

Functional Automata

Tobias Nipkow

March 17, 2025

Abstract

This theory defines deterministic and nondeterministic automata in a functional representation: the transition function/relation and the finality predicate are just functions. Hence the state space may be infinite. It is shown how to convert regular expressions into such automata. A scanner (generator) is implemented with the help of functional automata: the scanner chops the input up into longest recognized substrings. Finally we also show how to convert a certain subclass of functional automata (essentially the finite deterministic ones) into regular sets.

1 Overview

The theories are structured as follows:

- Automata: `AutoProj`, `NA`, `NAe`, `DA`, `Automata`
- Conversion of regular expressions into automata: `RegExp2NA`, `RegExp2NAe`, `AutoRegExp`.
- Scanning: `MaxPrefix`, `MaxChop`, `AutoMaxChop`.

For a full description see [1].

In contrast to that paper, the latest version of the theories provides a fully executable scanner generator. The non-executable bits (transitive closure) have been eliminated by going from regular expressions directly to nondeterministic automata, thus bypassing epsilon-moves.

Not described in the paper is the conversion of certain functional automata (essentially the finite deterministic ones) into regular sets contained in `RegSet_of_nat_DA`.

2 Projection functions for automata

```
theory AutoProj
imports Main
```

begin

definition *start* :: "'a * 'b * 'c ⇒ 'a" where "start A = fst A"
definition "next" :: "'a * 'b * 'c ⇒ 'b" where "next A = fst(snd(A))"
definition *fin* :: "'a * 'b * 'c ⇒ 'c" where "fin A = snd(snd(A))"

lemma [*simp*]: "start(q,d,f) = q"
<*proof*>

lemma [*simp*]: "next(q,d,f) = d"
<*proof*>

lemma [*simp*]: "fin(q,d,f) = f"
<*proof*>

end

3 Deterministic automata

theory DA

imports AutoProj

begin

type_synonym ('a,'s)da = "'s * ('a ⇒ 's ⇒ 's) * ('s ⇒ bool)"

definition

foldl2 :: "('a ⇒ 'b ⇒ 'b) ⇒ 'a list ⇒ 'b ⇒ 'b" where
"foldl2 f xs a = foldl1 (λa b. f b a) a xs"

definition

delta :: "('a,'s)da ⇒ 'a list ⇒ 's ⇒ 's" where
"delta A = foldl2 (next A)"

definition

accepts :: "('a,'s)da ⇒ 'a list ⇒ bool" where
"accepts A = (λw. fin A (delta A w (start A)))"

lemma [*simp*]: "foldl2 f [] a = a ∧ foldl2 f (x#xs) a = foldl2 f xs (f x a)"
<*proof*>

lemma *delta_Nil*[*simp*]: "delta A [] s = s"
<*proof*>

lemma *delta_Cons*[*simp*]: "delta A (a#w) s = delta A w (next A a s)"
<*proof*>

lemma *delta_append*[*simp*]:
"∧q ys. delta A (xs@ys) q = delta A ys (delta A xs q)"

<proof>

end

4 Nondeterministic automata

theory NA

imports AutoProj

begin

type_synonym ('a,'s) na = "'s * ('a \Rightarrow 's \Rightarrow 's set) * ('s \Rightarrow bool)"

primrec delta :: "('a,'s)na \Rightarrow 'a list \Rightarrow 's \Rightarrow 's set" where
"delta A [] p = {p}" |
"delta A (a#w) p = Union(delta A w ' next A a p)"

definition

accepts :: "('a,'s)na \Rightarrow 'a list \Rightarrow bool" where
"accepts A w = (\exists q \in delta A w (start A). fin A q)"

definition

step :: "('a,'s)na \Rightarrow 'a \Rightarrow ('s * 's)set" where
"step A a = {(p,q) . q : next A a p}"

primrec steps :: "('a,'s)na \Rightarrow 'a list \Rightarrow ('s * 's)set" where
"steps A [] = Id" |
"steps A (a#w) = step A a 0 steps A w"

lemma steps_append[simp]:

"steps A (v@w) = steps A v 0 steps A w"
<proof>

lemma in_steps_append[iff]:

"(p,r) : steps A (v@w) = ((p,r) : (steps A v 0 steps A w))"
<proof>

lemma delta_conv_steps: " \bigwedge p. delta A w p = {q. (p,q) : steps A w}"

<proof>

lemma accepts_conv_steps:

"accepts A w = (\exists q. (start A,q) \in steps A w \wedge fin A q)"
<proof>

abbreviation

Cons_syn :: "'a \Rightarrow 'a list set \Rightarrow 'a list set" (infixr <##> 65) where
"x ## S \equiv Cons x ' S"

end

5 Nondeterministic automata with epsilon transitions

```
theory NAε
imports NA
begin

type_synonym ('a,'s)naε = "('a option,'s)na"

abbreviation
  eps :: "('a,'s)naε ⇒ ('s * 's)set" where
    "eps A ≡ step A None"

primrec steps :: "('a,'s)naε ⇒ 'a list ⇒ ('s * 's)set" where
  "steps A [] = (eps A)*" |
  "steps A (a#w) = (eps A)* 0 step A (Some a) 0 steps A w"

definition
  accepts :: "('a,'s)naε ⇒ 'a list ⇒ bool" where
  "accepts A w = (∃q. (start A,q) ∈ steps A w ∧ fin A q)"

lemma steps_epsclosure[simp]: "(eps A)* 0 steps A w = steps A w"
  <proof>

lemma in_steps_epsclosure:
  "[| (p,q) : (eps A)*; (q,r) : steps A w |] ==> (p,r) : steps A w"
  <proof>

lemma epsclosure_steps: "steps A w 0 (eps A)* = steps A w"
  <proof>

lemma in_epsclosure_steps:
  "[| (p,q) : steps A w; (q,r) : (eps A)* |] ==> (p,r) : steps A w"
  <proof>

lemma steps_append[simp]: "steps A (v@w) = steps A v 0 steps A w"
  <proof>

lemma in_steps_append[iff]:
  "(p,r) : steps A (v@w) = ((p,r) : (steps A v 0 steps A w))"
  <proof>

end
```

