pnorm-alt (Atom a) = Atom a
| pnorm-alt (Plus r s) = pnPlus (pnorm-alt r) (pnorm-alt s)
| pnorm-alt (Times r s) = pnTimes (pnorm-alt r) s
| pnorm-alt (Star r) = Star r

lemma pset-pnPlus:
  pset (pnPlus r s) = pset (Plus r s)
  by (induct r s rule: pnPlus.induct) auto

lemma pset-pnTimes:
  pset (pnTimes r s) = pset (Times r s)
  by (induct r s rule: pnTimes.induct) (auto simp: pset-pnPlus)

lemma pset-pnorm-alt-Times: s ∈ pset r ⟹ pnTimes (pnorm-alt s) t = Times
(pnorm-alt s) t
  by (induct r arbitrary: s t) auto

lemma pset-pnorm-alt:
  pset (pnorm-alt r) = pnorm-alt ‘ pset r
  by (induct r) (auto simp: pset-pnPlus pset-pnTimes pset-pnorm-alt-Times im-
age-iff)

lemma pset-pnTimes-Times: s ∈ pset r ⟹ pnTimes s t = Times s t
  by (induct r arbitrary: s t) auto

lemma pset-pnorm-alt-id: s ∈ pset r ⟹ pnorm-alt s = s
  by (induct r arbitrary: s) (auto simp: pset-pnTimes-Times)

lemma pnorm-alt-image-pset: pnorm-alt ‘ pset r = pset r
```

by (*induction r*) (*auto*, *auto simp add*: *pset-pnorm-alt-id pset-pnTimes-Times image-iff*)

lemma *pnorm-pnorm-alt*: *pnorm* (*pnorm-alt r*) = *pnorm r*
by (*induct r*) (*auto simp*: *pnorm-def pset-pnPlus pset-pnTimes pset-pnorm-alt pnorm-alt-image-pset*)

lemma *pnPlus-singleton-PLUS*:
⟦*xs* ≠ []; *sorted xs*; *distinct xs*; ∀ *x* ∈ {*x*} ∪ *set xs*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧ ⟹
pnPlus x (*PLUS xs*) = (*if x* ∈ *set xs then PLUS xs else PLUS* (*insort x xs*))
proof (*induct xs rule*: *list-singleton-induct*)
case (*single y*)
thus *?case* **unfolding** *is-Zero-def is-Plus-def*
apply (*cases x y rule*: *linorder-cases*)
apply (*induct x y rule*: *pnPlus.induct*)
apply (*auto simp*: *less-rexp-def less-eq-rexp-def*)
apply (*cases y*)
apply *auto*
apply (*induct x y rule*: *pnPlus.induct*)
apply (*auto simp*: *less-rexp-def less-eq-rexp-def*)
apply (*induct x y rule*: *pnPlus.induct*)
apply (*auto simp*: *less-rexp-def less-eq-rexp-def*)
done
next
case (*cons y1 y2 ys*) **thus** *?case* **unfolding** *is-Zero-def is-Plus-def*
apply (*cases x*)
apply (*metis UnCI insertI1*)
apply *simp* **apply** (*metis antisym less-eq-rexp-def*)
apply *simp* **apply** (*metis antisym less-eq-rexp-def*)
apply (*metis UnCI insertI1*)
apply *simp* **apply** (*metis antisym less-eq-rexp-def*)
apply *simp* **apply** (*metis antisym less-eq-rexp-def*)
done
qed *simp*

lemma *pnPlus-PlusL*[*simp*]: *t* ≠ *Zero* ⟹ *pnPlus* (*Plus r s*) *t* = *pnPlus r* (*pnPlus s t*)
by (*induct t*) *auto*

lemma *pnPlus-ZeroR*[*simp*]: *pnPlus r Zero* = *r*
by (*induct r*) *auto*

lemma *PLUS-eq-Zero*: *PLUS xs* = *Zero* ⟷ *xs* = [] ∨ *xs* = [*Zero*]
by (*induct xs rule*: *list-singleton-induct*) *auto*

lemma *pnPlus-PLUS*:
⟦*xs1* ≠ []; *xs2* ≠ []; ∀ *x* ∈ *set* (*xs1* @ *xs2*). ¬*is-Zero x* ∧ ¬*is-Plus x*; *sorted xs2*; *distinct xs2*⟧⟹
pnPlus (*PLUS xs1*) (*PLUS xs2*) = *flatten PLUS* (*set* (*xs1* @ *xs2*))

proof (*induct xs1 arbitrary*: *xs2 rule*: *list-singleton-induct*)
case (*single x1*)
thus *?case*
apply (*auto intro*!: *trans*[*OF pnPlus-singleton-PLUS*]
simp: *insert-absorb simp del*: *sorted-list-of-set-insert-remove*)
apply (*metis List.finite-set finite-sorted-distinct-unique sorted-list-of-set*)
apply (*rule arg-cong*[*of - - PLUS*])
apply (*metis remdups-id-iff-distinct sorted-list-of-set-sort-remdups sorted-sort-id*)
done
next
case (*cons x11 x12 xs1*)
then show *?case* **unfolding** *rexp-of-list.simps*
apply (*subst pnPlus-PlusL*)
apply (*unfold PLUS-eq-Zero*) []
apply (*metis in-set-conv-decomp rexp.disc*(*1*))
apply (*subst cons*(*1*))
apply (*simp-all del*: *sorted-list-of-set-insert-remove*)
apply (*rule trans*[*OF pnPlus-singleton-PLUS*])
apply (*simp-all add*: *sorted-insort set-insort-key del*: *sorted-list-of-set-insert-remove*)
apply *safe*
unfolding *insert-commute*[*of x11*]
apply (*auto simp only*: *Un-insert-left*[*of x11*, *symmetric*] *insert-absorb*) []
apply (*auto simp only*: *Un-insert-right*[*of - x11*, *symmetric*] *insert-absorb*) []
done
qed *simp*

lemma *pnPlus-flatten-PLUS*:
⟦*X1* ≠ {}; *X2* ≠ {}; *finite X1*; *finite X2*; ∀ *x* ∈ *X1* ∪ *X2*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧⟹
pnPlus (*flatten PLUS X1*) (*flatten PLUS X2*) = *flatten PLUS* (*X1* ∪ *X2*)
by (*rule trans*[*OF pnPlus-PLUS*]) *auto*

lemma *pnPlus-pnorm*: *pnPlus* (*pnorm r*) (*pnorm s*) = *pnorm* (*Plus r s*)
by (*cases pset r* = {} ∨ *pset s* = {})
(*auto simp*: *pnorm-def pset-pnPlus pset-pnorm-alt intro*: *pnPlus-flatten-PLUS*)

lemma *pnTimes-not-Zero-or-Plus*[*simp*]: ⟦¬ *is-Zero x*; ¬ *is-Plus x*⟧ ⟹ *pnTimes x r* = *Times x r*
by (*cases x*) *auto*

lemma *pnTimes-PLUS*:
⟦*xs* ≠ []; ∀ *x* ∈ *set xs*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧⟹
pnTimes (*PLUS xs*) *r* = *flatten PLUS* (*Timess* (*set xs*) *r*)
proof (*induct xs arbitrary*: *r rule*: *list-singleton-induct*)
case (*cons x y xs*) **then show** *?case* **unfolding** *rexp-of-list.simps pnTimes.simps*
apply (*subst pnTimes-not-Zero-or-Plus*)
apply (*simp-all add*: *sorted-insort set-insort-key del*: *sorted-list-of-set-insert-remove*)
apply (*subst pnPlus-singleton-PLUS*)
apply (*simp-all add*: *sorted-insort set-insort-key del*: *sorted-list-of-set-insert-remove*)

unfolding *insert-commute*[*of Times y r*]
apply (*simp del*: *sorted-list-of-set-insert-remove*)
apply *safe*
apply (*subst insert-absorb*[*of Times x r*])
apply *simp-all*
done
qed *auto*

lemma *pnTimes-flatten-PLUS*:
⟦*X1* ≠ {}; *finite X1*; ∀ *x* ∈ *X1*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧⟹
pnTimes (*flatten PLUS X1*) *r* = *flatten PLUS* (*Timess X1 r*)
by (*rule trans*[*OF pnTimes-PLUS*]) *auto*

lemma *pnTimes-pnorm*: *pnTimes* (*pnorm r1*) *r2* = *pnorm* (*Times r1 r2*)
by (*cases pset r1* = {})
(*auto simp*: *pnorm-def pset-pnTimes pset-pnorm-alt intro*: *pnTimes-flatten-PLUS*)

lemma *pnorm-alt*[*symmetric*]: *pnorm-alt r* = *pnorm r*
by (*induct r*) (*simp-all only*: *pnorm-alt.simps pnPlus-pnorm pnTimes-pnorm*, *auto simp*: *pnorm-def*)

lemma *insort-eq-Cons*: ⟦∀ *a* ∈ *set xs*. *b* < *a*; *sorted xs*⟧ ⟹ *insort b xs* = *b* # *xs*
by (*cases xs*) *auto*

lemma *pderiv-PLUS*: *pderiv a* (*PLUS* (*x* # *xs*)) = *pderiv a x* ∪ *pderiv a* (*PLUS xs*)
by (*cases xs*) *auto*

lemma *pderiv-set-flatten-PLUS*:
finite X ⟹ *pderiv* (*a* :: *'a* :: *linorder*) (*flatten PLUS X*) = *pderiv-set a X*
proof (*induction X rule*: *finite-linorder-min-induct*)
case (*insert b X*)
then have *b* ∉ *X* **by** *auto*
then have *pderiv a* (*flatten PLUS* (*insert b X*)) = *pderiv a b* ∪ *pderiv a* (*flatten PLUS X*)
by (*simp add*: *pderiv-PLUS insort-eq-Cons insert.hyps*)
also from *insert.IH* **have** ... = *pderiv a b* ∪ *pderiv-set a X* **by** *simp*
finally show *?case* **by** *simp*
qed *simp*

lemma *fold-pderiv-set-flatten-PLUS*:
⟦*w* ≠ []; *finite X*⟧ ⟹ *fold pderiv-set w* {*flatten PLUS X*} = *fold pderiv-set w X*
by (*induct w arbitrary*: *X*) (*simp-all add*: *pderiv-set-flatten-PLUS*)

lemma *fold-pnorm-deriv*:
fold (λ*a r*. *pnorm* (*deriv a r*)) *w s* = *flatten PLUS* (*fold pderiv-set w* {*s*})
proof (*induction w arbitrary*: *s*)
case (*Cons x w*) **then show** *?case*
proof (*cases w* = [])

