

# Regular Expression Equivalence

Tobias Nipkow      Dmitriy Traytel

January 31, 2023

## 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

<b>1</b>	<b>Regular Expressions Equivalence Framework</b>	<b>2</b>
1.1	The overall procedure . . . . .	4
<b>2</b>	<b>Finiteness of Derivatives Modulo ACI</b>	<b>6</b>
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
<b>3</b>	<b>Connection Between Derivatives and Partial Derivatives</b>	<b>17</b>
<b>4</b>	<b>Framework Instantiations using (Partial) Derivatives</b>	<b>23</b>
4.1	Brzowski Derivatives Modulo ACI . . . . .	23
4.2	Brzowski Derivatives Modulo ACI Operating on the Quotient Type . . . . .	24
4.3	Brzowski Derivatives Modulo ACI++ (Only Soundness) . . . . .	25
4.4	Partial Derivatives . . . . .	25
4.5	Languages as States . . . . .	26

<b>5</b>	<b>Framework Instantiations using Marked Regular Expressions</b>	<b>27</b>
5.1	Marked Regular Expressions . . . . .	27
5.2	Mark Before Atom . . . . .	29
5.3	Mark After Atom . . . . .	30
<b>6</b>	<b>Linear Time Optimization for “Mark After Atom”</b>	<b>33</b>
<b>7</b>	<b>Linear Time Optimization for “Mark Before Atom” (for a Fixed Alphabet)</b>	<b>36</b>
<b>8</b>	<b>Various Algorithms for Regular Expression Equivalence</b>	<b>41</b>

## 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*:  $set (add-atoms r as) = atoms r \cup set as$   
**by** (*induction r arbitrary: as*) *auto*

