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"*
by*(simp add:start_def)*

lemma *[simp]: "next(q,d,f) = d"*
by*(simp add:next_def)*

lemma *[simp]: "fin(q,d,f) = f"*
by*(simp add:fin_def)*

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 = foldl (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)"*
by*(simp add:foldl2_def)*

lemma *delta_Nil[simp]: "delta A [] s = s"*
by*(simp add:delta_def)*

lemma *delta_Cons[simp]: "delta A (a#w) s = delta A w (next A a s)"*
by*(simp add:delta_def)*

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

```
by(induct xs) simp_all

end
```

4 Nondeterministic automata

```
theory NA
imports AutoProj
begin

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

primrec delta :: "('a,'s)na ⇒ 'a list ⇒ 's ⇒ '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 ⇒ 'a list ⇒ bool" where
"accepts A w = (∃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}"

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

lemma steps_append[simp]:
 "steps A (v@w) = steps A v  O  steps A w"
by(induct v, simp_all add:O_assoc)

lemma in_steps_append[iff]:
  "(p,r) : steps A (v@w) = ((p,r) : (steps A v O steps A w))"
apply(rule steps_append[THEN equalityE])
apply blast
done

lemma delta_conv_steps: "⋀p. delta A w p = {q. (p,q) : steps A w}"
by(induct w)(auto simp:step_def)

lemma accepts_conv_steps:
 "accepts A w = (∃q. (start A,q) ∈ steps A w ∧ fin A q)"
by(simp add: delta_conv_steps accepts_def)

abbreviation
  Cons_syn :: "'a ⇒ 'a list set ⇒ 'a list set" (infixr ‹##› 65) where
  "x ## S ≡ Cons x ` S"
```

end

5 Nondeterministic automata with epsilon transitions

```
theory NAe
imports NA
begin

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

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

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

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



lemma steps_epsclosure[simp]: "(eps A)* O steps A w = steps A w"
by (cases w) (simp_all add: O_assoc[symmetric])

lemma in_steps_epsclosure:
  "[| (p,q) : (eps A)*; (q,r) : steps A w |] ==> (p,r) : steps A w"
apply(rule steps_epsclosure[THEN equalityE])
apply blast
done

lemma epsclosure_steps: "steps A w O (eps A)* = steps A w"
apply(induct w)
 apply simp
apply(simp add:O_assoc)
done

lemma in_epsclosure_steps:
  "[| (p,q) : steps A w; (q,r) : (eps A)* |] ==> (p,r) : steps A w"
apply(rule epsclosure_steps[THEN equalityE])
apply blast
done

lemma steps_append[simp]:  "steps A (v@w) = steps A v  O  steps A w"
by(induct v)(simp_all add:O_assoc[symmetric])
```

```
lemma in_steps_append[iff]:
  "(p,r) : steps A (v@w) = ((p,r) : (steps A v O steps A w))"
apply(rule steps_append[THEN equalityE])
apply blast
done



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}, λa Q. Union(next A a ‘ Q), λQ. ∃q∈Q. fin A q)"

definition
 nae2da :: "('a,'s)nae ⇒ ('a,'s set)da" where
"nae2da A = ({start A},
             λa Q. Union(next A (Some a) ‘ ((eps A)* ‘‘ Q)),
             λQ. ∃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)"
by (induct w)(auto simp:na2da_def)

lemma NA_DA_equiv:
  "NA.accepts A w = DA.accepts (na2da A) w"
apply (simp add: DA.accepts_def NA.accepts_def DA_delta_is_lift_NA_delta)
apply (simp add: na2da_def)
done



lemma espclosure_DA_delta_is_steps:
 "⋀Q. (eps A)* ‘‘ (DA.delta (nae2da A) w Q) = steps A w ‘‘ Q"
apply (induct w)
 apply(simp)
apply (simp add: step_def nae2da_def)
apply (blast)
done
```

```
lemma NAe_DA_equiv:
  "DA.accepts (nae2da A) w = NAe.accepts A w"
proof -
  have "⋀Q. fin (nae2da A) Q = (∃q ∈ (eps A)* `` Q. fin A q)"
    by(simp add:nae2da_def)
  thus ?thesis
    apply(simp add:espclosure_DA_delta_is_steps NAe.accepts_def DA.accepts_def)
    apply(simp add:nae2da_def)
    apply blast
    done
qed

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(ql,dl,fl)(qr,dr,fr).
   ([],
    λa s. case s of
            [] ⇒ (True ## dl a ql) ∪ (False ## dr a qr)
          | left#s ⇒ if left then True ## dl a s
                              else False ## dr a s,
    λs. 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 = (λ(ql,dl,fl)(qr,dr,fr).
   (True#ql,
    λa s. case s of
            [] ⇒ {}
          | left#s ⇒ if left then (True ## dl a s) ∪
                                   (if fl s then False ## dr a qr else
```

```
{})
                                   else False ## dr a s,
    λs. case s of [] ⇒ False | left#s ⇒ left ∧ fl 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])"*
by*(simp add:atom_def)*

lemma *start_atom: "start (atom a) = [True]"*
by*(simp add:atom_def)*

lemma *in_step_atom_Some[simp]:*
```
 "(p,q) : step (atom a) b = (p=[True] ∧ q=[False] ∧ b=a)"
```
by *(simp add: atom_def step_def)*

lemma *False_False_in_steps_atom:*
```
 "([False],[False]) : steps (atom a) w = (w = [])"
```
apply *(induct "w")*
 apply *simp*
apply *(simp add: relcomp_unfold)*
done

lemma *start_fin_in_steps_atom:*

