Regular Expression Equivalence

Tobias Nipkow Dmitriy Traytel

May 26, 2024

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

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

lemma *set-add-atoms*: *set* (*add-atoms r as*) = *atoms r* ∪ *set as*
⟨*proof*⟩

lemma *rtrancl-fold-product*:
shows {((*r*,*s*),(*f a r*,*f a s*))| *r s a*. *a* : *A*}^* =
{((*r*,*s*),(*fold f w r*,*fold f w s*))| *r s w*. *w* : *lists A*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *rtrancl-fold-product1*:
shows {(*r*,*s*). ∃ *a* ∈ *A*. *s* = *f a r*}^* =
{(*r*,*fold f w r*) | *r w*. *w* : *lists A*} (**is** *?L* = *?R*)
⟨*proof*⟩

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

locale *rexp-DA* =

fixes *init* :: *'a rexp* ⇒ *'s*
fixes *delta* :: *'a* ⇒ *'s* ⇒ *'s*
fixes *final* :: *'s* ⇒ *bool*
fixes *L* :: *'s* ⇒ *'a lang*
assumes *L-init*: *L* (*init r*) = *lang r*
assumes *L-delta*: *L*(*delta a s*) = *Deriv a* (*L s*)
assumes *final-iff-Nil*: *final s* ⟷ [] ∈ *L s*
begin

lemma *L-deltas*: *L* (*fold delta w s*) = *fold Deriv w* (*L s*)
⟨*proof*⟩

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

theorem *closure-sound-complete*:
assumes *result*: *closure as* (*init r*,*init s*) = *Some*(*ws*,*R*)
and *atoms*: *set as* = *atoms r* ∪ *atoms s*
shows *ws* = [] ⟷ *lang r* = *lang s*
⟨*proof*⟩

1.1 The overall procedure

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

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

Auxiliary functions:

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

definition *automaton* :: *'a list* ⇒ *'a rexp* ⇒ ((*'s* ∗ *'a*) ∗ *'s*) *set* **where**
automaton as s =
snd (*the*
(*let i* = *init s*;
start = (([*i*], {*i*}), {});
test = λ((*ws*, *Z*), *A*). *ws* ≠ [];
step = λ((*ws*, *Z*), *A*).
(*let s* = *hd ws*;

```
        new-edges = map (λa. ((s, a), delta a s)) as;
        new = remdups (filter (λss. ss ∉ Z) (map snd new-edges))
      in ((new @ tl ws, set new ∪ Z), set new-edges ∪ A))
  in while-option test step start))

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

lemma match-correct: match s w ⟷ w ∈ lang s
  ⟨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: ∃ st. closure as (init r,init s) = Some st (is ∃ -. closure as ?i
= -)
⟨proof⟩

lemma completeness:
assumes lang r = lang s shows check-eqv r s
⟨proof⟩

lemma finite-rtrancl-delta-Image1:
  finite ({(r,s). ∃ a ∈ A. s = delta a r}^* `` {init r})
  ⟨proof⟩

lemma reachable: reachable as r = {fold delta w (init r) | w. w ∈ 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* ≡ *rexp-of-list Plus Zero*
abbreviation *TIMES* ≡ *rexp-of-list Times One*

lemma *list-singleton-induct* [*case-names nil single cons*]:
assumes *P* [] **and** ⋀*x*. *P* [*x*] **and** ⋀*x y xs*. *P* (*y* # *xs*) ⟹ *P* (*x* # (*y* # *xs*))
shows *P xs*
⟨*proof*⟩

2.1 ACI normalization

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

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

lemma *toplevel-summands-nonempty*[*simp*]:
toplevel-summands r ≠ {}
⟨*proof*⟩

lemma *toplevel-summands-finite*[*simp*]:
finite (*toplevel-summands r*)
⟨*proof*⟩

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

lemma *Plus-toplevel-summands*: *Plus r s* ∈ *toplevel-summands t* ⟹ *False*
⟨*proof*⟩

lemma *toplevel-summands-not-Plus*[*simp*]:
(∀ *r s*. *x* ≠ *Plus r s*) ⟹ *toplevel-summands x* = {*x*}
⟨*proof*⟩

lemma *toplevel-summands-PLUS-strong*:
⟦*xs* ≠ []; *list-all* (λ*x*. ¬(∃ *r s*. *x* = *Plus r s*)) *xs*⟧ ⟹ *toplevel-summands* (*PLUS xs*) = *set xs*
⟨*proof*⟩

lemma *toplevel-summands-flatten*:
⟦*X* ≠ {}; *finite X*; ∀ *x* ∈ *X*. ¬(∃ *r s*. *x* = *Plus r s*)⟧ ⟹ *toplevel-summands* (*flatten PLUS X*) = *X*
⟨*proof*⟩

lemma *ACI-norm-Plus*: $«r» = Plus\ s\ t \Longrightarrow \exists\, s\ t.\ r = Plus\ s\ t$
⟨*proof*⟩

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

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

lemma *toplevel-summands-ACI-norm*:
$toplevel\text{-}summands\ «r» = ACI\text{-}norm\ `\ toplevel\text{-}summands\ r$
⟨*proof*⟩

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

theorem *ACI-norm-idem*[*simp*]: $««r»» = «r»$
⟨*proof*⟩

2.2 Atoms

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

lemma *wf-PLUS*: $atoms\ (PLUS\ xs) \subseteq \Sigma \longleftrightarrow (\forall\, r \in set\ xs.\ atoms\ r \subseteq \Sigma)$
⟨*proof*⟩

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

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

theorem *atoms-ACI-norm*: $atoms\ «r» = atoms\ r$
⟨*proof*⟩

2.3 Language

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

lemma *toplevel-summands-lang-UN*:

lang s = (⋃ *r*∈*toplevel-summands s*. *lang r*)
⟨*proof*⟩

lemma *toplevel-summands-in-lang*:
w ∈ *lang s* = (∃ *r*∈*toplevel-summands s*. *w* ∈ *lang r*)
⟨*proof*⟩

lemma *lang-PLUS*: *lang* (*PLUS xs*) = (⋃ *r* ∈ *set xs*. *lang r*)
⟨*proof*⟩

lemma *lang-PLUS-map*[*simp*]:
lang (*PLUS* (*map f xs*)) = (⋃ *a*∈*set xs*. *lang* (*f a*))
⟨*proof*⟩

lemma *lang-flatten-PLUS*[*simp*]:
finite X ⟹ *lang* (*flatten PLUS X*) = (⋃ *r* ∈ *X*. *lang r*)
⟨*proof*⟩

theorem *lang-ACI-norm*[*simp*]: *lang* «*r*» = *lang r*
⟨*proof*⟩

2.4 Finiteness of ACI-Equivalent Derivatives

lemma *toplevel-summands-deriv*:
toplevel-summands (*deriv as r*) = (⋃ *s*∈*toplevel-summands r*. *toplevel-summands* (*deriv as s*))
⟨*proof*⟩

lemma *derivs-Zero*[*simp*]: *derivs xs Zero* = *Zero*
⟨*proof*⟩

lemma *derivs-One*: *derivs xs One* ∈ {*Zero*, *One*}
⟨*proof*⟩

lemma *derivs-Atom*: *derivs xs* (*Atom as*) ∈ {*Zero*, *One*, *Atom as*}
⟨*proof*⟩

lemma *derivs-Plus*: *derivs xs* (*Plus r s*) = *Plus* (*derivs xs r*) (*derivs xs s*)
⟨*proof*⟩

lemma *derivs-PLUS*: *derivs xs* (*PLUS ys*) = *PLUS* (*map* (*derivs xs*) *ys*)
⟨*proof*⟩

lemma *toplevel-summands-derivs-Times*: *toplevel-summands* (*derivs xs* (*Times r s*)) ⊆
{*Times* (*derivs xs r*) *s*} ∪
{*r*′. ∃ *ys zs*. *r*′ ∈ *toplevel-summands* (*derivs ys s*) ∧ *ys* ≠ [] ∧ *zs* @ *ys* = *xs*}
⟨*proof*⟩

lemma *toplevel-summands-derivs-Star-nonempty*:
$xs \neq [] \Longrightarrow$ *toplevel-summands* (*derivs xs* (*Star r*)) $\subseteq$
$\{$*Times* (*derivs ys r*) (*Star r*) | *ys*. $\exists$ *zs*. *ys* $\neq [] \wedge$ *zs* @ *ys* = *xs*$\}$
⟨*proof*⟩

lemma *toplevel-summands-derivs-Star*:
toplevel-summands (*derivs xs* (*Star r*)) $\subseteq$
$\{$*Star r*$\} \cup \{$*Times* (*derivs ys r*) (*Star r*) | *ys*. $\exists$ *zs*. *ys* $\neq [] \wedge$ *zs* @ *ys* = *xs*$\}$
⟨*proof*⟩

lemma *toplevel-summands-PLUS*:
xs $\neq []$ $\Longrightarrow$ *toplevel-summands* (*PLUS* (*map f xs*)) = ($\bigcup$ *r* $\in$ *set xs*. *toplevel-summands* (*f r*))
⟨*proof*⟩

lemma *ACI-norm-toplevel-summands-Zero*: *toplevel-summands r* $\subseteq$ $\{$*Zero*$\}$ $\Longrightarrow$ «*r*» = *Zero*
⟨*proof*⟩

lemma *finite-ACI-norm-toplevel-summands*:
finite $\{$*f* «*s*» |*s*. *toplevel-summands s* $\subseteq$ *B*$\}$ **if** *finite B*
⟨*proof*⟩

theorem *finite-derivs*: *finite* $\{$«*derivs xs r*» | *xs* . *True*$\}$
⟨*proof*⟩

2.5 Deriving preserves ACI-equivalence

lemma *ACI-norm-PLUS*:
list-all2 (λ*r s*. «*r*» = «*s*») *xs ys* $\Longrightarrow$ «*PLUS xs*» = «*PLUS ys*»
⟨*proof*⟩

lemma *toplevel-summands-ACI-norm-deriv*:
($\bigcup$ *a*$\in$*toplevel-summands r*. *toplevel-summands* «*deriv as* «*a*»») = *toplevel-summands* «*deriv as* «*r*»»
⟨*proof*⟩

lemma *toplevel-summands-nullable*:
nullable s = ($\exists$ *r*$\in$*toplevel-summands s*. *nullable r*)
⟨*proof*⟩

lemma *nullable-PLUS*:
nullable (*PLUS xs*) = ($\exists$ *r* $\in$ *set xs*. *nullable r*)
⟨*proof*⟩

theorem *ACI-norm-nullable*: *nullable* «*r*» = *nullable r*
⟨*proof*⟩

theorem *ACI-norm-deriv*: «*deriv as* «*r*»» = «*deriv as r*»
⟨*proof*⟩

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

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

theorem *ACI-norm-derivs*: «*derivs xs* «*r*»» = «*derivs xs r*»
⟨*proof*⟩

2.6 Alternative ACI defintions

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

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

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

lemma *toplevel-summands-ACI-nPlus*:
toplevel-summands (*ACI-nPlus r s*) = *toplevel-summands* (*Plus r s*)
⟨*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* ⟹ ∃ *s t*. *r* = *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* $\Longrightarrow$ *lang r* = *lang s*
⟨*proof*⟩

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

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

lemma *ACI-Plus-idemI*[*intro*]: ⟦*ACI r s1*; *ACI r s2*⟧ $\Longrightarrow$ *ACI r* (*Plus s1 s2*)
⟨*proof*⟩

lemma *ACI-Plus-idemI'*[*intro*]:
⟦*ACI r1 s1*; *ACI* (*Plus r1 r2*) *s2*⟧ $\Longrightarrow$ *ACI* (*Plus r1 r2*) (*Plus s1 s2*)
⟨*proof*⟩

lemma *ACI-ACI-nPlus*: ⟦*ACI r1 s1*; *ACI r2 s2*⟧ $\Longrightarrow$ *ACI* (*ACI-nPlus r1 r2*) (*Plus s1 s2*)
⟨*proof*⟩

lemma *ACI-ACI-norm*: *ACI* «*r*» *r*
⟨*proof*⟩

lemma *ACI-norm-eqI*: *ACI r s* $\Longrightarrow$ «*r*» = «*s*»
⟨*proof*⟩