```
    case False
    show ?thesis using fold-pderiv-set-flatten-PLUS[OF False] Cons.IH
      by (auto simp: pnorm-def pset-deriv)
  qed (simp add: pnorm-def pset-deriv)
qed simp

primrec
  pnderiv :: 'a :: linorder ⇒ 'a rexp ⇒ 'a rexp
where
  pnderiv c (Zero) = Zero
| pnderiv c (One) = Zero
| pnderiv c (Atom c') = (if c = c' then One else Zero)
| pnderiv c (Plus r1 r2) = pnPlus (pnderiv c r1) (pnderiv c r2)
| pnderiv c (Times r1 r2) =
    (if nullable r1 then pnPlus (pnTimes (pnderiv c r1) r2) (pnderiv c r2) else
pnTimes (pnderiv c r1) r2)
| pnderiv c (Star r) = pnTimes (pnderiv c r) (Star r)

lemma pnderiv-alt[code]: pnderiv a r = pnorm (deriv a r)
  by (induct r) (auto simp: pnorm-alt)

lemma pnderiv-pderiv: pnderiv a r = flatten PLUS (pderiv a r)
  unfolding pnderiv-alt pnorm-def o-apply pset-deriv ..
```

4 Framework Instantiations using (Partial) Derivatives

4.1 Brzozowski Derivatives Modulo ACI

```
lemma ACI-norm-derivs-alt: «derivs w r» = fold (λa r. «deriv a r») w «r»
  by (induct w arbitrary: r) (auto simp: ACI-norm-deriv)

