

# Functional Automata

Tobias Nipkow

December 12, 2009

## 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`
- Regular expressions and their conversion to automata: `RegSet`, `RegExp`, `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

types ('a,'s)da = "'s * ('a => 's => 's) * ('s => bool)"

definition
  foldl2 :: "('a => 'b => 'b) => 'a list => 'b => 'b" where
"foldl2 f xs a = foldl (%a 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 = (%w. 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

```

```

types ('a,'s)na = "'s * ('a => 's => 's set) * ('s => bool)"

```

```

consts delta :: "('a,'s)na => 'a list => 's => 's set"

```

```

primrec

```

```

"delta A [] p = {p}"

```

```

"delta A (a#w) p = Union(delta A w ' next A a p)"

```

```

definition

```

```

  accepts :: "('a,'s)na => 'a list => bool" where

```

```

"accepts A w = (EX q : delta A w (start A). fin A q)"

```

```

definition

```

```

  step :: "('a,'s)na => 'a => ('s * 's)set" where

```

```

"step A a = {(p,q) . q : next A a p}"

```

```

consts steps :: "('a,'s)na => 'a list => ('s * 's)set"

```

```

primrec

```

```

"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: "!!p. delta A w p = {q. (p,q) : steps A w}"

```

```

<proof>

```

```

lemma accepts_conv_steps:

```

```

  "accepts A w = (? q. (start A,q) : steps A w & fin A q)"

```

```

<proof>

```

```

end

```

## 5 Nondeterministic automata with epsilon transitions

```
theory NAε
imports NA
begin

types ('a,'s)naε = "('a option,'s)na"

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

consts steps :: "('a,'s)naε => 'a list => ('s * 's)set"
primrec
  "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 => ('a,'s set)da" where
"na2da A = ({start A}, %a Q. Union(next A a ' Q), %Q. ? q:Q. fin A q)"
```

**definition**

```
nae2da :: "('a,'s)nae => ('a,'s set)da" where
"nae2da A = ({start A},
             %a Q. Union(next A (Some a) ' ((eps A)^* ' Q)),
             %Q. ? p: (eps A)^* ' Q. fin A p)"
```

**lemma** *DA\_delta\_is\_lift\_NA\_delta*:

```
"!!Q. DA.delta (na2da A) w Q = Union(NA.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*:

```
"!!Q. (eps A)^* ' (DA.delta (nae2da A) w Q) = steps A w ' Q"
⟨proof⟩
```

**lemma** *NAe\_DA\_equiv*:

```
"DA.accepts (nae2da A) w = NAe.accepts A w"
⟨proof⟩
```

**end**

## 7 Regular sets

```
theory RegSet
imports Main
begin
```

**definition**

```
conc :: "'a list set => 'a list set => 'a list set" where
"conc A B = {xs@ys | xs ys. xs:A & ys:B}"
```

```

inductive_set
  star :: "'a list set => 'a list set"
  for A :: "'a list set"
where
  NilI[iff]:   "[] : star A"
  | ConsI[intro,simp]: "[| a:A; as : star A |] ==> a@as : star A"

lemma concat_in_star: "!xs: set xss. xs:S ==> concat xss : star S"
<proof>

lemma in_star:
  "w : star U = (? us. (!u : set(us). u : U) & (w = concat us))"
<proof>

end

```

## 8 Regular expressions

```

theory RegExp
imports RegSet
begin

datatype 'a rexp = Empty
              | Atom 'a
              | Or  "('a rexp)" "('a rexp)"
              | Conc "('a rexp)" "('a rexp)"
              | Star "('a rexp)"

consts lang :: "'a rexp => 'a list set"
primrec
  "lang Empty = {}"
  "lang (Atom a) = {[a]}"
  "lang (Or e1 er) = (lang e1) Un (lang er)"
  "lang (Conc e1 er) = RegSet.concat (lang e1) (lang er)"
  "lang (Star e) = RegSet.star(lang e)"

end

```

## 9 From regular expressions directly to nondeterministic automata

```

theory RegExp2NA
imports RegExp NA
begin

types 'a bitsNA = "('a,bool list)na"

```

### abbreviation

```
Cons_syn :: "'a => 'a list set => 'a list set" (infixr "##" 65) where
"x ## S == Cons x ' S"
```

### definition

```
"atom" :: "'a => 'a bitsNA" where
"atom a = ([True],
           %b s. if s=[True] & b=a then {[False]} else {},
           %s. s=[False])"
```

### definition

```
or :: "'a bitsNA => 'a bitsNA => 'a bitsNA" where
"or = (%(ql,dl,fl)(qr,dr,fr).
      ([],
       %a s. case s of
         [] => (True ## dl a ql) Un (False ## dr a qr)
       | left#s => if left then True ## dl a s
                   else False ## dr a s,
       %s. case s of [] => (fl ql | fr qr)
                   | left#s => if left then fl s else fr s)))"
```