```
 "(start (atom a), [False]) : steps (atom a) w = (w = [a])"
apply (induct "w")
 apply (simp add: start_atom)
apply (simp add: False_False_in_steps_atom relcomp_unfold start_atom)
done

lemma accepts_atom:
 "accepts (atom a) w = (w = [a])"
by (simp add: accepts_conv_steps start_fin_in_steps_atom fin_atom)



lemma fin_or_True[iff]:
 "⋀L R. fin (or L R) (True#p) = fin L p"
by(simp add:or_def)

lemma fin_or_False[iff]:
 "⋀L R. fin (or L R) (False#p) = fin R p"
by(simp add:or_def)


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)"
apply (simp add:or_def step_def)
apply blast
done

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)"
apply (simp add:or_def step_def)
apply blast
done


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)"
apply (induct "w")
 apply force
apply force
```

```
done

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)"
apply (induct "w")
 apply force
apply force
done



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))"
apply (simp add:or_def step_def)
apply blast
done

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)))"
apply (case_tac "w")
 apply (simp)
 apply blast
apply (simp)
apply blast
done

lemma fin_start_or[iff]:
 "⋀L R. fin (or L R) (start(or L R)) = (fin L (start L) | fin R (start
R))"
by (simp add:or_def)

lemma accepts_or[iff]:
 "accepts (or L R) w = (accepts L w | accepts R w)"
apply (simp add: accepts_conv_steps steps_or)

apply (case_tac "w = []")
 apply auto
done
```

```
lemma fin_conc_True[iff]:
 "⋀L R. fin (conc L R) (True#p) = (fin L p ∧ fin R (start R))"
by(simp add:conc_def)

lemma fin_conc_False[iff]:
 "⋀L R. fin (conc L R) (False#p) = fin R p"
by(simp add:conc_def)


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)))"
apply (simp add:conc_def step_def)
apply blast
done

lemma False_step_conc[iff]:
 "⋀L R. (False#p,q) : step (conc L R) a =
       (∃r. q = False#r ∧ (p,r) : step R a)"
apply (simp add:conc_def step_def)
apply blast
done


lemma False_steps_conc[iff]:
 "⋀p. (False#p,q): steps (conc L R) w = (∃r. q=False#r ∧ (p,r): steps
R w)"
apply (induct "w")
 apply fastforce
apply force
done


lemma True_True_steps_concI:
 "⋀L R p. (p,q) : steps L w ⟹ (True#p,True#q) : steps (conc L R) w"
apply (induct "w")
 apply simp
apply simp
apply fast
done

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)"
by simp

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)))))"
apply (induct "w")
 apply simp
apply simp
apply (clarify del:disjCI)
apply (erule disjE)
 apply (clarify del:disjCI)
 apply (erule allE, erule impE, assumption)
 apply (erule disjE)
  apply blast
 apply (rule disjI2)
 apply (clarify)
 apply simp
 apply (rule_tac x = "a#u" in exI)
 apply simp
 apply blast
apply (rule disjI2)
apply (clarify)
apply simp
apply (rule_tac x = "[]" in exI)
apply simp
apply blast
done

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)))))"
by(force dest!: True_steps_concD intro!: True_True_steps_concI)


lemma start_conc:
  "⋀L R. start(conc L R) = True#start L"
by (simp add:conc_def)
```

```
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))"
apply (simp add:conc_def split: list.split)
apply blast
done

lemma accepts_conc:
 "accepts (conc L R) w = (∃u v. w = u@v ∧ accepts L u ∧ accepts R v)"
apply (simp add: accepts_conv_steps True_steps_conc final_conc start_conc)
apply (rule iffI)
 apply (clarify)
 apply (erule disjE)
  apply (clarify)
  apply (erule disjE)
   apply (rule_tac x = "w" in exI)
   apply simp
   apply blast
  apply blast
 apply (erule disjE)
  apply blast
 apply (clarify)
 apply (rule_tac x = "u" in exI)
 apply simp
 apply blast
apply (clarify)
apply (case_tac "v")
 apply simp
 apply blast
apply simp
apply blast
done
```

```
lemma step_epsilon[simp]: "step epsilon a = {}"
by(simp add:epsilon_def step_def)

lemma steps_epsilon: "((p,q) : steps epsilon w) = (w=[] ∧ p=q)"
by (induct "w") auto

lemma accepts_epsilon[iff]: "accepts epsilon w = (w = [])"
apply (simp add: steps_epsilon accepts_conv_steps)
apply (simp add: epsilon_def)
done
```

```
lemma start_plus[simp]: "⋀A. start (plus A) = start A"
by(simp add:plus_def)

lemma fin_plus[iff]: "⋀A. fin (plus A) = fin A"
by(simp add:plus_def)

lemma step_plusI:
  "⋀A. (p,q) : step A a ⟹ (p,q) : step (plus A) a"
by(simp add:plus_def step_def)

lemma steps_plusI: "⋀p. (p,q) : steps A w ⟹ (p,q) ∈ steps (plus A)
w"
apply (induct "w")
 apply simp
apply simp
apply (blast intro: step_plusI)
done

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 )"
by(simp add:plus_def step_def)

lemma fin_steps_plusI:
 "[| (start A,q) : steps A u; u ≠ []; fin A p |]
 ==> (p,q) : steps (plus A) u"
apply (case_tac "u")
 apply blast
apply simp
apply (blast intro: steps_plusI)
done