**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**–

```

{ fix r s r' s'
  have  $((r,s),(r',s')) : ?L \Longrightarrow ((r,s),(r',s')) : ?R$ 
  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  $\forall x \in set w. x \in A \Longrightarrow ((r, s), fold f w r, fold f w s) : ?L$ 
  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  $((r,s),(r',s')) : ?R \implies ((r,s),(r',s')) : ?L$  **by auto**  
**} ultimately show** *?thesis* **by** (*auto simp: in-lists-conv-set*) *blast*  
**qed**

**lemma** *rtrancl-fold-product1*:

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

**proof** –

{ **fix** *r s*  
**have**  $(r,s) : ?L \implies (r,s) : ?R$   
**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**  $\forall x \in set w. x \in A \implies (r, fold f w r) : ?L$   
**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**  $(r,s) : ?R \implies (r,s) : ?L$  **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**  $\mathfrak{B} \equiv \{(fold Deriv w (lang r), fold Deriv w (lang s)) \mid w. w \in lists (atoms r \cup atoms s)\}$

**shows**  $lang r = lang s \iff (\forall (K, L) \in \mathfrak{B}. [] \in K \iff [] \in L)$

**unfolding** *lang-eq-ext*  $\mathfrak{B}$ -*def* **by** (*subst (1 2) in-fold-Deriv[of [], simplified, symmetric]*) *auto*

**locale** *rexp-DA* =

**fixes** *init* ::  $'a \text{ rexp} \Rightarrow 's$

**fixes** *delta* ::  $'a \Rightarrow 's \Rightarrow 's$

**fixes** *final* ::  $'s \Rightarrow bool$

**fixes** *L* ::  $'s \Rightarrow 'a \text{ lang}$

**assumes** *L-init*:  $L (init r) = lang r$

**assumes** *L-delta*:  $L(delta a s) = Deriv a (L s)$

**assumes** *final-iff-Nil*:  $final s \iff [] \in L s$

**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  $\Rightarrow$  's \* 's  $\Rightarrow$  (('s \* 's) list \* ('s \* 's) set) option  
**where**  
*closure* as = rtrancl-while ( $\lambda(p,q).$  final p = final q)  
( $\lambda(p,q).$  map ( $\lambda a.$  (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  $\cup$  atoms s

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

**proof** –

**have** *leq*: (lang r = lang s) =  
( $\forall (r', s') \in \{((r0, s0), (delta a r0, delta a s0)) \mid r0 s0 a. a : set as\}^*$  “{(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  $\in$  set (( $\lambda(p,q).$  map ( $\lambda a.$  (delta a p, delta a q)) as) x)} =  
{((r, s), delta a r, delta a s)  $\mid$  r s a. a  $\in$  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  $\Rightarrow$  'a rexp  $\Rightarrow$  bool **where**

*check-eqv* r s =

(let as = add-atoms r (add-atoms s []))

in case *closure* as (init r, init s) of

Some([], -)  $\Rightarrow$  True  $\mid$  -  $\Rightarrow$  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  $\Rightarrow$  'a rexp  $\Rightarrow$  's set **where**

*reachable* as s =

snd(the(rtrancl-while ( $\lambda.$  True) ( $\lambda s.$  map ( $\lambda a.$  delta a s) as) (init s)))

**definition** *automaton* :: 'a list  $\Rightarrow$  'a rexp  $\Rightarrow$  (('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), 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  
**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)<sup>\*</sup> “ {(init r, init s)}  
**unfolding** *rtrancl-fold-product Image-singleton*  
**by** (auto intro: finite-subset[OF - finite-cartesian-product[OF fin fin]])

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

**unfolding** *closure-def* **proof** (rule rtrancl-while-finite-Some)  
**show** finite ({(x, st). st  $\in$  set (( $\lambda$ (p,q). map ( $\lambda$ a. (delta a p, delta a q)) as) x)}<sup>\*</sup>  
 “ {?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-equiv 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- $equiv$  r s* **by** (*simp add: check- $equiv$ -def set-add-atoms*)  
**qed**

**lemma** *finite-rtrancl-delta-Image1*:

*finite* ( $\{(r,s). \exists a \in A. s = \text{delta } a \ r\}^* \hat{\ } \{init \ r\}$ )

**unfolding** *rtrancl-fold-product1* **by** (*auto intro: finite-subset[OF - fin]*)

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

**proof** –

**obtain** *wsZ* **where**  $*$ : *rtrancl-while* ( $\lambda s. True$ ) ( $\lambda s. \text{map } (\lambda a. \text{delta } a \ s)$  *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  $\in$  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*  $\equiv$  *rexp-of-list Plus Zero*

**abbreviation** *TIMES*  $\equiv$  *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*

**using** *assms* **by** *induction-schema (pat-completeness, lexicographic-order)*

### 2.1 ACI normalization

**fun** *toplevel-summands* :: *'a rexp*  $\Rightarrow$  *'a rexp set* **where**

*toplevel-summands* (*Plus r s*) = *toplevel-summands r*  $\cup$  *toplevel-summands s*

| *toplevel-summands r* =  $\{r\}$

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

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

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

**primrec** *ACI-norm* :: ( $'a::\text{linorder}$ ) *rexp*  $\Rightarrow$   $'a$  *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*  $r$   $s \in$  *toplevel-summands*  $t \Longrightarrow$  *False*  
**by** (*induction*  $t$ ) *auto*

**lemma** *toplevel-summands-not-Plus*[simp]:  
 $(\forall r s. x \neq \text{Plus } r s) \Longrightarrow$  *toplevel-summands*  $x = \{x\}$   
**by** (*induction*  $x$ ) *auto*

**lemma** *toplevel-summands-PLUS-strong*:  
 $\llbracket xs \neq []; \text{list-all } (\lambda x. \neg(\exists r s. x = \text{Plus } r s)) \text{ } xs \rrbracket \Longrightarrow$  *toplevel-summands* (*PLUS*  $xs$ ) = *set*  $xs$   
**by** (*induct*  $xs$  *rule: list-singleton-induct*) *auto*

**lemma** *toplevel-summands-flatten*:  
 $\llbracket X \neq \{\}; \text{finite } X; \forall x \in X. \neg(\exists r s. x = \text{Plus } r s) \rrbracket \Longrightarrow$  *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*  $s$   $t \Longrightarrow \exists s t. r = \text{Plus } s t$   
**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*  $r \cup$  *ACI-norm*  
 ‘*toplevel-summands*  $s$ )) =  
*ACI-norm* ‘*toplevel-summands*  $r \cup$  *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*»  $\cup$  *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*»  $\cup$  *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* = ( $\bigcup_{r \in \text{toplevel-summands } s} \text{atoms } r$ )  
**by** (*induct* *s*) *auto*

**lemma** *wf-PLUS*: *atoms* (*PLUS* *xs*)  $\subseteq \Sigma \iff (\forall r \in \text{set } xs. \text{atoms } r \subseteq \Sigma)$   
**by** (*induct* *xs* *rule: list-singleton-induct*) *auto*

**lemma** *atoms-PLUS*: *atoms* (*PLUS* *xs*) = ( $\bigcup_{r \in \text{set } xs} \text{atoms } r$ )  
**by** (*induct* *xs* *rule: list-singleton-induct*) *auto*

**lemma** *atoms-flatten-PLUS*:  
*finite* *X*  $\implies \text{atoms} (\text{flatten } PLUS \text{ } X) = (\bigcup_{r \in X} \text{atoms } r)$   
**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*:  $r \in \text{toplevel-summands } s \implies \text{lang } r \subseteq \text{lang } s$   
**by** (*induct s*) *auto*

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

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

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

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

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

**theorem** *lang-ACI-norm[simp]*:  $\text{lang } \langle r \rangle = \text{lang } r$

**proof** (*induct r*)

**case** (*Plus r1 r2*)

**moreover**

**from** *Plus[symmetric]* **have**  $\text{lang } (\text{Plus } r1 \ r2) \subseteq \text{lang } \langle \text{Plus } r1 \ r2 \rangle$

**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*:

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

**by** (*induction r*) (*auto simp: Let-def*)

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

**by** (*induction xs*) *auto*

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

**by** (*induction xs*) *auto*

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

**proof** (*induction xs*)

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

**lemma** *derivs-Plus*:  $\text{derivs } xs \text{ (Plus } r \text{ s)} = \text{Plus (derivs } xs \text{ r) (derivs } xs \text{ s)}$   
**by** (*induction xs arbitrary: r s*) *auto*

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

**lemma** *toplevel-summands-derivs-Times*:  $\text{toplevel-summands (derivs } xs \text{ (Times } r \text{ s))} \subseteq$   
 $\{ \text{Times (derivs } xs \text{ r) } s \} \cup$   
 $\{ r'. \exists ys \text{ } zs. r' \in \text{toplevel-summands (derivs } ys \text{ s)} \wedge ys \neq [] \wedge zs @ ys = xs \}$   
**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*:  
 $xs \neq [] \implies \text{toplevel-summands (derivs } xs \text{ (Star } r)) \subseteq$   
 $\{ \text{Times (derivs } ys \text{ r) (Star } r) \mid ys. \exists zs. ys \neq [] \wedge zs @ ys = xs \}$   
**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*:  
 $\text{toplevel-summands (derivs } xs \text{ (Star } r)) \subseteq$   
 $\{ \text{Star } r \} \cup \{ \text{Times (derivs } ys \text{ r) (Star } r) \mid ys. \exists zs. ys \neq [] \wedge zs @ ys = xs \}$   
**by** (*cases xs = []*) (*auto dest!: toplevel-summands-derivs-Star-nonempty*)

**lemma** *toplevel-summands-PLUS*:  
 $xs \neq [] \implies \text{toplevel-summands (PLUS (map } f \text{ } xs)) = (\bigcup r \in \text{set } xs. \text{toplevel-summands (f } r))$   
**by** (*induction xs rule: list-singleton-induct*) *simp-all*

**lemma** *ACI-norm-toplevel-summands-Zero*:  $\text{toplevel-summands } r \subseteq \{ \text{Zero} \} \implies$   
 $\llbracket r \rrbracket = \text{Zero}$   
**by** (*subst ACI-norm-flatten*) (*auto dest: subset-singletonD*)

**lemma** *finite-ACI-norm-toplevel-summands*:  
 $\text{finite } \{ f \llbracket s \rrbracket \mid s. \text{toplevel-summands } s \subseteq B \}$  **if** *finite B*  
**proof** –  
**have** \*:  $\{ f \llbracket s \rrbracket \mid s. \text{toplevel-summands } s \subseteq B \} \subseteq (f \circ \text{flatten PLUS} \circ (\cdot) \text{ ACI-norm}) \text{ ` Pow } B$

```

    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 derivs-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* ( $\lambda r s. \langle\langle r \rangle\rangle = \langle\langle s \rangle\rangle$ ) *xs ys*  $\implies \langle\langle PLUS\ xs \rangle\rangle = \langle\langle PLUS\ ys \rangle\rangle$

**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*:

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

**proof** (*induct r*)

**case** (*Plus r1 r2*) **thus** ?*case*

**unfolding** *toplevel-summands.simps toplevel-summands-ACI-norm*

*toplevel-summands-deriv*[of as  $\langle\langle Plus\ r1\ r2 \rangle\rangle$ ] *image-Un Union-Un-distrib*