### definition

```
conc :: "'a bitsNA => 'a bitsNA => 'a bitsNA" where
"conc = (%(ql,dl,fl)(qr,dr,fr).
        (True#ql,
         %a s. case s of
           [] => {}
         | left#s => if left then (True ## dl a s) Un
                               (if fl s then False ## dr a qr else
                                {}))
        else False ## dr a s,
        %s. case s of [] => False | left#s => left & fl s & fr qr | ~left
        & fr s))"
```

### definition

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

### definition

```
plus :: "'a bitsNA => 'a bitsNA" where
"plus = (%(q,d,f). (q, %a s. d a s Un (if f s then d a q else {}), f))"
```

### definition

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

```
consts rexp2na :: "'a rexp => 'a bitsNA"
primrec
```

```

"rexp2na Empty      = ([], %a s. {}, %s. False)"
"rexp2na(Atom a)    = atom a"
"rexp2na(Or r s)    = or  (rexp2na r) (rexp2na s)"
"rexp2na(Conc 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]:
  "!!L R. fin (or L R) (True#p) = fin L p"
<proof>

```

```

lemma fin_or_False[iff]:
  "!!L R. fin (or L R) (False#p) = fin R p"
<proof>

```

```

lemma True_in_step_or[iff]:
  "!!L R. (True#p,q) : step (or L R) a = (? r. q = True#r & (p,r) : step
  L a)"
  <proof>

lemma False_in_step_or[iff]:
  "!!L R. (False#p,q) : step (or L R) a = (? r. q = False#r & (p,r) : step
  R a)"
  <proof>

lemma lift_True_over_steps_or[iff]:
  "!!p. (True#p,q):steps (or L R) w = (? r. q = True # r & (p,r):steps
  L w)"
  <proof>

lemma lift_False_over_steps_or[iff]:
  "!!p. (False#p,q):steps (or L R) w = (? r. q = False#r & (p,r):steps
  R w)"
  <proof>

lemma start_step_or[iff]:
  "!!L R. (start(or L R),q) : step(or L R) a =
  (? p. (q = True#p & (start L,p) : step L a) |
  (q = False#p & (start R,p) : step R a))"
  <proof>

lemma steps_or:
  "(start(or L R), q) : steps (or L R) w =
  ( (w = [] & q = start(or L R)) |
  (w ~= [] & (? p. q = True # p & (start L,p) : steps L w |
  q = False # p & (start R,p) : steps R w)))"
  <proof>

lemma fin_start_or[iff]:
  "!!L R. fin (or L R) (start(or L R)) = (fin L (start L) | fin R (start
  R))"
  <proof>

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

```

```

lemma fin_conc_True[iff]:
  "!!L R. fin (conc L R) (True#p) = (fin L p & fin R (start R))"
  <proof>

lemma fin_conc_False[iff]:
  "!!L R. fin (conc L R) (False#p) = fin R p"
  <proof>

lemma True_step_conc[iff]:
  "!!L R. (True#p,q) : step (conc L R) a =
    ((? r. q=True#r & (p,r): step L a) |
     (fin L p & (? r. q=False#r & (start R,r) : step R a)))"
  <proof>

lemma False_step_conc[iff]:
  "!!L R. (False#p,q) : step (conc L R) a =
    (? r. q = False#r & (p,r) : step R a)"
  <proof>

lemma False_steps_conc[iff]:
  "!!p. (False#p,q): steps (conc L R) w = (? r. q=False#r & (p,r): steps
  R w)"
  <proof>

lemma True_True_steps_concI:
  "!!L R p. (p,q) : steps L w ==> (True#p,True#q) : steps (conc L R) w"
  <proof>

lemma True_False_step_conc[iff]:
  "!!L R. (True#p,False#q) : step (conc L R) a =
    (fin L p & (start R,q) : step R a)"
  <proof>

lemma True_steps_concD[rule_format]:
  "!!p. (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 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: "!!p. (p,q) : steps A w ==> (p,q) : steps (plus A)
w"
<proof>

lemma step_plus_conv[iff]:
  "!!A. (p,r): step (plus A) a =
    ( (p,r): step A a | fin A p & (start A,r) : step A a )"
<proof>

lemma fin_steps_plusI:
  "[| (start A,q) : steps A u; u ~= []; fin A p |]
  ==> (p,q) : steps (plus A) u"
<proof>

lemma start_steps_plusD[rule_format]:
  "!r. (start A,r) : steps (plus A) w -->
    (? us v. w = concat us @ v &
      (!u:set us. accepts A u) &
      (start A,r) : steps A v)"
<proof>

lemma steps_star_cycle[rule_format]:
  "us ~= [] --> (!u : set us. accepts A u) --> accepts (plus A) (concat
us)"
<proof>

lemma accepts_plus[iff]:
  "accepts (plus A) w =
    (? us. us ~= [] & w = concat us & (!u : set us. accepts A u))"
<proof>

lemma accepts_star:
  "accepts (star A) w = (? us. (!u : set us. accepts A u) & w = concat
us)"
<proof>

```

```

lemma accepts_rexp2na:
  "!!w. accepts (rexp2na r) w = (w : lang r)"
  <proof>

end

