

Regular Expression Equivalence

Tobias Nipkow Dmitriy Traytel

February 23, 2021

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	3
2	Finiteness of Derivatives Modulo ACI	4
2.1	ACI normalization	5
2.2	Atoms	6
2.3	Language	6
2.4	Finiteness of ACI-Equivalent Derivatives	7
2.5	Deriving preserves ACI-equivalence	8
2.6	Alternative ACI defintions	9
3	Connection Between Derivatives and Partial Derivatives	11
4	Framework Instantiations using (Partial) Derivatives	15
4.1	Brzowski Derivatives Modulo ACI	15
4.2	Brzowski Derivatives Modulo ACI Operating on the Quotient Type	16
4.3	Brzowski Derivatives Modulo ACI++ (Only Soundness)	16
4.4	Partial Derivatives	17
4.5	Languages as States	17

5	Framework Instantiations using Marked Regular Expressions	18
5.1	Marked Regular Expressions	18
5.2	Mark Before Atom	19
5.3	Mark After Atom	20
6	Linear Time Optimization for “Mark After Atom”	22
7	Linear Time Optimization for “Mark Before Atom” (for a Fixed Alphabet)	26
8	Various Algorithms for Regular Expression Equivalence	30

1 Regular Expressions Equivalence Framework

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

where

add-atoms Zero = *id*
| *add-atoms* One = *id*
| *add-atoms* (Atom *a*) = List.insert *a*
| *add-atoms* (Plus *r s*) = *add-atoms s* o *add-atoms r*
| *add-atoms* (Times *r s*) = *add-atoms s* o *add-atoms r*
| *add-atoms* (Star *r*) = *add-atoms r*

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

lemma *rtrancl-fold-product*:

shows $\{((r,s),(f\ a\ r,f\ a\ s)) \mid r\ s\ a.\ a : A\}^{\widehat{*}} =$
 $\{((r,s),(fold\ f\ w\ r,fold\ f\ w\ s)) \mid r\ s\ w.\ w : lists\ A\}$ (is ?L = ?R)
<proof>

lemma *rtrancl-fold-product1*:

shows $\{(r,s).\ \exists a \in A.\ s = f\ a\ r\}^{\widehat{*}} =$
 $\{(r,fold\ f\ w\ r) \mid r\ w.\ w : lists\ A\}$ (is ?L = ?R)
<proof>

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

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

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*: $L(\text{init } r) = \text{lang } r$
assumes *L-delta*: $L(\text{delta } a \ s) = \text{Deriv } a \ (L \ s)$
assumes *final-iff-Nil*: $\text{final } s \longleftrightarrow [] \in L \ s$
begin

lemma *L-deltas*: $L(\text{fold } \text{delta } w \ s) = \text{fold } \text{Deriv } w \ (L \ s)$
 ⟨proof⟩

definition *closure* :: 'a list ⇒ 's * 's ⇒ (('s * 's) list * ('s * 's) set) option
where
closure as = *rtrancl-while* ($\lambda(p,q). \text{final } p = \text{final } q$)
 ($\lambda(p,q). \text{map } (\lambda a. (\text{delta } a \ p, \text{delta } a \ q)) \ as$)

theorem *closure-sound-complete*:
assumes *result*: $\text{closure } as \ (\text{init } r, \text{init } s) = \text{Some}(ws, R)$
and *atoms*: $\text{set } as = \text{atoms } r \cup \text{atoms } s$
shows $ws = [] \longleftrightarrow \text{lang } r = \text{lang } s$
 ⟨proof⟩

1.1 The overall procedure

definition *check-equiv* :: 'a rexp ⇒ 'a rexp ⇒ bool **where**
check-equiv 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-equiv r s* **shows** $\text{lang } r = \text{lang } s$
 ⟨proof⟩

Auxiliary functions:

definition *reachable* :: 'a list ⇒ 'a rexp ⇒ 's set **where**
reachable as s =
 snd(*the(rtrancl-while* ($\lambda-. \text{True}$) ($\lambda s. \text{map } (\lambda a. \text{delta } a \ s) \ as$) (*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 = $\lambda((ws, Z), A). ws \neq []$;
 step = $\lambda((ws, Z), A).$
 (let *s* = *hd ws*;

```

    new-edges = map ( $\lambda a. ((s, a), \text{delta } a \ s)$ ) as;
    new = remdups (filter ( $\lambda ss. ss \notin Z$ ) (map snd new-edges))
    in ((new @ tl ws, set new  $\cup$  Z), set new-edges  $\cup$  A))
  in while-option test step start))

```

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

lemma *match-correct*: *match* s w \longleftrightarrow w \in lang s
 <proof>

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)}
 <proof>

lemma *termination*: \exists st. closure as (init r,init s) = Some st (is \exists -. closure as ?i
 = -)
 <proof>

lemma *completeness*:
assumes lang r = lang s **shows** check-equiv r s
 <proof>

lemma *finite-rtrancl-delta-Image1*:
 finite ({(r,s). \exists a \in A. s = delta a r} * “ {init r})
 <proof>

lemma *reachable*: reachable as r = {fold delta w (init r) | w. w \in lists (set as)}
and *finite-reachable*: finite (reachable as r)
 <proof>

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 $PLUS \equiv \text{rexp-of-list Plus Zero}$
abbreviation $TIMES \equiv \text{rexp-of-list Times One}$

lemma *list-singleton-induct* [*case-names nil single cons*]:
assumes $P []$ **and** $\bigwedge x. P [x]$ **and** $\bigwedge x y xs. P (y \# xs) \implies P (x \# (y \# xs))$
shows $P xs$
<proof>

2.1 ACI normalization

fun *toplevel-summands* :: $'a \text{ rexp} \Rightarrow 'a \text{ rexp set}$ **where**
 $\text{toplevel-summands } (Plus\ r\ s) = \text{toplevel-summands } r \cup \text{toplevel-summands } s$
 $\text{toplevel-summands } r = \{r\}$

abbreviation *flatten LISTOP* $X \equiv LISTOP$ (*sorted-list-of-set* X)

lemma *toplevel-summands-nonempty*[*simp*]:
 $\text{toplevel-summands } r \neq \{\}$
<proof>

lemma *toplevel-summands-finite*[*simp*]:
 $\text{finite } (\text{toplevel-summands } r)$
<proof>

primrec *ACI-norm* :: $('a::\text{linorder}) \text{ rexp} \Rightarrow 'a \text{ rexp}$ ($\llcorner - \lrcorner$) **where**
 $\llcorner Zero \lrcorner = Zero$
 $\llcorner One \lrcorner = One$
 $\llcorner Atom\ a \lrcorner = Atom\ a$
 $\llcorner Plus\ r\ s \lrcorner = \text{flatten } PLUS\ (\text{toplevel-summands } (Plus\ \llcorner r \lrcorner\ \llcorner s \lrcorner))$
 $\llcorner Times\ r\ s \lrcorner = Times\ \llcorner r \lrcorner\ \llcorner s \lrcorner$
 $\llcorner Star\ r \lrcorner = Star\ \llcorner r \lrcorner$

lemma *Plus-toplevel-summands*: $Plus\ r\ s \in \text{toplevel-summands } t \implies False$
<proof>

lemma *toplevel-summands-not-Plus*[*simp*]:
 $(\forall r\ s. x \neq Plus\ r\ s) \implies \text{toplevel-summands } x = \{x\}$
<proof>

lemma *toplevel-summands-PLUS-strong*:
 $\llbracket xs \neq []; \text{list-all } (\lambda x. \neg(\exists r\ s. x = Plus\ r\ s))\ xs \rrbracket \implies \text{toplevel-summands } (PLUS\ xs) = \text{set } xs$
<proof>