global-interpretation brzozowski: rexp-DFA λr. «r» λa r. «deriv a r» nullable
lang
  defines brzozowski-closure = brzozowski.closure
    and check-eqv-brz = brzozowski.check-eqv
    and reachable-brz = brzozowski.reachable
    and automaton-brz = brzozowski.automaton
    and match-brz = brzozowski.match
proof (unfold-locales, goal-cases)
  case 1 show ?case by (rule lang-ACI-norm)
next
  case 2 show ?case by (rule trans[OF lang-ACI-norm lang-deriv])
next
  case 3 show ?case by (rule nullable-iff)
next
  case 4 show ?case by (simp only: ACI-norm-derivs-alt[symmetric] finite-derivs)
qed
```

4.2 Brzozowski Derivatives Modulo ACI Operating on the Quotient Type

lemma *derivs-alt*: *derivs* = *fold deriv*
proof
  fix *w* :: *'a list* **show** *derivs w* = *fold deriv w* **by** (*induct w*) *auto*
qed

functor *map-rexp* **by** (*simp-all only*: *o-def id-def map-map-rexp map-rexp-ident*)

quotient-type *'a ACI-rexp* = *'a rexp* / *ACI*
  morphisms *rep-ACI-rexp ACI-class*
 by (*intro equivpI reflpI sympI transpI*) (*blast intro*: *ACI-refl ACI-sym ACI-trans*)+

instantiation *ACI-rexp* :: ({*equal*, *linorder*}) {*equal*, *linorder*}
begin
lift-definition *less-eq-ACI-rexp* :: *'a ACI-rexp* ⇒ *'a ACI-rexp* ⇒ *bool* **is** λ*r s*. *less-eq* «*r*» «*s*»
   by (*simp add*: *ACI-decidable*)
lift-definition *less-ACI-rexp* :: *'a ACI-rexp* ⇒ *'a ACI-rexp* ⇒ *bool* **is** λ*r s*. *less* «*r*» «*s*»
   by (*simp add*: *ACI-decidable*)
lift-definition *equal-ACI-rexp* :: *'a ACI-rexp* ⇒ *'a ACI-rexp* ⇒ *bool* **is** λ*r s*. «*r*» = «*s*»
   by (*simp add*: *ACI-decidable*)
instance by *intro-classes* (*transfer*, *force simp*: *ACI-decidable*)+
end

lift-definition *ACI-deriv* :: *'a* :: *linorder* ⇒ *'a ACI-rexp* ⇒ *'a ACI-rexp* **is** *deriv*
  by (*rule ACI-deriv*)
lift-definition *ACI-nullable* :: *'a* :: *linorder ACI-rexp* ⇒ *bool* **is** *nullable*
  by (*rule ACI-nullable*)
lift-definition *ACI-lang* :: *'a* :: *linorder ACI-rexp* ⇒ *'a lang* **is** *lang*
  by (*rule ACI-lang*)

lemma [*transfer-rule*]: *rel-fun* (*rel-set* (*pcr-ACI-rexp* (=))) (=) (*finite o image ACI-norm*) *finite*
  unfolding *rel-fun-def rel-set-def cr-ACI-rexp-def ACI-rexp.pcr-cr-eq*
  apply (*auto simp*: *elim*!: *finite-surj*[*of - - ACI-class*] *finite-surj*[*of - - ACI-norm o rep-ACI-rexp*])
 apply (*metis* (*opaque-lifting*, *no-types*) *ACI-norm-idem ACI-rexp.abs-eq-iff ACI-decidable imageI*)
  apply (*rule image-eqI*)
  apply (*subst ACI-decidable*[*symmetric*])
  apply (*rule ACI-sym*)
  apply (*rule Quotient-rep-abs*[*OF Quotient-ACI-rexp*, *OF ACI-refl*])
  apply *blast*
  done

global-interpretation *brzozowski-quotient*: *rexp-DFA ACI-class ACI-deriv ACI-nullable*

ACI-lang
defines *brzozowski-quotient-closure* = *brzozowski-quotient.closure*
and *check-eqv-brzq* = *brzozowski-quotient.check-eqv*
and *reachable-brzq* = *brzozowski-quotient.reachable*
and *automaton-brzq* = *brzozowski-quotient.automaton*
and *match-brzq* = *brzozowski-quotient.match*
proof (*unfold-locales*, *goal-cases*)
case *1* **show** *?case* **by** *transfer* (*rule refl*)
next
case *2* **show** *?case* **by** *transfer* (*rule lang-deriv*)
next
case *3* **show** *?case* **by** *transfer* (*rule nullable-iff*)
next
case *4* **show** *?case* **by** *transfer*
(*auto simp*: *ACI-decidable derivs-alt intro*!: *finite-subset*[*OF* - *finite-derivs*])
qed

4.3 Brzozowski Derivatives Modulo ACI++ (Only Soundness)

global-interpretation *nderiv*: *rexp-DA* $\lambda x.$ *norm x nderiv nullable lang*
defines *nderiv-closure* = *nderiv.closure*
and *check-eqv-n* = *nderiv.check-eqv*
and *reachable-n* = *nderiv.reachable*
and *automaton-n* = *nderiv.automaton*
and *match-n* = *nderiv.match*
proof (*unfold-locales*, *goal-cases*)
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **by** (*rule lang-nderiv*)
next
case *3* **show** *?case* **by** (*rule nullable-iff*)
qed

4.4 Partial Derivatives

global-interpretation *pderiv*: *rexp-DFA* $\lambda r.$ $\{r\}$ *pderiv-set* $\lambda P.$ $\exists\, p{\in}P.$ *nullable p* $\lambda P.$ $\bigcup$(*lang* ‘ *P*)
defines *pderiv-closure* = *pderiv.closure*
and *check-eqv-p* = *pderiv.check-eqv*
and *reachable-p* = *pderiv.reachable*
and *automaton-p* = *pderiv.automaton*
and *match-p* = *pderiv.match*
proof (*unfold-locales*, *goal-cases*)
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **by** (*simp add*: *Deriv-pderiv*)
next
case *3* **show** *?case* **by** (*simp add*: *nullable-iff*)

next
case (*4 s*) **note** *pderivs-lang-def*[*simp*]
{ fix *w* :: *'a list*
have *fold pderiv-set w* = *Union o image* (*pderivs-lang* {*w*}) **by** (*induct w*) *auto*
}
hence {*fold pderiv-set w* {*s*} |*w*. *True*} ⊆ *Pow* (*pderivs-lang UNIV s*) **by** *auto*
then show *?case* **by** (*rule finite-subset*) (*auto simp only*: *finite-pderivs-lang*)
qed

global-interpretation *pnderiv*: *rexp-DFA* λ*r*. *r pnderiv nullable lang*
defines *pnderiv-closure* = *pnderiv.closure*
and *check-eqv-pn* = *pnderiv.check-eqv*
and *reachable-pn* = *pnderiv.reachable*
and *automaton-pn* = *pnderiv.automaton*
and *match-pn* = *pnderiv.match*
proof (*unfold-locales*, *goal-cases*)
case *1* **show** *?case* **by** *simp*
next
case *2* **show** *?case* **by** (*simp add*: *pnorm-def pset-deriv Deriv-pderiv pnderiv-alt*)
next
case *3* **show** *?case* **by** (*simp add*: *nullable-iff*)
next
case (*4 s*)
then show *?case* **unfolding** *pnderiv-alt*[*abs-def*]
by (*rule finite-surj*[*OF pderiv.fin*, *of - flatten PLUS s*]) (*auto simp*: *fold-pnorm-deriv*)
qed

4.5 Languages as States

Not executable but still instructive.

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

interpretation *language*: *rexp-DFA lang Deriv* λ*L*. [] ∈ *L id*
proof (*unfold-locales*, *goal-cases*)
case (*4 s*)
have {*fold Deriv w* (*lang s*) |*w*. *True*} ⊆ (λ*X*. ⋃(*lang* ‘ *X*)) ‘ *Pow* (*pderivs-lang UNIV s*)
by (*auto simp*: *sym*[*OF Derivs-alt-def*] *Derivs-pderivs pderivs-lang-def*)
then show *?case* **by** (*rule finite-surj*[*OF iffD2*[*OF finite-Pow-iff finite-pderivs-lang-UNIV*]])
qed *simp-all*

definition *str-eq* :: *'a lang* => (*'a list* × *'a list*) *set* (‹≈-› [*100*] *100*)
where ≈*A* ≡ {(*x*, *y*). (∀ *z*. *x* @ *z* ∈ *A* ⟷ *y* @ *z* ∈ *A*)}

lemma *str-eq-alt*: ≈*A* = {(*x*, *y*). *fold Deriv x A* = *fold Deriv y A*}
unfolding *str-eq-def set-eq-iff in-fold-Deriv* **by** *simp*

lemma *Myhill-Nerode2*: *finite* (*UNIV* // ≈*lang r*)

unfolding *str-eq-alt quotient-def Image-def*
by (*rule finite-surj*[*OF language.fin, of* - $\lambda X.\ \{y.\ X = fold\ Deriv\ y\ (lang\ r)\}\ r$])
auto