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)"
apply (induct w rule: rev_induct)
 apply simp
 apply (rule_tac x = "[]" in exI)
 apply simp
apply simp
apply (clarify)
apply (erule allE, erule impE, assumption)
```

```
apply (clarify)
apply (erule disjE)
 apply (rule_tac x = "us" in exI)
 apply (simp)
 apply blast
apply (rule_tac x = "us@[v]" in exI)
apply (simp add: accepts_conv_steps)
apply blast
done

lemma steps_star_cycle[rule_format]:
 "us ≠ [] ⟶ (∀u ∈ set us. accepts A u) ⟶ accepts (plus A) (concat
us)"
apply (simp add: accepts_conv_steps)
apply (induct us rule: rev_induct)
 apply simp
apply (rename_tac u us)
apply simp
apply (clarify)
apply (case_tac "us = []")
 apply (simp)
 apply (blast intro: steps_plusI fin_steps_plusI)
apply (clarify)
apply (case_tac "u = []")
 apply (simp)
 apply (blast intro: steps_plusI fin_steps_plusI)
apply (blast intro: steps_plusI fin_steps_plusI)
done

lemma accepts_plus[iff]:
 "accepts (plus A) w =
 (∃us. us ≠ [] ∧ w = concat us ∧ (∀u ∈ set us. accepts A u))"
apply (rule iffI)
 apply (simp add: accepts_conv_steps)
 apply (clarify)
 apply (drule start_steps_plusD)
 apply (clarify)
 apply (rule_tac x = "us@[v]" in exI)
 apply (simp add: accepts_conv_steps)
 apply blast
apply (blast intro: steps_star_cycle)
done

lemma accepts_star:
 "accepts (star A) w = (∃us. (∀u ∈ set us. accepts A u) ∧ w = concat
```

```
us)"
apply(unfold star_def)
apply (rule iffI)
 apply (clarify)
 apply (erule disjE)
  apply (rule_tac x = "[]" in exI)
  apply simp
 apply blast
apply force
done
```

```
lemma accepts_rexp2na:
 "⋀w. accepts (rexp2na r) w = (w : lang r)"
apply (induct "r")
     apply (simp add: accepts_conv_steps)
    apply simp
   apply (simp add: accepts_atom)
  apply (simp)
 apply (simp add: accepts_conc Regular_Set.conc_def)
apply (simp add: accepts_star in_star_iff_concat subset_iff Ball_def)
done

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 = ([],λa s. {}, λs. s=[])"

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

definition
 or :: "'a bitsNAe ⇒ 'a bitsNAe ⇒ 'a bitsNAe" where
"or = (λ(ql,dl,fl)(qr,dr,fr).
```

```
   ([],
    λa 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,
    λs. case s of [] ⇒ False | left#s ⇒ if left then fl s else fr s))"
```

definition
 conc :: "'a bitsNAe ⇒ 'a bitsNAe ⇒ 'a bitsNAe" **where**

```
"conc = (λ(ql,dl,fl)(qr,dr,fr).
   (True#ql,
    λa s. case s of
            [] ⇒ {}
          | left#s ⇒ if left then (True ## dl a s) ∪
                                  (if fl s ∧ a=None then {False#qr} else
{})
                               else False ## dr a s,
    λs. case s of [] ⇒ False | left#s ⇒ ¬left ∧ fr s))"
```

definition
 star :: "'a bitsNAe ⇒ 'a bitsNAe" **where**

```
"star = (λ(q,d,f).
   ([],
    λa 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 = {}"*
by*(simp add:epsilon_def step_def)*

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

```
by (induct "w") auto

lemma accepts_epsilon[simp]: "accepts epsilon w = (w = [])"
apply (simp add: steps_epsilon accepts_def)
apply (simp add: epsilon_def)
done

lemma fin_atom: "(fin (atom a) q) = (q = [False])"
by(simp add:atom_def)

lemma start_atom: "start (atom a) = [True]"
by(simp add:atom_def)

lemma eps_atom[simp]:
 "eps(atom a) = {}"
by (simp add:atom_def step_def)

lemma in_step_atom_Some[simp]:
 "(p,q) : step (atom a) (Some b) = (p=[True] ∧ q=[False] ∧ b=a)"
by (simp add:atom_def step_def)

lemma False_False_in_steps_atom:
  "([False],[False]) : steps (atom a) w = (w = [])"
apply (induct "w")
 apply (simp)
apply (simp add: relcomp_unfold)
done

lemma start_fin_in_steps_atom:
  "(start (atom a), [False]) : steps (atom a) w = (w = [a])"
apply (induct "w")
 apply (simp add: start_atom rtrancl_empty)
apply (simp add: False_False_in_steps_atom relcomp_unfold start_atom)
done

lemma accepts_atom: "accepts (atom a) w = (w = [a])"
by (simp add: accepts_def start_fin_in_steps_atom fin_atom)
```

lemma *fin_or_True[iff]:*
 "⋀L R. fin (or L R) (True#p) = fin L p"
by*(simp add:or_def)*

lemma *fin_or_False[iff]:*
 "⋀L R. fin (or L R) (False#p) = fin R p"
by*(simp add:or_def)*

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)"
apply *(simp add:or_def step_def)*
apply *blast*
done