lemma *ACI-I*: «*r*» = «*s*» $\Longrightarrow$ *ACI r s*
⟨*proof*⟩

lemma *ACI-decidable*: *ACI r s* = («*r*» = «*s*»)
⟨*proof*⟩

3 Connection Between Derivatives and Partial Derivatives

lemma *pderiv-not-is-Zero-is-Plus*[*simp*]: ∀ *x* ∈ *pderiv a r*. ¬ *is-Zero x* ∧ ¬ *is-Plus x*
⟨*proof*⟩

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

lemma *PLUS-inject*: ⟦∀ *x* ∈ *set xs* ∪ *set ys*. ¬ *is-Zero x* ∧ ¬ *is-Plus x*; *sorted xs*; *sorted ys*⟧ $\Longrightarrow$

$(PLUS\ xs = PLUS\ ys) \longleftrightarrow xs = ys$
⟨*proof*⟩

lemma *sorted-list-of-set-inject*: ⟦*finite R*; *finite S*⟧ ⟹
$(sorted\text{-}list\text{-}of\text{-}set\ R = sorted\text{-}list\text{-}of\text{-}set\ S) \longleftrightarrow R = S$
⟨*proof*⟩

lemma *flatten-PLUS-inject*: ⟦∀ $x \in R \cup S$. ¬ *is-Zero* $x \wedge \neg$ *is-Plus* x; *finite R*; *finite S*⟧ ⟹
$(flatten\ PLUS\ R = flatten\ PLUS\ S) = (R = S)$
⟨*proof*⟩

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

lemma *pset-not-is-Zero-is-Plus*[*simp*]: ∀ $x \in pset\ r$. ¬ *is-Zero* $x \wedge \neg$ *is-Plus* x
⟨*proof*⟩

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

lemma *pset-deriv*: $pset\ (deriv\ a\ r) = pderiv\ a\ r$
⟨*proof*⟩

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

lemma *pnorm-deriv-eq-iff-pderiv-eq*:
$pnorm\ (deriv\ a\ r) = pnorm\ (deriv\ a\ s) \longleftrightarrow pderiv\ a\ r = pderiv\ a\ s$
⟨*proof*⟩

fun *pnPlus* :: *'a*::*linorder rexp* ⇒ *'a rexp* ⇒ *'a rexp* **where**
$pnPlus\ Zero\ r = r$
| $pnPlus\ r\ Zero = r$
| $pnPlus\ (Plus\ r\ s)\ t = pnPlus\ r\ (pnPlus\ s\ t)$
| $pnPlus\ r\ (Plus\ s\ t) =$
(*if* $r = s$ *then* $(Plus\ s\ t)$
else if le-rexp $r\ s$ *then* $Plus\ r\ (Plus\ s\ t)$
else $Plus\ s\ (pnPlus\ r\ t)$)
| $pnPlus\ r\ s =$
(*if* $r = s$ *then* r
else if le-rexp $r\ s$ *then* $Plus\ r\ s$
else $Plus\ s\ r$)

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

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

lemma *pset-pnPlus*:
  pset (*pnPlus r s*) = *pset* (*Plus r s*)
  ⟨*proof*⟩

lemma *pset-pnTimes*:
  pset (*pnTimes r s*) = *pset* (*Times r s*)
  ⟨*proof*⟩

lemma *pset-pnorm-alt-Times*: *s* ∈ *pset r* ⟹ *pnTimes* (*pnorm-alt s*) *t* = *Times* (*pnorm-alt s*) *t*
  ⟨*proof*⟩

lemma *pset-pnorm-alt*:
  pset (*pnorm-alt r*) = *pnorm-alt* ‘ *pset r*
  ⟨*proof*⟩

lemma *pset-pnTimes-Times*: *s* ∈ *pset r* ⟹ *pnTimes s t* = *Times s t*
  ⟨*proof*⟩

lemma *pset-pnorm-alt-id*: *s* ∈ *pset r* ⟹ *pnorm-alt s* = *s*
  ⟨*proof*⟩

lemma *pnorm-alt-image-pset*: *pnorm-alt* ‘ *pset r* = *pset r*
  ⟨*proof*⟩

lemma *pnorm-pnorm-alt*: *pnorm* (*pnorm-alt r*) = *pnorm r*
  ⟨*proof*⟩

lemma *pnPlus-singleton-PLUS*:
  ⟦*xs* ≠ []; *sorted xs*; *distinct xs*; ∀ *x* ∈ {*x*} ∪ *set xs*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧ ⟹
  pnPlus x (*PLUS xs*) = (*if x* ∈ *set xs then PLUS xs else PLUS* (*insort x xs*))
⟨*proof*⟩

lemma *pnPlus-PlusL*[*simp*]: *t* ≠ *Zero* ⟹ *pnPlus* (*Plus r s*) *t* = *pnPlus r* (*pnPlus s t*)
  ⟨*proof*⟩

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

lemma *PLUS-eq-Zero*: *PLUS xs* = *Zero* ⟷ *xs* = [] ∨ *xs* = [*Zero*]
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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

lemma *pnTimes-not-Zero-or-Plus*[*simp*]: ⟦¬ *is-Zero x*; ¬ *is-Plus x*⟧ ⟹ *pnTimes x r* = *Times x r*
⟨*proof*⟩

lemma *pnTimes-PLUS*:
⟦*xs* ≠ []; ∀ *x* ∈ *set xs*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧⟹
pnTimes (*PLUS xs*) *r* = *flatten PLUS* (*Timess* (*set xs*) *r*)
⟨*proof*⟩

lemma *pnTimes-flatten-PLUS*:
⟦*X1* ≠ {}; *finite X1*; ∀ *x* ∈ *X1*. ¬*is-Zero x* ∧ ¬*is-Plus x*⟧⟹
pnTimes (*flatten PLUS X1*) *r* = *flatten PLUS* (*Timess X1 r*)
⟨*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*: ⟦∀ *a* ∈ *set xs*. *b* < *a*; *sorted xs*⟧ ⟹ *insort b xs* = *b* # *xs*