```

## 10 From regular expressions to nondeterministic automata with epsilon

```

theory RegExp2NAe
imports RegExp NAe
begin

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

abbreviation
  Cons_syn :: "'a => 'a list set => 'a list set" (infixr "##" 65) where
  "x ## S == Cons x ' S"

definition
  "atom" :: "'a => 'a bitsNAe" where
  "atom a = ([True],
             %b s. if s=[True] & b=Some a then {[False]} else {},
             %s. s=[False])"

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

definition
  conc :: "'a bitsNAe => 'a bitsNAe => 'a bitsNAe" where
  "conc = (%(ql,dl,fl)(qr,dr,fr).
          (True#ql,
           %a s. case s of
             [] => {}
           | left#s => if left then (True ## dl a s) Un
                       (if fl s & a=None then {False#qr} else
              {}))
          else False ## dr a s,

```

```

    %s. case s of [] => False | left#s => ~left & fr s))"

definition
  star :: "'a bitsNAe => 'a bitsNAe" where
"star = (%(q,d,f).
  ([],
  %a s. case s of
    [] => if a=None then {True#q} else {}
    | left#s => if left then (True ## d a s) Un
                      (if f s & a=None then {True#q} else
{}))
    else {},
  %s. case s of [] => True | left#s => left & f s))"

consts rexp2nae :: "'a rexp => 'a bitsNAe"
primrec
"rexp2nae Empty      = ([], %a s. {}, %s. False)"
"rexp2nae(Atom a)    = atom a"
"rexp2nae(Or r s)    = or   (rexp2nae r) (rexp2nae s)"
"rexp2nae(Conc r s)  = conc (rexp2nae r) (rexp2nae s)"
"rexp2nae(Star r)    = star (rexp2nae 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 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]) : 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]:  
"!!L R. fin (or L R) (True#p) = fin L p"  
<proof>

**lemma** fin\_or\_False[iff]:  
"!!L R. fin (or L R) (False#p) = fin R p"  
<proof>

**lemma** True\_in\_step\_or[iff]:  
"!!L R. (True#p,q) : step (or L R) a = (? r. q = True#r & (p,r) : step L a)"  
<proof>

**lemma** False\_in\_step\_or[iff]:  
"!!L R. (False#p,q) : step (or L R) a = (? r. q = False#r & (p,r) : step R a)"  
<proof>

**lemma** lemma1a:  
"(tp,tq) : (eps(or L R))^\* ==>  
(!!p. tp = True#p ==> ? q. (p,q) : (eps L)^\* & tq = True#q)"  
<proof>

**lemma** lemma1b:  
"(tp,tq) : (eps(or L R))^\* ==>  
(!!p. tp = False#p ==> ? q. (p,q) : (eps R)^\* & tq = False#q)"  
<proof>

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

```

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

lemma True_epsclosure_or[iff]:
  "(True#p,q) : (eps(or L R))^* = (? r. q = True#r & (p,r) : (eps L)^*)"
  <proof>

lemma False_epsclosure_or[iff]:
  "(False#p,q) : (eps(or L R))^* = (? r. q = False#r & (p,r) : (eps R)^*)"
  <proof>

lemma lift_True_over_steps_or[iff]:
  "!!p. (True#p,q):steps (or L R) w = (? r. q = True # r & (p,r):steps
  L w)"
  <proof>

lemma lift_False_over_steps_or[iff]:
  "!!p. (False#p,q):steps (or L R) w = (? r. q = False#r & (p,r):steps
  R w)"
  <proof>

lemma unfold_rtrancl2:
  "R^* = Id Un (R O R^*)"
  <proof>

lemma in_unfold_rtrancl2:
  "(p,q) : R^* = (q = p | (? r. (p,r) : R & (r,q) : R^*))"
  <proof>

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

lemma start_eps_or[iff]:
  "!!L R. (start(or L R),q) : eps(or L R) =
  (q = True#start L | q = False#start R)"
  <proof>

lemma not_start_step_or_Some[iff]:
  "!!L R. (start(or L R),q) ~: step (or L R) (Some a)"
  <proof>

lemma steps_or:
  "(start(or L R), q) : steps (or L R) w =
  ( (w = [] & q = start(or L R)) |

```

```

      (? p. q = True # p & (start L,p) : steps L w |
         q = False # p & (start R,p) : steps R w )"
⟨proof⟩

```

```

lemma start_or_not_final[iff]:
  "!!L R. ~ fin (or L R) (start(or L R))"
⟨proof⟩

```

```

lemma accepts_or:
  "accepts (or L R) w = (accepts L w | accepts R w)"
⟨proof⟩

```

```

lemma in_conc_True[iff]:
  "!!L R. fin (conc L R) (True#p) = False"
⟨proof⟩

```

```

lemma fin_conc_False[iff]:
  "!!L R. fin (conc L R) (False#p) = fin R p"
⟨proof⟩

```

```

lemma True_step_conc[iff]:
  "!!L R. (True#p,q) : step (conc L R) a =
    ((? r. q=True#r & (p,r): step L a) |
     (fin L p & a=None & q=False#start R))"
⟨proof⟩

```

```

lemma False_step_conc[iff]:
  "!!L R. (False#p,q) : step (conc L R) a =
    (? r. q = False#r & (p,r) : step R a)"
⟨proof⟩

```