**by** (*simp add: image-UN*)

**qed** (*auto simp: Let-def*)

**lemma** *toplevel-summands-nullable*:

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

**by** (*induction s*) *auto*

**lemma** *nullable-PLUS*:

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

**by** (*induction xs rule: list-singleton-induct*) *auto*

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

**proof** (*induction r*)

**case** (*Plus r1 r2*) **thus** ?*case using toplevel-summands-nullable*

**by** (*auto simp: nullable-PLUS*)

**qed** *auto*

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

**proof** (*induction r arbitrary: as*)

**case** (*Plus r1 r2*) **thus** ?*case*

**unfolding** *deriv.simps ACI-norm-flatten*[of *deriv as*  $\langle\langle Plus\ r1\ r2 \rangle\rangle$ ]

*toplevel-summands-deriv*[of as  $\langle\langle Plus\ r1\ r2 \rangle\rangle$ ] *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-preserved:*  $\langle\langle r \rangle\rangle = \langle\langle s \rangle\rangle \implies \langle\langle \text{deriv as } r \rangle\rangle = \langle\langle \text{deriv as } s \rangle\rangle$

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

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

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

**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  $\langle r \rangle$ ] **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  $\Rightarrow$  '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)

**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  $\implies \exists s t. r = Plus s t$

**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*:

$\llbracket xs \neq []; \text{sorted } xs; \text{distinct } xs; \forall x \in \{x\} \cup \text{set } xs. \neg(\exists r s. x = \text{Plus } r s) \rrbracket \implies$   
*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*:

$\llbracket xs1 \neq []; xs2 \neq []; \forall x \in \text{set } (xs1 @ xs2). \neg(\exists r s. x = \text{Plus } r s); \text{sorted } xs2; \text{distinct } xs2 \rrbracket \implies$

*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*:

$\llbracket X1 \neq \{\}; X2 \neq \{\}; \text{finite } X1; \text{finite } X2; \forall x \in X1 \cup X2. \neg(\exists r s. x = \text{Plus } r s) \rrbracket \implies$   
 $\text{ACI-nPlus } (\text{flatten PLUS } X1) (\text{flatten PLUS } X2) = \text{flatten PLUS } (X1 \cup X2)$   
**by** (rule trans[OF ACI-nPlus-PLUS]) auto

**lemma** ACI-nPlus-ACI-norm [simp]:  $\text{ACI-nPlus } \langle r \rangle \langle s \rangle = \langle \text{Plus } r s \rangle$   
**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:  
 $\text{ACI-norm-alt } r = \langle r \rangle$   
**by** (induct r) auto

**declare** ACI-norm-alt[symmetric, code]

**inductive** ACI **where**

$\text{ACI-refl: } \text{ACI } r r \mid$   
 $\text{ACI-sym: } \text{ACI } r s \implies \text{ACI } s r \mid$   
 $\text{ACI-trans: } \text{ACI } r s \implies \text{ACI } s t \implies \text{ACI } r t \mid$   
 $\text{ACI-Plus-cong: } \llbracket \text{ACI } r1 s1; \text{ACI } r2 s2 \rrbracket \implies \text{ACI } (\text{Plus } r1 r2) (\text{Plus } s1 s2) \mid$   
 $\text{ACI-Times-cong: } \llbracket \text{ACI } r1 s1; \text{ACI } r2 s2 \rrbracket \implies \text{ACI } (\text{Times } r1 r2) (\text{Times } s1 s2) \mid$   
 $\text{ACI-Star-cong: } \text{ACI } r s \implies \text{ACI } (\text{Star } r) (\text{Star } s) \mid$   
 $\text{ACI-assoc: } \text{ACI } (\text{Plus } (\text{Plus } r s) t) (\text{Plus } r (\text{Plus } s t)) \mid$   
 $\text{ACI-comm: } \text{ACI } (\text{Plus } r s) (\text{Plus } s r) \mid$   
 $\text{ACI-idem: } \text{ACI } (\text{Plus } r r) r$

**lemma** ACI-atoms:  $\text{ACI } r s \implies \text{atoms } r = \text{atoms } s$   
**by** (induct rule: ACI.induct) auto

**lemma** ACI-nullable:  $\text{ACI } r s \implies \text{nullable } r = \text{nullable } s$   
**by** (induct rule: ACI.induct) auto

**lemma** ACI-lang:  $\text{ACI } r s \implies \text{lang } r = \text{lang } s$   
**by** (induct rule: ACI.induct) auto

**lemma** ACI-deriv:  $\text{ACI } r s \implies \text{ACI } (\text{deriv } a r) (\text{deriv } a s)$

**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.ACITimes-cong ACI-Plus-cong)

**qed** (auto intro: ACI.intros)

**lemma** ACI-Plus-assocI[intro]:

$\text{ACI } (\text{Plus } r1 r2) s2 \implies \text{ACI } (\text{Plus } r1 (\text{Plus } s1 r2)) (\text{Plus } s1 s2)$   
 $\text{ACI } (\text{Plus } r1 r2) s2 \implies \text{ACI } (\text{Plus } r1 (\text{Plus } r2 s1)) (\text{Plus } s1 s2)$

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

**lemma** *ACI-Plus-idemI*[intro]:  $\llbracket \text{ACI } r \text{ } s1; \text{ACI } r \text{ } s2 \rrbracket \implies \text{ACI } r \text{ } (\text{Plus } s1 \text{ } s2)$   
by (metis ACI-Plus-cong ACI-idem ACI-sym ACI-trans)

**lemma** *ACI-Plus-idemI'*[intro]:  
 $\llbracket \text{ACI } r1 \text{ } s1; \text{ACI } (\text{Plus } r1 \text{ } r2) \text{ } s2 \rrbracket \implies \text{ACI } (\text{Plus } r1 \text{ } r2) \text{ } (\text{Plus } s1 \text{ } s2)$   
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*:  $\llbracket \text{ACI } r1 \text{ } s1; \text{ACI } r2 \text{ } s2 \rrbracket \implies \text{ACI } (\text{ACI-nPlus } r1 \text{ } r2) \text{ } (\text{Plus } s1 \text{ } s2)$   
**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*:  $\text{ACI } \langle\langle r \rangle\rangle r$   
**unfolding** *ACI-norm-alt*[symmetric]  
by (induct r) (auto intro: ACI.intros simp: ACI-ACI-nPlus)

**lemma** *ACI-norm-eqI*:  $\text{ACI } r \text{ } s \implies \langle\langle r \rangle\rangle = \langle\langle s \rangle\rangle$   
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*:  $\langle\langle r \rangle\rangle = \langle\langle s \rangle\rangle \implies \text{ACI } r \text{ } s$   
by (metis ACI-ACI-norm ACI-sym ACI-trans)



**lemma** *ACI-decidable*:  $ACI\ r\ s = (\llbracket r \rrbracket = \llbracket s \rrbracket)$   
**by** (*metis ACI-I ACI-norm-eqI*)