⟨*proof*⟩

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

lemma *pderiv-set-flatten-PLUS*:
finite X $\Longrightarrow$ *pderiv* (*a* :: *'a* :: *linorder*) (*flatten PLUS X*) = *pderiv-set a X*
⟨*proof*⟩

lemma *fold-pderiv-set-flatten-PLUS*:
⟦*w* ≠ []; *finite X*⟧ $\Longrightarrow$ *fold pderiv-set w* {*flatten PLUS X*} = *fold pderiv-set w X*
⟨*proof*⟩

lemma *fold-pnorm-deriv*:
fold (λ*a r*. *pnorm* (*deriv a r*)) *w s* = *flatten PLUS* (*fold pderiv-set w* {*s*})
⟨*proof*⟩

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

lemma *pnderiv-alt*[*code*]: *pnderiv a r* = *pnorm* (*deriv a r*)
⟨*proof*⟩

lemma *pnderiv-pderiv*: *pnderiv 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*: «*derivs w r*» = *fold* (λ*a r*. «*deriv a r*») *w* «*r*»
⟨*proof*⟩

global-interpretation *brzozowski*: *rexp-DFA* λ*r*. «*r*» λ*a r*. «*deriv a r*» *nullable lang*
defines *brzozowski-closure* = *brzozowski.closure*
and *check-eqv-brz* = *brzozowski.check-eqv*
and *reachable-brz* = *brzozowski.reachable*
and *automaton-brz* = *brzozowski.automaton*
and *match-brz* = *brzozowski.match*
⟨*proof*⟩

4.2 Brzozowski 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 *brzozowski-quotient*: *rexp-DFA ACI-class ACI-deriv ACI-nullable ACI-lang*
defines *brzozowski-quotient-closure* = *brzozowski-quotient.closure*
and *check-eqv-brzq* = *brzozowski-quotient.check-eqv*
and *reachable-brzq* = *brzozowski-quotient.reachable*
and *automaton-brzq* = *brzozowski-quotient.automaton*
and *match-brzq* = *brzozowski-quotient.match*
⟨*proof*⟩

4.3 Brzozowski 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* λ*r*. {*r*} *pderiv-set* λ*P*. ∃ *p*∈*P*. *nullable p* λ*P*. ⋃(*lang* ‘ *P*)
defines *pderiv-closure* = *pderiv.closure*
and *check-eqv-p* = *pderiv.check-eqv*
and *reachable-p* = *pderiv.reachable*
and *automaton-p* = *pderiv.automaton*
and *match-p* = *pderiv.match*
⟨*proof*⟩

global-interpretation *pnderiv*: *rexp-DFA* λ*r*. *r pnderiv nullable lang*
defines *pnderiv-closure* = *pnderiv.closure*
and *check-eqv-pn* = *pnderiv.check-eqv*
and *reachable-pn* = *pnderiv.reachable*
and *automaton-pn* = *pnderiv.automaton*
and *match-pn* = *pnderiv.match*
⟨*proof*⟩

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* λ*L*. [] ∈ *L id*
⟨*proof*⟩

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

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

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

5 Framework Instantiations using Marked Regular Expressions

5.1 Marked Regular Expressions

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

abbreviation *strip* ≡ *map-rexp snd*

primrec *mrexps* :: *'a rexp* ⇒ (*'a mrexp*) *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*)
⟨*proof*⟩

lemma *strip-mrexps*: *strip* ‘ *mrexps r* = {*r*}
⟨*proof*⟩

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

lemma *read-mrexps*[*simp*]: *r* ∈ *mrexps s* ⟹ *read a r* ∈ *mrexps s*
⟨*proof*⟩

fun *follow* :: *bool* ⇒ *'a mrexp* ⇒ *'a mrexp* **where**
follow m Zero = *Zero* |
follow m One = *One* |
follow m (*Atom*(-,*a*)) = *Atom*(*m*,*a*) |

follow m (Plus r s) = Plus (follow m r) (follow m s) |
follow m (Times r s) =
Times (follow m r) (follow (final r ∨ m ∧ nullable r) s) |
follow m (Star r) = Star(follow (final r ∨ m) r)

lemma *follow-mrexps*[*simp*]: *r ∈ mrexps s ⟹ follow b r ∈ mrexps s*
⟨*proof*⟩

lemma *strip-read*[*simp*]: *strip (read a r) = strip r*
⟨*proof*⟩

lemma *Nil-notin-Lm*[*simp*]: *[] ∉ Lm r*
⟨*proof*⟩

lemma *Nil-in-lang-strip*[*simp*]: *[] ∈ lang(r) ⟷ [] ∈ lang(strip r)*
⟨*proof*⟩

lemma *strip-follow*[*simp*]: *strip(follow m r) = strip r*
⟨*proof*⟩

lemma *conc-lemma*: *[] ∉ A ⟹ {w : A @@ B. w ≠ [] ∧ P(hd w)} = {w : A. w ≠ [] ∧ P(hd w)} @@ B*