lemma *toplevel-summands-flatten*:
 $\llbracket X \neq \{\}; \text{finite } X; \forall x \in X. \neg(\exists r\ s. x = Plus\ r\ s) \rrbracket \implies \text{toplevel-summands } (\text{flatten } PLUS\ X) = X$
<proof>

lemma *ACI-norm-Plus*: $\langle r \rangle = \text{Plus } s \ t \implies \exists s \ t. r = \text{Plus } s \ t$
 ⟨proof⟩

lemma *toplevel-summands-flatten-ACI-norm-image*:
 $\text{toplevel-summands } (\text{flatten PLUS } (\text{ACI-norm } \text{' toplevel-summands } r)) = \text{ACI-norm } \text{' toplevel-summands } r$
 ⟨proof⟩

lemma *toplevel-summands-flatten-ACI-norm-image-Union*:
 $\text{toplevel-summands } (\text{flatten PLUS } (\text{ACI-norm } \text{' toplevel-summands } r \cup \text{ACI-norm } \text{' toplevel-summands } s)) =$
 $\text{ACI-norm } \text{' toplevel-summands } r \cup \text{ACI-norm } \text{' toplevel-summands } s$
 ⟨proof⟩

lemma *toplevel-summands-ACI-norm*:
 $\text{toplevel-summands } \langle r \rangle = \text{ACI-norm } \text{' toplevel-summands } r$
 ⟨proof⟩

lemma *ACI-norm-flatten*:
 $\langle r \rangle = \text{flatten PLUS } (\text{ACI-norm } \text{' toplevel-summands } r)$
 ⟨proof⟩

theorem *ACI-norm-idem[simp]*: $\langle \langle r \rangle \rangle = \langle r \rangle$
 ⟨proof⟩

2.2 Atoms

lemma *atoms-toplevel-summands*:
 $\text{atoms } s = (\bigcup r \in \text{toplevel-summands } s. \text{atoms } r)$
 ⟨proof⟩

lemma *wf-PLUS*: $\text{atoms } (\text{PLUS } xs) \subseteq \Sigma \iff (\forall r \in \text{set } xs. \text{atoms } r \subseteq \Sigma)$
 ⟨proof⟩

lemma *atoms-PLUS*: $\text{atoms } (\text{PLUS } xs) = (\bigcup r \in \text{set } xs. \text{atoms } r)$
 ⟨proof⟩

lemma *atoms-flatten-PLUS*:
 $\text{finite } X \implies \text{atoms } (\text{flatten PLUS } X) = (\bigcup r \in X. \text{atoms } r)$
 ⟨proof⟩

theorem *atoms-ACI-norm*: $\text{atoms } \langle r \rangle = \text{atoms } r$
 ⟨proof⟩

2.3 Language

lemma *toplevel-summands-lang*: $r \in \text{toplevel-summands } s \implies \text{lang } r \subseteq \text{lang } s$
 ⟨proof⟩

lemma *toplevel-summands-lang-UN*:

$lang\ s = (\bigcup_{r \in \text{toplevel-summands } s} lang\ r)$
(proof)

lemma *toplevel-summands-in-lang*:

$w \in lang\ s = (\exists r \in \text{toplevel-summands } s. w \in lang\ r)$
(proof)

lemma *lang-PLUS*: $lang\ (PLUS\ xs) = (\bigcup_{r \in \text{set } xs} lang\ r)$
(proof)

lemma *lang-PLUS-map[simp]*:

$lang\ (PLUS\ (\text{map } f\ xs)) = (\bigcup_{a \in \text{set } xs} lang\ (f\ a))$
(proof)

lemma *lang-flatten-PLUS[simp]*:

$\text{finite } X \implies lang\ (\text{flatten } PLUS\ X) = (\bigcup_{r \in X} lang\ r)$
(proof)

theorem *lang-ACI-norm[simp]*: $lang\ \langle r \rangle = lang\ r$
(proof)

2.4 Finiteness of ACI-Equivalent Derivatives

lemma *toplevel-summands-deriv*:

$\text{toplevel-summands } (\text{deriv } as\ r) = (\bigcup_{s \in \text{toplevel-summands } r} \text{toplevel-summands } (\text{deriv } as\ s))$
(proof)

lemma *derivs-Zero[simp]*: $\text{derivs } xs\ Zero = Zero$
(proof)

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

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

lemma *derivs-Plus*: $\text{derivs } xs\ (\text{Plus } r\ s) = \text{Plus } (\text{derivs } xs\ r)\ (\text{derivs } xs\ s)$
(proof)

lemma *derivs-PLUS*: $\text{derivs } xs\ (PLUS\ ys) = PLUS\ (\text{map } (\text{derivs } xs)\ ys)$
(proof)

lemma *toplevel-summands-deriv-Times*: $\text{toplevel-summands } (\text{deriv } xs\ (\text{Times } r\ s)) \subseteq$
 $\{\text{Times } (\text{derivs } xs\ r)\ s\} \cup$
 $\{r'. \exists ys\ zs. r' \in \text{toplevel-summands } (\text{derivs } ys\ s) \wedge ys \neq [] \wedge zs @ ys = xs\}$
(proof)

lemma *toplevel-summands-derivs-Star-nonempty*:

$xs \neq [] \implies \text{toplevel-summands } (\text{derivs } xs \text{ (Star } r)) \subseteq$
 $\{\text{Times } (\text{derivs } ys \text{ } r) \text{ (Star } r) \mid ys. \exists zs. ys \neq [] \wedge zs @ ys = xs\}$
(proof)

lemma *toplevel-summands-derivs-Star*:

$\text{toplevel-summands } (\text{derivs } xs \text{ (Star } r)) \subseteq$
 $\{\text{Star } r\} \cup \{\text{Times } (\text{derivs } ys \text{ } r) \text{ (Star } r) \mid ys. \exists zs. ys \neq [] \wedge zs @ ys = xs\}$
(proof)

lemma *toplevel-summands-PLUS*:

$xs \neq [] \implies \text{toplevel-summands } (\text{PLUS } (\text{map } f \text{ } xs)) = (\bigcup r \in \text{set } xs. \text{toplevel-summands } (f \text{ } r))$
(proof)

lemma *ACI-norm-toplevel-summands-Zero*: $\text{toplevel-summands } r \subseteq \{\text{Zero}\} \implies$

$\langle r \rangle = \text{Zero}$
(proof)

lemma *finite-ACI-norm-toplevel-summands*:

$\text{finite } \{f \text{ } \langle s \rangle \mid s. \text{toplevel-summands } s \subseteq B\}$ if finite B
(proof)

theorem *finite-derivs*: $\text{finite } \{\langle \text{derivs } xs \text{ } r \rangle \mid xs. \text{True}\}$

(proof)

2.5 Deriving preserves ACI-equivalence

lemma *ACI-norm-PLUS*:

$\text{list-all2 } (\lambda r \text{ } s. \langle r \rangle = \langle s \rangle) \text{ } xs \text{ } ys \implies \langle \text{PLUS } xs \rangle = \langle \text{PLUS } ys \rangle$
(proof)

lemma *toplevel-summands-ACI-norm-deriv*:

$(\bigcup a \in \text{toplevel-summands } r. \text{toplevel-summands } \langle \text{deriv as } \langle a \rangle \rangle) = \text{toplevel-summands } \langle \text{deriv as } \langle r \rangle \rangle$
(proof)

lemma *toplevel-summands-nullable*:

$\text{nullable } s = (\exists r \in \text{toplevel-summands } s. \text{nullable } r)$
(proof)

lemma *nullable-PLUS*:

$\text{nullable } (\text{PLUS } xs) = (\exists r \in \text{set } xs. \text{nullable } r)$
(proof)

theorem *ACI-norm-nullable*: $\text{nullable } \langle r \rangle = \text{nullable } r$

(proof)

theorem *ACI-norm-deriv*: $\langle\langle \text{deriv as } \langle r \rangle \rangle\rangle = \langle\langle \text{deriv as } r \rangle\rangle$
 ⟨proof⟩

corollary *deriv-preserves*: $\langle r \rangle = \langle s \rangle \implies \langle\langle \text{deriv as } r \rangle\rangle = \langle\langle \text{deriv as } s \rangle\rangle$
 ⟨proof⟩

lemma *derivs-snoc[simp]*: $\text{derivs } (xs @ [x]) r = (\text{deriv } x (\text{derivs } xs r))$
 ⟨proof⟩

theorem *ACI-norm-deriv*: $\langle\langle \text{derivs } xs \langle r \rangle \rangle\rangle = \langle\langle \text{derivs } xs r \rangle\rangle$
 ⟨proof⟩

2.6 Alternative ACI defintions

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

fun *ACI-nPlus* :: 'a::linorder rexp \Rightarrow 'a rexp \Rightarrow '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*)
 ⟨proof⟩

lemma *toplevel-summands-ACI-norm-alt*:

toplevel-summands (*ACI-norm-alt* *r*) = *ACI-norm-alt* ' *toplevel-summands* *r*
 ⟨proof⟩

lemma *ACI-norm-alt-Plus*:

ACI-norm-alt *r* = *Plus* *s* *t* $\implies \exists s t. r = \text{Plus } s t$
 ⟨proof⟩

lemma *toplevel-summands-flatten-ACI-norm-alt-image*:

toplevel-summands (*flatten* *PLUS* (*ACI-norm-alt* ' *toplevel-summands* *r*)) =

ACI-norm-alt ‘*toplevel-summands r*
(*proof*)

lemma *ACI-norm-ACI-norm-alt*: «*ACI-norm-alt r*» = «*r*»
(*proof*)

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*)

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*)

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)*
(*proof*)

lemma *ACI-nPlus-ACI-norm [simp]*: *ACI-nPlus* «*r*» «*s*» = «*Plus r s*»
(*proof*)

lemma *ACI-norm-alt*:
ACI-norm-alt r = «*r*»
(*proof*)

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*
(*proof*)

lemma *ACI-nullable*: *ACI r s* ⇒ *nullable r* = *nullable s*
(*proof*)

lemma *ACI-lang*: $ACI\ r\ s \implies lang\ r = lang\ s$
 ⟨proof⟩

lemma *ACI-deriv*: $ACI\ r\ s \implies ACI\ (deriv\ a\ r)\ (deriv\ a\ s)$
 ⟨proof⟩

lemma *ACI-Plus-assocI*[intro]:
 $ACI\ (Plus\ r1\ r2)\ s2 \implies ACI\ (Plus\ r1\ (Plus\ s1\ r2))\ (Plus\ s1\ s2)$
 $ACI\ (Plus\ r1\ r2)\ s2 \implies ACI\ (Plus\ r1\ (Plus\ r2\ s1))\ (Plus\ s1\ s2)$
 ⟨proof⟩

lemma *ACI-Plus-idemI*[intro]: $\llbracket ACI\ r\ s1; ACI\ r\ s2 \rrbracket \implies ACI\ r\ (Plus\ s1\ s2)$
 ⟨proof⟩

lemma *ACI-Plus-idemI'*[intro]:
 $\llbracket ACI\ r1\ s1; ACI\ (Plus\ r1\ r2)\ s2 \rrbracket \implies ACI\ (Plus\ r1\ r2)\ (Plus\ s1\ s2)$
 ⟨proof⟩

lemma *ACI-ACI-nPlus*: $\llbracket ACI\ r1\ s1; ACI\ r2\ s2 \rrbracket \implies ACI\ (ACI-nPlus\ r1\ r2)\ (Plus\ s1\ s2)$
 ⟨proof⟩

lemma *ACI-ACI-norm*: $ACI\ \llbracket r \rrbracket\ r$
 ⟨proof⟩

lemma *ACI-norm-eqI*: $ACI\ r\ s \implies \llbracket r \rrbracket = \llbracket s \rrbracket$
 ⟨proof⟩

lemma *ACI-I*: $\llbracket r \rrbracket = \llbracket s \rrbracket \implies ACI\ r\ s$
 ⟨proof⟩

lemma *ACI-decidable*: $ACI\ r\ s = (\llbracket r \rrbracket = \llbracket s \rrbracket)$
 ⟨proof⟩

3 Connection Between Derivatives and Partial Derivatives

lemma *pderiv-not-is-Zero-is-Plus*[simp]: $\forall x \in pderiv\ a\ r. \neg is-Zero\ x \wedge \neg is-Plus\ x$
 ⟨proof⟩

lemma *finite-pderiv*[simp]: $finite\ (pderiv\ a\ r)$
 ⟨proof⟩

lemma *PLUS-inject*: $\llbracket \forall x \in set\ xs \cup set\ ys. \neg is-Zero\ x \wedge \neg is-Plus\ x; sorted\ xs; sorted\ ys \rrbracket \implies$
 $(PLUS\ xs = PLUS\ ys) \iff xs = ys$

<proof>

lemma *sorted-list-of-set-inject*: $\llbracket \text{finite } R; \text{finite } S \rrbracket \implies$
 $(\text{sorted-list-of-set } R = \text{sorted-list-of-set } S) \iff R = S$
<proof>

lemma *flatten-PLUS-inject*: $\llbracket \forall x \in R \cup S. \neg \text{is-Zero } x \wedge \neg \text{is-Plus } x; \text{finite } R;$
 $\text{finite } S \rrbracket \implies$
 $(\text{flatten PLUS } R = \text{flatten PLUS } S) = (R = S)$
<proof>

primrec *pset* **where**

$\text{pset Zero} = \{\}$
 $\text{pset One} = \{\text{One}\}$
 $\text{pset (Atom } a) = \{\text{Atom } a\}$
 $\text{pset (Plus } r \ s) = \text{pset } r \cup \text{pset } s$
 $\text{pset (Times } r \ s) = \text{Times } (\text{pset } r) \ s$
 $\text{pset (Star } r) = \{\text{Star } r\}$

lemma *pset-not-is-Zero-is-Plus[simp]*: $\forall x \in \text{pset } r. \neg \text{is-Zero } x \wedge \neg \text{is-Plus } x$
<proof>

lemma *finite-pset[simp]*: $\text{finite } (\text{pset } r)$
<proof>

lemma *pset-deriv*: $\text{pset } (\text{deriv } a \ r) = \text{pderiv } a \ r$
<proof>

definition *pnorm* **where**

$\text{pnorm} = \text{flatten PLUS } o \ \text{pset}$

lemma *pnorm-deriv-eq-iff-pderiv-eq*:

$\text{pnorm } (\text{deriv } a \ r) = \text{pnorm } (\text{deriv } a \ s) \iff \text{pderiv } a \ r = \text{pderiv } a \ s$
<proof>

fun *pnPlus* :: $'a::\text{linorder rexp} \Rightarrow 'a \ \text{rexp} \Rightarrow 'a \ \text{rexp}$ **where**