### 3 Connection Between Derivatives and Partial Derivatives

**lemma** *pderiv-not-is-Zero-is-Plus[simp]*:  $\forall x \in pderiv\ a\ r. \neg is-Zero\ x \wedge \neg is-Plus\ x$   
**by** (*induct r*) *auto*

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

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

$(PLUS\ xs = PLUS\ ys) \longleftrightarrow xs = ys$

**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*:  $\llbracket finite\ R; finite\ S \rrbracket \implies$

$(sorted-list-of-set\ R = sorted-list-of-set\ S) \longleftrightarrow R = S$

**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.premis(1) set-sorted-list-of-set apply fastforce
using insert.premis(1) set-sorted-list-of-set apply fastforce
using insert.premis(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*:  $\llbracket \forall x \in R \cup S. \neg \text{is-Zero } x \wedge \neg \text{is-Plus } x; \text{ finite } R; \text{ finite } S \rrbracket \implies$   
 $(\text{flatten PLUS } R = \text{flatten PLUS } S) = (R = S)$   
**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  $\cup$  *pset* s  
| *pset* (Times r s) = *Times* (*pset* r) s  
| *pset* (Star r) = {Star r}

**lemma** *pset-not-is-Zero-is-Plus*[simp]:  $\forall x \in \text{pset } r. \neg \text{is-Zero } x \wedge \neg \text{is-Plus } x$   
**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)  $\iff$  *pderiv* a r = *pderiv* a s  
**unfolding** *pnorm-def* o-apply *pset-deriv*  
**by** (rule *flatten-PLUS-inject*) auto

**fun** *pnPlus* :: 'a::linorder rexp  $\Rightarrow$  'a rexp  $\Rightarrow$  '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*:  $\text{pnorm } (\text{pnorm-alt } r) = \text{pnorm } r$

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

**lemma** *pnPlus-singleton-PLUS*:

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

**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-rop-def less-eq-rop-def*)

**apply** (*cases y*)

**apply** *auto*

**apply** (*induct x y rule: pnPlus.induct*)

**apply** (*auto simp: less-rop-def less-eq-rop-def*)

**apply** (*induct x y rule: pnPlus.induct*)

**apply** (*auto simp: less-rop-def less-eq-rop-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-rop-def)*

**apply** *simp apply (metis antisym less-eq-rop-def)*

**apply** (*metis UnCI insertI1*)

**apply** *simp apply (metis antisym less-eq-rop-def)*

**apply** *simp apply (metis antisym less-eq-rop-def)*

**done**

**qed** *simp*

**lemma** *pnPlus-PlusL[simp]*:  $t \neq \text{Zero} \implies \text{pnPlus } (\text{Plus } r \text{ } s) \text{ } t = \text{pnPlus } r \text{ } (\text{pnPlus } s \text{ } t)$

**by** (*induct t*) *auto*

**lemma** *pnPlus-ZeroR[simp]*:  $\text{pnPlus } r \text{ } \text{Zero} = r$

**by** (*induct r*) *auto*

**lemma** *PLUS-eq-Zero*:  $\text{PLUS } xs = \text{Zero} \iff xs = [] \vee xs = [\text{Zero}]$

**by** (*induct xs rule: list-singleton-induct*) *auto*

**lemma** *pnPlus-PLUS*:

$\llbracket xs1 \neq []; xs2 \neq []; \forall x \in \text{set } (xs1 @ xs2). \neg \text{is-Zero } x \wedge \neg \text{is-Plus } x; \text{sorted } xs2; \text{distinct } xs2 \rrbracket \implies$

$\text{pnPlus } (\text{PLUS } xs1) \text{ } (\text{PLUS } xs2) = \text{flatten PLUS (set } (xs1 @ xs2))$

```

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 (Times (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*:  
 $\llbracket X1 \neq \{\}; \text{finite } X1; \forall x \in X1. \neg \text{is-Zero } x \wedge \neg \text{is-Plus } x \rrbracket \implies$   
 $\text{pnTimes } (\text{flatten PLUS } X1) r = \text{flatten PLUS } (\text{Times } X1 r)$   
**by** (*rule trans[OF pnTimes-PLUS]*) *auto*

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

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

**lemma** *insort-eq-Cons*:  $\llbracket \forall a \in \text{set } xs. b < a; \text{sorted } xs \rrbracket \implies \text{insort } b xs = b \# xs$   
**by** (*cases xs*) *auto*

**lemma** *pderiv-PLUS*:  $\text{pderiv } a (\text{PLUS } (x \# xs)) = \text{pderiv } a x \cup \text{pderiv } a (\text{PLUS } xs)$   
**by** (*cases xs*) *auto*

**lemma** *pderiv-set-flatten-PLUS*:  
 $\text{finite } X \implies \text{pderiv } (a :: 'a :: \text{linorder}) (\text{flatten PLUS } X) = \text{pderiv-set } a X$   
**proof** (*induction X rule: finite-linorder-min-induct*)  
**case** (*insert b X*)  
**then have**  $b \notin X$  **by** *auto*  
**then have**  $\text{pderiv } a (\text{flatten PLUS } (\text{insert } b X)) = \text{pderiv } a b \cup \text{pderiv } a (\text{flatten PLUS } X)$   
**by** (*simp add: pderiv-PLUS insort-eq-Cons insert.hyps*)  
**also from** *insert.IH* **have**  $\dots = \text{pderiv } a b \cup \text{pderiv-set } a X$  **by** *simp*  
**finally show** *?case* **by** *simp*  
**qed** *simp*

**lemma** *fold-pderiv-set-flatten-PLUS*:  
 $\llbracket w \neq \{\}; \text{finite } X \rrbracket \implies \text{fold pderiv-set } w \{\text{flatten PLUS } X\} = \text{fold pderiv-set } w X$   
**by** (*induct w arbitrary: X*) (*simp-all add: pderiv-set-flatten-PLUS*)

**lemma** *fold-pnorm-deriv*:  
 $\text{fold } (\lambda a r. \text{pnorm } (\text{deriv } a r)) w s = \text{flatten PLUS } (\text{fold pderiv-set } w \{s\})$   
**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**

```

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) =
  (if nullable r1 then pnPlus (pnTimes (pderiv c r1) r2) (pderiv c r2) else
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)
  by (induct r) (auto simp: pnorm-alt)

```

```

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

```

## 4 Framework Instantiations using (Partial) Derivatives

### 4.1 Brzowski Derivatives Modulo ACI

```

lemma ACI-norm-derivs-alt:  $\langle\langle$  derivs w r  $\rangle\rangle$  = fold ( $\lambda$  a r.  $\langle\langle$  deriv a r  $\rangle\rangle$ ) w  $\langle\langle$  r  $\rangle\rangle$ 
  by (induct w arbitrary: r) (auto simp: ACI-norm-deriv)

```