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)"
apply *(simp add:or_def step_def)*
apply *blast*
done

lemma *lemma1a:*
 "(tp,tq) : (eps(or L R)) ⟹*
 (⋀p. tp = True#p ⟹ ∃q. (p,q) : (eps L) ∧ tq = True#q)"*
apply *(induct rule:rtrancl_induct)*
 apply *(blast)*
apply *(clarify)*
apply *(simp)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *lemma1b:*
 "(tp,tq) : (eps(or L R)) ⟹*
 (⋀p. tp = False#p ⟹ ∃q. (p,q) : (eps R) ∧ tq = False#q)"*
apply *(induct rule:rtrancl_induct)*
 apply *(blast)*
apply *(clarify)*
apply *(simp)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *lemma2a:*
 "(p,q) : (eps L)* $\Longrightarrow$ (True#p, True#q) : (eps(or L R))*"
apply *(induct rule: rtrancl_induct)*
 apply *(blast)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *lemma2b:*
 "(p,q) : (eps R)* $\Longrightarrow$ (False#p, False#q) : (eps(or L R))*"
apply *(induct rule: rtrancl_induct)*
 apply *(blast)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *True_epsclosure_or[iff]:*
 "(True#p,q) : (eps(or L R))* = ($\exists$r. q = True#r $\wedge$ (p,r) : (eps L)*)"
by *(blast dest: lemma1a lemma2a)*

lemma *False_epsclosure_or[iff]:*
 "(False#p,q) : (eps(or L R))* = ($\exists$r. q = False#r $\wedge$ (p,r) : (eps R)*)"
by *(blast dest: lemma1b lemma2b)*

lemma *lift_True_over_steps_or[iff]:*
 "$\bigwedge$p. (True#p,q):steps (or L R) w = ($\exists$r. q = True # r $\wedge$ (p,r):steps L w)"
apply *(induct "w")*
 apply *auto*
apply *force*
done

lemma *lift_False_over_steps_or[iff]:*
 "$\bigwedge$p. (False#p,q):steps (or L R) w = ($\exists$r. q = False#r $\wedge$ (p,r):steps R w)"
apply *(induct "w")*
 apply *auto*
apply *(force)*
done

lemma *unfold_rtrancl2:*
 "R* = Id $\cup$ (R O R*)"
apply *(rule set_eqI)*
apply *(simp)*
apply *(rule iffI)*
 apply *(erule rtrancl_induct)*
  apply *(blast)*

```
 apply (blast intro: rtrancl_into_rtrancl)
apply (blast intro: converse_rtrancl_into_rtrancl)
done

lemma in_unfold_rtrancl2:
 "(p,q) : R* = (q = p | (∃r. (p,r) : R ∧ (r,q) : R*))"
apply (rule unfold_rtrancl2[THEN equalityE])
apply (blast)
done

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

lemma start_eps_or[iff]:
 "⋀L R. (start(or L R),q) : eps(or L R) =
       (q = True#start L | q = False#start R)"
by (simp add:or_def step_def)

lemma not_start_step_or_Some[iff]:
 "⋀L R. (start(or L R),q) ∉ step (or L R) (Some a)"
by (simp add:or_def step_def)

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) )"
apply (case_tac "w")
 apply (simp)
 apply (blast)
apply (simp)
apply (blast)
done

lemma start_or_not_final[iff]:
 "⋀L R. ¬ fin (or L R) (start(or L R))"
by (simp add:or_def)

lemma accepts_or:
 "accepts (or L R) w = (accepts L w | accepts R w)"
apply (simp add:accepts_def steps_or)
 apply auto
done
```

```
lemma in_conc_True[iff]:
 "⋀L R. fin (conc L R) (True#p) = False"
by (simp add:conc_def)

lemma fin_conc_False[iff]:
 "⋀L R. fin (conc L R) (False#p) = fin R p"
by (simp add:conc_def)



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))"
by (simp add:conc_def step_def) (blast)

lemma False_step_conc[iff]:
 "⋀L R. (False#p,q) : step (conc L R) a =
       (∃r. q = False#r ∧ (p,r) : step R a)"
by (simp add:conc_def step_def) (blast)



lemma lemma1b':
 "(tp,tq) : (eps(conc L R))* ⟹
  (⋀p. tp = False#p ⟹ ∃q. (p,q) : (eps R)* ∧ tq = False#q)"
apply (induct rule: rtrancl_induct)
 apply (blast)
apply (blast intro: rtrancl_into_rtrancl)
done

lemma lemma2b':
 "(p,q) : (eps R)* ⟹ (False#p, False#q) : (eps(conc L R))*"
apply (induct rule: rtrancl_induct)
 apply (blast)
apply (blast intro: rtrancl_into_rtrancl)
done

lemma False_epsclosure_conc[iff]:
 "((False # p, q) : (eps (conc L R))*) =
 (∃r. q = False # r ∧ (p, r) : (eps R)*)"
apply (rule iffI)
 apply (blast dest: lemma1b')
apply (blast dest: lemma2b')
done
```

lemma *False_steps_conc[iff]:*
"⋀p. (False#p,q): steps (conc L R) w = (∃r. q=False#r ∧ (p,r): steps R w)"
apply *(induct "w")*
 apply *(simp)*
apply *(simp)*
apply *(fast)*
done

lemma *True_True_eps_concI:*
"(p,q): (eps L) ⟹ (True#p,True#q) : (eps(conc L R))*"*
apply *(induct rule: rtrancl_induct)*
 apply *(blast)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *True_True_steps_concI:*
"⋀p. (p,q) : steps L w ⟹ (True#p,True#q) : steps (conc L R) w"
apply *(induct "w")*
 apply *(simp add: True_True_eps_concI)*
apply *(simp)*
apply *(blast intro: True_True_eps_concI)*
done