⟨*proof*⟩

lemma *Lm-read*: *Lm (read a r) = {w : Lm r. w ≠ [] ∧ hd w = a}*
⟨*proof*⟩

lemma *tl-conc*[*simp*]: *[] ∉ A ⟹tl ‘ (A @@ B) = tl ‘ A @@ B*
⟨*proof*⟩

lemma *Nil-in-tl-Lm-if-final*[*simp*]: *final r ⟹ [] : tl ‘ Lm r*
⟨*proof*⟩

lemma *Nil-notin-tl-if-not-final*: *¬ final r ⟹ [] ∉ tl ‘ Lm r*
⟨*proof*⟩

lemma *Lm-follow*: *Lm (follow m r) = tl ‘ Lm r ∪ (if m then lang(strip r) else {}) − {[]}*
⟨*proof*⟩

5.2 Mark Before Atom

Position automaton where mark is placed before atoms.

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

lemma *empty-mrexp-mrexps*[*simp*]: *empty-mrexp r ∈ mrexps r*
⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *init-b-mrexps*[*simp*]: *init-b r* ∈ *mrexps r* × *UNIV*
⟨*proof*⟩

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

lemma *delta-b-mrexps*[*simp*]: *rb* ∈ *mrexps r* × *UNIV* ⟹ *delta-b a rb* ∈ *mrexps r* × *UNIV*
⟨*proof*⟩

lemma *fold-delta-b-init-b-mrexps*[*simp*]: *fold delta-b w* (*init-b s*) ∈ *mrexps s* × *UNIV*
⟨*proof*⟩

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

abbreviation *final-b* ≡ *snd*

lemma *Lm-empty*: *Lm* (*empty-mrexp r*) = {}
⟨*proof*⟩

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

global-interpretation *before*: *rexp-DFA init-b delta-b final-b L-b*
defines *before-closure* = *before.closure*
and *check-eqv-b* = *before.check-eqv*
and *reachable-b* = *before.reachable*
and *automaton-b* = *before.automaton*
and *match-b* = *before.match*
⟨*proof*⟩

5.3 Mark After Atom

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

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

lemma *init-a-mrexps*[*simp*]: *init-a r* ∈ *UNIV* × *mrexps r*
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *tl-eq-Cons-iff*[*simp*]: $tl\ ys = x\ \#\ xs \longleftrightarrow (\exists\, y.\ ys = y\ \#\ x\ \#\ xs)$
⟨*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" atomaton is a quotient of the "after" automaton.

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

fun *hom-ab* **where**
  $hom\text{-}ab\ (b,\ r) = (follow\ b\ r,\ final\text{-}a\ (b,\ r))$

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

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

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

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

theorem *card-reachable-ab*: *card* (*reachable-b as r*) ≤ *card* (*reachable-a as r*)
⟨*proof*⟩

The implementation by Fischer et al.:

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

lemma *shift-read-follow*: *shift m r x* = *read x* (*follow m r*)
⟨*proof*⟩

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

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

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

lemma *nullable-read*[*simp*]: *nullable* (*read c r*) = *nullable r*
⟨*proof*⟩

lemma *final-read-final1*: *final* (*read c r*) = *final1 r c*
⟨*proof*⟩

lemma *move-follow-read*: *move c r m* = *follow m* (*read c r*)
⟨*proof*⟩

6 Linear Time Optimization for "Mark After Atom"

datatype *'a mrexp2* =

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

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

lemma finite-mrexps3[simp]: finite (mrexps2 r)
⟨proof⟩

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

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

primrec shift2 :: bool ⇒ 'a mrexp2 ⇒ 'a ⇒ 'a mrexp2 where
shift2 - One2 - = One2 |
shift2 - Zero2 - = Zero2 |
shift2 m (Atom2 - x) c = Atom2 (m ∧ (x=c)) x |
shift2 m (Plus2 r s - -) c = plus2 (shift2 m r c) (shift2 m s c) |
shift2 m (Times2 r s - -) c = times2 (shift2 m r c) (shift2 (m ∧ nul r ∨ fin r) s
c) |
shift2 m (Star2 r -) c =  star2 (shift2 (m ∨ fin r) r c)

primrec strip2 where
```