6 Conversions between automata

```
theory Automata
imports DA NAe
begin
```

definition

```
na2da :: "('a,'s)na  $\Rightarrow$  ('a,'s set)da" where
"na2da A = ({start A},  $\lambda a Q. \text{Union}(\text{next } A \ a \ ' \ Q), \lambda Q. \exists q \in Q. \text{fin } A \ q)$ "
```

definition

```
nae2da :: "('a,'s)nae  $\Rightarrow$  ('a,'s set)da" where
"nae2da A = ({start A},
 $\lambda a Q. \text{Union}(\text{next } A \ (\text{Some } a) \ ' \ ((\text{eps } A)^* \ ' \ Q)),$ 
 $\lambda Q. \exists p \in (\text{eps } A)^* \ ' \ Q. \text{fin } A \ p)$ "
```

lemma *DA_delta_is_lift_NA_delta*:

```
" $\bigwedge Q. \text{DA}.\text{delta} \ (\text{na2da } A) \ w \ Q = \text{Union}(\text{NA}.\text{delta} \ A \ w \ ' \ Q)$ "
<proof>
```

lemma *NA_DA_equiv*:

```
"NA.accepts A w = DA.accepts (na2da A) w"
<proof>
```

lemma *espclosure_DA_delta_is_steps*:

```
" $\bigwedge Q. (\text{eps } A)^* \ ' \ (\text{DA}.\text{delta} \ (\text{nae2da } A) \ w \ Q) = \text{steps } A \ w \ ' \ Q$ "
<proof>
```

lemma *NAe_DA_equiv*:

```
"DA.accepts (nae2da A) w = NAe.accepts A w"
<proof>
```

end

7 From regular expressions directly to nondeterministic automata

```
theory RegExp2NA
imports "Regular-Sets.Regular_Exp" NA
begin
```

```
type_synonym 'a bitsNA = "('a, bool list)na"
```

definition

```

"atom"  :: "'a ⇒ 'a bitsNA" where
"atom a = ([True],
           λb s. if s=[True] ∧ b=a then {[False]} else {},
           λs. s=[False])"

```

definition

```

or  :: "'a bitsNA ⇒ 'a bitsNA ⇒ 'a bitsNA" where
"or = (λ(q1,d1,f1)(qr,dr,fr).
      ([],
       λa s. case s of
         [] ⇒ (True ## d1 a q1) ∪ (False ## dr a qr)
       | left#s ⇒ if left then True ## d1 a s
                   else False ## dr a s,
       λs. case s of [] ⇒ (f1 q1 | fr qr)
                   | left#s ⇒ if left then f1 s else fr s)))"

```

definition

```

conc :: "'a bitsNA ⇒ 'a bitsNA ⇒ 'a bitsNA" where
"conc = (λ(q1,d1,f1)(qr,dr,fr).
        (True#q1,
         λa s. case s of
           [] ⇒ {}
         | left#s ⇒ if left then (True ## d1 a s) ∪
                                (if f1 s then False ## dr a qr else
                                 {}))
        else False ## dr a s,
        λs. case s of [] ⇒ False | left#s ⇒ left ∧ f1 s ∧ fr qr | ¬left
        ∧ fr s)))"

```

definition

```

epsilon :: "'a bitsNA" where
"epsilon = ([], λa s. {}, λs. s=[])"

```

definition

```

plus :: "'a bitsNA ⇒ 'a bitsNA" where
"plus = (λ(q,d,f). (q, λa s. d a s ∪ (if f s then d a q else {}), f))"

```

definition

```

star :: "'a bitsNA ⇒ 'a bitsNA" where
"star A = or epsilon (plus A)"

```

primrec rexp2na :: "'a rexp ⇒ 'a bitsNA" where

```

"rexp2na Zero      = ([], λa s. {}, λs. False)" |
"rexp2na One       = epsilon" |
"rexp2na (Atom a)  = atom a" |
"rexp2na (Plus r s) = or (rexp2na r) (rexp2na s)" |
"rexp2na (Times r s) = conc (rexp2na r) (rexp2na s)" |
"rexp2na (Star r)  = star (rexp2na r)"

```

declare *split_paired_all*[simp]

lemma *fin_atom*: "(fin (atom a) q) = (q = [False])"
<proof>

lemma *start_atom*: "start (atom a) = [True]"
<proof>

lemma *in_step_atom_Some*[simp]:
"(p,q) : step (atom a) b = (p=[True] ∧ q=[False] ∧ b=a)"
<proof>

lemma *False_False_in_steps_atom*:
"([False],[False]) : steps (atom a) w = (w = [])"
<proof>

lemma *start_fin_in_steps_atom*:
"(start (atom a), [False]) : steps (atom a) w = (w = [a])"
<proof>

lemma *accepts_atom*:
"accepts (atom a) w = (w = [a])"
<proof>

lemma *fin_or_True*[iff]:
" $\bigwedge L R. \text{fin (or L R) (True\#p)} = \text{fin L p}$ "
<proof>

lemma *fin_or_False*[iff]:
" $\bigwedge L R. \text{fin (or L R) (False\#p)} = \text{fin R p}$ "
<proof>

lemma *True_in_step_or*[iff]:
" $\bigwedge L R. (\text{True\#p},q) : \text{step (or L R) a} = (\exists r. q = \text{True\#r} \wedge (p,r) \in \text{step L a})$ "
<proof>

lemma False_in_step_or[iff]:
 " $\bigwedge L R. (\text{False}\#p,q) : \text{step (or L R) a} = (\exists r. q = \text{False}\#r \wedge (p,r) \in \text{step R a})$ "
 <proof>

lemma lift_True_over_steps_or[iff]:
 " $\bigwedge p. (\text{True}\#p,q) \in \text{steps (or L R) w} = (\exists r. q = \text{True}\#r \wedge (p,r) \in \text{steps L w})$ "
 <proof>

lemma lift_False_over_steps_or[iff]:
 " $\bigwedge p. (\text{False}\#p,q) \in \text{steps (or L R) w} = (\exists r. q = \text{False}\#r \wedge (p,r) \in \text{steps R w})$ "
 <proof>

lemma start_step_or[iff]:
 " $\bigwedge L R. (\text{start (or L R), q} : \text{step (or L R) a} = (\exists p. (q = \text{True}\#p \wedge (\text{start L}, p) : \text{step L a}) \mid (q = \text{False}\#p \wedge (\text{start R}, p) : \text{step R a}))$ "
 <proof>

lemma steps_or:
 " $(\text{start (or L R), q} : \text{steps (or L R) w} = ((w = [] \wedge q = \text{start (or L R)}) \mid (w \neq [] \wedge (\exists p. q = \text{True}\#p \wedge (\text{start L}, p) : \text{steps L w} \mid q = \text{False}\#p \wedge (\text{start R}, p) : \text{steps R w})))$ "
 <proof>

lemma fin_start_or[iff]:
 " $\bigwedge L R. \text{fin (or L R) (start (or L R))} = (\text{fin L (start L)} \mid \text{fin R (start R)})$ "
 <proof>

lemma accepts_or[iff]:
 " $\text{accepts (or L R) w} = (\text{accepts L w} \mid \text{accepts R w})$ "
 <proof>

lemma fin_conc_True[iff]:

" $\bigwedge L R. \text{fin} (\text{conc } L R) (\text{True}\#p) = (\text{fin } L p \wedge \text{fin } R (\text{start } R))$ "
 <proof>

lemma *fin_conc_False*[iff]:
 " $\bigwedge L R. \text{fin} (\text{conc } L R) (\text{False}\#p) = \text{fin } R p$ "
 <proof>

lemma *True_step_conc*[iff]:
 " $\bigwedge L R. (\text{True}\#p,q) : \text{step} (\text{conc } L R) a =$
 $(\exists r. q=\text{True}\#r \wedge (p,r) : \text{step } L a) \mid$
 $(\text{fin } L p \wedge (\exists r. q=\text{False}\#r \wedge (\text{start } R,r) : \text{step } R a))$ "
 <proof>

lemma *False_step_conc*[iff]:
 " $\bigwedge L R. (\text{False}\#p,q) : \text{step} (\text{conc } L R) a =$
 $(\exists r. q = \text{False}\#r \wedge (p,r) : \text{step } R a)$ "
 <proof>

lemma *False_steps_conc*[iff]:
 " $\bigwedge p. (\text{False}\#p,q) : \text{steps} (\text{conc } L R) w = (\exists r. q=\text{False}\#r \wedge (p,r) : \text{steps}$
 $R w)$ "
 <proof>

lemma *True_True_steps_concI*:
 " $\bigwedge L R p. (p,q) : \text{steps } L w \implies (\text{True}\#p,\text{True}\#q) : \text{steps} (\text{conc } L R) w$ "
 <proof>

lemma *True_False_step_conc*[iff]:
 " $\bigwedge L R. (\text{True}\#p,\text{False}\#q) : \text{step} (\text{conc } L R) a =$
 $(\text{fin } L p \wedge (\text{start } R,q) : \text{step } R a)$ "
 <proof>

lemma *True_steps_concD*[rule_format]:
 " $\forall p. (\text{True}\#p,q) : \text{steps} (\text{conc } L R) w \implies$
 $(\exists r. (p,r) : \text{steps } L w \wedge q = \text{True}\#r) \vee$
 $(\exists u a v. w = u@a\#v \wedge$
 $(\exists r. (p,r) : \text{steps } L u \wedge \text{fin } L r \wedge$
 $(\exists s. (\text{start } R,s) : \text{step } R a \wedge$
 $(\exists t. (s,t) : \text{steps } R v \wedge q = \text{False}\#t))))$ "
 <proof>

lemma *True_steps_conc*:

```

"(True#p,q) : steps (conc L R) w =
((∃r. (p,r) : steps L w ∧ q = True#r) ∨
(∃u a v. w = u@a#v ∧
(∃r. (p,r) : steps L u ∧ fin L r ∧
(∃s. (start R,s) : step R a ∧
(∃t. (s,t) : steps R v ∧ q = False#t))))))"
⟨proof⟩

```

```

lemma start_conc:
"∧L R. start(conc L R) = True#start L"
⟨proof⟩

```

```

lemma final_conc:
"∧L R. fin(conc L R) p = ((fin R (start R) ∧ (∃s. p = True#s ∧ fin
L s)) ∨
(∃s. p = False#s ∧ fin R s))"
⟨proof⟩

```

```

lemma accepts_conc:
"accepts (conc L R) w = (∃u v. w = u@v ∧ accepts L u ∧ accepts R v)"
⟨proof⟩

```

```

lemma step_epsilon[simp]: "step epsilon a = {}"
⟨proof⟩

```

```

lemma steps_epsilon: "(p,q) : steps epsilon w = (w=[] ∧ p=q)"
⟨proof⟩

```

```

lemma accepts_epsilon[iff]: "accepts epsilon w = (w = [])"
⟨proof⟩

```

```

lemma start_plus[simp]: "∧A. start (plus A) = start A"
⟨proof⟩

```

```

lemma fin_plus[iff]: "∧A. fin (plus A) = fin A"
⟨proof⟩

```

```

lemma step_plusI:
"∧A. (p,q) : step A a ⇒ (p,q) : step (plus A) a"

```

<proof>

lemma *steps_plusI*: " $\bigwedge p. (p,q) : \text{steps } A \ w \implies (p,q) \in \text{steps } (\text{plus } A) \ w$ "

<proof>

lemma *step_plus_conv[iff]*:

" $\bigwedge A. (p,r) : \text{step } (\text{plus } A) \ a =$
 $((p,r) : \text{step } A \ a \mid \text{fin } A \ p \wedge (\text{start } A, r) : \text{step } A \ a)$ "

<proof>

lemma *fin_steps_plusI*:

" $[\mid (\text{start } A, q) : \text{steps } A \ u; u \neq []; \text{fin } A \ p \mid]$
 $\implies (p,q) : \text{steps } (\text{plus } A) \ u$ "

<proof>

lemma *start_steps_plusD[rule_format]*:

" $\forall r. (\text{start } A, r) \in \text{steps } (\text{plus } A) \ w \longrightarrow$
 $(\exists us \ v. w = \text{concat } us \ @ \ v \wedge$
 $(\forall u \in \text{set } us. \text{accepts } A \ u) \wedge$
 $(\text{start } A, r) \in \text{steps } A \ v)$ "

<proof>

lemma *steps_star_cycle[rule_format]*:

" $us \neq [] \longrightarrow (\forall u \in \text{set } us. \text{accepts } A \ u) \longrightarrow \text{accepts } (\text{plus } A) \ (\text{concat } us)$ "

<proof>

lemma *accepts_plus[iff]*:

" $\text{accepts } (\text{plus } A) \ w =$
 $(\exists us. us \neq [] \wedge w = \text{concat } us \wedge (\forall u \in \text{set } us. \text{accepts } A \ u))$ "

<proof>

lemma *accepts_star*:

" $\text{accepts } (\text{star } A) \ w = (\exists us. (\forall u \in \text{set } us. \text{accepts } A \ u) \wedge w = \text{concat } us)$ "

<proof>

lemma *accepts_rexp2na*:

" $\bigwedge w. \text{accepts } (\text{rexp2na } r) \ w = (w : \text{lang } r)$ "