$\text{pnPlus Zero } r = r$
 $\text{pnPlus } r \ \text{Zero} = r$
 $\text{pnPlus (Plus } r \ s) \ t = \text{pnPlus } r \ (\text{pnPlus } s \ t)$
 $\text{pnPlus } r \ (\text{Plus } s \ t) =$
 $(\text{if } r = s \ \text{then } (\text{Plus } s \ t)$
 $\text{else if le-rexp } r \ s \ \text{then Plus } r \ (\text{Plus } s \ t)$
 $\text{else Plus } s \ (\text{pnPlus } r \ t))$
 $\text{pnPlus } r \ s =$
 $(\text{if } r = s \ \text{then } r$
 $\text{else if le-rexp } r \ s \ \text{then Plus } r \ s$
 $\text{else Plus } s \ r)$

fun *pnTimes* :: $'a::\text{linorder rexp} \Rightarrow 'a \ \text{rexp} \Rightarrow 'a \ \text{rexp}$ **where**

$pnTimes\ Zero\ r = Zero$
 $| pnTimes\ (Plus\ r\ s)\ t = pnPlus\ (pnTimes\ r\ t)\ (pnTimes\ s\ t)$
 $| pnTimes\ r\ s = Times\ r\ s$

primrec $pnorm-alt :: 'a::linorder\ rexp \Rightarrow 'a\ rexp$ **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 $pset-pnPlus$:

$pset\ (pnPlus\ r\ s) = pset\ (Plus\ r\ s)$
 $\langle proof \rangle$

lemma $pset-pnTimes$:

$pset\ (pnTimes\ r\ s) = pset\ (Times\ r\ s)$
 $\langle proof \rangle$

lemma $pset-pnorm-alt-Times$: $s \in pset\ r \Longrightarrow pnTimes\ (pnorm-alt\ s)\ t = Times\ (pnorm-alt\ s)\ t$

$\langle proof \rangle$

lemma $pset-pnorm-alt$:

$pset\ (pnorm-alt\ r) = pnorm-alt\ ` pset\ r$
 $\langle proof \rangle$

lemma $pset-pnTimes-Times$: $s \in pset\ r \Longrightarrow pnTimes\ s\ t = Times\ s\ t$

$\langle proof \rangle$

lemma $pset-pnorm-alt-id$: $s \in pset\ r \Longrightarrow pnorm-alt\ s = s$

$\langle proof \rangle$

lemma $pnorm-alt-image-pset$: $pnorm-alt\ ` pset\ r = pset\ r$

$\langle proof \rangle$

lemma $pnorm-pnorm-alt$: $pnorm\ (pnorm-alt\ r) = pnorm\ r$

$\langle proof \rangle$

lemma $pnPlus-singleton-PLUS$:

$\llbracket xs \neq []; sorted\ xs; distinct\ xs; \forall x \in \{x\} \cup set\ xs. \neg is-Zero\ x \wedge \neg is-Plus\ x \rrbracket \Longrightarrow$
 $pnPlus\ x\ (PLUS\ xs) = (if\ x \in set\ xs\ then\ PLUS\ xs\ else\ PLUS\ (insort\ x\ xs))$
 $\langle proof \rangle$

lemma $pnPlus-PlusL[simp]$: $t \neq Zero \Longrightarrow pnPlus\ (Plus\ r\ s)\ t = pnPlus\ r\ (pnPlus\ s\ t)$

$\langle proof \rangle$

lemma *pnPlus-ZeroR[simp]*: $pnPlus\ r\ Zero = r$
 ⟨proof⟩

lemma *PLUS-eq-Zero*: $PLUS\ xs = Zero \longleftrightarrow xs = [] \vee xs = [Zero]$
 ⟨proof⟩

lemma *pnPlus-PLUS*:
 $\llbracket xs1 \neq []; xs2 \neq []; \forall x \in set\ (xs1\ @\ xs2). \neg is-Zero\ x \wedge \neg is-Plus\ x; sorted\ xs2; distinct\ xs2 \rrbracket \implies$
 $pnPlus\ (PLUS\ xs1)\ (PLUS\ xs2) = flatten\ PLUS\ (set\ (xs1\ @\ xs2))$
 ⟨proof⟩

lemma *pnPlus-flatten-PLUS*:
 $\llbracket X1 \neq \{\}; X2 \neq \{\}; finite\ X1; finite\ X2; \forall x \in X1 \cup X2. \neg is-Zero\ x \wedge \neg is-Plus\ x \rrbracket \implies$
 $pnPlus\ (flatten\ PLUS\ X1)\ (flatten\ PLUS\ X2) = flatten\ PLUS\ (X1 \cup X2)$
 ⟨proof⟩

lemma *pnPlus-pnorm*: $pnPlus\ (pnorm\ r)\ (pnorm\ s) = pnorm\ (Plus\ r\ s)$
 ⟨proof⟩

lemma *pnTimes-not-Zero-or-Plus[simp]*: $\llbracket \neg is-Zero\ x; \neg is-Plus\ x \rrbracket \implies pnTimes\ x\ r = Times\ x\ r$
 ⟨proof⟩

lemma *pnTimes-PLUS*:
 $\llbracket xs \neq []; \forall x \in set\ xs. \neg is-Zero\ x \wedge \neg is-Plus\ x \rrbracket \implies$
 $pnTimes\ (PLUS\ xs)\ r = flatten\ PLUS\ (Times\ (set\ xs)\ r)$
 ⟨proof⟩

lemma *pnTimes-flatten-PLUS*:
 $\llbracket X1 \neq \{\}; finite\ X1; \forall x \in X1. \neg is-Zero\ x \wedge \neg is-Plus\ x \rrbracket \implies$
 $pnTimes\ (flatten\ PLUS\ X1)\ r = flatten\ PLUS\ (Times\ X1\ r)$
 ⟨proof⟩

lemma *pnTimes-pnorm*: $pnTimes\ (pnorm\ r1)\ r2 = pnorm\ (Times\ r1\ r2)$
 ⟨proof⟩

lemma *pnorm-alt[symmetric]*: $pnorm-alt\ r = pnorm\ r$
 ⟨proof⟩

lemma *insort-eq-Cons*: $\llbracket \forall a \in set\ xs. b < a; sorted\ xs \rrbracket \implies insort\ b\ xs = b \# xs$
 ⟨proof⟩

lemma *pderiv-PLUS*: $pderiv\ a\ (PLUS\ (x\ \# \ xs)) = pderiv\ a\ x \cup pderiv\ a\ (PLUS\ xs)$
 ⟨proof⟩

lemma *pderiv-set-flatten-PLUS*:

$finite\ X \implies pderiv\ (a :: 'a :: linorder)\ (flatten\ PLUS\ X) = pderiv\text{-}set\ a\ X$
 ⟨proof⟩

lemma *fold-pderiv-set-flatten-PLUS*:

$\llbracket w \neq []; finite\ X \rrbracket \implies fold\ pderiv\text{-}set\ w\ \{flatten\ PLUS\ X\} = fold\ pderiv\text{-}set\ w\ X$
 ⟨proof⟩

lemma *fold-pnorm-deriv*:

$fold\ (\lambda a\ r.\ pnorm\ (deriv\ a\ r))\ w\ s = flatten\ PLUS\ (fold\ pderiv\text{-}set\ w\ \{s\})$
 ⟨proof⟩

primrec

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

where

$pderiv\ c\ (Zero) = Zero$
 $| pderiv\ c\ (One) = Zero$
 $| pderiv\ c\ (Atom\ c') = (if\ c = c'\ then\ One\ else\ Zero)$
 $| pderiv\ c\ (Plus\ r1\ r2) = pnPlus\ (pderiv\ c\ r1)\ (pderiv\ c\ r2)$
 $| pderiv\ c\ (Times\ r1\ r2) =$
 $\quad (if\ nullable\ r1\ then\ pnPlus\ (pnTimes\ (pderiv\ c\ r1)\ r2)\ (pderiv\ c\ r2)\ else$
 $\quad pnTimes\ (pderiv\ c\ r1)\ r2)$
 $| pderiv\ c\ (Star\ r) = pnTimes\ (pderiv\ c\ r)\ (Star\ r)$

lemma *pderiv-alt*[code]: $pderiv\ a\ r = pnorm\ (deriv\ a\ r)$

⟨proof⟩

lemma *pderiv-pderiv*: $pderiv\ a\ r = flatten\ PLUS\ (pderiv\ a\ r)$

⟨proof⟩

4 Framework Instantiations using (Partial) Derivatives

4.1 Brzozowski Derivatives Modulo ACI

lemma *ACI-norm-derivs-alt*: $\llbracket derivs\ w\ r \rrbracket = fold\ (\lambda a\ r.\ \llbracket deriv\ a\ r \rrbracket)\ w\ \llbracket r \rrbracket$