5 Framework Instantiations using Marked Regular Expressions

5.1 Marked Regular Expressions

type-synonym $'a\ mrexp = (bool * 'a)\ rexp$

abbreviation $strip \equiv map\text{-}rexp\ snd$

primrec $mrexps :: 'a\ rexp \Rightarrow ('a\ mrexp)\ set$ **where**
$mrexps\ Zero = \{Zero\}$
$| mrexps\ One = \{One\}$
$| mrexps\ (Atom\ a) = \{Atom\ (True, a), Atom\ (False, a)\}$
$| mrexps\ (Plus\ r\ s) = case\text{-}prod\ Plus\ `\ (mrexps\ r \times mrexps\ s)$
$| mrexps\ (Times\ r\ s) = case\text{-}prod\ Times\ `\ (mrexps\ r \times mrexps\ s)$
$| mrexps\ (Star\ r) = Star\ `\ mrexps\ r$

lemma *finite-mrexps*[*simp*]: $finite\ (mrexps\ r)$
by (*induction r*) *auto*

lemma *strip-mrexps*: $strip\ `\ mrexps\ r = \{r\}$
by (*induction r*) (*auto simp*: *set-eq-subset subset-iff image-iff*)

fun $Lm :: 'a\ mrexp \Rightarrow 'a\ lang$ **where**
$Lm\ Zero = \{\}\ |$
$Lm\ One = \{\}\ |$
$Lm\ (Atom(m,a)) = (if\ m\ then\ \{[a]\}\ else\ \{\})\ |$
$Lm\ (Plus\ r\ s) = Lm\ r \cup Lm\ s\ |$
$Lm\ (Times\ r\ s) = Lm\ r\ @@\ lang(strip\ s) \cup Lm\ s\ |$
$Lm\ (Star\ r) = Lm\ r\ @@\ star(lang(strip\ r))$

fun $final :: 'a\ mrexp \Rightarrow bool$ **where**
$final\ Zero = False\ |$
$final\ One = False\ |$
$final\ (Atom(m,a)) = m\ |$
$final\ (Plus\ r\ s) = (final\ r \vee final\ s)\ |$
$final\ (Times\ r\ s) = (final\ s \vee nullable\ s \wedge final\ r)\ |$
$final\ (Star\ r) = final\ r$

abbreviation $read :: 'a \Rightarrow 'a\ mrexp \Rightarrow 'a\ mrexp$ **where**
$read\ a \equiv map\text{-}rexp\ (\lambda(m,x).\ (m \wedge a{=}x,\ x))$

lemma *read-mrexps*[*simp*]: $r \in mrexps\ s \Longrightarrow read\ a\ r \in mrexps\ s$
by (*induction s arbitrary*: *a r*) (*auto simp*: *image-iff*)

fun *follow* :: *bool* ⇒ *'a mrexp* ⇒ *'a mrexp* **where**
follow m Zero = *Zero* |
follow m One = *One* |
follow m (*Atom*(-,*a*)) = *Atom*(*m*,*a*) |
follow m (*Plus r s*) = *Plus* (*follow m r*) (*follow m s*) |
follow m (*Times r s*) =
  Times (*follow m r*) (*follow* (*final r* ∨ *m* ∧ *nullable r*) *s*) |
follow m (*Star r*) = *Star*(*follow* (*final r* ∨ *m*) *r*)

lemma *follow-mrexps*[*simp*]: *r* ∈ *mrexps s* ⟹ *follow b r* ∈ *mrexps s*
by (*induction s arbitrary*: *b r*) (*auto simp*: *image-iff*)

lemma *strip-read*[*simp*]: *strip* (*read a r*) = *strip r*
by (*simp add*: *map-map-rexp split-def*)

lemma *Nil-notin-Lm*[*simp*]: [] ∉ *Lm r*
by (*induction r*) (*auto split*: *if-splits*)

lemma *Nil-in-lang-strip*[*simp*]: [] ∈ *lang*(*r*) ⟷ [] ∈ *lang*(*strip r*)
by (*induction r*) *auto*

lemma *strip-follow*[*simp*]: *strip*(*follow m r*) = *strip r*
by (*induction r arbitrary*: *m*) (*auto split*: *if-splits*)

lemma *conc-lemma*: [] ∉ *A* ⟹ {*w* : *A* @@ *B*. *w* ≠ [] ∧ *P*(*hd w*)} = {*w* : *A*. *w* ≠
[] ∧ *P*(*hd w*)} @@ *B*
  unfolding *conc-def* **by** *auto* (*metis hd-append2*)+

lemma *Lm-read*: *Lm* (*read a r*) = {*w* : *Lm r*. *w* ≠ [] ∧ *hd w* = *a*}
proof (*induction r*)
  case (*Times r1 r2*) **thus** *?case*
    using *conc-lemma*[*OF Nil-notin-Lm*, **where** *P* = λ*x*. *x*=*a* **and** *r1* = *r1*] **by**
auto
next
  case *Star* **thus** *?case* **using** *conc-lemma*[*OF Nil-notin-Lm*, **where** *P* = λ*x*. *x*=*a*]
by *simp*
qed (*auto split*: *if-splits*)

lemma *tl-conc*[*simp*]: [] ∉ *A* ⟹*tl* ‘ (*A* @@ *B*) = *tl* ‘ *A* @@ *B*
by (*fastforce simp*: *image-def Bex-def tl-append split*: *list.split*)

lemma *Nil-in-tl-Lm-if-final*[*simp*]: *final r* ⟹ [] : *tl* ‘ *Lm r*
by (*induction r*) (*auto simp*: *nullable-iff image-Un*)

lemma *Nil-notin-tl-if-not-final*: ¬ *final r* ⟹ [] ∉ *tl* ‘ *Lm r*
by (*induction r*) (*auto simp*: *nullable-iff Nil-tl singleton-in-conc intro*!: *image-eqI*[*rotated*])

lemma *Lm-follow*: *Lm* (*follow m r*) = *tl* ‘ *Lm r* ∪ (*if m then lang*(*strip r*) *else*

{}) − {[]}
proof (*induction r arbitrary*: *m*)
case (*Atom mb*) **thus** *?case* **by** (*cases mb*) *auto*
next
case (*Times r s*) **thus** *?case*
by (*simp add*: *Un-Diff image-Un conc-Un-distrib nullable-iff conc-Diff-if-Nil1 Nil-notin-tl-if-not-final Un-ac*)
next
case (*Star r*) **thus** *?case*
by (*simp add*: *Un-Diff conc-Un-distrib conc-Diff-if-Nil1 Nil-notin-tl-if-not-final star-Diff-Nil-fold*)
qed *auto*

5.2 Mark Before Atom

Position automaton where mark is placed before atoms.

abbreviation *empty-mrexp* ≡ *map-rexp* (λ*a*. (*False*,*a*))

lemma *empty-mrexp-mrexps*[*simp*]: *empty-mrexp r* ∈ *mrexps r*
by (*induction r*) *auto*

lemma *nullable-empty-mrexp*[*simp*]: *nullable* (*empty-mrexp r*) = *nullable r*
by (*induct r*) *auto*

definition *init-b r* = (*follow True* (*empty-mrexp r*), *nullable r*)

lemma *init-b-mrexps*[*simp*]: *init-b r* ∈ *mrexps r* × *UNIV*
unfolding *init-b-def* **by** *auto*

fun *delta-b* **where**
delta-b a (*r*,*b*) = (*let r*′ = *read a r in* (*follow False r*′, *final r*′))

lemma *delta-b-mrexps*[*simp*]: *rb* ∈ *mrexps r* × *UNIV* ⟹ *delta-b a rb* ∈ *mrexps r* × *UNIV*
by (*auto simp*: *Let-def*)

lemma *fold-delta-b-init-b-mrexps*[*simp*]: *fold delta-b w* (*init-b s*) ∈ *mrexps s* × *UNIV*
by (*induction w arbitrary*: *s rule*: *rev-induct*) *auto*

fun *L-b* **where**
L-b (*r*,*b*) = *Lm r* ∪ (*if b then* {[]} *else* {})

abbreviation *final-b* ≡ *snd*

lemma *Lm-empty*: *Lm* (*empty-mrexp r*) = {}
by (*induction r*) *auto*

lemma *final-read-Lm*: *final*(*read a r*) ⟷ [*a*] ∈ *Lm r*

by (*induction r*) (*auto simp*: *nullable-iff concI-if-Nil2 singleton-in-conc split*: *if-splits*)

global-interpretation *before*: *rexp-DFA init-b delta-b final-b L-b*
  defines *before-closure* = *before.closure*
    and *check-eqv-b* = *before.check-eqv*
    and *reachable-b* = *before.reachable*
    and *automaton-b* = *before.automaton*
    and *match-b* = *before.match*
proof (*standard*, *goal-cases*)
  case (*1 r*) **show** *L-b* (*init-b r*) = *lang r*
    by(*auto simp add*: *init-b-def Lm-follow Lm-empty map-map-rexp nullable-iff*)
next
  case (*2 a rb*) **show** *L-b* (*delta-b a rb*) = *Deriv a* (*L-b rb*)
     by (*cases rb*) (*auto simp add*: *Deriv-def final-read-Lm image-def Lm-read Lm-follow*)
next
  case (*3 rb*) **show** *final-b rb* ⟷ [] ∈ *L-b rb* **by** (*cases rb*) *simp*
next
  case (*4 s*)
  have {*fold delta-b w* (*init-b s*) |*w*. *True*} ⊆ *mrexps s* × *UNIV*
    by (*intro subsetI*, *elim CollectE exE*) (*simp only*: *fold-delta-b-init-b-mrexps*)
  then show *finite* {*fold delta-b w* (*init-b s*) |*w*. *True*} **by** (*rule finite-subset*) *simp*
qed

5.3 Mark After Atom

Position automaton where mark is placed after atoms. This is the Glushkov and McNaughton/Yamada construction.

definition *init-a r* = (*True*, *empty-mrexp r*)