<proof>

end

8 From regular expressions to nondeterministic automata with epsilon

```
theory RegExp2NAe
imports "Regular-Sets.Regular_Exp" NAe
begin

type_synonym 'a bitsNAe = "('a,bool list)nae"

definition
  epsilon :: "'a bitsNAe" where
  "epsilon = ([], $\lambda$ a s. {},  $\lambda$ s. s=[])"

definition
  "atom" :: "'a  $\Rightarrow$  'a bitsNAe" where
  "atom a = ([True],
              $\lambda$ b s. if s=[True]  $\wedge$  b=Some a then {[False]} else {},
              $\lambda$ s. s=[False])"

definition
  or :: "'a bitsNAe  $\Rightarrow$  'a bitsNAe  $\Rightarrow$  'a bitsNAe" where
  "or = ( $\lambda$ (ql,dl,fl)(qr,dr,fr).
        ([],
          $\lambda$ a s. case s of
           []  $\Rightarrow$  if a=None then {True#ql,False#qr} else {}
         | left#s  $\Rightarrow$  if left then True ## dl a s
                       else False ## dr a s,
          $\lambda$ s. case s of []  $\Rightarrow$  False | left#s  $\Rightarrow$  if left then fl s else fr s))"

definition
  conc :: "'a bitsNAe  $\Rightarrow$  'a bitsNAe  $\Rightarrow$  'a bitsNAe" where
  "conc = ( $\lambda$ (ql,dl,fl)(qr,dr,fr).
          (True#ql,
            $\lambda$ a s. case s of
             []  $\Rightarrow$  {}
           | left#s  $\Rightarrow$  if left then (True ## dl a s)  $\cup$ 
                               (if fl s  $\wedge$  a=None then {False#qr} else
                                {})
           else False ## dr a s,
            $\lambda$ s. case s of []  $\Rightarrow$  False | left#s  $\Rightarrow$   $\neg$ left  $\wedge$  fr s))"

definition
  star :: "'a bitsNAe  $\Rightarrow$  'a bitsNAe" where
  "star = ( $\lambda$ (q,d,f).
          ([],
            $\lambda$ a s. case s of
```

```

    [] ⇒ if a=None then {True#q} else {}
  | left#s ⇒ if left then (True ## d a s) ∪
                (if f s ∧ a=None then {True#q} else
                {})
                else {},
  λs. case s of [] ⇒ True | left#s ⇒ left ∧ f s))"

```

```

primrec rexp2nae :: "'a rexp ⇒ 'a bitsNAe" where
"rexp2nae Zero      = ([], λa s. {}, λs. False)" |
"rexp2nae One      = epsilon" |
"rexp2nae (Atom a) = atom a" |
"rexp2nae (Plus r s) = or (rexp2nae r) (rexp2nae s)" |
"rexp2nae (Times r s) = conc (rexp2nae r) (rexp2nae s)" |
"rexp2nae (Star r) = star (rexp2nae r)"

```

```

declare split_paired_all[simp]

```

```

lemma step_epsilon[simp]: "step epsilon a = {}"
<proof>

```

```

lemma steps_epsilon: "(p,q) : steps epsilon w = (w=[] ∧ p=q)"
<proof>

```

```

lemma accepts_epsilon[simp]: "accepts epsilon w = (w = [])"
<proof>

```

```

lemma fin_atom: "(fin (atom a) q) = (q = [False])"
<proof>

```

```

lemma start_atom: "start (atom a) = [True]"
<proof>

```

```

lemma eps_atom[simp]:
  "eps(atom a) = {}"
<proof>

```

```

lemma in_step_atom_Some[simp]:
  "(p,q) : step (atom a) (Some b) = (p=[True] ∧ q=[False] ∧ b=a)"
<proof>

```

lemma *False_False_in_steps_atom*:
 " $([False],[False]) : \text{steps } (\text{atom } a) \ w = (w = [])$ "
 $\langle \text{proof} \rangle$

lemma *start_fin_in_steps_atom*:
 " $(\text{start } (\text{atom } a), [False]) : \text{steps } (\text{atom } a) \ w = (w = [a])$ "
 $\langle \text{proof} \rangle$

lemma *accepts_atom*: " $\text{accepts } (\text{atom } a) \ w = (w = [a])$ "
 $\langle \text{proof} \rangle$

lemma *fin_or_True[iff]*:
 " $\bigwedge L \ R. \ \text{fin } (\text{or } L \ R) \ (\text{True}\#p) = \text{fin } L \ p$ "
 $\langle \text{proof} \rangle$

lemma *fin_or_False[iff]*:
 " $\bigwedge L \ R. \ \text{fin } (\text{or } L \ R) \ (\text{False}\#p) = \text{fin } R \ p$ "
 $\langle \text{proof} \rangle$

lemma *True_in_step_or[iff]*:
 " $\bigwedge L \ R. \ (\text{True}\#p,q) : \text{step } (\text{or } L \ R) \ a = (\exists r. \ q = \text{True}\#r \wedge (p,r) : \text{step } L \ a)$ "
 $\langle \text{proof} \rangle$

lemma *False_in_step_or[iff]*:
 " $\bigwedge L \ R. \ (\text{False}\#p,q) : \text{step } (\text{or } L \ R) \ a = (\exists r. \ q = \text{False}\#r \wedge (p,r) : \text{step } R \ a)$ "
 $\langle \text{proof} \rangle$

lemma *lemma1a*:
 " $(tp,tq) : (\text{eps}(\text{or } L \ R))^* \implies$
 $(\bigwedge p. \ tp = \text{True}\#p \implies \exists q. \ (p,q) : (\text{eps } L)^* \wedge tq = \text{True}\#q)$ "
 $\langle \text{proof} \rangle$

lemma *lemma1b*:
 " $(tp,tq) : (\text{eps}(\text{or } L \ R))^* \implies$

$(\bigwedge p. tp = \text{False}\#p \implies \exists q. (p,q) : (\text{eps } R)^* \wedge tq = \text{False}\#q)$ "
 <proof>

lemma lemma2a:
 "(p,q) : (eps L)* \implies (True#p, True#q) : (eps(or L R))*"
 <proof>

lemma lemma2b:
 "(p,q) : (eps R)* \implies (False#p, False#q) : (eps(or L R))*"
 <proof>

lemma True_epsclosure_or[iff]:
 "(True#p,q) : (eps(or L R))* = ($\exists r. q = \text{True}\#r \wedge (p,r) : (\text{eps } L)^*$)"
 <proof>