```

global-interpretation brzowski: rexp-DFA  $\lambda$  r.  $\langle\langle$  r  $\rangle\rangle$   $\lambda$  a r.  $\langle\langle$  deriv a r  $\rangle\rangle$  nullable
lang

```

```

defines brzowski-closure = brzowski.closure
  and check-equiv-brz = brzowski.check-equiv
  and reachable-brz = brzowski.reachable
  and automaton-brz = brzowski.automaton
  and match-brz = brzowski.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 Brzowski 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*  $\Rightarrow$  'a *ACI-rexp*  $\Rightarrow$  *bool* **is**  $\lambda r s.$  *less-eq* «*r*» «*s*»

**by** (*simp add: ACI-decidable*)

**lift-definition** *less-ACI-rexp* :: 'a *ACI-rexp*  $\Rightarrow$  'a *ACI-rexp*  $\Rightarrow$  *bool* **is**  $\lambda r s.$  *less* «*r*» «*s*»

**by** (*simp add: ACI-decidable*)

**lift-definition** *equal-ACI-rexp* :: 'a *ACI-rexp*  $\Rightarrow$  'a *ACI-rexp*  $\Rightarrow$  *bool* **is**  $\lambda r s.$  «*r*» = «*s*»

**by** (*simp add: ACI-decidable*)

**instance** **by** *intro-classes* (*transfer, force simp: ACI-decidable*)+

**end**

**lift-definition** *ACI-deriv* :: 'a :: *linorder*  $\Rightarrow$  'a *ACI-rexp*  $\Rightarrow$  'a *ACI-rexp* **is** *deriv*

**by** (*rule ACI-deriv*)

**lift-definition** *ACI-nullable* :: 'a :: *linorder* *ACI-rexp*  $\Rightarrow$  *bool* **is** *nullable*

**by** (*rule ACI-nullable*)

**lift-definition** *ACI-lang* :: 'a :: *linorder* *ACI-rexp*  $\Rightarrow$  '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** *brzowski-quotient*: *rexp-DFA* *ACI-class* *ACI-deriv* *ACI-nullable*



*ACI-lang*

```

defines brzowski-quotient-closure = brzowski-quotient.closure
and check-equiv-brzq = brzowski-quotient.check-equiv
and reachable-brzq = brzowski-quotient.reachable
and automaton-brzq = brzowski-quotient.automaton
and match-brzq = brzowski-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 Brzowski Derivatives Modulo ACI++ (Only Soundness)

**global-interpretation** *nderiv: rexp-DA  $\lambda x. norm\ x\ nderiv\ nullable\ lang$*

```

defines nderiv-closure = nderiv.closure
and check-equiv-n = nderiv.check-equiv
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-equiv-p = pderiv.check-equiv
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 pderiv: rexp-DFA λr. r pderiv nullable lang
  defines pderiv-closure = pderiv.closure
  and check-eqv-pn = pderiv.check-eqv
  and reachable-pn = pderiv.reachable
  and automaton-pn = pderiv.automaton
  and match-pn = pderiv.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 pderiv-alt)
next
  case 3 show ?case by (simp add: nullable-iff)
next
  case (4 s)
  then show ?case unfolding pderiv-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*:  $rexp\ DFA\ lang\ Deriv\ \lambda L. [] \in L\ id$

**proof** (unfold-locales, goal-cases)

```

  case (4 s)
  have {fold Deriv w (lang s) |w. True} ⊆ (λX. ⋃ (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  $\approx A \equiv \{(x, y). (\forall z. x @ z \in A \longleftrightarrow y @ z \in A)\}$

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

**lemma** *Myhill-Nerode2*:  $finite\ (UNIV\ //\ \approx lang\ r)$

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

## 5 Framework Instantiations using Marked Regular Expressions

### 5.1 Marked Regular Expressions

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

**abbreviation** *strip* ≡ *map-rexp snd*

**primrec** *mrexps* :: *'a rexp* ⇒ (*'a mrexpr*) **set** **where**  
*mrexps Zero* = {*Zero*}  
| *mrexps One* = {*One*}  
| *mrexps (Atom a)* = {*Atom (True, a), Atom (False, a)*}  
| *mrexps (Plus r s)* = *case-prod Plus ' (mrexps r × mrexps s)*  
| *mrexps (Times r s)* = *case-prod Times ' (mrexps r × 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 mrexpr* ⇒ *'a lang* **where**  
*Lm Zero* = {} |  
*Lm One* = {} |  
*Lm (Atom(m,a))* = (*if m then {[a] else {}*) |  
*Lm (Plus r s)* = *Lm r* ∪ *Lm s* |  
*Lm (Times r s)* = *Lm r* @@ *lang(strip s)* ∪ *Lm s* |  
*Lm (Star r)* = *Lm r* @@ *star(lang(strip r))*

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

**abbreviation** *read* :: *'a* ⇒ *'a mrexpr* ⇒ *'a mrexpr* **where**  
*read a* ≡ *map-rexp (λ(m,x). (m ∧ a=x, x))*

**lemma** *read-mrexps[simp]*: *r* ∈ *mrexps s* ⇒ *read a r* ∈ *mrexps s*  
**by** (*induction s arbitrary: a r*) (*auto simp: image-iff*)

**fun** *follow* :: *bool*  $\Rightarrow$  'a *mrexp*  $\Rightarrow$  '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  $\vee$  m  $\wedge$  nullable r) s)* |  
*follow m (Star r)* = *Star(follow (final r  $\vee$  m) r)*

**lemma** *follow-mrexp[simp]*:  $r \in mrexp\ s \Longrightarrow follow\ b\ r \in mrexp\ s$   
**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]*:  $\square \notin Lm\ r$   
**by** (*induction r*) (*auto split: if-splits*)

**lemma** *Nil-in-lang-strip[simp]*:  $\square \in lang(r) \longleftrightarrow \square \in lang(strip\ r)$   
**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*:  $\square \notin A \Longrightarrow \{w : A\ @\@ B.\ w \neq \square \wedge P(hd\ w)\} = \{w : A.\ w \neq \square \wedge P(hd\ w)\}\ @\@ B$   
**unfolding** *conc-def* **by** *auto (metis hd-append2)+*

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

**proof** (*induction r*)

**case** (*Times r1 r2*) **thus** *?case*

**using** *conc-lemma[OF Nil-notin-Lm, where P =  $\lambda x.\ x=a$  and  $r1 = r1$ ]* **by**  
*auto*

**next**

**case** *Star* **thus** *?case* **using** *conc-lemma[OF Nil-notin-Lm, where P =  $\lambda x.\ x=a$ ]*

**by** *simp*

**qed** (*auto split: if-splits*)

**lemma** *tl-conc[simp]*:  $\square \notin A \Longrightarrow tl\ '(A\ @\@ B) = tl\ 'A\ @\@ B$   
**by** (*fastforce simp: image-def Bex-def tl-append split: list.split*)

**lemma** *Nil-in-tl-Lm-if-final[simp]*: *final r*  $\Longrightarrow \square : tl\ 'Lm\ r$   
**by** (*induction r*) (*auto simp: nullable-iff image-Un*)

**lemma** *Nil-notin-tl-if-not-final*:  $\neg\ final\ r \Longrightarrow \square \notin tl\ 'Lm\ r$

**by** (*induction r*) (*auto simp: nullable-iff Nil-tl singleton-in-conc intro!: image-eqI[rotated]*)

**lemma** *Lm-follow*:  $Lm\ (follow\ m\ r) = tl\ 'Lm\ r \cup (if\ m\ then\ lang(strip\ r)\ else$