lemma *init-a-mrexps*[*simp*]: *init-a r* ∈ *UNIV* × *mrexps r*
  unfolding *init-a-def* **by** *auto*

fun *delta-a* **where**
delta-a a (*b*,*r*) = (*False*, *read a* (*follow b r*))

lemma *delta-a-mrexps*[*simp*]: *br* ∈ *UNIV* × *mrexps r* ⟹ *delta-a a br* ∈ *UNIV* × *mrexps r*
by *auto*

lemma *fold-delta-a-init-a-mrexps*[*simp*]: *fold delta-a w* (*init-a s*) ∈ *UNIV* × *mrexps s*
by (*induction w arbitrary*: *s rule*: *rev-induct*) *auto*

fun *final-a* **where**
final-a (*b*,*r*) ⟷ *final r* ∨ *b* ∧ *nullable r*

fun *L-a* **where**
L-a (*b*,*r*) = *Lm* (*follow b r*) ∪ (*if final-a*(*b*,*r*) *then* {[]} *else* {})

lemma *nonfinal-empty-mrexp*: ¬ *final* (*empty-mrexp r*)
by (*induction r*) *auto*

lemma *Cons-eq-tl-iff*[*simp*]: *x # xs = tl ys* ⟷ (∃ *y*. *ys = y # x # xs*)
by (*cases ys*) *auto*

lemma *tl-eq-Cons-iff*[*simp*]: *tl ys = x # xs* ⟷ (∃ *y*. *ys = y # x # xs*)
by (*cases ys*) *auto*

global-interpretation *after*: *rexp-DFA init-a delta-a final-a L-a*
  defines *after-closure = after.closure*
    and *check-eqv-a = after.check-eqv*
    and *reachable-a = after.reachable*
    and *automaton-a = after.automaton*
    and *match-a = after.match*
proof (*standard*, *goal-cases*)
  case (*1 r*) **show** *L-a* (*init-a r*) = *lang r*
   by (*auto simp*: *init-a-def nonfinal-empty-mrexp Lm-follow Lm-empty map-map-rexp nullable-iff*)
next
  case (*2 a br*) **show** *L-a* (*delta-a a br*) = *Deriv a* (*L-a br*)
    by (*cases br*) (*simp add*: *Deriv-def final-read-Lm Lm-read Lm-follow*,
      fastforce simp: *image-def neq-Nil-conv*)
next
  case (*3 br*) **show** *final-a br* ⟷ [] ∈ *L-a br* **by** (*cases br*) *simp*
next
  case (*4 s*)
  have {*fold delta-a w* (*init-a s*) |*w*. *True*} ⊆ *UNIV* × *mrexps s*
    by (*intro subsetI*, *elim CollectE exE*) (*simp only*: *fold-delta-a-init-a-mrexps*)
  then show *finite* {*fold delta-a w* (*init-a s*) |*w*. *True*} **by** (*rule finite-subset*) *simp*
qed

The "before" atomaton is a quotient of the "after" automaton.

The proof below follows an informal proof given by Helmut Seidl in personal communication.

fun *hom-ab* **where**
  hom-ab (*b*, *r*) = (*follow b r*, *final-a* (*b*, *r*))

lemma *hom-delta*: *hom-ab* (*delta-a x br*) = *delta-b x* (*hom-ab br*)
by(*cases br*) (*auto simp add*: *Let-def*)

lemma *hom-deltas*: *hom-ab* (*fold delta-a w br*) = *fold delta-b w* (*hom-ab br*)
  by (*induct w arbitrary*: *br*) (*auto simp add*: *hom-delta*)

lemma *hom-init*: *hom-ab* (*init-a r*) = *init-b r*
 unfolding *init-a-def init-b-def hom-ab.simps* **by** (*simp add*: *nonfinal-empty-mrexp*)

lemma *reachable-ab*: *reachable-b as r = hom-ab* ‘ *reachable-a as r*

unfolding *after.reachable before.reachable* **by** (*force simp*: *hom-init hom-deltas*)

theorem *card-reachable-ab*: *card* (*reachable-b as r*) ≤ *card* (*reachable-a as r*)
unfolding *reachable-ab* **using** *after.finite-reachable* **by** (*rule card-image-le*)

The implementation by Fischer et al.:

fun *shift* :: *bool* ⇒ *'a mrexp* ⇒ *'a* ⇒ *'a mrexp* **where**
shift - One - = One |
shift - Zero - = Zero |
shift m (*Atom* (*-*,*x*)) *c* = *Atom* (*m* ∧ (*x*=*c*),*x*) |
shift m (*Plus r s*) *c* = *Plus* (*shift m r c*) (*shift m s c*) |
shift m (*Times r s*) *c* =
Times (*shift m r c*) (*shift* (*final r* ∨ *m* ∧ *nullable r*) *s c*) |
shift m (*Star r*) *c* = *Star* (*shift* (*final r* ∨ *m*) *r c*)

lemma *shift-read-follow*: *shift m r x* = *read x* (*follow m r*)
by (*induction m r x rule*: *shift.induct*) *auto*

In the spirit of Asperti, and similarly quadratic because of need to call final1 in move.

fun *final1* :: *'a mrexp* ⇒ *'a* ⇒ *bool* **where**
final1 Zero - = False |
final1 One - = False |
final1 (*Atom*(*m*,*a*)) *x* = (*m* ∧ *a*=*x*) |
final1 (*Plus r s*) *x* = (*final1 r x* ∨ *final1 s x*) |
final1 (*Times r s*) *x* = (*final1 s x* ∨ *nullable s* ∧ *final1 r x*) |
final1 (*Star r*) *x* = *final1 r x*

fun *move* :: *'a* ⇒ *'a mrexp* ⇒ *bool* ⇒ *'a mrexp* **where**
move - One - = One |
move - Zero - = Zero |
move c (*Atom* (*-*,*a*)) *m* = *Atom* (*m*, *a*) |
move c (*Plus r s*) *m* = *Plus* (*move c r m*) (*move c s m*) |
move c (*Times r s*) *m* =
Times (*move c r m*) (*move c s* (*final1 r c* ∨ *m* ∧ *nullable r*)) |
move c (*Star r*) *m* = *Star* (*move c r* (*final1 r c* ∨ *m*))

lemma *nullable-read*[*simp*]: *nullable* (*read c r*) = *nullable r*
by (*induction r*) *auto*

lemma *final-read-final1*: *final* (*read c r*) = *final1 r c*
by (*induction r*) *auto*

lemma *move-follow-read*: *move c r m* = *follow m* (*read c r*)
by (*induction c r m rule*: *move.induct*) (*auto simp*: *final-read-final1*)

6 Linear Time Optimization for "Mark After Atom"

datatype *'a mrexp2* =
  Zero2 |
  One2 |
  Atom2 (*fin*: *bool*) *'a* |
  Plus2 'a mrexp2 'a mrexp2 (*fin*: *bool*) (*nul*: *bool*) |
  Times2 'a mrexp2 'a mrexp2 (*fin*: *bool*) (*nul*: *bool*) |
  Star2 'a mrexp2 (*fin*: *bool*)
where
  fin Zero2 = *False*
| *nul Zero2* = *False*
| *fin One2* = *False*
| *nul One2* = *True*
| *nul* (*Atom2* - -) = *False*
| *nul* (*Star2* - -) = *True*

primrec *mrexps2* :: *'a rexp* ⇒ (*'a mrexp2*) *set* **where**
  mrexps2 Zero = {*Zero2*}
| *mrexps2 One* = {*One2*}
| *mrexps2* (*Atom a*) = {*Atom2 True a*, *Atom2 False a*}
| *mrexps2* (*Plus r s*) = (λ(*r*, *s*, *f*, *n*). *Plus2 r s f n*) ‘ (*mrexps2 r* × *mrexps2 s* × *UNIV*)
| *mrexps2* (*Times r s*) = (λ(*r*, *s*, *f*, *n*). *Times2 r s f n*) ‘ (*mrexps2 r* × *mrexps2 s* × *UNIV*)
| *mrexps2* (*Star r*) = (λ(*r*, *f*). *Star2 r f*) ‘ (*mrexps2 r* × *UNIV*)

lemma *finite-mrexps3*[*simp*]: *finite* (*mrexps2 r*)
by (*induction r*) *auto*

definition[*simp*]: *plus2 r s* == *Plus2 r s* (*fin r* ∨ *fin s*) (*nul r* ∨ *nul s*)
definition[*simp*]: *times2 r s* == *Times2 r s* (*fin r* ∧ *nul s* ∨ *fin s*) (*nul r* ∧ *nul s*)
definition[*simp*]: *star2 r* == *Star2 r* (*fin r*)

primrec *empty-mrexp2* :: *'a rexp* ⇒ *'a mrexp2* **where**
empty-mrexp2 Zero = *Zero2* |
empty-mrexp2 One = *One2* |
empty-mrexp2 (*Atom x*) = *Atom2 False x* |
empty-mrexp2 (*Plus r s*) = *plus2* (*empty-mrexp2 r*) (*empty-mrexp2 s*) |
empty-mrexp2 (*Times r s*) = *times2* (*empty-mrexp2 r*) (*empty-mrexp2 s*) |
empty-mrexp2 (*Star r*) = *star2* (*empty-mrexp2 r*)

primrec *shift2* :: *bool* ⇒ *'a mrexp2* ⇒ *'a* ⇒ *'a mrexp2* **where**
shift2 - *One2* - = *One2* |
shift2 - *Zero2* - = *Zero2* |
shift2 m (*Atom2* - *x*) *c* = *Atom2* (*m* ∧ (*x*=*c*)) *x* |
shift2 m (*Plus2 r s* - -) *c* = *plus2* (*shift2 m r c*) (*shift2 m s c*) |
shift2 m (*Times2 r s* - -) *c* = *times2* (*shift2 m r c*) (*shift2* (*m* ∧ *nul r* ∨ *fin r*) *s*