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)*))"*
apply *(induct rule: rtrancl_induct)*
 apply *(blast)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *lemma2a':*
"(p, q) : (eps L) ⟹ (True#p, True#q) : (eps(conc L R))*"*
apply *(induct rule: rtrancl_induct)*
 apply *(blast)*
apply *(blast intro: rtrancl_into_rtrancl)*
done

lemma *lem:*
"⋀L R. (p,q) : step R None ⟹ (False#p, False#q) : step (conc L R) None"
by *(simp add: conc_def step_def)*

```
lemma lemma2b'':
 "(p,q) : (eps R)* ⟹ (False#p, False#q) : (eps(conc L R))*"
apply (induct rule: rtrancl_induct)
 apply (blast)
apply (drule lem)
apply (blast intro: rtrancl_into_rtrancl)
done

lemma True_False_eps_concI:
 "⋀L R. fin L p ⟹ (True#p, False#start R) : eps(conc L R)"
by(simp add: conc_def step_def)

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)))"
apply (rule iffI)
 apply (blast dest: lemma1a')
apply (erule disjE)
 apply (blast intro: lemma2a')
apply (clarify)
apply (rule rtrancl_trans)
apply (erule lemma2a')
apply (rule converse_rtrancl_into_rtrancl)
apply (erule True_False_eps_concI)
apply (erule lemma2b'')
done



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))))"
apply (induct "w")
 apply (simp)
apply (simp)
apply (clarify del: disjCI)
 apply (erule disjE)
 apply (clarify del: disjCI)
 apply (erule disjE)
  apply (clarify del: disjCI)
  apply (erule allE, erule impE, assumption)
  apply (erule disjE)
   apply (blast)
  apply (rule disjI2)
  apply (clarify)
```

```
  apply (simp)
  apply (rule_tac x = "a#u" in exI)
  apply (simp)
  apply (blast)
 apply (blast)
apply (rule disjI2)
apply (clarify)
apply (simp)
apply (rule_tac x = "[]" in exI)
apply (simp)
apply (blast)
done

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))))"
by (blast dest: True_steps_concD
    intro: True_True_steps_concI in_steps_epsclosure)



lemma start_conc:
  "⋀L R. start(conc L R) = True#start L"
by (simp add: conc_def)

lemma final_conc:
 "⋀L R. fin(conc L R) p = (∃s. p = False#s ∧ fin R s)"
by (simp add:conc_def split: list.split)

lemma accepts_conc:
 "accepts (conc L R) w = (∃u v. w = u@v ∧ accepts L u ∧ accepts R v)"
apply (simp add: accepts_def True_steps_conc final_conc start_conc)
apply (blast)
done




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) )"
by (simp add:star_def step_def) (blast)

lemma True_True_step_starI:
  "⋀A. (p,q) : step A a ⟹ (True#p, True#q) : step (star A) a"
```

```
by (simp add:star_def step_def)

lemma True_True_eps_starI:
  "(p,r) : (eps A)* ⟹ (True#p, True#r) : (eps(star A))*"
apply (induct rule: rtrancl_induct)
 apply (blast)
apply (blast intro: True_True_step_starI rtrancl_into_rtrancl)
done

lemma True_start_eps_starI:
 "⋀A. fin A p ⟹ (True#p,True#start A) : eps(star A)"
by (simp add:star_def step_def)

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))"
apply (induct rule: rtrancl_induct)
 apply (simp)
apply (clarify)
apply (simp)
apply (blast intro: rtrancl_into_rtrancl)
done

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)"
apply (rule iffI)
 apply (drule lem')
 apply (blast)

apply (clarify)
apply (erule disjE)
apply (erule True_True_eps_starI)
apply (clarify)
apply (rule rtrancl_trans)
apply (erule True_True_eps_starI)
apply (rule rtrancl_trans)
apply (rule r_into_rtrancl)
apply (erule True_start_eps_starI)
apply (erule True_True_eps_starI)
done
```

```
lemma True_step_star[iff]:
 "⋀A. (True#p,r) ∈ step (star A) (Some a) =
     (∃q. (p,q) ∈ step A (Some a) ∧ r=True#q)"
by (simp add:star_def step_def) (blast)
```

```
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))"
apply (induct w rule: rev_induct)
 apply (simp)
 apply (clarify)
 apply (rule_tac x = "[]" in exI)
 apply (erule disjE)
  apply (simp)
 apply (clarify)
 apply (simp)
apply (simp add: O_assoc[symmetric] epsclosure_steps)
apply (clarify)
apply (erule allE, erule impE, assumption)
apply (clarify)
apply (erule disjE)
 apply (rule_tac x = "us" in exI)
 apply (rule_tac x = "v@[x]" in exI)
 apply (simp add: O_assoc[symmetric] epsclosure_steps)
 apply (blast)
apply (clarify)
apply (rule_tac x = "us@[v@[x]]" in exI)
apply (rule_tac x = "[]" in exI)
apply (simp add: accepts_def)
apply (blast)
done
```

```
lemma True_True_steps_starI:
  "⋀p. (p,q) : steps A w ⟹ (True#p,True#q) : steps (star A) w"
apply (induct "w")
 apply (simp)
apply (simp)
apply (blast intro: True_True_eps_starI True_True_step_starI)
done
```

```
lemma steps_star_cycle:
 "(∀u ∈ set us. accepts A u) ⟹
 (True#start A,True#start A) ∈ steps (star A) (concat us)"
```

```
apply (induct "us")
 apply (simp add:accepts_def)
apply (simp add:accepts_def)
by(blast intro: True_True_steps_starI True_start_eps_starI in_epsclosure_steps)


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))"
apply (rule iffI)
 apply (erule True_start_steps_starD)
apply (clarify)
apply (blast intro: steps_star_cycle True_True_steps_starI)
done



lemma start_step_star[iff]:
  "⋀A. (start(star A),r) : step (star A) a = (a=None ∧ r = True#start
A)"
by (simp add:star_def step_def)

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

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)"
apply (rule iffI)
 apply (case_tac "w")
  apply (simp add: epsclosure_start_step_star)
 apply (simp)
 apply (clarify)
 apply (simp add: epsclosure_start_step_star)
 apply (blast)
apply (erule disjE)
 apply (simp)
apply (blast intro: in_steps_epsclosure)
done

lemma fin_star_True[iff]: "⋀A. fin (star A) (True#p) = fin A p"
by (simp add:star_def)

lemma fin_star_start[iff]: "⋀A. fin (star A) (start(star A))"
by (simp add:star_def)
```

```
lemma accepts_star:
 "accepts (star A) w =
 (∃us. (∀u ∈ set(us). accepts A u) ∧ (w = concat us))"
apply(unfold accepts_def)
apply (simp add: start_steps_star True_start_steps_star)
apply (rule iffI)
 apply (clarify)
 apply (erule disjE)
  apply (clarify)
  apply (simp)
  apply (rule_tac x = "[]" in exI)
  apply (simp)
 apply (clarify)
 apply (rule_tac x = "us@[v]" in exI)
 apply (simp add: accepts_def)
 apply (blast)
apply (clarify)
apply (rule_tac xs = "us" in rev_exhaust)
 apply (simp)
 apply (blast)
apply (clarify)
apply (simp add: accepts_def)
apply (blast)
done
```

```
lemma accepts_rexp2nae:
 "⋀w. accepts (rexp2nae r) w = (w : lang r)"
apply (induct "r")
     apply (simp add: accepts_def)
    apply simp
   apply (simp add: accepts_atom)
  apply (simp add: accepts_or)
 apply (simp add: accepts_conc Regular_Set.conc_def)
apply (simp add: accepts_star in_star_iff_concat subset_iff Ball_def)
done

end
```