```

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

**lemma**  $empty-mrexp-mrexp[simp]: empty-mrexp r \in mrexp\ r$   
**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**  $init-b-mrexp[simp]: init-b\ r \in mrexp\ r \times UNIV$   
**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**  $delta-b-mrexp[simp]: rb \in mrexp\ r \times UNIV \implies delta-b\ a\ rb \in mrexp\ r$   
 $\times UNIV$   
**by** (auto simp: Let-def)

**lemma**  $fold-delta-b-init-b-mrexp[simp]: fold\ delta-b\ w\ (init-b\ s) \in mrexp\ s \times UNIV$   
**by** (induction w arbitrary: s rule: rev-induct) auto

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

**abbreviation**  $final-b \equiv snd$

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

**lemma**  $final-read-Lm: final(read\ a\ r) \longleftrightarrow [a] \in Lm\ r$

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-egv-b = before.check-egv

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  $\longleftrightarrow$  []  $\in$  L-b rb by (cases rb) simp

next

case (4 s)

have {fold delta-b w (init-b s) |w. True}  $\subseteq$  mrexp s  $\times$  UNIV

by (intro subsetI, elim CollectE exE) (simp only: fold-delta-b-init-b-mrexp)

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-mrexp[simp]: init-a r  $\in$  UNIV  $\times$  mrexp 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-mrexp[simp]: br  $\in$  UNIV  $\times$  mrexp r  $\implies$  delta-a a br  $\in$  UNIV  $\times$  mrexp r

by auto

**lemma** fold-delta-a-init-a-mrexp[simp]: fold delta-a w (init-a s)  $\in$  UNIV  $\times$  mrexp s

by (induction w arbitrary: s rule: rev-induct) auto

**fun** final-a where

final-a (b,r)  $\longleftrightarrow$  final r  $\vee$  b  $\wedge$  nullable r

**fun** L-a where

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

```

lemma nonfinal-empty-mrexp:  $\neg \text{final } (\text{empty-mrexp } r)$ 
by (induction r) auto

lemma Cons-eq-tl-iff[simp]:  $x \# xs = \text{tl } ys \longleftrightarrow (\exists y. ys = y \# x \# xs)$ 
by (cases ys) auto

lemma tl-eq-Cons-iff[simp]:  $\text{tl } ys = x \# xs \longleftrightarrow (\exists y. ys = y \# x \# xs)$ 
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  $L\text{-}a \text{ (init-}a \text{ } r) = \text{lang } r$ 
  by (auto simp: init-a-def nonfinal-empty-mrexp Lm-follow Lm-empty map-map-rexp
  nullable-iff)
  next
  case (2 a br) show  $L\text{-}a \text{ (delta-}a \text{ } a \text{ } br) = \text{Deriv } a \text{ (} L\text{-}a \text{ } br)$ 
  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  $\text{final-}a \text{ } br \longleftrightarrow \square \in L\text{-}a \text{ } br$  by (cases br) simp
  next
  case (4 s)
  have  $\{\text{fold delta-}a \text{ } w \text{ (init-}a \text{ } s) \mid w. \text{True}\} \subseteq UNIV \times \text{mrexp}s$ 
  by (intro subsetI, elim CollectE exE) (simp only: fold-delta-a-init-a-mrexp)
  then show finite  $\{\text{fold delta-}a \text{ } w \text{ (init-}a \text{ } s) \mid w. \text{True}\}$  by (rule finite-subset) simp
qed

```

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*) = (*follow b r, final-a (b, r)*)

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

**by**(*cases br*) (*auto simp add: Let-def*)

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

**by** (*induct w arbitrary: br*) (*auto simp add: hom-delta*)

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

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

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

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

**theorem** *card-reachable-ab*:  $\text{card } (\text{reachable-b as } r) \leq \text{card } (\text{reachable-a as } r)$   
**unfolding** *reachable-ab* **using** *after.finite-reachable* **by** (*rule card-image-le*)

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*:  $\text{shift } m \ r \ x = \text{read } x \ (\text{follow } m \ r)$   
**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* ::  $'a \text{ mrexpr} \Rightarrow 'a \Rightarrow \text{bool}$  **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]*:  $\text{nullable } (\text{read } c \ r) = \text{nullable } r$   
**by** (*induction* *r*) *auto*

**lemma** *final-read-final1*:  $\text{final } (\text{read } c \ r) = \text{final1 } r \ c$   
**by** (*induction* *r*) *auto*

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



## 6 Linear Time Optimization for “Mark After Atom”

**datatype** 'a mrex2 =

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** mrex2 :: 'a rexp ⇒ ('a mrex2) set **where**

mrex2 Zero = {Zero2}  
 | mrex2 One = {One2}  
 | mrex2 (Atom a) = {Atom2 True a, Atom2 False a}  
 | mrex2 (Plus r s) = (λ(r, s, f, n). Plus2 r s f n) ‘(mrex2 r × mrex2 s × UNIV)  
 | mrex2 (Times r s) = (λ(r, s, f, n). Times2 r s f n) ‘(mrex2 r × mrex2 s × UNIV)  
 | mrex2 (Star r) = (λ(r, f). Star2 r f) ‘(mrex2 r × UNIV)

**lemma** finite-mrex2[simp]: finite (mrex2 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-mrex2 :: 'a rexp ⇒ '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 ⇒ 'a mrex2 ⇒ 'a ⇒ 'a mrex2 **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) |  
 $\text{shift2 } m \text{ (Star2 } r \text{ -) } c = \text{star2 (shift2 (} m \vee \text{fn } r \text{) } r \text{ } c)$

**primrec strip2 where**

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

**lemma strip-mrexp2:**  $(\text{strip } o \text{ strip2}) \text{ ' } \text{mrexp2 } r = \{r\}$   
**by** (induction r) (auto simp: set-eq-subset subset-iff image-iff)