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

lemma *strip-mrexps2*: *(strip o strip2) ‘ mrexps2 r = {r}*
⟨*proof*⟩

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

lemma *ok2-fin-final*[*simp*]: *ok2 r ⟹ fin r = final (strip2 r)*
⟨*proof*⟩

lemma *ok2-nul-nullable*[*simp*]: *ok2 r ⟹ nul r = nullable (strip2 r)*
⟨*proof*⟩

lemma *strip2-shift2*: *ok2 r ⟹ strip2(shift2 m r c) = shift m (strip2 r) c*
⟨*proof*⟩

lemma *ok2-empty-mrexp2*: *ok2 (empty-mrexp2 r)*
⟨*proof*⟩

lemma *ok2-shift2*: *ok2 r ⟹ ok2(shift2 m r c)*
⟨*proof*⟩

lemma *strip2-empty-mrexp2*[*simp*]: *strip2 (empty-mrexp2 r) = empty-mrexp r*
⟨*proof*⟩

lemma *nul-empty-mrexp2*[*simp*]: *nul (empty-mrexp2 r) = nullable r*
⟨*proof*⟩

lemma *nonfin-empty-mrexp2*[*simp*]: *¬ fin (empty-mrexp2 r)*
⟨*proof*⟩

lemma *empty-mrexp2-mrexps2*[*simp*]: *empty-mrexp2 s ∈ mrexps2 s*
⟨*proof*⟩

lemma *shift2-mrexps2*[*simp*]: *r ∈ mrexps2 s ⟹ shift2 x r a ∈ mrexps2 s*
⟨*proof*⟩

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** λ*r*. (*True*, *empty-mrexp2 r*)
⟨*proof*⟩
lift-definition *delta-okm* :: *'a* ⇒ *'a ok-mrexp2* ⇒ *'a ok-mrexp2* **is**
λ*a* (*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 *mrexps3* :: *'a rexp* ⇒ (*'a mrexp3*) *set* **where**
mrexps3 Zero = {*Zero3*}
| *mrexps3 One* = {*One3*}
| *mrexps3* (*Atom a*) = {*Atom3 True a*, *Atom3 False a*}
| *mrexps3* (*Plus r s*) = (λ(*r*, *s*, *f1*, *n*). *Plus3 r s f1 n*) ' (*mrexps3 r* × *mrexps3 s* × *Pow* (*atoms* (*Plus r s*)) × *UNIV*)
| *mrexps3* (*Times r s*) = (λ(*r*, *s*, *f1*, *n*). *Times3 r s f1 n*) ' (*mrexps3 r* × *mrexps3 s* × *Pow* (*atoms* (*Times r s*)) × *UNIV*)
| *mrexps3* (*Star r*) = (λ(*r*, *f1*). *Star3 r f1*) ' (*mrexps3 r* × *Pow* (*atoms* (*Star r*)))

lemma *finite-atoms*[*simp*]: *finite* (*atoms r*)
⟨*proof*⟩

lemma *finite-mrexps3*[*simp*]: *finite* (*mrexps3 r*)
⟨*proof*⟩

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

primrec *follow3* :: *bool* ⇒ *'a mrexp3* ⇒ *'a mrexp3* **where**
follow3 m Zero3 = *Zero3* |
follow3 m One3 = *One3* |
follow3 m (*Atom3* - *a*) = *Atom3 m a* |
follow3 m (*Plus3 r s* - -) = *plus3* (*follow3 m r*) (*follow3 m s*) |
follow3 m (*Times3 r s* - -) =
times3 (*follow3 m r*) (*follow3* (*final3 r* ∨ *m* ∧ *nul r*) *s*) |
follow3 m (*Star3 r* -) = *star3*(*follow3* (*final3 r* ∨ *m*) *r*)

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

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

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

lemma *strip-mrexps3*: (*strip o strip3*) ‘ *mrexps3 r* = {*r*}
⟨*proof*⟩

primrec *ok3* :: *'a mrexp3* ⇒ *bool* **where**
ok3 Zero3 = *True* |
ok3 One3 = *True* |