```
c) |
shift2 m (Star2 r -) c =  star2 (shift2 (m ∨ fin r) r c)

primrec strip2 where
strip2 Zero2 = Zero |
strip2 One2 = One |
strip2 (Atom2 m x) = Atom (m, x) |
strip2 (Plus2 r s - -) = Plus (strip2 r) (strip2 s) |
strip2 (Times2 r s - -) = Times (strip2 r) (strip2 s) |
strip2 (Star2 r -) = Star (strip2 r)

lemma strip-mrexps2: (strip o strip2) ‘ mrexps2 r = {r}
by (induction r) (auto simp: set-eq-subset subset-iff image-iff)

primrec ok2 :: 'a mrexp2 ⇒ bool where
ok2 Zero2 = True |
ok2 One2 = True |
ok2 (Atom2 - -) = True |
ok2 (Plus2 r s f n) = (ok2 r ∧ ok2 s ∧
   (let rs = Plus (strip2 r) (strip2 s) in f = final rs ∧ n = nullable rs)) |
ok2 (Times2 r s f n) = (ok2 r ∧ ok2 s ∧
   (let rs = Times (strip2 r) (strip2 s) in f = final rs ∧ n = nullable rs)) |
ok2 (Star2 r f) = (ok2 r ∧ f = final (strip2 r))

lemma ok2-fin-final[simp]: ok2 r ⟹ fin r = final (strip2 r)
  by (induct r) auto

lemma ok2-nul-nullable[simp]: ok2 r ⟹ nul r = nullable (strip2 r)
  by (induct r) auto

lemma strip2-shift2: ok2 r ⟹ strip2(shift2 m r c) = shift m (strip2 r) c
apply(induction r arbitrary: m)
apply (auto simp: disj-commute)
done

lemma ok2-empty-mrexp2: ok2 (empty-mrexp2 r)
apply(induction r)
apply auto
done

lemma ok2-shift2: ok2 r ⟹ ok2(shift2 m r c)
apply(induction r arbitrary: m)
apply auto
done

lemma strip2-empty-mrexp2[simp]: strip2 (empty-mrexp2 r) = empty-mrexp r
  by (induct r) auto

lemma nul-empty-mrexp2[simp]: nul (empty-mrexp2 r) = nullable r
```

by (*induct r*) *auto*

lemma *nonfin-empty-mrexp2*[*simp*]: ¬ *fin* (*empty-mrexp2 r*)
by (*induct r*) *auto*

lemma *empty-mrexp2-mrexps2*[*simp*]: *empty-mrexp2 s* ∈ *mrexps2 s*
by (*induct s*) (*auto simp*: *image-iff*)

lemma *shift2-mrexps2*[*simp*]: *r* ∈ *mrexps2 s* ⟹ *shift2 x r a* ∈ *mrexps2 s*
by (*induct s arbitrary*: *r x*) (*auto simp*: *image-iff*)

typedef *'a ok-mrexp2* = {(*b* :: *bool*, *r* :: *'a mrexp2*). *ok2 r*}
unfolding *mem-Collect-eq split-beta* **by** (*metis snd-eqD ok2-empty-mrexp2*)

setup-lifting *type-definition-ok-mrexp2*

lift-definition *init-okm* :: *'a rexp* ⇒ *'a ok-mrexp2* **is** λ*r*. (*True*, *empty-mrexp2 r*)
by (*simp add*: *ok2-empty-mrexp2 del*: *ok2.simps*)
lift-definition *delta-okm* :: *'a* ⇒ *'a ok-mrexp2* ⇒ *'a ok-mrexp2* **is**
λ*a* (*m*, *r*). (*False*, *shift2 m r a*)
unfolding *mem-Collect-eq split-beta snd-conv* **by** (*intro ok2-shift2*) *simp*
lift-definition *nullable-okm* :: *'a ok-mrexp2* ⇒ *bool* **is** λ(*m*, *r*). *fin r* ∨ *m* ∧ *nul r*
.
lift-definition *lang-okm* :: *'a ok-mrexp2* ⇒ *'a lang* **is** λ(*m*, *r*). *L-a* (*m*, *strip2 r*) .

instantiation *ok-mrexp2* :: (*equal*) *equal*
begin

fun *eq-mrexp2* **where**
eq-mrexp2 Zero2 Zero2 = *True*
| *eq-mrexp2 One2 One2* = *True*
| *eq-mrexp2* (*Atom2 m x*) (*Atom2 m'* *y*) = (*m* = *m'* ∧ *x* = *y*)
| *eq-mrexp2* (*Plus2 r1 s1* - -) (*Plus2 r2 s2* - -) = (*eq-mrexp2 r1 r2* ∧ *eq-mrexp2 s1 s2*)
| *eq-mrexp2* (*Times2 r1 s1* - -) (*Times2 r2 s2* - -) = (*eq-mrexp2 r1 r2* ∧ *eq-mrexp2 s1 s2*)
| *eq-mrexp2* (*Star2 r1* -) (*Star2 r2* -) = (*eq-mrexp2 r1 r2*)
| *eq-mrexp2 r s* = *False*

lemma *eq-mrexp2-imp-eq*: ⟦*eq-mrexp2 r s*; *ok2 r*; *ok2 s*⟧ ⟹ (*r* = *s*)
by (*induct rule*: *eq-mrexp2.induct*) *auto*

lemma *eq-mrexp2-refl*[*simplified*, *simp*]: *r* = *s* ⟹ *eq-mrexp2 r s*
by (*induct rule*: *eq-mrexp2.induct*) *auto*

lemma *eq-mrexp2-eq*: ⟦*ok2 r*; *ok2 s*⟧ ⟹ *eq-mrexp2 r s* = (*r* = *s*)
by (*metis eq-mrexp2-imp-eq eq-mrexp2-refl*)

lift-definition *equal-ok-mrexp2* :: *'a ok-mrexp2* ⇒ *'a ok-mrexp2* ⇒ *bool*
  is λ(*b1*,*r1*) (*b2*, *r2*). *b1* = *b2* ∧ *eq-mrexp2 r1 r2* .

instance by *intro-classes* (*transfer*, *auto simp*: *eq-mrexp2-eq*)

end

global-interpretation *after2*: *rexp-DFA init-okm delta-okm nullable-okm lang-okm*
  defines *after2-closure* = *after2.closure*
    and *check-eqv-a2* = *after2.check-eqv*
    and *reachable-a2* = *after2.reachable*
    and *automaton-a2* = *after2.automaton*
    and *match-a2* = *after2.match*
proof (*standard*, *goal-cases*)
  case (*1 r*) **show** *lang-okm* (*init-okm r*) = *lang r*
    by *transfer* (*auto simp*: *split-beta init-a-def nonfinal-empty-mrexp Lm-follow Lm-empty*
      map-map-rexp nullable-iff)
next
  case (*2 a br*) **show** *lang-okm* (*delta-okm a br*) = *Deriv a* (*lang-okm br*)
    apply *transfer*
    unfolding *split-beta fst-conv snd-conv mem-Collect-eq after.L-delta*[*symmetric*] *delta-a.simps*
      shift-read-follow[*symmetric*]
    by (*subst strip2-shift2*) *simp-all*
next
  case (*3 br*) **show** *nullable-okm br* = ([] ∈ *lang-okm br*)
    by *transfer* (*simp add*: *split-beta*)
next
  case (*4 s*)
  have {*fold* (λ*a* (*m*, *r*). (*False*, *shift2 m r a*)) *w* (*True*, *empty-mrexp2 s*) |*w*. *True*} ⊆
    UNIV × *mrexps2 s*
  proof (*intro subsetI*, *elim CollectE exE conjE*, *hypsubst*)
    fix *w* **show** *fold* (λ*a* (*m*, *r*). (*False*, *shift2 m r a*)) *w* (*True*, *empty-mrexp2 s*) ∈
      UNIV × *mrexps2 s*
    by (*induct w rule*: *rev-induct*) (*auto simp*: *split*: *prod.splits intro*!: *shift2-mrexps2*)
  qed
  then show *finite* {*fold delta-okm w* (*init-okm s*) |*w*. *True*}
    by *transfer* (*erule finite-subset*[*OF subset-trans*[*rotated*]], *auto*)
qed

7 Linear Time Optimization for "Mark Before Atom" (for a Fixed Alphabet)

declare *Let-def*[*simp*]

datatype *'a mrexp3* =
  Zero3 |
  One3 |
  Atom3 bool 'a |
  Plus3 'a mrexp3 'a mrexp3 (*fin1*: *'a set*) (*nul*: *bool*) |
  Times3 'a mrexp3 'a mrexp3 (*fin1*: *'a set*) (*nul*: *bool*) |
  Star3 'a mrexp3 (*fin1*: *'a set*)
where
  fin1 Zero3 = {}
| *nul Zero3* = *False*
| *fin1 One3* = {}
| *nul One3* = *True*
| *fin1* (*Atom3 m a*) = (*if m then* {*a*} *else* {})
| *nul* (*Atom3* - -) = *False*
| *nul* (*Star3* - -) = *True*

primrec *final3* **where**
  final3 Zero3 = *False*
| *final3 One3* = *False*
| *final3* (*Atom3 m a*) = *m*
| *final3* (*Plus3 r s* - -) = (*final3 r* ∨ *final3 s*)
| *final3* (*Times3 r s* - -) = (*final3 s* ∨ *nul s* ∧ *final3 r*)
| *final3* (*Star3 r* -) = *final3 r*

primrec *mrexps3* :: *'a rexp* ⇒ (*'a mrexp3*) *set* **where**
  mrexps3 Zero = {*Zero3*}
| *mrexps3 One* = {*One3*}
| *mrexps3* (*Atom a*) = {*Atom3 True a*, *Atom3 False a*}
| *mrexps3* (*Plus r s*) = (λ(*r*, *s*, *f1*, *n*). *Plus3 r s f1 n*) ‘ (*mrexps3 r* × *mrexps3 s* × *Pow* (*atoms* (*Plus r s*)) × *UNIV*)
| *mrexps3* (*Times r s*) = (λ(*r*, *s*, *f1*, *n*). *Times3 r s f1 n*) ‘ (*mrexps3 r* × *mrexps3 s* × *Pow* (*atoms* (*Times r s*)) × *UNIV*)
| *mrexps3* (*Star r*) = (λ(*r*, *f1*). *Star3 r f1*) ‘ (*mrexps3 r* × *Pow* (*atoms* (*Star r*)))

lemma *finite-atoms*[*simp*]: *finite* (*atoms r*)
  by (*induct r*) *auto*

lemma *finite-mrexps3*[*simp*]: *finite* (*mrexps3 r*)
  by (*induct r*) *auto*

definition[*simp*]: *plus3 r s* == *Plus3 r s* (*fin1 r* ∪ *fin1 s*) (*nul r* ∨ *nul s*)
definition[*simp*]: *times3 r s* ==
  let ns = *nul s in Times3 r s* (*fin1 s* ∪ (*if ns then fin1 r else* {})) (*nul r* ∧ *ns*)
definition[*simp*]: *star3 r* == *Star3 r* (*fin1 r*)

primrec *follow3* :: *bool* ⇒ *'a mrexp3* ⇒ *'a mrexp3* **where**
follow3 m Zero3 = *Zero3* |
follow3 m One3 = *One3* |