lemma False_epsclosure_or[iff]:
 "(False#p,q) : (eps(or L R))* = ($\exists r. q = \text{False}\#r \wedge (p,r) : (\text{eps } R)^*$)"
 <proof>

lemma lift_True_over_steps_or[iff]:
 " $\bigwedge p. (\text{True}\#p,q) : \text{steps } (\text{or } L \ R) \ w = (\exists r. q = \text{True}\#r \wedge (p,r) : \text{steps } L \ w)$ "
 <proof>

lemma lift_False_over_steps_or[iff]:
 " $\bigwedge p. (\text{False}\#p,q) : \text{steps } (\text{or } L \ R) \ w = (\exists r. q = \text{False}\#r \wedge (p,r) : \text{steps } R \ w)$ "
 <proof>

lemma unfold_rtrancl2:
 " $R^* = \text{Id} \cup (R \ 0 \ R^*)$ "
 <proof>

lemma in_unfold_rtrancl2:
 "(p,q) : R* = (q = p | ($\exists r. (p,r) : R \wedge (r,q) : R^*$))"
 <proof>

lemmas [iff] = in_unfold_rtrancl2[where ?p = "start(or L R)"] for L R

lemma start_eps_or[iff]:
 " $\bigwedge L \ R. (\text{start } (\text{or } L \ R), q) : \text{eps } (\text{or } L \ R) =$
 (q = True#start L | q = False#start R)"
 <proof>

lemma not_start_step_or_Some[iff]:

" $\bigwedge L R. (\text{start}(\text{or } L R), q) \notin \text{step } (\text{or } L R) (\text{Some } a)$ "
<proof>

lemma steps_or:
"(start(or L R), q) : steps (or L R) w =
((w = [] \wedge q = start(or L R)) |
 ($\exists p. q = \text{True} \# p \wedge (\text{start } L, p) : \text{steps } L w$ |
 q = False $\# p \wedge (\text{start } R, p) : \text{steps } R w$))"
<proof>

lemma start_or_not_final[iff]:
" $\bigwedge L R. \neg \text{fin } (\text{or } L R) (\text{start}(\text{or } L R))$ "
<proof>

lemma accepts_or:
"accepts (or L R) w = (accepts L w | accepts R w)"
<proof>

lemma in_conc_True[iff]:
" $\bigwedge L R. \text{fin } (\text{conc } L R) (\text{True}\#p) = \text{False}$ "
<proof>

lemma fin_conc_False[iff]:
" $\bigwedge L R. \text{fin } (\text{conc } L R) (\text{False}\#p) = \text{fin } R p$ "
<proof>

lemma True_step_conc[iff]:
" $\bigwedge L R. (\text{True}\#p, q) : \text{step } (\text{conc } L R) a =$
 ($\exists r. q = \text{True}\#r \wedge (p, r) : \text{step } L a$ |
 (fin L p \wedge a=None \wedge q=False#start R))"
<proof>

lemma False_step_conc[iff]:
" $\bigwedge L R. (\text{False}\#p, q) : \text{step } (\text{conc } L R) a =$
 ($\exists r. q = \text{False}\#r \wedge (p, r) : \text{step } R a$)"
<proof>

lemma lemma1b':

"(tp,tq) : (eps(conc L R))* \implies
 ($\bigwedge p. tp = \text{False}\#p \implies \exists q. (p,q) : (\text{eps } R)^* \wedge tq = \text{False}\#q$)"
 <proof>

lemma lemma2b':
 "(p,q) : (eps R)* \implies (False#p, False#q) : (eps(conc L R))*"
 <proof>

lemma False_epsclosure_conc[iff]:
 "((False # p, q) : (eps (conc L R))*) =
 ($\exists r. q = \text{False}\#r \wedge (p, r) : (\text{eps } R)^*$)"
 <proof>

lemma False_steps_conc[iff]:
 " $\bigwedge p. (\text{False}\#p, q) : \text{steps (conc L R) } w = (\exists r. q = \text{False}\#r \wedge (p, r) : \text{steps } R \ w)$ "
 <proof>

lemma True_True_eps_concI:
 "(p,q) : (eps L)* \implies (True#p, True#q) : (eps(conc L R))*"
 <proof>

lemma True_True_steps_concI:
 " $\bigwedge p. (p, q) : \text{steps L } w \implies (\text{True}\#p, \text{True}\#q) : \text{steps (conc L R) } w$ "
 <proof>

lemma lemma1a':
 "(tp,tq) : (eps(conc L R))* \implies
 ($\bigwedge p. tp = \text{True}\#p \implies$
 ($\exists q. tq = \text{True}\#q \wedge (p,q) : (\text{eps } L)^*$) |
 ($\exists q r. tq = \text{False}\#q \wedge (p,r) : (\text{eps } L)^* \wedge \text{fin L } r \wedge (\text{start } R, q) : (\text{eps } R)^*$))"
 <proof>

lemma lemma2a':
 "(p, q) : (eps L)* \implies (True#p, True#q) : (eps(conc L R))*"
 <proof>

lemma lem:
 " $\bigwedge L R. (p, q) : \text{step R None} \implies (\text{False}\#p, \text{False}\#q) : \text{step (conc L R) None}$ "
 <proof>

lemma lemma2b'':
 "(p,q) : (eps R)* \implies (False#p, False#q) : (eps(conc L R))*"

$\langle proof \rangle$

lemma *True_False_eps_concI*:

" $\bigwedge L R. \text{fin } L \ p \implies (\text{True}\#p, \text{False}\#\text{start } R) : \text{eps}(\text{conc } L \ R)$ "
 $\langle proof \rangle$

lemma *True_epsclosure_conc[iff]*:

" $((\text{True}\#p, q) \in (\text{eps}(\text{conc } L \ R))^*) =$
 $(\exists r. (p, r) \in (\text{eps } L)^* \wedge q = \text{True}\#r) \vee$
 $(\exists r. (p, r) \in (\text{eps } L)^* \wedge \text{fin } L \ r \wedge$
 $(\exists s. (\text{start } R, s) \in (\text{eps } R)^* \wedge q = \text{False}\#s))$ "
 $\langle proof \rangle$

lemma *True_steps_concD[rule_format]*:

" $\forall p. (\text{True}\#p, q) : \text{steps } (\text{conc } L \ R) \ w \longrightarrow$
 $(\exists r. (p, r) : \text{steps } L \ w \wedge q = \text{True}\#r) \vee$
 $(\exists u \ v. w = u@v \wedge (\exists r. (p, r) \in \text{steps } L \ u \wedge \text{fin } L \ r \wedge$
 $(\exists s. (\text{start } R, s) \in \text{steps } R \ v \wedge q = \text{False}\#s)))$ "
 $\langle proof \rangle$

lemma *True_steps_conc*:

" $(\text{True}\#p, q) \in \text{steps } (\text{conc } L \ R) \ w =$
 $(\exists r. (p, r) \in \text{steps } L \ w \wedge q = \text{True}\#r) \mid$
 $(\exists u \ v. w = u@v \wedge (\exists r. (p, r) : \text{steps } L \ u \wedge \text{fin } L \ r \wedge$
 $(\exists s. (\text{start } R, s) : \text{steps } R \ v \wedge q = \text{False}\#s)))$ "
 $\langle proof \rangle$

lemma *start_conc*:

" $\bigwedge L R. \text{start}(\text{conc } L \ R) = \text{True}\#\text{start } L$ "
 $\langle proof \rangle$

lemma *final_conc*:

" $\bigwedge L R. \text{fin}(\text{conc } L \ R) \ p = (\exists s. p = \text{False}\#s \wedge \text{fin } R \ s)$ "
 $\langle proof \rangle$

lemma *accepts_conc*:

" $\text{accepts } (\text{conc } L \ R) \ w = (\exists u \ v. w = u@v \wedge \text{accepts } L \ u \wedge \text{accepts } R \ v)$ "
 $\langle proof \rangle$

lemma *True_in_eps_star[iff]*:

" $\bigwedge A. (\text{True}\#p, q) \in \text{eps}(\text{star } A) =$

($\exists r. q = \text{True}\#r \wedge (p,r) \in \text{eps } A$) \vee ($\text{fin } A \ p \wedge q = \text{True}\#\text{start } A$))"

<proof>

lemma True_True_step_starI:
 $\wedge A. (p,q) : \text{step } A \ a \implies (\text{True}\#p, \text{True}\#q) : \text{step } (\text{star } A) \ a$ "

<proof>

lemma True_True_eps_starI:
 $(p,r) : (\text{eps } A)^* \implies (\text{True}\#p, \text{True}\#r) : (\text{eps}(\text{star } A))^*$ "

<proof>

lemma True_start_eps_starI:
 $\wedge A. \text{fin } A \ p \implies (\text{True}\#p, \text{True}\#\text{start } A) : \text{eps}(\text{star } A)$ "

<proof>

lemma lem':
 $(tp,s) : (\text{eps}(\text{star } A))^* \implies (\forall p. tp = \text{True}\#p \longrightarrow$
 $(\exists r. ((p,r) \in (\text{eps } A)^* \vee$
 $(\exists q. (p,q) \in (\text{eps } A)^* \wedge \text{fin } A \ q \wedge (\text{start } A,r) : (\text{eps } A)^*)) \wedge$
 $s = \text{True}\#r))$ "

<proof>

lemma True_eps_star[iff]:
 $((\text{True}\#p,s) \in (\text{eps}(\text{star } A))^*) =$
 $(\exists r. ((p,r) \in (\text{eps } A)^* \vee$
 $(\exists q. (p,q) : (\text{eps } A)^* \wedge \text{fin } A \ q \wedge (\text{start } A,r) : (\text{eps } A)^*)) \wedge$
 $s = \text{True}\#r)$ "

<proof>

lemma True_step_star[iff]:
 $\wedge A. (\text{True}\#p,r) \in \text{step } (\text{star } A) \ (\text{Some } a) =$
 $(\exists q. (p,q) \in \text{step } A \ (\text{Some } a) \wedge r = \text{True}\#q)$ "

<proof>

lemma True_start_steps_starD[rule_format]:
 $\forall rr. (\text{True}\#\text{start } A, rr) \in \text{steps } (\text{star } A) \ w \longrightarrow$
 $(\exists us \ v. w = \text{concat } us \ @ \ v \wedge$
 $(\forall u \in \text{set } us. \text{accepts } A \ u) \wedge$
 $(\exists r. (\text{start } A,r) \in \text{steps } A \ v \wedge rr = \text{True}\#r))$ "

<proof>

```

lemma True_True_steps_starI:
  " $\bigwedge p. (p,q) : \text{steps } A \ w \implies (\text{True}\#p, \text{True}\#q) : \text{steps } (\text{star } A) \ w$ "
  <proof>

lemma steps_star_cycle:
  " $(\forall u \in \text{set } us. \text{accepts } A \ u) \implies$ 
   $(\text{True}\#\text{start } A, \text{True}\#\text{start } A) \in \text{steps } (\text{star } A) \ (\text{concat } us)$ "
  <proof>

lemma True_start_steps_star:
  " $(\text{True}\#\text{start } A, rr) : \text{steps } (\text{star } A) \ w =$ 
   $(\exists us \ v. w = \text{concat } us \ @ \ v \ \wedge$ 
   $(\forall u \in \text{set } us. \text{accepts } A \ u) \ \wedge$ 
   $(\exists r. (\text{start } A, r) \in \text{steps } A \ v \ \wedge \ rr = \text{True}\#r))$ "
  <proof>

lemma start_step_star[iff]:
  " $\bigwedge A. (\text{start}(\text{star } A), r) : \text{step } (\text{star } A) \ a = (a = \text{None} \ \wedge \ r = \text{True}\#\text{start } A)$ "
  <proof>

lemmas epsclosure_start_step_star =
  in_unfold_rtrancl2[where ?p = "start (star A)"] for A

lemma start_steps_star:
  " $(\text{start}(\text{star } A), r) : \text{steps } (\text{star } A) \ w =$ 
   $((w = [] \ \wedge \ r = \text{start}(\text{star } A)) \ | \ (\text{True}\#\text{start } A, r) : \text{steps } (\text{star } A) \ w)$ "
  <proof>

lemma fin_star_True[iff]: " $\bigwedge A. \text{fin } (\text{star } A) \ (\text{True}\#p) = \text{fin } A \ p$ "
  <proof>

lemma fin_star_start[iff]: " $\bigwedge A. \text{fin } (\text{star } A) \ (\text{start}(\text{star } A))$ "
  <proof>

lemma accepts_star:
  " $\text{accepts } (\text{star } A) \ w =$ 
   $(\exists us. (\forall u \in \text{set}(us). \text{accepts } A \ u) \ \wedge \ (w = \text{concat } us))$ "
  <proof>

lemma accepts_rexp2nae:
  " $\bigwedge w. \text{accepts } (\text{rexp2nae } r) \ w = (w : \text{lang } r)$ "