9 Combining automata and regular expressions

```
theory AutoRegExp
imports Automata RegExp2NA RegExp2NAe
begin

theorem "DA.accepts (na2da(rexp2na r)) w = (w : lang r)"
by (simp add: NA_DA_equiv[THEN sym] accepts_rexp2na)
```

```
theorem "DA.accepts (nae2da(rexp2nae r)) w = (w : lang r)"
by (simp add: NAe_DA_equiv accepts_rexp2nae)

end
```

10 Maximal prefix

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

definition
 is_maxpref :: "('a list ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool" where
"is_maxpref P xs ys =
 (prefix xs ys ∧ (xs=[] ∨ P xs) ∧ (∀zs. prefix zs ys ∧ P zs ⟶ prefix
zs xs))"

type_synonym '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))"

fun maxsplit :: "('a list ⇒ bool) ⇒ 'a list * 'a list ⇒ 'a list ⇒
'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 ∃us. prefix us qs ∧ P(ps@us) then xs@ys=ps@qs ∧ is_maxpref P xs
(ps@qs)
   else (xs,ys)=res)"
proof (induction P res ps qs rule: maxsplit.induct)
  case 1
  thus ?case by (auto simp: is_maxpref_def split: if_splits)
next
  case (2 P res ps q qs)
  show ?case
  proof (cases "∃us. prefix us qs ∧ P ((ps @ [q]) @ us)")
    case ex1: True
    then obtain us where "prefix us qs" "P ((ps @ [q]) @ us)" by blast
    hence ex2: "∃us. prefix us (q # qs) ∧ P (ps @ us)"
      by (intro exI[of _ "q#us"]) auto
    with ex1 and 2 show ?thesis by simp
```

```
  next
    case ex1: False
    show ?thesis
    proof (cases "∃us. prefix us (q#qs) ∧ P (ps @ us)")
      case False
      from 2 show ?thesis
        by (simp only: ex1 False) (insert ex1 False, auto simp: prefix_Cons)
    next
      case True
      note ex2 = this
      show ?thesis
      proof (cases "P ps")
        case True
        with 2 have "(maxsplit P (ps, q # qs) (ps @ [q]) qs = (xs, ys))
⟷ (xs = ps ∧ ys = q # qs)"
          by (simp only: ex1 ex2) simp_all
        also have "... ⟷ (xs @ ys = ps @ q # qs ∧ is_maxpref P xs (ps
@ q # qs))"
          using ex1 True
          by (auto simp: is_maxpref_def prefix_append prefix_Cons append_eq_append_conv2)
        finally show ?thesis using True by (simp only: ex1 ex2) simp_all
      next
        case False
        with 2 have "(maxsplit P res (ps @ [q]) qs = (xs, ys)) ⟷ ((xs,
ys) = res)"
          by (simp only: ex1 ex2) simp
        also have "... ⟷ (xs @ ys = ps @ q # qs ∧ is_maxpref P xs (ps
@ q # qs))"
          using ex1 ex2 False
          by (auto simp: append_eq_append_conv2 is_maxpref_def prefix_Cons)
        finally show ?thesis
          using False by (simp only: ex1 ex2) simp
      qed
    qed
  qed
qed

declare if_split[split]

lemma is_maxpref_Nil[simp]:
 "¬(∃us. prefix us xs ∧ P us) ⟹ is_maxpref P ps xs = (ps = [])"
  by (auto simp: is_maxpref_def)

lemma is_maxsplitter_maxsplit:
 "is_maxsplitter P (λxs. maxsplit P ([],xs) [] xs)"
  by (auto simp: maxsplit_lemma is_maxsplitter_def)