```

lemma lemma1b':
  "(tp,tq) : (eps(conc L R))^* ==>
   (!!p. tp = False#p ==> ? q. (p,q) : (eps R)^* & tq = False#q)"
⟨proof⟩

```

```

lemma lemma2b':
  "(p,q) : (eps R)^* ==> (False#p, False#q) : (eps(conc L R))^*"

```

*<proof>*

**lemma** *False\_epsclosure\_conc[iff]*:  
"((False # p, q) : (eps (conc L R))^\*) =  
(? r. q = False # r & (p, r) : (eps R)^\*)" *<proof>*

**lemma** *False\_steps\_conc[iff]*:  
"!!p. (False#p,q): steps (conc L R) w = (? r. q=False#r & (p,r): steps  
R w)" *<proof>*

**lemma** *True\_True\_eps\_concI*:  
"(p,q): (eps L)^\* ==> (True#p,True#q) : (eps(conc L R))^\*" *<proof>*

**lemma** *True\_True\_steps\_concI*:  
"!!p. (p,q) : steps L w ==> (True#p,True#q) : steps (conc L R) w" *<proof>*

**lemma** *lemma1a'*:  
"(tp,tq) : (eps(conc L R))^\* ==>  
(!!p. tp = True#p ==>  
(? q. tq = True#q & (p,q) : (eps L)^\*) |  
(? q r. tq = False#q & (p,r):(eps L)^\* & fin L r & (start R,q) : (eps  
R)^\*))" *<proof>*

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

**lemma** *lem*:  
"!!L R. (p,q) : step R None ==> (False#p, False#q) : step (conc L R)  
None" *<proof>*

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

**lemma** *True\_False\_eps\_concI*:  
"!!L R. fin L p ==> (True#p, False#start R) : eps(conc L R)" *<proof>*

```

lemma True_epsclosure_conc[iff]:
  "((True#p,q) : (eps(conc L R))^*) =
  ((? r. (p,r) : (eps L)^* & q = True#r) |
  (? r. (p,r) : (eps L)^* & fin L r &
  (? s. (start R, s) : (eps R)^* & q = False#s)))"
<proof>

```

```

lemma True_steps_concD[rule_format]:
  "!p. (True#p,q) : steps (conc L R) w -->
  ((? r. (p,r) : steps L w & q = True#r) |
  (? u v. w = u@v & (? r. (p,r) : steps L u & fin L r &
  (? s. (start R,s) : steps R v & q = False#s))))"
<proof>

```

```

lemma True_steps_conc:
  "(True#p,q) : steps (conc L R) w =
  ((? r. (p,r) : steps L w & q = True#r) |
  (? u v. w = u@v & (? r. (p,r) : steps L u & fin L r &
  (? s. (start R,s) : steps R v & q = False#s))))"
<proof>

```

```

lemma start_conc:
  "!!L R. start(conc L R) = True#start L"
<proof>

```

```

lemma final_conc:
  "!!L R. fin(conc L R) p = (? 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 True_in_eps_star[iff]:
  "!!A. (True#p,q) : eps(star A) =
  ( (? r. q = True#r & (p,r) : eps A) | (fin A p & q = True#start A)
  )"
<proof>

```

```

lemma True_True_step_starI:
  "!!A. (p,q) : step A a ==> (True#p, True#q) : step (star A) a"

```

*<proof>*

**lemma True\_True\_eps\_starI:**

"(p,r) : (eps A)^\* ==> (True#p, True#r) : (eps(star A))^\*" *<proof>*

**lemma True\_start\_eps\_starI:**

"!!A. fin A p ==> (True#p, True#start A) : eps(star A)" *<proof>*

**lemma lem':**

"(tp,s) : (eps(star A))^\* ==> (! p. tp = True#p -->  
(? r. ((p,r) : (eps A)^\* |  
          (? q. (p,q) : (eps A)^\* & fin A q & (start A,r) : (eps A)^\*))  
&  
      s = True#r))" *<proof>*

**lemma True\_eps\_star[iff]:**

"((True#p,s) : (eps(star A))^\*) =  
(? r. ((p,r) : (eps A)^\* |  
          (? q. (p,q) : (eps A)^\* & fin A q & (start A,r) : (eps A)^\*))  
&  
      s = True#r)" *<proof>*

**lemma True\_step\_star[iff]:**

"!!A. (True#p,r) : step (star A) (Some a) =  
      (? q. (p,q) : step A (Some a) & r=True#q)" *<proof>*

**lemma True\_start\_steps\_starD[rule\_format]:**

"!rr. (True#start A,rr) : steps (star A) w -->  
(? us v. w = concat us @ v &  
          (!u:set us. accepts A u) &  
          (? r. (start A,r) : steps A v & rr = True#r))" *<proof>*

**lemma True\_True\_steps\_starI:**

"!!p. (p,q) : steps A w ==> (True#p, True#q) : steps (star A) w" *<proof>*

**lemma steps\_star\_cycle:**

```

"!u : set us. accepts A u) ==>
(True#start A,True#start A) : steps (star A) (concat us)"
<proof>

```