```

<proof>

end

9 Combining automata and regular expressions

```
theory AutoRegExp
imports Automata RegExp2NA RegExp2NAe
begin
```

```
theorem "DA.accepts (na2da(rexp2na r)) w = (w : lang r)"
<proof>
```

```
theorem "DA.accepts (nae2da(rexp2nae r)) w = (w : lang r)"
<proof>
```

end

10 Maximal prefix

```
theory MaxPrefix
imports "HOL-Library.Sublist"
begin
```

```
definition
  is_maxpref :: "('a list  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  'a list  $\Rightarrow$  bool" where
  "is_maxpref P xs ys =
    (prefix xs ys  $\wedge$  (xs=[]  $\vee$  P xs)  $\wedge$  ( $\forall$ zs. prefix zs ys  $\wedge$  P zs  $\longrightarrow$  prefix
    zs xs))"
```

```
type_synonym 'a splitter = "'a list  $\Rightarrow$  'a list * 'a list"
```

```
definition
  is_maxsplitter :: "('a list  $\Rightarrow$  bool)  $\Rightarrow$  'a splitter  $\Rightarrow$  bool" where
  "is_maxsplitter P f =
    ( $\forall$ xs ps qs. f xs = (ps,qs) = (xs=ps@qs  $\wedge$  is_maxpref P ps xs))"
```

```
fun maxsplit :: "('a list  $\Rightarrow$  bool)  $\Rightarrow$  'a list * 'a list  $\Rightarrow$  'a list  $\Rightarrow$ 
'a splitter" where
  "maxsplit P res ps [] = (if P ps then (ps,[]) else res)" |
  "maxsplit P res ps (q#qs) = maxsplit P (if P ps then (ps,q#qs) else res)
    (ps@[q]) qs"
```

```
declare if_split[split del]
```

```
lemma maxsplit_lemma: "(maxsplit P res ps qs = (xs,ys)) =
  (if  $\exists$ us. prefix us qs  $\wedge$  P(ps@us) then xs@ys=ps@qs  $\wedge$  is_maxpref P xs
  (ps@qs))"
```

```

    else (xs,ys)=res)"
⟨proof⟩

declare if_split[split]

lemma is_maxpref_Nil[simp]:
  "¬(∃ us. prefix us xs ∧ P us) ⇒ is_maxpref P ps xs = (ps = [])"
  ⟨proof⟩

lemma is_maxsplitter_maxsplit:
  "is_maxsplitter P (λxs. maxsplit P ([],xs) [] xs)"
  ⟨proof⟩

lemmas maxsplit_eq = is_maxsplitter_maxsplit[simplified is_maxsplitter_def]

end

```

11 Generic scanner

```

theory MaxChop
imports MaxPrefix
begin

type_synonym 'a chopper = "'a list ⇒ 'a list list * 'a list"

definition
  is_maxchopper :: "('a list ⇒ bool) ⇒ 'a chopper ⇒ bool" where
  "is_maxchopper P chopper =
    (∀ xs zs yss.
      (chopper(xs) = (yss,zs)) =
      (xs = concat yss @ zs ∧ (∀ ys ∈ set yss. ys ≠ []) ∧
      (case yss of
        [] ⇒ is_maxpref P [] xs
        | us#uss ⇒ is_maxpref P us xs ∧ chopper(concat(uss)@zs) = (uss,zs)))))"

definition
  reducing :: "'a splitter ⇒ bool" where
  "reducing splitf =
    (∀ xs ys zs. splitf xs = (ys,zs) ∧ ys ≠ [] → length zs < length xs)"

function chop :: "'a splitter ⇒ 'a list ⇒ 'a list list × 'a list" where
  [simp del]: "chop splitf xs = (if reducing splitf
    then let pp = splitf xs
      in if fst pp = [] then ([], xs)
      else let qq = chop splitf (snd pp)
        in (fst pp # fst qq, snd qq)
    else undefined)"

⟨proof⟩

```

```

termination ⟨proof⟩

lemma chop_rule: "reducing splitf  $\implies$ 
  chop splitf xs = (let (pre, post) = splitf xs
    in if pre = [] then ([], xs)
    else let (xss, zs) = chop splitf post
    in (pre # xss,zs))"
⟨proof⟩

lemma reducing_maxsplit: "reducing( $\lambda$ qs. maxsplit P ([],qs) [] qs)"
⟨proof⟩

lemma is_maxsplitter_reducing:
  "is_maxsplitter P splitf  $\implies$  reducing splitf"
⟨proof⟩

lemma chop_concat[rule_format]: "is_maxsplitter P splitf  $\implies$ 
  ( $\forall$ yss zs. chop splitf xs = (yss,zs)  $\longrightarrow$  xs = concat yss @ zs)"
⟨proof⟩

lemma chop_nonempty: "is_maxsplitter P splitf  $\implies$ 
   $\forall$ yss zs. chop splitf xs = (yss,zs)  $\longrightarrow$  ( $\forall$ ys  $\in$  set yss. ys  $\neq$  [])"
⟨proof⟩

lemma is_maxchopper_chop:
  assumes prem: "is_maxsplitter P splitf" shows "is_maxchopper P (chop
  splitf)"
⟨proof⟩

end

```

12 Automata based scanner

```

theory AutoMaxChop
imports DA MaxChop
begin

primrec auto_split :: "('a,'s)da  $\Rightarrow$  's  $\Rightarrow$  'a list * 'a list  $\Rightarrow$  'a list
 $\Rightarrow$  'a splitter" where
  "auto_split A q res ps [] = (if fin A q then (ps,[]) else res)" |
  "auto_split A q res ps (x#xs) =
    auto_split A (next A x q) (if fin A q then (ps,x#xs) else res) (ps@[x])
  xs"

definition
  auto_chop :: "('a,'s)da  $\Rightarrow$  'a chopper" where
  "auto_chop A = chop ( $\lambda$ xs. auto_split A (start A) ([],xs) [] xs)"

```

```

lemma delta_snoc: "delta A (xs@[y]) q = next A y (delta A xs q)"
<proof>

lemma auto_split_lemma:
  "\q ps res. auto_split A (delta A ps q) res ps xs =
    maxsplit (\ys. fin A (delta A ys q)) res ps xs"
<proof>

lemma auto_split_is_maxsplit:
  "auto_split A (start A) res [] xs = maxsplit (accepts A) res [] xs"
<proof>

lemma is_maxsplitter_auto_split:
  "is_maxsplitter (accepts A) (\xs. auto_split A (start A) ([],xs) [] xs)"
<proof>

lemma is_maxchopper_auto_chop:
  "is_maxchopper (accepts A) (auto_chop A)"
<proof>

end