**primrec ok2 :: 'a mrep2  $\Rightarrow$  bool where**

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

**lemma ok2-fin-final[simp]:**  $\text{ok2 } r \Longrightarrow \text{fn } r = \text{final (strip2 } r \text{)}$   
**by** (induct r) auto

**lemma ok2-nul-nullable[simp]:**  $\text{ok2 } r \Longrightarrow \text{nul } r = \text{nullable (strip2 } r \text{)}$   
**by** (induct r) auto

**lemma strip2-shift2:**  $\text{ok2 } r \Longrightarrow \text{strip2}(\text{shift2 } m \text{ } r \text{ } c) = \text{shift } m \text{ (strip2 } r \text{) } c$   
**apply**(induction r arbitrary: m)  
**apply** (auto simp: disj-commute)  
**done**

**lemma ok2-empty-mrep2:**  $\text{ok2 (empty-mrep2 } r \text{)}$   
**apply**(induction r)  
**apply** auto  
**done**

**lemma ok2-shift2:**  $\text{ok2 } r \Longrightarrow \text{ok2}(\text{shift2 } m \text{ } r \text{ } c)$   
**apply**(induction r arbitrary: m)  
**apply** auto  
**done**

**lemma strip2-empty-mrep2[simp]:**  $\text{strip2 (empty-mrep2 } r \text{) } = \text{empty-mrep } r$   
**by** (induct r) auto

**lemma nul-empty-mrep2[simp]:**  $\text{nul (empty-mrep2 } r \text{) } = \text{nullable } r$

**by** (*induct r*) *auto*

**lemma** *nonfin-empty-mrexp2[simp]*:  $\neg \text{fin } (\text{empty-mrexp2 } r)$   
**by** (*induct r*) *auto*

**lemma** *empty-mrexp2-mrexp2[simp]*:  $\text{empty-mrexp2 } s \in \text{mrexp2 } s$   
**by** (*induct s*) (*auto simp: image-iff*)

**lemma** *shift2-mrexp2[simp]*:  $r \in \text{mrexp2 } s \implies \text{shift2 } x \ r \ a \in \text{mrexp2 } s$   
**by** (*induct s arbitrary: r x*) (*auto simp: image-iff*)

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

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

**lift-definition** *init-okm* ::  $'a \text{ rexp} \Rightarrow 'a \text{ ok-mrexp2}$  **is**  $\lambda r. (\text{True}, \text{empty-mrexp2 } r)$   
**by** (*simp add: ok2-empty-mrexp2 del: ok2.simps*)

**lift-definition** *delta-okm* ::  $'a \Rightarrow 'a \text{ ok-mrexp2} \Rightarrow 'a \text{ ok-mrexp2}$  **is**  
 $\lambda a \ (m, r). (\text{False}, \text{shift2 } m \ r \ a)$   
**unfolding** *mem-Collect-eq split-beta snd-conv* **by** (*intro ok2-shift2*) *simp*

**lift-definition** *nullable-okm* ::  $'a \text{ ok-mrexp2} \Rightarrow \text{bool}$  **is**  $\lambda(m, r). \text{fin } r \vee m \wedge \text{nul } r$   
**lift-definition** *lang-okm* ::  $'a \text{ ok-mrexp2} \Rightarrow 'a \text{ lang}$  **is**  $\lambda(m, r). L\text{-a } (m, \text{strip2 } r)$  .

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

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

**lemma** *eq-mrexp2-imp-eq*:  $\llbracket \text{eq-mrexp2 } r \ s; \text{ok2 } r; \text{ok2 } s \rrbracket \implies (r = s)$   
**by** (*induct rule: eq-mrexp2.induct*) *auto*

**lemma** *eq-mrexp2-refl[simplified, simp]*:  $r = s \implies \text{eq-mrexp2 } r \ s$   
**by** (*induct rule: eq-mrexp2.induct*) *auto*

**lemma** *eq-mrexp2-eq*:  $\llbracket \text{ok2 } r; \text{ok2 } s \rrbracket \implies \text{eq-mrexp2 } r \ s = (r = s)$   
**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-egu-a2 = after2.check-egu
    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 (λa (m, r). (False, shift2 m r a)) w (True, empty-mrexp2 s) |w. True}
  ⊆
    UNIV × mrexp2 s
  proof (intro subsetI, elim CollectE exE conjE, hypsubst)
    fix w show fold (λa (m, r). (False, shift2 m r a)) w (True, empty-mrexp2 s) ∈
      UNIV × mrexp2 s
    by (induct w rule: rev-induct) (auto simp: split: prod.splits intro!: shift2-mrexp2)
  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 mrexps3 =  
 Zero3 |  
 One3 |  
 Atom3 bool 'a |  
 Plus3 'a mrexps3 'a mrexps3 (fin1: 'a set) (nul: bool) |  
 Times3 'a mrexps3 'a mrexps3 (fin1: 'a set) (nul: bool) |  
 Star3 'a mrexps3 (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  $\vee$  final3 s)  
 | final3 (Times3 r s -) = (final3 s  $\vee$  nul s  $\wedge$  final3 r)  
 | final3 (Star3 r -) = final3 r

**primrec** mrexps3 :: 'a rexp  $\Rightarrow$  ('a mrexps3) set **where**  
 mrexps3 Zero = {Zero3}  
 | mrexps3 One = {One3}  
 | mrexps3 (Atom a) = {Atom3 True a, Atom3 False a}  
 | mrexps3 (Plus r s) = ( $\lambda(r, s, f1, n).$  Plus3 r s f1 n) ' (mrexps3 r  $\times$  mrexps3 s  $\times$  Pow (atoms (Plus r s))  $\times$  UNIV)  
 | mrexps3 (Times r s) = ( $\lambda(r, s, f1, n).$  Times3 r s f1 n) ' (mrexps3 r  $\times$  mrexps3 s  $\times$  Pow (atoms (Times r s))  $\times$  UNIV)  
 | mrexps3 (Star r) = ( $\lambda(r, f1).$  Star3 r f1) ' (mrexps3 r  $\times$  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  $\cup$  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 mrexps3  $\Rightarrow$  'a mrexps3 **where**  
 follow3 m Zero3 = Zero3 |  
 follow3 m One3 = One3 |

$follow3\ m\ (Atom3\ -\ a) = Atom3\ m\ a\ |$   
 $follow3\ m\ (Plus3\ r\ s\ -) = plus3\ (follow3\ m\ r)\ (follow3\ m\ s)\ |$   
 $follow3\ m\ (Times3\ r\ s\ -) =$   
 $\quad times3\ (follow3\ m\ r)\ (follow3\ (final3\ r\ \vee\ m\ \wedge\ nul\ r)\ s)\ |$   
 $follow3\ m\ (Star3\ r\ -) = star3\ (follow3\ (final3\ r\ \vee\ m)\ r)$