```

lemma True_start_steps_star:
"(True#start A,rr) : steps (star A) w =
(? us v. w = concat us @ v &
(!u:set us. accepts A u) &
(? r. (start A,r) : steps A v & rr = True#r))"
<proof>

```

```

lemma start_step_star[iff]:
"!!A. (start(star A),r) : step (star A) a = (a=None & r = True#start
A)"
<proof>

```

```

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

```

```

lemma start_steps_star:
"(start(star A),r) : steps (star A) w =
((w=[] & r= start(star A)) | (True#start A,r) : steps (star A) w)"
<proof>

```

```

lemma fin_star_True[iff]: "!!A. fin (star A) (True#p) = fin A p"
<proof>

```

```

lemma fin_star_start[iff]: "!!A. fin (star A) (start(star A))"
<proof>

```

```

lemma accepts_star:
"accepts (star A) w =
(? us. (!u : set(us). accepts A u) & (w = concat us) )"
<proof>

```

```

lemma accepts_rexp2nae:
"!!w. accepts (rexp2nae r) w = (w : lang r)"
<proof>

```

```

end

```

## 11 Relating (finite) sets and lists

```
theory List_Set
imports Main
begin
```

### 11.1 Various additional list functions

```
definition insert :: "'a ⇒ 'a list ⇒ 'a list" where
  "insert x xs = (if x ∈ set xs then xs else x # xs)"
```

```
definition remove_all :: "'a ⇒ 'a list ⇒ 'a list" where
  "remove_all x xs = filter (Not o op = x) xs"
```

### 11.2 Various additional set functions

```
definition is_empty :: "'a set ⇒ bool" where
  "is_empty A ↔ A = {}"
```

```
definition remove :: "'a ⇒ 'a set ⇒ 'a set" where
  "remove x A = A - {x}"
```

```
lemma fun_left_comm_idem_remove:
  "fun_left_comm_idem remove"
⟨proof⟩
```

```
lemma minus_fold_remove:
  assumes "finite A"
  shows "B - A = fold remove B A"
⟨proof⟩
```

```
definition project :: "('a ⇒ bool) ⇒ 'a set ⇒ 'a set" where
  "project P A = {a∈A. P a}"
```

### 11.3 Basic set operations

```
lemma is_empty_set:
  "is_empty (set xs) ↔ null xs"
⟨proof⟩
```

```
lemma ball_set:
  "(∀x∈set xs. P x) ↔ list_all P xs"
⟨proof⟩
```

```
lemma bex_set:
  "(∃x∈set xs. P x) ↔ list_ex P xs"
⟨proof⟩
```

```
lemma empty_set:
  "{} = set []"
```

*<proof>*

**lemma insert\_set:**

"Set.insert x (set xs) = set (insert x xs)"  
*<proof>*

**lemma insert\_set\_compl:**

"Set.insert x (- set xs) = - set (List\_Set.remove\_all x xs)"  
*<proof>*

**lemma remove\_set:**

"remove x (set xs) = set (remove\_all x xs)"  
*<proof>*

**lemma remove\_set\_compl:**

"List\_Set.remove x (- set xs) = - set (List\_Set.insert x xs)"  
*<proof>*

**lemma image\_set:**

"image f (set xs) = set (map f xs)"  
*<proof>*

**lemma project\_set:**

"project P (set xs) = set (filter P xs)"  
*<proof>*

## 11.4 Functorial set operations

**lemma union\_set:**

"set xs  $\cup$  A = foldl ( $\lambda A x.$  Set.insert x A) A xs"  
*<proof>*

**lemma minus\_set:**

"A - set xs = foldl ( $\lambda A x.$  remove x A) A xs"  
*<proof>*

**lemma Inter\_set:**

"Inter (set As) = foldl (op  $\cap$ ) UNIV As"  
*<proof>*

**lemma Union\_set:**

"Union (set As) = foldl (op  $\cup$ ) {} As"  
*<proof>*

**lemma INTER\_set:**

"INTER (set As) f = foldl ( $\lambda B A.$  f A  $\cap$  B) UNIV As"  
*<proof>*

**lemma UNION\_set:**

```
"UNION (set As) f = foldl ( $\lambda B A. f A \cup B$ ) {} As"
<proof>
```

## 11.5 Derived set operations

```
lemma member:
  " $a \in A \iff (\exists x \in A. a = x)$ "
  <proof>
```

```
lemma subset_eq:
  " $A \subseteq B \iff (\forall x \in A. x \in B)$ "
  <proof>
```

```
lemma subset:
  " $A \subset B \iff A \subseteq B \wedge \neg B \subseteq A$ "
  <proof>
```

```
lemma set_eq:
  " $A = B \iff A \subseteq B \wedge B \subseteq A$ "
  <proof>
```

```
lemma inter:
  " $A \cap B = \text{project } (\lambda x. x \in A) B$ "
  <proof>
```

```
hide (open) const insert
```

```
end
```

## 12 Executable finite sets

```
theory Fset
imports List_Set
begin
```

```
declare mem_def [simp]
```

### 12.1 Lifting

```
datatype 'a fset = Fset "'a set"
```

```
primrec member :: "'a fset  $\Rightarrow$  'a set" where
  "member (Fset A) = A"
```