```

13 From deterministic automata to regular sets

```

theory RegSet_of_nat_DA
imports "Regular-Sets.Regular_Set" DA
begin

type_synonym 'a nat_next = "'a  $\Rightarrow$  nat  $\Rightarrow$  nat"

abbreviation
  deltas :: "'a nat_next  $\Rightarrow$  'a list  $\Rightarrow$  nat  $\Rightarrow$  nat" where
  "deltas  $\equiv$  foldl2"

primrec trace :: "'a nat_next  $\Rightarrow$  nat  $\Rightarrow$  'a list  $\Rightarrow$  nat list" where
  "trace d i [] = []" |
  "trace d i (x#xs) = d x i # trace d (d x i) xs"

primrec regset :: "'a nat_next  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a list set" where
  "regset d i j 0 = (if i=j then insert [] {[a] | a. d a i = j}
    else {[a] | a. d a i = j})" |
  "regset d i j (Suc k) =
    regset d i j k  $\cup$ 
    (regset d i k k) @@ (star(regset d k k k)) @@ (regset d k j k)"

definition

```


`regset_of_DA :: "('a,nat)da ⇒ nat ⇒ 'a list set" where`
`"regset_of_DA A k = (⋃j∈{j. j<k ∧ fin A j}. regset (next A) (start A) j k)"`

definition

`bounded :: "'a nat_next ⇒ nat ⇒ bool" where`
`"bounded d k = (∀n. n < k → (∀x. d x n < k))"`

declare

`in_set_butlast_appendI[simp,intro] less_SucI[simp] image_eqI[simp]`

lemma butlast_empty[iff]:

`"(butlast xs = []) = (case xs of [] ⇒ True | y#ys ⇒ ys=[])"`
`<proof>`

lemma in_set_butlast_concatI:

`"x:set(butlast xs) ⇒ xs:set xss ⇒ x:set(butlast(concat xss))"`
`<proof>`

lemma decompose[rule_format]:

`"∀i. k ∈ set(trace d i xs) → (∃pref mids suf.`
`xs = pref @ concat mids @ suf ∧`
`deltas d pref i = k ∧ (∀n∈set(butlast(trace d i pref)). n ≠ k) ∧`
`(∀mid∈set mids. (deltas d mid k = k) ∧`
`(∀n∈set(butlast(trace d k mid)). n ≠ k)) ∧`
`(∀n∈set(butlast(trace d k suf)). n ≠ k))"`
`<proof>`

lemma length_trace[simp]: `"∧i. length(trace d i xs) = length xs"`

`<proof>`

lemma deltas_append[simp]:

`"∧i. deltas d (xs@ys) i = deltas d ys (deltas d xs i)"`
`<proof>`

lemma trace_append[simp]:

`"∧i. trace d i (xs@ys) = trace d i xs @ trace d (deltas d xs i) ys"`
`<proof>`

lemma trace_concat[simp]:

`"(∀xs ∈ set xss. deltas d xs i = i) ⇒`
`trace d i (concat xss) = concat (map (trace d i) xss)"`
`<proof>`

```

lemma trace_is_Nil[simp]: " $\bigwedge i. (\text{trace } d \ i \ xs = []) = (xs = [])$ "
<proof>

lemma trace_is_Cons_conv[simp]:
  " $(\text{trace } d \ i \ xs = n\#ns) =$ 
   $(\text{case } xs \text{ of } [] \Rightarrow \text{False} \mid y\#ys \Rightarrow n = d \ y \ i \wedge ns = \text{trace } d \ n \ ys)$ "
<proof>

lemma set_trace_conv:
  " $\bigwedge i. \text{set}(\text{trace } d \ i \ xs) =$ 
   $(\text{if } xs=[] \text{ then } \{\} \text{ else } \text{insert}(\text{deltas } d \ xs \ i)(\text{set}(\text{butlast}(\text{trace } d \ i \ xs))))$ "
<proof>

lemma deltas_concat[simp]:
  " $(\forall mid \in \text{set } mids. \text{deltas } d \ mid \ k = k) \implies \text{deltas } d \ (\text{concat } mids) \ k =$ 
   $k$ "
<proof>

lemma lem: " $[| \ n < \text{Suc } k; \ n \neq k \ |] \implies \ n < k$ "
<proof>

lemma regset_spec:
  " $\bigwedge i \ j \ xs. xs \in \text{regset } d \ i \ j \ k =$ 
   $(\forall n \in \text{set}(\text{butlast}(\text{trace } d \ i \ xs)). \ n < k) \wedge \text{deltas } d \ xs \ i = j$ "
<proof>

lemma trace_below:
  " $\text{bounded } d \ k \implies \forall i. \ i < k \longrightarrow (\forall n \in \text{set}(\text{trace } d \ i \ xs). \ n < k)$ "
<proof>

lemma regset_below:
  " $[| \ \text{bounded } d \ k; \ i < k; \ j < k \ |] \implies$ 
   $\text{regset } d \ i \ j \ k = \{xs. \ \text{deltas } d \ xs \ i = j\}$ "
<proof>

lemma deltas_below:
  " $\bigwedge i. \ \text{bounded } d \ k \implies \ i < k \implies \ \text{deltas } d \ w \ i < k$ "
<proof>

lemma regset_DA_equiv:
  " $[| \ \text{bounded } (\text{next } A) \ k; \ \text{start } A < k; \ j < k \ |] \implies$ 
   $w : \text{regset\_of\_DA } A \ k = \text{accepts } A \ w$ "
<proof>

end

```

14 Executing Automata and membership of Regular Expressions

```
theory Execute
imports AutoRegExp
begin
```

14.1 Example

```
definition example_expression
where
  "example_expression = (let r0 = Atom (0::nat); r1 = Atom (1::nat)
    in Times (Star (Plus (Times r1 r1) r0)) (Star (Plus (Times r0 r0)
    r1)))"

value "NA.accepts (rexp2na example_expression) [0,1,1,0,0,1]"

value "DA.accepts (na2da (rexp2na example_expression)) [0,1,1,0,0,1]"

end
theory Functional_Automata
imports AutoRegExp AutoMaxChop RegSet_of_nat_DA Execute
begin

end
```

References

- [1] T. Nipkow. Verified lexical analysis. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, volume 1479, pages 1–15, 1998. <http://www4.informatik.tu-muenchen.de/~nipkow/pubs/tphols98.html>.