⟨proof⟩

global-interpretation *brzozowski*: *rexp-DFA* $\lambda r.\ \llbracket r \rrbracket\ \lambda a\ r.\ \llbracket deriv\ a\ r \rrbracket\ nullable\ lang$

defines *brzozowski-closure* = *brzozowski.closure*

and *check-equiv-brz* = *brzozowski.check-equiv*

and *reachable-brz* = *brzozowski.reachable*

and *automaton-brz* = *brzozowski.automaton*

and *match-brz* = *brzozowski.match*

⟨proof⟩

4.2 Brzowski Derivatives Modulo ACI Operating on the Quotient Type

lemma *derivs-alt*: *derivs* = *fold deriv*
 ⟨*proof*⟩

functor *map-rexp* ⟨*proof*⟩

quotient-type 'a *ACI-rexp* = 'a *rexp* / *ACI*
morphisms *rep-ACI-rexp* *ACI-class*
 ⟨*proof*⟩

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*»
 ⟨*proof*⟩

lift-definition *less-ACI-rexp* :: 'a *ACI-rexp* ⇒ 'a *ACI-rexp* ⇒ *bool* **is** λ*r s*. *less*
 «*r*» «*s*»
 ⟨*proof*⟩

lift-definition *equal-ACI-rexp* :: 'a *ACI-rexp* ⇒ 'a *ACI-rexp* ⇒ *bool* **is** λ*r s*. «*r*»
 = «*s*»
 ⟨*proof*⟩

instance ⟨*proof*⟩

end

lift-definition *ACI-deriv* :: 'a :: *linorder* ⇒ 'a *ACI-rexp* ⇒ 'a *ACI-rexp* **is** *deriv*
 ⟨*proof*⟩

lift-definition *ACI-nullable* :: 'a :: *linorder* *ACI-rexp* ⇒ *bool* **is** *nullable*
 ⟨*proof*⟩

lift-definition *ACI-lang* :: 'a :: *linorder* *ACI-rexp* ⇒ 'a *lang* **is** *lang*
 ⟨*proof*⟩

lemma [*transfer-rule*]: *rel-fun* (*rel-set* (*pcr-ACI-rexp* (=))) (=) (*finite* o *image*
ACI-norm) *finite*
 ⟨*proof*⟩

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

defines *brzowski-quotient-closure* = *brzowski-quotient.closure*

and *check-eqv-brzq* = *brzowski-quotient.check-eqv*

and *reachable-brzq* = *brzowski-quotient.reachable*

and *automaton-brzq* = *brzowski-quotient.automaton*

and *match-brzq* = *brzowski-quotient.match*

⟨*proof*⟩

4.3 Brzowski Derivatives Modulo ACI++ (Only Soundness)

global-interpretation *nderiv*: *rexp-DA* λ*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⟩

```

4.4 Partial Derivatives

```

global-interpretation pderiv: rexp-DFA  $\lambda r. \{r\}$  pderiv-set  $\lambda P. \exists p \in P. \text{nullable } p$ 
 $\lambda P. \bigcup (lang \text{ ' } P)$ 
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⟩

```

```

global-interpretation pnderiv: rexp-DFA  $\lambda r. r$  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⟩

```

4.5 Languages as States

Not executable but still instructive.

```

lemma Derivs-alt-def: Derivs w L = fold Deriv w L
⟨proof⟩

```

```

interpretation language: rexp-DFA lang Deriv  $\lambda L. [] \in L \text{ id}$ 
⟨proof⟩

```

```

definition str-eq :: 'a lang => ('a list × 'a list) set (≈- [100] 100)
where  $\approx A \equiv \{(x, y). (\forall z. x @ z \in A \longleftrightarrow y @ z \in A)\}$ 

```

```

lemma str-eq-alt:  $\approx A = \{(x, y). fold Deriv x A = fold Deriv y A\}$ 
⟨proof⟩

```