```
lemma Fset_member [simp]:
  "Fset (member A) = A"
  <proof>
```

```

definition Set :: "'a list  $\Rightarrow$  'a fset" where
  "Set xs = Fset (set xs)"

lemma member_Set [simp]:
  "member (Set xs) = set xs"
  <proof>

definition Coset :: "'a list  $\Rightarrow$  'a fset" where
  "Coset xs = Fset (- set xs)"

lemma member_Coset [simp]:
  "member (Coset xs) = - set xs"
  <proof>

code_datatype Set Coset

lemma member_code [code]:
  "member (Set xs) y  $\longleftrightarrow$  List.member y xs"
  "member (Coset xs) y  $\longleftrightarrow$   $\neg$  List.member y xs"
  <proof>

lemma member_image_UNIV [simp]:
  "member ' UNIV = UNIV"
  <proof>

12.2 Basic operations

definition is_empty :: "'a fset  $\Rightarrow$  bool" where
  [simp]: "is_empty A  $\longleftrightarrow$  List_Set.is_empty (member A)"

lemma is_empty_Set [code]:
  "is_empty (Set xs)  $\longleftrightarrow$  null xs"
  <proof>

definition empty :: "'a fset" where
  [simp]: "empty = Fset {}"

lemma empty_Set [code]:
  "empty = Set []"
  <proof>

definition insert :: "'a  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" where
  [simp]: "insert x A = Fset (Set.insert x (member A))"

lemma insert_Set [code]:
  "insert x (Set xs) = Set (List_Set.insert x xs)"
  "insert x (Coset xs) = Coset (remove_all x xs)"
  <proof>

```

```

definition remove :: "'a ⇒ 'a fset ⇒ 'a fset" where
  [simp]: "remove x A = Fset (List_Set.remove x (member A))"

lemma remove_Set [code]:
  "remove x (Set xs) = Set (remove_all x xs)"
  "remove x (Coset xs) = Coset (List_Set.insert x xs)"
  ⟨proof⟩

definition map :: "('a ⇒ 'b) ⇒ 'a fset ⇒ 'b fset" where
  [simp]: "map f A = Fset (image f (member A))"

lemma map_Set [code]:
  "map f (Set xs) = Set (remdups (List.map f xs))"
  ⟨proof⟩

definition filter :: "('a ⇒ bool) ⇒ 'a fset ⇒ 'a fset" where
  [simp]: "filter P A = Fset (List_Set.project P (member A))"

lemma filter_Set [code]:
  "filter P (Set xs) = Set (List.filter P xs)"
  ⟨proof⟩

definition forall :: "('a ⇒ bool) ⇒ 'a fset ⇒ bool" where
  [simp]: "forall P A ⟷ Ball (member A) P"

lemma forall_Set [code]:
  "forall P (Set xs) ⟷ list_all P xs"
  ⟨proof⟩

definition exists :: "('a ⇒ bool) ⇒ 'a fset ⇒ bool" where
  [simp]: "exists P A ⟷ Bex (member A) P"

lemma exists_Set [code]:
  "exists P (Set xs) ⟷ list_ex P xs"
  ⟨proof⟩

definition card :: "'a fset ⇒ nat" where
  [simp]: "card A = Finite_Set.card (member A)"

lemma card_Set [code]:
  "card (Set xs) = length (remdups xs)"
  ⟨proof⟩

```

### 12.3 Derived operations

```

definition subfset_eq :: "'a fset ⇒ 'a fset ⇒ bool" where
  [simp]: "subfset_eq A B ⟷ member A ⊆ member B"

lemma subfset_eq_forall [code]:

```

```
"subset_eq A B  $\longleftrightarrow$  forall (member B) A"  
<proof>
```

```
definition subset :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  bool" where  
[simp]: "subset A B  $\longleftrightarrow$  member A  $\subset$  member B"
```

```
lemma subset_subset_eq [code]:  
"subset A B  $\longleftrightarrow$  subset_eq A B  $\wedge$   $\neg$  subset_eq B A"  
<proof>
```

```
lemma eq_fset_subset_eq [code]:  
"eq_class.eq A B  $\longleftrightarrow$  subset_eq A B  $\wedge$  subset_eq B A"  
<proof>
```

## 12.4 Functorial operations

```
definition inter :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" where  
[simp]: "inter A B = Fset (member A  $\cap$  member B)"
```

```
lemma inter_project [code]:  
"inter A (Set xs) = Set (List.filter (member A) xs)"  
"inter A (Coset xs) = foldl ( $\lambda$ A x. remove x A) A xs"  
<proof>
```

```
definition subtract :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" where  
[simp]: "subtract A B = Fset (member B - member A)"
```

```
lemma subtract_remove [code]:  
"subtract (Set xs) A = foldl ( $\lambda$ A x. remove x A) A xs"  
"subtract (Coset xs) A = Set (List.filter (member A) xs)"  
<proof>
```