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

lemma *ok3-fin1-final1*[*simp*]: *ok3 r* ⟹ *fin1 r* = *Collect* (*final1* (*strip3 r*))
⟨*proof*⟩

lemma *ok3-nul-nullable*[*simp*]: *ok3 r* ⟹ *nul r* = *nullable* (*strip3 r*)
⟨*proof*⟩

lemma *ok3-final3-final*[*simp*]: *ok3 r* ⟹ *final3 r* = *final* (*strip3 r*)
⟨*proof*⟩

lemma *follow3-follow*[*simp*]: *ok3 r* ⟹ *strip3* (*follow3 m r*) = *follow m* (*strip3 r*)
⟨*proof*⟩

lemma *nul-follow3*[*simp*]: *ok3 r* ⟹ *nul* (*follow3 m r*) = *nul r*
⟨*proof*⟩

lemma *ok3-follow3*[*simp*]: *ok3 r* ⟹ *ok3* (*follow3 m r*)
⟨*proof*⟩

lemma *fin1-atoms*: ⟦*x* ∈ *fin1 mr*; *mr* ∈ *mrexps3 r*⟧ ⟹ *x* ∈ *atoms r*
⟨*proof*⟩

lemma *follow3-mrexps3*[*simp*]: *r* ∈ *mrexps3 s* ⟹ *follow3 m r* ∈ *mrexps3 s*
⟨*proof*⟩

lemma *empty-mrexp3-mrexps*[*simp*]: *empty-mrexp3 r* ∈ *mrexps3 r*
⟨*proof*⟩

lemma *strip3-empty-mrexp3*[*simp*]: *strip3* (*empty-mrexp3 r*) = *empty-mrexp r*
⟨*proof*⟩

lemma *strip3-move3*: *ok3 r* ⟹ *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* ⟹ *ok3*(*move3 m r c*)

⟨*proof*⟩

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

lemma *move3-mrexps3*[*simp*]: $r \in \mathit{mrexps3}\ s \Longrightarrow \mathit{move3}\ x\ r\ a \in \mathit{mrexps3}\ s$
⟨*proof*⟩

typedef $'a$ *ok-mrexp3* $= \{(r :: 'a\ \mathit{mrexp3},\ b :: \mathit{bool}).\ \mathit{ok3}\ r\}$
⟨*proof*⟩

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

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

lift-definition *init-okm* :: $'a$ *rexp* $\Rightarrow$ $'a$ *ok-mrexp3* **is** *init-m*
⟨*proof*⟩
lift-definition *delta-okm* :: $'a \Rightarrow 'a$ *ok-mrexp3* $\Rightarrow 'a$ *ok-mrexp3* **is**
$\lambda a\ (r,\ m).\ (\mathit{move3}\ a\ r\ \mathit{False},\ a \in \mathit{fin1}\ r)$
⟨*proof*⟩
lift-definition *nullable-okm* :: $'a$ *ok-mrexp3* $\Rightarrow$ *bool* **is** *snd* ⟨*proof*⟩
lift-definition *lang-okm* :: $'a$ *ok-mrexp3* $\Rightarrow 'a$ *lang* **is** $\lambda(r,\ m).\ L\text{-}b\ (\mathit{strip3}\ r,\ m)$
⟨*proof*⟩

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* - -) = (*eq-mrexp3 r1 r3* $\wedge$ *eq-mrexp3 s1 s3*)
| *eq-mrexp3* (*Times3 r1 s1* - -) (*Times3 r3 s3* - -) = (*eq-mrexp3 r1 r3* $\wedge$ *eq-mrexp3 s1 s3*)
| *eq-mrexp3* (*Star3 r1* -) (*Star3 r3* -) = (*eq-mrexp3 r1 r3*)
| *eq-mrexp3 r s* = *False*

lemma *eq-mrexp3-imp-eq*: $[\![\mathit{eq\text{-}mrexp3}\ r\ s;\ \mathit{ok3}\ r;\ \mathit{ok3}\ s]\!] \Longrightarrow (r = s)$
⟨*proof*⟩

lemma *eq-mrexp3-refl*[*simplified*, *simp*]: $r = s \Longrightarrow \mathit{eq\text{-}mrexp3}\ r\ s$
⟨*proof*⟩

lemma *eq-mrexp3-eq*: $[\![\mathit{ok3}\ r;\ \mathit{ok3}\ s]\!] \Longrightarrow \mathit{eq\text{-}mrexp3}\ r\ s = (r = s)$
⟨*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-eqv-b2* = *before2.check-eqv*
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-eqv-brz
check-eqv-brzq
check-eqv-n
check-eqv-p
check-eqv-pn
check-eqv-b
check-eqv-b2
check-eqv-a
check-eqv-a2
match-brz
match-brzq
match-n
match-p
match-pn
match-b
match-b2
match-a
match-a2
in *SML* **module-name** *Rexp*

References

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