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)"
by pat_completeness auto

termination apply (relation "measure (length ∘ snd)")
apply (auto simp: reducing_def)
apply (case_tac "splitf xs")
apply auto
done

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))"
apply (simp add: chop.simps)
apply (simp add: Let_def split: prod.split)
done
```

```
lemma reducing_maxsplit: "reducing(λqs. maxsplit P ([],qs) [] qs)"
by (simp add: reducing_def maxsplit_eq)

lemma is_maxsplitter_reducing:
 "is_maxsplitter P splitf ⟹ reducing splitf"
by(simp add:is_maxsplitter_def reducing_def)

lemma chop_concat[rule_format]: "is_maxsplitter P splitf ⟹
  (∀yss zs. chop splitf xs = (yss,zs) ⟶ xs = concat yss @ zs)"
apply (induct xs rule:length_induct)
apply (simp (no_asm_simp) split del: if_split
            add: chop_rule[OF is_maxsplitter_reducing])
apply (simp add: Let_def is_maxsplitter_def split: prod.split)
done

lemma chop_nonempty: "is_maxsplitter P splitf ⟹
  ∀yss zs. chop splitf xs = (yss,zs) ⟶ (∀ys ∈ set yss. ys ≠ [])"
apply (induct xs rule:length_induct)
apply (simp (no_asm_simp) add: chop_rule is_maxsplitter_reducing)
apply (simp add: Let_def is_maxsplitter_def split: prod.split)
apply (intro allI impI)
apply (rule ballI)
apply (erule exE)
apply (erule allE)
apply auto
done

lemma is_maxchopper_chop:
 assumes prem: "is_maxsplitter P splitf" shows "is_maxchopper P (chop
splitf)"
apply(unfold is_maxchopper_def)
apply clarify
apply (rule iffI)
 apply (rule conjI)
  apply (erule chop_concat[OF prem])
 apply (rule conjI)
  apply (erule prem[THEN chop_nonempty[THEN spec, THEN spec, THEN mp]])
 apply (erule rev_mp)
 apply (subst prem[THEN is_maxsplitter_reducing[THEN chop_rule]])
 apply (simp add: Let_def prem[simplified is_maxsplitter_def]
             split: prod.split)
 apply clarify
 apply (rule conjI)
  apply (clarify)
 apply (clarify)
 apply simp
 apply (frule chop_concat[OF prem])
 apply (clarify)
apply (subst prem[THEN is_maxsplitter_reducing, THEN chop_rule])
```

```
apply (simp add: Let_def prem[simplified is_maxsplitter_def]
             split: prod.split)
apply (clarify)
apply (rename_tac xs1 ys1 xss1 ys)
apply (simp split: list.split_asm)
 apply (simp add: is_maxpref_def)
 apply (blast intro: prefix_append[THEN iffD2])
apply (rule conjI)
 apply (clarify)
 apply (simp (no_asm_use) add: is_maxpref_def)
 apply (blast intro: prefix_append[THEN iffD2])
apply (clarify)
apply (rename_tac us uss)
apply (subgoal_tac "xs1=us")
 apply simp
apply simp
apply (simp (no_asm_use) add: is_maxpref_def)
apply (blast intro: prefix_append[THEN iffD2] prefix_order.antisym)
done

end
```

12 Automata based scanner

```
theory AutoMaxChop
imports DA MaxChop
begin

primrec auto_split :: "('a,'s)da ⇒ 's  ⇒ 'a list * 'a list ⇒ 'a list
⇒ '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 ⇒ 'a chopper" where
"auto_chop A = chop (λxs. auto_split A (start A) ([],xs) [] xs)"


lemma delta_snoc: "delta A (xs@[y]) q = next A y (delta A xs q)"
by simp

lemma auto_split_lemma:
 "⋀q ps res. auto_split A (delta A ps q) res ps xs =
              maxsplit (λys. fin A (delta A ys q)) res ps xs"
apply (induct xs)
 apply simp
apply (simp add: delta_snoc[symmetric] del: delta_append)
```

```
done

lemma auto_split_is_maxsplit:
 "auto_split A (start A) res [] xs = maxsplit (accepts A) res [] xs"
apply (unfold accepts_def)
apply (subst delta_Nil[where ?s = "start A", symmetric])
apply (subst auto_split_lemma)
apply simp
done

lemma is_maxsplitter_auto_split:
 "is_maxsplitter (accepts A) (λxs. auto_split A (start A) ([],xs) [] xs)"
by (simp add: auto_split_is_maxsplit is_maxsplitter_maxsplit)


lemma is_maxchopper_auto_chop:
 "is_maxchopper (accepts A) (auto_chop A)"
apply (unfold auto_chop_def)
apply (rule is_maxchopper_chop)
apply (rule is_maxsplitter_auto_split)
done