```
definition union :: "'a fset  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset" where  
[simp]: "union A B = Fset (member A  $\cup$  member B)"
```

```
lemma union_insert [code]:  
"union (Set xs) A = foldl ( $\lambda$ A x. insert x A) A xs"  
"union (Coset xs) A = Coset (List.filter (Not  $\circ$  member A) xs)"  
<proof>
```

```
definition Inter :: "'a fset fset  $\Rightarrow$  'a fset" where  
[simp]: "Inter A = Fset (Complete_Lattice.Inter (member  $\circ$  member A))"
```

```
lemma Inter_inter [code]:  
"Inter (Set As) = foldl inter (Coset []) As"  
"Inter (Coset []) = empty"  
<proof>
```

```
definition Union :: "'a fset fset  $\Rightarrow$  'a fset" where
```

```

[simp]: "Union A = Fset (Complete_Lattice.Union (member ' member A))"

lemma Union_union [code]:
  "Union (Set As) = foldl union empty As"
  "Union (Coset []) = Coset []"
⟨proof⟩

12.5 Misc operations

lemma size_fset [code]:
  "fset_size f A = 0"
  "size A = 0"
⟨proof⟩

lemma fset_case_code [code]:
  "fset_case f A = f (member A)"
⟨proof⟩

lemma fset_rec_code [code]:
  "fset_rec f A = f (member A)"
⟨proof⟩

12.6 Simplified simprules

lemma is_empty_simp [simp]:
  "is_empty A  $\longleftrightarrow$  member A = {}"
⟨proof⟩
declare is_empty_def [simp del]

lemma remove_simp [simp]:
  "remove x A = Fset (member A - {x})"
⟨proof⟩
declare remove_def [simp del]

lemma filter_simp [simp]:
  "filter P A = Fset {x  $\in$  member A. P x}"
⟨proof⟩
declare filter_def [simp del]

declare mem_def [simp del]

hide (open) const is_empty empty insert remove map filter forall exists
card
  subset_eq subset inter union subtract Inter Union

end

```

## 13 Install quickcheck of SML code generator

```
theory SML_Quickcheck
imports Main
begin

⟨ML⟩

end
```

## 14 Implementation of finite sets by lists

```
theory Executable_Set
imports Main Fset SML_Quickcheck
begin
```

### 14.1 Preprocessor setup

```
declare member [code]

definition empty :: "'a set" where
  "empty = {}"

declare empty_def [symmetric, code_unfold]

definition inter :: "'a set ⇒ 'a set ⇒ 'a set" where
  "inter = op ∩"

declare inter_def [symmetric, code_unfold]

definition union :: "'a set ⇒ 'a set ⇒ 'a set" where
  "union = op ∪"

declare union_def [symmetric, code_unfold]

definition subset :: "'a set ⇒ 'a set ⇒ bool" where
  "subset = op ≤"

declare subset_def [symmetric, code_unfold]

lemma [code]:
  "subset A B ⟷ (∀x∈A. x ∈ B)"
  ⟨proof⟩

definition eq_set :: "'a set ⇒ 'a set ⇒ bool" where
  [code del]: "eq_set = op ="
```

```

lemma [code]:
  "eq_set A B  $\longleftrightarrow$  A  $\subseteq$  B  $\wedge$  B  $\subseteq$  A"
  <proof>

declare inter [code]

declare List_Set.project_def [symmetric, code_unfold]

definition Inter :: "'a set set  $\Rightarrow$  'a set" where
  "Inter = Complete_Lattice.Inter"

declare Inter_def [symmetric, code_unfold]

definition Union :: "'a set set  $\Rightarrow$  'a set" where
  "Union = Complete_Lattice.Union"

declare Union_def [symmetric, code_unfold]

```

## 14.2 Code generator setup

<ML>

```

definition flip :: "('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow$  'c" where
  "flip f a b = f b a"

```

types\_code

```

  fset ("(_/ <module>fset)")
attach {*
datatype 'a fset = Set of 'a list | Coset of 'a list;
*}

```

consts\_code

```

  Set ("<module>Set")
  Coset ("<module>Coset")

```

consts\_code

```

  "empty"          ("{*Fset.empty*}")
  "List_Set.is_empty" ("{*Fset.is_empty*}")
  "Set.insert"     ("{*Fset.insert*}")
  "List_Set.remove" ("{*Fset.remove*}")
  "Set.image"      ("{*Fset.map*}")
  "List_Set.project" ("{*Fset.filter*}")
  "Ball"          ("{*flip Fset.forall*}")
  "Bex"           ("{*flip Fset.exists*}")
  "union"         ("{*Fset.union*}")
  "inter"         ("{*Fset.inter*}")
  "op - :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  'a set" ("{*flip Fset.subtract*}")
  "Union"        ("{*Fset.Union*}")

```

```

    "Inter"          (*Fset.Inter*)
    card             (*Fset.card*)
    fold             (*foldl o flip*)

hide (open) const empty inter union subset eq_set Inter Union flip

end

```

## 15 Combining automata and regular expressions, including code generation