```

lemma Myhill-Nerode2: finite (UNIV //  $\approx lang r$ )
⟨proof⟩

```

5 Framework Instantiations using Marked Regular Expressions

5.1 Marked Regular Expressions

type-synonym $'a \text{ mrexps} = (\text{bool} * 'a) \text{ rexp}$

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

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

lemma $\text{finite-mrexps}[\text{simp}]$: $\text{finite } (\text{mrexps } r)$
 $\langle \text{proof} \rangle$

lemma strip-mrexps : $\text{strip } ' \text{mrexps } r = \{r\}$
 $\langle \text{proof} \rangle$

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

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

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

lemma $\text{read-mrexps}[\text{simp}]$: $r \in \text{mrexps } s \Longrightarrow \text{read } a \ r \in \text{mrexps } s$
 $\langle \text{proof} \rangle$

fun $\text{follow} :: \text{bool} \Rightarrow 'a \text{ mrexps} \Rightarrow 'a \text{ mrexps}$ **where**
 $\text{follow } m \ \text{Zero} = \text{Zero}$ |
 $\text{follow } m \ \text{One} = \text{One}$ |
 $\text{follow } m \ (\text{Atom}(-,a)) = \text{Atom}(m,a)$ |

$follow\ m\ (Plus\ r\ s) = Plus\ (follow\ m\ r)\ (follow\ m\ s) \mid$
 $follow\ m\ (Times\ r\ s) =$
 $\quad Times\ (follow\ m\ r)\ (follow\ (final\ r\ \vee\ m\ \wedge\ nullable\ r)\ s) \mid$
 $follow\ m\ (Star\ r) = Star(follow\ (final\ r\ \vee\ m)\ r)$

lemma *follow-mrexp[simp]*: $r \in mrexp\ s \implies follow\ b\ r \in mrexp\ s$
 $\langle proof \rangle$

lemma *strip-read[simp]*: $strip\ (read\ a\ r) = strip\ r$
 $\langle proof \rangle$

lemma *Nil-notin-Lm[simp]*: $\square \notin Lm\ r$
 $\langle proof \rangle$

lemma *Nil-in-lang-strip[simp]*: $\square \in lang(r) \longleftrightarrow \square \in lang(strip\ r)$
 $\langle proof \rangle$

lemma *strip-follow[simp]*: $strip(follow\ m\ r) = strip\ r$
 $\langle proof \rangle$

lemma *conc-lemma*: $\square \notin A \implies \{w : A\ @\@ B.\ w \neq \square \wedge P(hd\ w)\} = \{w : A.\ w \neq \square \wedge P(hd\ w)\} @\@ B$
 $\langle proof \rangle$

lemma *Lm-read*: $Lm\ (read\ a\ r) = \{w : Lm\ r.\ w \neq \square \wedge hd\ w = a\}$
 $\langle proof \rangle$

lemma *tl-conc[simp]*: $\square \notin A \implies tl\ '(A\ @\@ B) = tl\ 'A\ @\@ B$
 $\langle proof \rangle$

lemma *Nil-in-tl-Lm-if-final[simp]*: $final\ r \implies \square : tl\ 'Lm\ r$
 $\langle proof \rangle$

lemma *Nil-notin-tl-if-not-final*: $\neg final\ r \implies \square \notin tl\ 'Lm\ r$
 $\langle proof \rangle$

lemma *Lm-follow*: $Lm\ (follow\ m\ r) = tl\ 'Lm\ r \cup (if\ m\ then\ lang(strip\ r)\ else\ \{\})$
 $- \{\square\}$
 $\langle proof \rangle$

5.2 Mark Before Atom

Position automaton where mark is placed before atoms.

abbreviation *empty-mrexp* $\equiv map-rexp\ (\lambda a.\ (False, a))$

lemma *empty-mrexp-mrexp[simp]*: $empty-mrexp\ r \in mrexp\ r$
 $\langle proof \rangle$

lemma *nullable-empty-mrexp[simp]*: $nullable\ (empty-mrexp\ r) = nullable\ r$

<proof>

definition *init-b* $r = (\text{follow True } (\text{empty-mrexp } r), \text{nullable } r)$

lemma *init-b-mrexs[simp]*: $\text{init-b } r \in \text{mrexs } r \times \text{UNIV}$
<proof>

fun *delta-b* **where**

delta-b $a (r,b) = (\text{let } r' = \text{read } a \text{ } r \text{ in } (\text{follow False } r', \text{final } r'))$

lemma *delta-b-mrexs[simp]*: $rb \in \text{mrexs } r \times \text{UNIV} \implies \text{delta-b } a \text{ } rb \in \text{mrexs } r \times \text{UNIV}$
<proof>

lemma *fold-delta-b-init-b-mrexs[simp]*: $\text{fold } \text{delta-b } w (\text{init-b } s) \in \text{mrexs } s \times \text{UNIV}$
<proof>

fun *L-b* **where**

L-b $(r,b) = Lm \text{ } r \cup (\text{if } b \text{ then } \{\}\ \text{else } \{\})$

abbreviation *final-b* $\equiv \text{snd}$

lemma *Lm-empty*: $Lm (\text{empty-mrexp } r) = \{\}$
<proof>

lemma *final-read-Lm*: $\text{final}(\text{read } a \text{ } r) \longleftrightarrow [a] \in Lm \text{ } r$
<proof>

global-interpretation *before*: *rexp-DFA* *init-b* *delta-b* *final-b* *L-b*

defines *before-closure* = *before.closure*

and *check-equiv-b* = *before.check-equiv*

and *reachable-b* = *before.reachable*

and *automaton-b* = *before.automaton*

and *match-b* = *before.match*

<proof>

5.3 Mark After Atom

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

definition *init-a* $r = (\text{True}, \text{empty-mrexp } r)$

lemma *init-a-mrexs[simp]*: $\text{init-a } r \in \text{UNIV} \times \text{mrexs } r$
<proof>

fun *delta-a* **where**

delta-a $a (b,r) = (\text{False}, \text{read } a \text{ } (\text{follow } b \text{ } r))$

lemma *delta-a-mrexp[simp]*: $br \in UNIV \times mrexp\ r \implies \text{delta-a } a\ br \in UNIV \times mrexp\ r$
 ⟨proof⟩

lemma *fold-delta-a-init-a-mrexp[simp]*: $\text{fold delta-a } w\ (\text{init-a } s) \in UNIV \times mrexp\ s$
 ⟨proof⟩

fun *final-a* **where**
final-a (b,r) $\longleftrightarrow \text{final } r \vee b \wedge \text{nullable } r$

fun *L-a* **where**
L-a (b,r) = $Lm\ (\text{follow } b\ r) \cup (\text{if } \text{final-a}(b,r)\ \text{then } \{\}\ \text{else } \{\})$

lemma *nonfinal-empty-mrexp*: $\neg \text{final}\ (\text{empty-mrexp } r)$
 ⟨proof⟩

lemma *Cons-eq-tl-iff[simp]*: $x \# xs = \text{tl } ys \longleftrightarrow (\exists y. ys = y \# x \# xs)$
 ⟨proof⟩

lemma *tl-eq-Cons-iff[simp]*: $\text{tl } ys = x \# xs \longleftrightarrow (\exists y. ys = y \# x \# xs)$
 ⟨proof⟩

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⟩

The “before” automaton 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) = $(\text{follow } b\ r, \text{final-a } (b, r))$

lemma *hom-delta*: $\text{hom-ab } (\text{delta-a } x\ br) = \text{delta-b } x\ (\text{hom-ab } br)$
 ⟨proof⟩

lemma *hom-deltas*: $\text{hom-ab } (\text{fold delta-a } w\ br) = \text{fold delta-b } w\ (\text{hom-ab } br)$
 ⟨proof⟩

lemma *hom-init*: $\text{hom-ab } (\text{init-a } r) = \text{init-b } r$
 ⟨proof⟩

lemma *reachable-ab*: $\text{reachable-b } as\ r = \text{hom-ab } \text{ ` } \text{reachable-a } as\ r$
 ⟨proof⟩

theorem *card-reachable-ab*: $\text{card}(\text{reachable-b as } r) \leq \text{card}(\text{reachable-a as } r)$
 ⟨proof⟩

The implementation by Fischer et al.:

fun *shift* :: $\text{bool} \Rightarrow 'a \text{ mrexpr} \Rightarrow 'a \Rightarrow 'a \text{ mrexpr}$ **where**
shift - *One* - = *One* |
shift - *Zero* - = *Zero* |
shift *m* (*Atom* (-, *x*)) *c* = *Atom* (*m* \wedge (*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* \vee *m* \wedge *nullable* *r*) *s c*) |
shift *m* (*Star* *r*) *c* = *Star* (*shift* (*final* *r* \vee *m*) *r c*)

lemma *shift-read-follow*: $\text{shift } m \ r \ x = \text{read } x \ (\text{follow } m \ r)$
 ⟨proof⟩

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

fun *final1* :: $'a \text{ mrexpr} \Rightarrow 'a \Rightarrow \text{bool}$ **where**
final1 *Zero* - = *False* |
final1 *One* - = *False* |
final1 (*Atom*(*m*, *a*)) *x* = (*m* \wedge *a*=*x*) |
final1 (*Plus* *r s*) *x* = (*final1* *r x* \vee *final1* *s x*) |
final1 (*Times* *r s*) *x* = (*final1* *s x* \vee *nullable* *s* \wedge *final1* *r x*) |
final1 (*Star* *r*) *x* = *final1* *r x*

fun *move* :: $'a \Rightarrow 'a \text{ mrexpr} \Rightarrow \text{bool} \Rightarrow 'a \text{ mrexpr}$ **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* \vee *m* \wedge *nullable* *r*)) |
move *c* (*Star* *r*) *m* = *Star* (*move* *c r* (*final1* *r c* \vee *m*))

lemma *nullable-read[simp]*: $\text{nullable}(\text{read } c \ r) = \text{nullable } r$
 ⟨proof⟩

lemma *final-read-final1*: $\text{final}(\text{read } c \ r) = \text{final1 } r \ c$
 ⟨proof⟩

lemma *move-follow-read*: $\text{move } c \ r \ m = \text{follow } m \ (\text{read } c \ r)$
 ⟨proof⟩

6 Linear Time Optimization for “Mark After Atom”

datatype $'a \text{ mrexpr2} =$