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 ⇒ nat ⇒ nat"

abbreviation
  deltas :: "'a nat_next ⇒ 'a list ⇒ nat ⇒ nat" where
  "deltas ≡ foldl2"

primrec trace :: "'a nat_next ⇒ nat ⇒ 'a list ⇒ 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 ⇒ nat ⇒ nat ⇒ nat ⇒ '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 ∪
  (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=[])"
by (cases xs) simp_all

lemma in_set_butlast_concatI:
 "x:set(butlast xs) ⟹ xs:set xss ⟹ x:set(butlast(concat xss))"
apply (induct "xss")
 apply simp
apply (simp add: butlast_append del: ball_simps)
apply (rule conjI)
 apply (clarify)
 apply (erule disjE)
  apply (blast)
 apply (subgoal_tac "xs=[]")
  apply simp
 apply (blast)
apply (blast dest: in_set_butlastD)
done



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))"
apply (induct "xs")
 apply (simp)
apply (rename_tac a as)
apply (intro strip)
apply (case_tac "d a i = k")
 apply (rule_tac x = "[a]" in exI)
```

```
 apply simp
 apply (case_tac "k : set(trace d (d a i) as)")
  apply (erule allE)
  apply (erule impE)
   apply (assumption)
  apply (erule exE)+
  apply (rule_tac x = "pref#mids" in exI)
  apply (rule_tac x = "suf" in exI)
  apply simp
 apply (rule_tac x = "[]" in exI)
 apply (rule_tac x = "as" in exI)
 apply simp
 apply (blast dest: in_set_butlastD)
apply simp
apply (erule allE)
apply (erule impE)
 apply (assumption)
apply (erule exE)+
apply (rule_tac x = "a#pref" in exI)
apply (rule_tac x = "mids" in exI)
apply (rule_tac x = "suf" in exI)
apply simp
done

lemma length_trace[simp]: "⋀i. length(trace d i xs) = length xs"
by (induct "xs") simp_all

lemma deltas_append[simp]:
  "⋀i. deltas d (xs@ys) i = deltas d ys (deltas d xs i)"
by (induct "xs") simp_all

lemma trace_append[simp]:
  "⋀i. trace d i (xs@ys) = trace d i xs @ trace d (deltas d xs i) ys"
by (induct "xs") simp_all

lemma trace_concat[simp]:
 "(∀xs ∈ set xss. deltas d xs i = i) ⟹
  trace d i (concat xss) = concat (map (trace d i) xss)"
by (induct "xss") simp_all

lemma trace_is_Nil[simp]: "⋀i. (trace d i xs = []) = (xs = [])"
by (case_tac "xs") simp_all

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)"
apply (case_tac "xs")
apply simp_all
apply (blast)
```

```
done

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))))"
apply (induct "xs")
 apply (simp)
apply (simp add: insert_commute)
done

lemma deltas_concat[simp]:
 "(∀mid∈set mids. deltas d mid k = k) ⟹ deltas d (concat mids) k =
k"
by (induct mids) simp_all

lemma lem: "[| n < Suc k; n ≠ k |] ==> n < k"
by arith

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)"
apply (induct k)
 apply(simp split: list.split)
 apply(fastforce)
apply (simp add: conc_def)
apply (rule iffI)
 apply (erule disjE)
  apply simp
 apply (erule exE conjE)+
 apply simp
 apply (subgoal_tac
      "(∀m∈set(butlast(trace d k xsb)). m < Suc k) ∧ deltas d xsb k =
k")
  apply (simp add: set_trace_conv butlast_append ball_Un)
 apply (erule star_induct)
  apply (simp)
 apply (simp add: set_trace_conv butlast_append ball_Un)
apply (case_tac "k : set(butlast(trace d i xs))")
 prefer 2 apply (rule disjI1)
 apply (blast intro:lem)
apply (rule disjI2)
apply (drule in_set_butlastD[THEN decompose])
apply (clarify)
apply (rule_tac x = "pref" in exI)
apply simp
apply (rule conjI)
 apply (rule ballI)
 apply (rule lem)
  prefer 2 apply simp
```

```
 apply (drule bspec) prefer 2 apply assumption
 apply simp
apply (rule_tac x = "concat mids" in exI)
apply (simp)
apply (rule conjI)
 apply (rule concat_in_star)
 apply (clarsimp simp: subset_iff)
 apply (rule lem)
  prefer 2 apply simp
 apply (drule bspec) prefer 2 apply assumption
 apply (simp add: image_eqI in_set_butlast_concatI)
apply (rule ballI)
apply (rule lem)
 apply auto
done

lemma trace_below:
 "bounded d k ⟹ ∀i. i < k ⟶ (∀n∈set(trace d i xs). n < k)"
apply (unfold bounded_def)
apply (induct "xs")
 apply simp
apply (simp (no_asm))
apply (blast)
done

lemma regset_below:
 "[| bounded d k; i < k; j < k |] ==>
  regset d i j k = {xs. deltas d xs i = j}"
apply (rule set_eqI)
apply (simp add: regset_spec)
apply (blast dest: trace_below in_set_butlastD)
done

lemma deltas_below:
 "⋀i. bounded d k ⟹ i < k ⟹ deltas d w i < k"
apply (unfold bounded_def)
apply (induct "w")
 apply simp_all
done

lemma regset_DA_equiv:
 "[| bounded (next A) k; start A < k; j < k |] ==>
  w : regset_of_DA A k = accepts A w"
apply(unfold regset_of_DA_def)
apply (simp cong: conj_cong
            add: regset_below deltas_below accepts_def delta_def)
done

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.