```

theory AutoRegExp
imports Automata RegExp2NA RegExp2NAe Executable_Set
begin

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

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

declare RegExp2NA.star_def [unfolded epsilon_def, code]

code_module Generated
contains
  test =
    "let r0 = Atom(0::nat);
      r1 = Atom(1::nat);
      re = Conc (Star(Or(Conc r1 r1)r0))
              (Star(Or(Conc r0 r0)r1));
      N = rexp2na re;
      D = na2da N
    in (NA.accepts N [0,1,1,0,0,1], DA.accepts D [0,1,1,0,0,1])"

end

```

## 16 Maximal prefix

```

theory MaxPrefix
imports List_Prefix
begin

definition
  is_maxpref :: "('a list => bool) => 'a list => 'a list => bool" where
  "is_maxpref P xs ys =

```

```

(xs <= ys & (xs=[] | P xs) & (!zs. zs <= ys & P zs --> zs <= xs))"

types 'a splitter = "'a list => 'a list * 'a list"

definition
  is_maxsplitter :: "('a list => bool) => 'a splitter => bool" where
"is_maxsplitter P f =
  (!xs ps qs. f xs = (ps,qs) = (xs=ps@qs & is_maxpref P ps xs))"

consts
  maxsplit :: "('a list => bool) => 'a list * 'a list => 'a list => 'a
splitter"
primrec
"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 split_if[split del]

lemma maxsplit_lemma: "!(ps::'a list) res.
  (maxsplit P res ps qs = (xs,ys)) =
  (if EX us. us <= qs & P(ps@us) then xs@ys=ps@qs & is_maxpref P xs (ps@qs)
   else (xs,ys)=res)"
<proof>

declare split_if[split add]

lemma is_maxpref_Nil[simp]:
  "~(? us. us<=xs & P us) ==> is_maxpref P ps xs = (ps = [])"
<proof>

lemma is_maxsplitter_maxsplit:
  "is_maxsplitter P (%xs. maxsplit P ([],xs) [] xs)"
<proof>

lemmas maxsplit_eq = is_maxsplitter_maxsplit[simplified is_maxsplitter_def]

end

```

## 17 Generic scanner

```

theory MaxChop
imports MaxPrefix
begin

types '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 ==>

```

  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(%qs. maxsplit P ([],qs) [] qs)"

*<proof>*

**lemma is\_maxsplitter\_reducing:**

```

  "is_maxsplitter P splitf ==> reducing splitf"
```

*<proof>*

**lemma chop\_concat[rule\_format]:** "is\_maxsplitter P splitf ==>

```

  (!yss zs. chop splitf xs = (yss,zs) --> xs = concat yss @ zs)"
```

*<proof>*

**lemma chop\_nonempty:** "is\_maxsplitter P splitf ==>

```

  !yss zs. chop splitf xs = (yss,zs) --> (!ys : set yss. ys ~= [])"
```

*<proof>*

**lemma is\_maxchopper\_chop:**

```

  assumes prem: "is_maxsplitter P splitf" shows "is_maxchopper P (chop
```

```
splitf)"
⟨proof⟩
```

```
end
```

## 18 Automata based scanner

```
theory AutoMaxChop
imports DA MaxChop
begin
```

```
consts
```

```
  auto_split :: "('a,'s)da => 's => 'a list * 'a list => 'a list => 'a
  splitter"
```

```
primrec
```

```
"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 => 'a chopper" where
"auto_chop A = chop (%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
```

## 19 From deterministic automata to regular sets

```

theory RegSet_of_nat_DA
imports RegSet DA
begin

types 'a nat_next = "'a => nat => nat"

abbreviation
  deltas :: "'a nat_next => 'a list => nat => nat" where
    "deltas == foldl2"

consts trace :: "'a nat_next => nat => 'a list => nat list"
primrec
  "trace d i [] = []"
  "trace d i (x#xs) = d x i # trace d (d x i) xs"

consts regset :: "'a nat_next => nat => nat => nat => 'a list set"
primrec
  "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 Un
    conc (regset d i k k)
    (conc (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 = (UN 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) --> (EX 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]: "!!i. (trace d i xs = []) = (xs = [])"
  <proof>

lemma trace_is_Cons_conv[simp]:
  "(trace d i xs = n#ns) =
    (case xs of [] => False | y#ys => n = d y i & ns = trace d n ys)"
  <proof>

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

lemma deltas_concat[simp]:
  "(!mid:set mids. deltas d mid k = k) ==> deltas d (concat mids) k = k"
  <proof>

lemma lem: "[| n < Suc k; n ~ = k |] ==> n < k"
  <proof>

```

```

lemma regset_spec:
  "!!i j xs. xs : regset d i j k =
    (!n:set(butlast(trace d i xs)). n < k) & deltas d xs i = j)"
  <proof>

lemma trace_below:
  "bounded d k ==> !i. i < k --> (!n:set(trace d i xs). n < k)"
  <proof>

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

lemma deltas_below:
  "!!i. bounded d k ==> i < k ==> deltas d w i < k"
  <proof>

lemma regset_DA_equiv:
  "[| bounded (next A) k; start A < k; j < k |] ==>
    w : regset_of_DA A k = accepts A w"
  <proof>

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>.