```
follow3 m (Atom3 - a) = Atom3 m a |
follow3 m (Plus3 r s - -) = plus3 (follow3 m r) (follow3 m s) |
follow3 m (Times3 r s - -) =
  times3 (follow3 m r) (follow3 (final3 r ∨ m ∧ nul r) s) |
follow3 m (Star3 r -) = star3(follow3 (final3 r ∨ m) r)

primrec empty-mrexp3 :: 'a rexp ⇒ 'a mrexp3 where
empty-mrexp3 Zero = Zero3 |
empty-mrexp3 One = One3 |
empty-mrexp3 (Atom x) = Atom3 False x |
empty-mrexp3 (Plus r s) = plus3 (empty-mrexp3 r) (empty-mrexp3 s) |
empty-mrexp3 (Times r s) = times3 (empty-mrexp3 r) (empty-mrexp3 s) |
empty-mrexp3 (Star r) = star3 (empty-mrexp3 r)

primrec move3 :: 'a ⇒ 'a mrexp3 ⇒ bool ⇒ 'a mrexp3 where
move3 - One3 - = One3 |
move3 - Zero3 - = Zero3 |
move3 c (Atom3 - a) m = Atom3 m a |
move3 c (Plus3 r s - -) m = plus3 (move3 c r m) (move3 c s m) |
move3 c (Times3 r s - -) m =
  times3 (move3 c r m) (move3 c s (c ∈ fin1 r ∨ m ∧ nul r)) |
move3 c (Star3 r -) m = star3 (move3 c r (c ∈ fin1 r ∨ m))

primrec strip3 where
strip3 Zero3 = Zero |
strip3 One3 = One |
strip3 (Atom3 m x) = Atom (m, x) |
strip3 (Plus3 r s - -) = Plus (strip3 r) (strip3 s) |
strip3 (Times3 r s - -) = Times (strip3 r) (strip3 s) |
strip3 (Star3 r -) = Star (strip3 r)

lemma strip-mrexps3: (strip o strip3) ‘ mrexps3 r = {r}
by (induction r) (auto simp: set-eq-subset subset-iff image-iff)

primrec ok3 :: 'a mrexp3 ⇒ bool where
ok3 Zero3 = True |
ok3 One3 = True |
ok3 (Atom3 - -) = True |
ok3 (Plus3 r s f1 n) = (ok3 r ∧ ok3 s ∧
  (let rs = Plus (strip3 r) (strip3 s) in f1 = Collect (final1 rs) ∧ n = nullable rs))
|
ok3 (Times3 r s f1 n) = (ok3 r ∧ ok3 s ∧
  (let rs = Times (strip3 r) (strip3 s) in f1 = Collect (final1 rs) ∧ n = nullable
rs)) |
ok3 (Star3 r f1) = (ok3 r ∧ f1 = Collect (final1 (strip3 r)))