```

Zero2 |
One2 |
Atom2 (fin: bool) 'a |
Plus2 'a mrex2 'a mrex2 (fin: bool) (nul: bool) |
Times2 'a mrex2 'a mrex2 (fin: bool) (nul: bool) |
Star2 'a mrex2 (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 \Rightarrow ('a mrex2) set **where**

```

mrexps2 Zero = {Zero2}
| mrexps2 One = {One2}
| mrexps2 (Atom a) = {Atom2 True a, Atom2 False a}
| mrexps2 (Plus r s) = ( $\lambda(r, s, f, n).$  Plus2 r s f n) ' (mrexps2 r  $\times$  mrexps2 s  $\times$  UNIV)
| mrexps2 (Times r s) = ( $\lambda(r, s, f, n).$  Times2 r s f n) ' (mrexps2 r  $\times$  mrexps2 s  $\times$  UNIV)
| mrexps2 (Star r) = ( $\lambda(r, f).$  Star2 r f) ' (mrexps2 r  $\times$  UNIV)

```

lemma *finite-mrexps3[simp]*: finite (mrexps2 r)
<proof>

definition^[simp]: plus2 r s == Plus2 r s (fin r \vee fin s) (nul r \vee nul s)

definition^[simp]: times2 r s == Times2 r s (fin r \wedge nul s \vee fin s) (nul r \wedge nul s)

definition^[simp]: star2 r == Star2 r (fin r)

primrec *empty-mrex2* :: 'a rexp \Rightarrow 'a mrex2 **where**

```

empty-mrex2 Zero = Zero2 |
empty-mrex2 One = One2 |
empty-mrex2 (Atom x) = Atom2 False x |
empty-mrex2 (Plus r s) = plus2 (empty-mrex2 r) (empty-mrex2 s) |
empty-mrex2 (Times r s) = times2 (empty-mrex2 r) (empty-mrex2 s) |
empty-mrex2 (Star r) = star2 (empty-mrex2 r)

```

primrec *shift2* :: bool \Rightarrow 'a mrex2 \Rightarrow 'a \Rightarrow 'a mrex2 **where**

```

shift2 - One2 - = One2 |
shift2 - Zero2 - = Zero2 |
shift2 m (Atom2 - x) c = Atom2 (m  $\wedge$  (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  $\wedge$  nul r  $\vee$  fin r) s c) |
shift2 m (Star2 r -) c = star2 (shift2 (m  $\vee$  fin r) r c)

```

primrec *strip2* **where**

$strip2\ Zero2 = Zero \mid$
 $strip2\ One2 = One \mid$
 $strip2\ (Atom2\ m\ x) = Atom\ (m,\ x) \mid$
 $strip2\ (Plus2\ r\ s\ -) = Plus\ (strip2\ r)\ (strip2\ s) \mid$
 $strip2\ (Times2\ r\ s\ -) = Times\ (strip2\ r)\ (strip2\ s) \mid$
 $strip2\ (Star2\ r\ -) = Star\ (strip2\ r)$

lemma *strip-mrexp2*: $(strip\ o\ strip2)\ 'mrexp2\ r = \{r\}$
 $\langle proof \rangle$

primrec *ok2* :: 'a mrexp2 \Rightarrow bool **where**

$ok2\ Zero2 = True \mid$
 $ok2\ One2 = True \mid$
 $ok2\ (Atom2\ -) = True \mid$
 $ok2\ (Plus2\ r\ s\ f\ n) = (ok2\ r \wedge ok2\ s \wedge$
 $(let\ rs = Plus\ (strip2\ r)\ (strip2\ s)\ in\ f = final\ rs \wedge n = nullable\ rs)) \mid$
 $ok2\ (Times2\ r\ s\ f\ n) = (ok2\ r \wedge ok2\ s \wedge$
 $(let\ rs = Times\ (strip2\ r)\ (strip2\ s)\ in\ f = final\ rs \wedge n = nullable\ rs)) \mid$
 $ok2\ (Star2\ r\ f) = (ok2\ r \wedge f = final\ (strip2\ r))$

lemma *ok2-fin-final[simp]*: $ok2\ r \Longrightarrow fin\ r = final\ (strip2\ r)$
 $\langle proof \rangle$

lemma *ok2-nul-nullable[simp]*: $ok2\ r \Longrightarrow nul\ r = nullable\ (strip2\ r)$
 $\langle proof \rangle$

lemma *strip2-shift2*: $ok2\ r \Longrightarrow strip2(shift2\ m\ r\ c) = shift\ m\ (strip2\ r)\ c$
 $\langle proof \rangle$

lemma *ok2-empty-mrexp2*: $ok2\ (empty-mrexp2\ r)$
 $\langle proof \rangle$

lemma *ok2-shift2*: $ok2\ r \Longrightarrow ok2(shift2\ m\ r\ c)$
 $\langle proof \rangle$

lemma *strip2-empty-mrexp2[simp]*: $strip2\ (empty-mrexp2\ r) = empty-mrexp\ r$
 $\langle proof \rangle$

lemma *nul-empty-mrexp2[simp]*: $nul\ (empty-mrexp2\ r) = nullable\ r$
 $\langle proof \rangle$

lemma *nonfin-empty-mrexp2[simp]*: $\neg fin\ (empty-mrexp2\ r)$
 $\langle proof \rangle$

lemma *empty-mrexp2-mrexp2[simp]*: $empty-mrexp2\ s \in mrexp2\ s$
 $\langle proof \rangle$

lemma *shift2-mrexp2[simp]*: $r \in mrexp2\ s \Longrightarrow shift2\ x\ r\ a \in mrexp2\ s$
 $\langle proof \rangle$

typedef 'a ok-mrexp2 = {(b :: bool, r :: 'a mrexp2). ok2 r}
⟨proof⟩

setup-lifting type-definition-ok-mrexp2

lift-definition init-okm :: 'a rexp ⇒ 'a ok-mrexp2 **is** λr. (True, empty-mrexp2 r)
⟨proof⟩

lift-definition delta-okm :: 'a ⇒ 'a ok-mrexp2 ⇒ 'a ok-mrexp2 **is**
λa (m, r). (False, shift2 m r a)
⟨proof⟩

lift-definition nullable-okm :: 'a ok-mrexp2 ⇒ bool **is** λ(m, r). fin r ∨ m ∧ nul r
⟨proof⟩

lift-definition lang-okm :: 'a ok-mrexp2 ⇒ 'a lang **is** λ(m, r). L-a (m, strip2 r)
⟨proof⟩

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)
⟨proof⟩

lemma eq-mrexp2-refl[simplified, simp]: r = s ⇒ eq-mrexp2 r s
⟨proof⟩

lemma eq-mrexp2-eq: [ok2 r; ok2 s] ⇒ eq-mrexp2 r s = (r = s)
⟨proof⟩

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

instance ⟨proof⟩

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⟩

```

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 mrexp3 :: 'a rexp ⇒ ('a mrexp3) set where

```

```

  mrexp3 Zero = {Zero3}
| mrexp3 One = {One3}
| mrexp3 (Atom a) = {Atom3 True a, Atom3 False a}
| mrexp3 (Plus r s) = (λ(r, s, f1, n). Plus3 r s f1 n) ‘ (mrexp3 r × mrexp3 s ×
  Pow (atoms (Plus r s)) × UNIV)
| mrexp3 (Times r s) = (λ(r, s, f1, n). Times3 r s f1 n) ‘ (mrexp3 r × mrexp3
  s × Pow (atoms (Times r s)) × UNIV)
| mrexp3 (Star r) = (λ(r, f1). Star3 r f1) ‘ (mrexp3 r × Pow (atoms (Star r)))

```

```

lemma finite-atoms[simp]: finite (atoms r)

```

```

  ⟨proof⟩

```

lemma *finite-mrexp3*[simp]: *finite* (*mrexp3* *r*)