**primrec**  $empty\_mrepr3 :: 'a\ rexp \Rightarrow 'a\ mrepr3$  **where**  
 $empty\_mrepr3\ Zero = Zero3\ |$   
 $empty\_mrepr3\ One = One3\ |$   
 $empty\_mrepr3\ (Atom\ x) = Atom3\ False\ x\ |$   
 $empty\_mrepr3\ (Plus\ r\ s) = plus3\ (empty\_mrepr3\ r)\ (empty\_mrepr3\ s)\ |$   
 $empty\_mrepr3\ (Times\ r\ s) = times3\ (empty\_mrepr3\ r)\ (empty\_mrepr3\ s)\ |$   
 $empty\_mrepr3\ (Star\ r) = star3\ (empty\_mrepr3\ r)$

**primrec**  $move3 :: 'a \Rightarrow 'a\ mrepr3 \Rightarrow bool \Rightarrow 'a\ mrepr3$  **where**  
 $move3\ -\ One3\ - = One3\ |$   
 $move3\ -\ Zero3\ - = Zero3\ |$   
 $move3\ c\ (Atom3\ -\ a)\ m = Atom3\ m\ a\ |$   
 $move3\ c\ (Plus3\ r\ s\ -)\ m = plus3\ (move3\ c\ r\ m)\ (move3\ c\ s\ m)\ |$   
 $move3\ c\ (Times3\ r\ s\ -)\ m =$   
 $\quad times3\ (move3\ c\ r\ m)\ (move3\ c\ s\ (c \in fin1\ r \vee m \wedge nul\ r))\ |$   
 $move3\ c\ (Star3\ r\ -)\ m = star3\ (move3\ c\ r\ (c \in fin1\ r \vee m))$

**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**  $strip\_mreprs3$ :  $(strip\ o\ strip3)\ 'mreprs3\ r = \{r\}$   
**by**  $(induction\ r)\ (auto\ simp:\ set\ eq\ subset\ subset\ iff\ image\ iff)$

**primrec**  $ok3 :: 'a\ mrepr3 \Rightarrow bool$  **where**  
 $ok3\ Zero3 = True\ |$   
 $ok3\ One3 = True\ |$   
 $ok3\ (Atom3\ -) = True\ |$   
 $ok3\ (Plus3\ r\ s\ f1\ n) = (ok3\ r \wedge ok3\ s \wedge$   
 $\quad (let\ rs = Plus\ (strip3\ r)\ (strip3\ s)\ in\ f1 = Collect\ (final1\ rs) \wedge n = nullable\ rs))$   
 $|$   
 $ok3\ (Times3\ r\ s\ f1\ n) = (ok3\ r \wedge ok3\ s \wedge$   
 $\quad (let\ rs = Times\ (strip3\ r)\ (strip3\ s)\ in\ f1 = Collect\ (final1\ rs) \wedge n = nullable$   
 $\quad rs))\ |$   
 $ok3\ (Star3\ r\ f1) = (ok3\ r \wedge f1 = Collect\ (final1\ (strip3\ r)))$

**lemma**  $ok3\_fin1\_final1[simp]$ :  $ok3\ r \Longrightarrow fin1\ r = Collect\ (final1\ (strip3\ r))$   
**by**  $(induct\ r)\ (auto\ simp\ add:\ set\ eq\ iff)$

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

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

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

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

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

**lemma** *fin1-atoms*:  $\llbracket x \in fin1\ mr; mr \in mrexp3\ r \rrbracket \implies x \in atoms\ r$   
**by** (*induct r*) *auto*

**lemma** *follow3-mrexp3[simp]*:  $r \in mrexp3\ s \implies follow3\ m\ r \in mrexp3\ s$   
**by** (*induct s arbitrary: m r*) (*fastforce simp add: image-iff dest: fin1-atoms*)+

**lemma** *empty-mrexp3-mrexp[simp]*:  $empty-mrexp3\ r \in mrexp3\ r$   
**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*:  $ok3\ r \implies strip3\ (move3\ m\ r\ c) = move\ m\ (strip3\ r)\ c$   
**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*:  $ok3\ r \implies ok3\ (move3\ m\ r\ c)$   
**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-mrexp3[simp]*:  $r \in \text{mrexp3 } s \implies \text{move3 } x \ r \ a \in \text{mrexp3 } s$   
**by** (*induct s arbitrary: r x a*) (*fastforce simp: image-iff dest: fin1-atoms*)<sup>+</sup>

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

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

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

**lift-definition** *init-okm* ::  $'a \ \text{rexp} \Rightarrow 'a \ \text{ok-mrexp3}$  **is** *init-m*  
**by** (*simp add: ok3-empty-mrexp3*)

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

**unfolding** *mem-Collect-eq split-beta fst-conv* **by** (*intro ok3-move3*) *simp*

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

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

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

**lemma** *eq-mrexp3-imp-eq*:  $\llbracket \text{eq-mrexp3 } r \ s; \text{ok3 } r; \text{ok3 } s \rrbracket \implies (r = s)$   
**by** (*induct rule: eq-mrexp3.induct*) *auto*

**lemma** *eq-mrexp3-refl[simplified, simp]*:  $r = s \implies \text{eq-mrexp3 } r \ s$   
**by** (*induct rule: eq-mrexp3.induct*) *auto*

**lemma** *eq-mrexp3-eq*:  $\llbracket \text{ok3 } r; \text{ok3 } s \rrbracket \implies \text{eq-mrexp3 } r \ s = (r = s)$   
**by** (*metis eq-mrexp3-imp-eq eq-mrexp3-refl*)

**lift-definition** *equal-ok-mrexp3* ::  $'a \ \text{ok-mrexp3} \Rightarrow 'a \ \text{ok-mrexp3} \Rightarrow \text{bool}$   
**is**  $\lambda(r1, b1) \ (r3, b3). b1 = b3 \ \wedge \ \text{eq-mrexp3 } r1 \ r3$  .

**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 = ( $\emptyset \in$  lang-okm br)
        by transfer (simp add: split-beta)
    next
      case (4 s)
      have {fold ( $\lambda a$  (r, m). (move3 a r False, a  $\in$  fin1 r)) w (init-m s) |w. True}  $\subseteq$ 
mrexp3 s  $\times$  UNIV
      proof (intro subsetI, elim CollectE exE conjE, hypsubst)
        fix w show fold ( $\lambda a$  (r, m). (move3 a r False, a  $\in$  fin1 r)) w (init-m s)  $\in$ 
mrexp3 s  $\times$  UNIV
        by (induct w rule: rev-induct) (auto simp: split: prod.splits intro!: move3-mrexp3)
      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** *Exp*

## References

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