lemma ok3-fin1-final1[simp]: ok3 r ⟹ fin1 r = Collect (final1 (strip3 r))
  by (induct r) (auto simp add: set-eq-iff)
```

lemma *ok3-nul-nullable*[*simp*]: *ok3 r* $\Longrightarrow$ *nul r* = *nullable* (*strip3 r*)
by (*induct r*) *auto*

lemma *ok3-final3-final*[*simp*]: *ok3 r* $\Longrightarrow$ *final3 r* = *final* (*strip3 r*)
by (*induct r*) *auto*

lemma *follow3-follow*[*simp*]: *ok3 r* $\Longrightarrow$ *strip3* (*follow3 m r*) = *follow m* (*strip3 r*)
by (*induct r arbitrary*: *m*) *auto*

lemma *nul-follow3*[*simp*]: *ok3 r* $\Longrightarrow$ *nul* (*follow3 m r*) = *nul r*
by (*induct r arbitrary*: *m*) *auto*

lemma *ok3-follow3*[*simp*]: *ok3 r* $\Longrightarrow$ *ok3* (*follow3 m r*)
by (*induct r arbitrary*: *m*) *auto*

lemma *fin1-atoms*: $[\![x \in$ *fin1 mr*; *mr* $\in$ *mrexps3 r*$]\!] \Longrightarrow x \in$ *atoms r*
by (*induct r*) *auto*

lemma *follow3-mrexps3*[*simp*]: *r* $\in$ *mrexps3 s* $\Longrightarrow$ *follow3 m r* $\in$ *mrexps3 s*
by (*induct s arbitrary*: *m r*) (*fastforce simp add*: *image-iff dest*: *fin1-atoms*)+

lemma *empty-mrexp3-mrexps*[*simp*]: *empty-mrexp3 r* $\in$ *mrexps3 r*
by (*induct r*) (*auto simp*: *image-iff dest*: *fin1-atoms*)

lemma *strip3-empty-mrexp3*[*simp*]: *strip3* (*empty-mrexp3 r*) = *empty-mrexp r*
by (*induct r*) *auto*

lemma *strip3-move3*: *ok3 r* $\Longrightarrow$ *strip3*(*move3 m r c*) = *move m* (*strip3 r*) *c*
apply(*induction r arbitrary*: *c*)
apply (*auto simp*: *disj-commute*)
done

lemma *nul-empty-mrexp3*[*simp*]: *nul* (*empty-mrexp3 r*) = *nullable r*
apply(*induction r*)
apply *auto*
done

lemma *ok3-empty-mrexp3*: *ok3* (*empty-mrexp3 r*)
apply(*induction r*)
apply *auto*
done

lemma *ok3-move3*: *ok3 r* $\Longrightarrow$ *ok3*(*move3 m r c*)
apply(*induction r arbitrary*: *c*)
apply *auto*
done

lemma *nonfin1-empty-mrexp3*[*simp*]: *c* $\notin$ *fin1* (*empty-mrexp3 r*)
by (*induct r*) *auto*

lemma *move3-mrexps3*[*simp*]: $r \in mrexps3\ s \Longrightarrow move3\ x\ r\ a \in mrexps3\ s$
by (*induct s arbitrary*: *r x a*) (*fastforce simp*: *image-iff dest*: *fin1-atoms*)+

typedef *'a ok-mrexp3* = {(*r* :: *'a mrexp3*, *b* :: *bool*). *ok3 r*}
unfolding *mem-Collect-eq split-beta* **by** (*metis fst-eqD ok3-empty-mrexp3*)

setup-lifting *type-definition-ok-mrexp3*

abbreviation *init-m r* ≡ *let mr* = *follow3 True* (*empty-mrexp3 r*) *in* (*mr*, *nul mr*)

lift-definition *init-okm* :: *'a rexp* ⇒ *'a ok-mrexp3* **is** *init-m*
by (*simp add*: *ok3-empty-mrexp3*)
lift-definition *delta-okm* :: *'a* ⇒ *'a ok-mrexp3* ⇒ *'a ok-mrexp3* **is**
λ*a* (*r*, *m*). (*move3 a r False*, *a* ∈ *fin1 r*)
unfolding *mem-Collect-eq split-beta fst-conv* **by** (*intro ok3-move3*) *simp*
lift-definition *nullable-okm* :: *'a ok-mrexp3* ⇒ *bool* **is** *snd* .
lift-definition *lang-okm* :: *'a ok-mrexp3* ⇒ *'a lang* **is** λ(*r*, *m*). *L-b* (*strip3 r*, *m*) .

instantiation *ok-mrexp3* :: (*equal*) *equal*
begin

fun *eq-mrexp3* **where**
eq-mrexp3 Zero3 Zero3 = *True*
| *eq-mrexp3 One3 One3* = *True*
| *eq-mrexp3* (*Atom3 m x*) (*Atom3 m' y*) = (*m* = *m'* ∧ *x* = *y*)
| *eq-mrexp3* (*Plus3 r1 s1* - -) (*Plus3 r3 s3* - -) = (*eq-mrexp3 r1 r3* ∧ *eq-mrexp3 s1 s3*)
| *eq-mrexp3* (*Times3 r1 s1* - -) (*Times3 r3 s3* - -) = (*eq-mrexp3 r1 r3* ∧ *eq-mrexp3 s1 s3*)
| *eq-mrexp3* (*Star3 r1* -) (*Star3 r3* -) = (*eq-mrexp3 r1 r3*)
| *eq-mrexp3 r s* = *False*

lemma *eq-mrexp3-imp-eq*: ⟦*eq-mrexp3 r s*; *ok3 r*; *ok3 s*⟧ ⟹ (*r* = *s*)
by (*induct rule*: *eq-mrexp3.induct*) *auto*

lemma *eq-mrexp3-refl*[*simplified*, *simp*]: *r* = *s* ⟹ *eq-mrexp3 r s*
by (*induct rule*: *eq-mrexp3.induct*) *auto*

lemma *eq-mrexp3-eq*: ⟦*ok3 r*; *ok3 s*⟧ ⟹ *eq-mrexp3 r s* = (*r* = *s*)
by (*metis eq-mrexp3-imp-eq eq-mrexp3-refl*)

lift-definition *equal-ok-mrexp3* :: *'a ok-mrexp3* ⇒ *'a ok-mrexp3* ⇒ *bool*
is λ(*r1*, *b1*) (*r3*, *b3*). *b1* = *b3* ∧ *eq-mrexp3 r1 r3* .

instance by *intro-classes* (*transfer*, *auto simp*: *eq-mrexp3-eq*)

end

global-interpretation *before2*: *rexp-DFA init-okm delta-okm nullable-okm lang-okm*
  defines *before2-closure* = *before2.closure*
    and *check-eqv-b2* = *before2.check-eqv*
    and *reachable-b2* = *before2.reachable*
    and *automaton-b2* = *before2.automaton*
    and *match-b2* = *before2.match*
proof (*standard*, *goal-cases*)
  case (*1 r*) **show** *lang-okm* (*init-okm r*) = *lang r*
    by *transfer* (*auto simp*: *split-beta init-a-def nonfinal-empty-mrexp Lm-follow Lm-empty*
      map-map-rexp nullable-iff ok3-empty-mrexp3)
next
  case (*2 a br*) **show** *lang-okm* (*delta-okm a br*) = *Deriv a* (*lang-okm br*)
    apply *transfer*
   unfolding *split-beta fst-conv snd-conv mem-Collect-eq before.L-delta*[*symmetric*] *delta-b.simps*
      move-follow-read[*symmetric*] *final-read-final1 Let-def*
    by (*subst strip3-move3*) *simp-all*
next
  case (*3 br*) **show** *nullable-okm br* = ([] ∈ *lang-okm br*)
    by *transfer* (*simp add*: *split-beta*)
next
  case (*4 s*)
  have {*fold* (λ*a* (*r*, *m*). (*move3 a r False*, *a* ∈ *fin1 r*)) *w* (*init-m s*) |*w*. *True*} ⊆
    mrexps3 s × *UNIV*
  proof (*intro subsetI*, *elim CollectE exE conjE*, *hypsubst*)
    fix *w* **show** *fold* (λ*a* (*r*, *m*). (*move3 a r False*, *a* ∈ *fin1 r*)) *w* (*init-m s*) ∈
      mrexps3 s × *UNIV*
   by (*induct w rule*: *rev-induct*) (*auto simp*: *split*: *prod.splits intro*!: *move3-mrexps3*)
  qed
  then show *finite* {*fold delta-okm w* (*init-okm s*) |*w*. *True*}
    by *transfer* (*erule finite-subset*[*OF subset-trans*[*rotated*]], *auto*)
qed

8 Various Algorithms for Regular Expression Equivalence

export-code
  check-eqv-brz
  check-eqv-brzq
  check-eqv-n
  check-eqv-p
  check-eqv-pn
  check-eqv-b

check-eqv-b2
check-eqv-a
check-eqv-a2
match-brz
match-brzq
match-n
match-p
match-pn
match-b
match-b2
match-a
match-a2
in *SML* **module-name** *Rexp*

References

[1] T. Nipkow and D. Traytel. Unified decision procedures for regular expression equivalence. http://www.in.tum.de/~nipkow/pubs/regex_equiv.pdf, 2014.