<proof>

definition[simp]: *plus3* *r s* == *Plus3* *r s* (*fin1* *r* \cup *fin1* *s*) (*nul* *r* \vee *nul* *s*)

definition[simp]: *times3* *r s* ==

let ns = nul s in Times3 *r s* (*fin1* *s* \cup (*if ns then fin1 r else {}*)) (*nul* *r* \wedge *ns*)

definition[simp]: *star3* *r* == *Star3* *r* (*fin1* *r*)

primrec *follow3* :: *bool* \Rightarrow '*a mrexp3* \Rightarrow '*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* \vee *m* \wedge *nul* *r*) *s*) |

follow3 *m* (*Star3* *r* -) = *star3*(*follow3* (*final3* *r* \vee *m*) *r*)

primrec *empty-mrexp3* :: '*a rexp* \Rightarrow '*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* \Rightarrow '*a mrexp3* \Rightarrow *bool* \Rightarrow '*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* \in *fin1* *r* \vee *m* \wedge *nul* *r*)) |

move3 *c* (*Star3* *r* -) *m* = *star3* (*move3* *c r* (*c* \in *fin1* *r* \vee *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-mrexp3*: (*strip* *o* *strip3*) '*mrexp3* *r* = {*r*}

<proof>

primrec *ok3* :: '*a mrexp3* \Rightarrow *bool* **where**

ok3 *Zero3* = *True* |

ok3 *One3* = *True* |

$ok3 (Atom3 -) = True$ |
 $ok3 (Plus3 r s f1 n) = (ok3 r \wedge ok3 s \wedge$
 $(let rs = Plus (strip3 r) (strip3 s) in f1 = Collect (final1 rs) \wedge n = nullable rs))$
 |
 $ok3 (Times3 r s f1 n) = (ok3 r \wedge ok3 s \wedge$
 $(let rs = Times (strip3 r) (strip3 s) in f1 = Collect (final1 rs) \wedge n = nullable$
 $rs))$ |
 $ok3 (Star3 r f1) = (ok3 r \wedge f1 = Collect (final1 (strip3 r)))$

lemma *ok3-fin1-final1[simp]*: $ok3 r \implies fin1 r = Collect (final1 (strip3 r))$
<proof>

lemma *ok3-nul-nullable[simp]*: $ok3 r \implies nul r = nullable (strip3 r)$
<proof>

lemma *ok3-final3-final[simp]*: $ok3 r \implies final3 r = final (strip3 r)$
<proof>

lemma *follow3-follow[simp]*: $ok3 r \implies strip3 (follow3 m r) = follow m (strip3 r)$
<proof>

lemma *nul-follow3[simp]*: $ok3 r \implies nul (follow3 m r) = nul r$
<proof>

lemma *ok3-follow3[simp]*: $ok3 r \implies ok3 (follow3 m r)$
<proof>

lemma *fin1-atoms*: $\llbracket x \in fin1 mr; mr \in mrexps3 r \rrbracket \implies x \in atoms r$
<proof>

lemma *follow3-mrexps3[simp]*: $r \in mrexps3 s \implies follow3 m r \in mrexps3 s$
<proof>

lemma *empty-mrexp3-mrexps[simp]*: $empty-mrexp3 r \in mrexps3 r$
<proof>

lemma *strip3-empty-mrexp3[simp]*: $strip3 (empty-mrexp3 r) = empty-mrexp r$
<proof>

lemma *strip3-move3*: $ok3 r \implies strip3 (move3 m r c) = move m (strip3 r) c$
<proof>

lemma *nul-empty-mrexp3[simp]*: $nul (empty-mrexp3 r) = nullable r$
<proof>

lemma *ok3-empty-mrexp3*: $ok3 (empty-mrexp3 r)$
<proof>

lemma *ok3-move3*: $ok3 r \implies ok3 (move3 m r c)$

<proof>

lemma *nonfin1-empty-mrexp3[simp]*: $c \notin \text{fin1 } (\text{empty-mrexp3 } r)$
<proof>

lemma *move3-mrexp3[simp]*: $r \in \text{mrexp3 } s \implies \text{move3 } x \ r \ a \in \text{mrexp3 } s$
<proof>

typedef *'a ok-mrexp3* = $\{(r :: 'a \text{ mrexp3}, b :: \text{bool}). \text{ok3 } r\}$
<proof>

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

abbreviation *init-m* $r \equiv \text{let } mr = \text{follow3 } \text{True } (\text{empty-mrexp3 } r) \text{ in } (mr, \text{nul } mr)$

lift-definition *init-okm* :: $'a \text{ rexp} \Rightarrow 'a \text{ ok-mrexp3}$ **is** *init-m*
<proof>

lift-definition *delta-okm* :: $'a \Rightarrow 'a \text{ ok-mrexp3} \Rightarrow 'a \text{ ok-mrexp3}$ **is**
 $\lambda a \ (r, m). (\text{move3 } a \ r \ \text{False}, a \in \text{fin1 } r)$
<proof>

lift-definition *nullable-okm* :: $'a \text{ ok-mrexp3} \Rightarrow \text{bool}$ **is** *snd* *<proof>*

lift-definition *lang-okm* :: $'a \text{ ok-mrexp3} \Rightarrow 'a \text{ lang}$ **is** $\lambda(r, m). L\text{-b } (\text{strip3 } r, m)$
<proof>

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

fun *eq-mrexp3* **where**

$\text{eq-mrexp3 } \text{Zero3 } \text{Zero3} = \text{True}$
 $|\ \text{eq-mrexp3 } \text{One3 } \text{One3} = \text{True}$
 $|\ \text{eq-mrexp3 } (\text{Atom3 } m \ x) (\text{Atom3 } m' \ y) = (m = m' \wedge x = y)$
 $|\ \text{eq-mrexp3 } (\text{Plus3 } r1 \ s1 \ -) (\text{Plus3 } r3 \ s3 \ -) = (\text{eq-mrexp3 } r1 \ r3 \wedge \text{eq-mrexp3 } s1 \ s3)$
 $|\ \text{eq-mrexp3 } (\text{Times3 } r1 \ s1 \ -) (\text{Times3 } r3 \ s3 \ -) = (\text{eq-mrexp3 } r1 \ r3 \wedge \text{eq-mrexp3 } s1 \ s3)$
 $|\ \text{eq-mrexp3 } (\text{Star3 } r1 \ -) (\text{Star3 } r3 \ -) = (\text{eq-mrexp3 } r1 \ r3)$
 $|\ \text{eq-mrexp3 } r \ s = \text{False}$

lemma *eq-mrexp3-imp-eq*: $\llbracket \text{eq-mrexp3 } r \ s; \text{ok3 } r; \text{ok3 } s \rrbracket \implies (r = s)$
<proof>

lemma *eq-mrexp3-refl[simplified, simp]*: $r = s \implies \text{eq-mrexp3 } r \ s$
<proof>

lemma *eq-mrexp3-eq*: $\llbracket \text{ok3 } r; \text{ok3 } s \rrbracket \implies \text{eq-mrexp3 } r \ s = (r = s)$
<proof>

```

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

instance <proof>

end

global-interpretation before2: rexp-DFA init-okm delta-okm nullable-okm lang-okm
  defines before2-closure = before2.closure
    and check-equiv-b2 = before2.check-equiv
    and reachable-b2 = before2.reachable
    and automaton-b2 = before2.automaton
    and match-b2 = before2.match
  <proof>

```

8 Various Algorithms for Regular Expression Equivalence

```

export-code
  check-equiv-brz
  check-equiv-brzq
  check-equiv-n
  check-equiv-p
  check-equiv-pn
  check-equiv-b
  check-equiv-b2
  check-equiv-a
  check-equiv-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.