Automata

Thomas Tuerk (tuerk@in.tum.de)

February 6, 2012

Contents

1 Labeled transition systems **2**
1.1 Auxiliary Definitions . . . 2
1.2 Basic Definitions . . . 3
1.2.1 Transition Relations . . . 3
1.2.2 Paths . . . 3
1.2.3 Reachability . . . 3
1.3 Deterministic Transition Relations . . . 5
1.3.1 Basic Definitions for DLTSs . . . 6
1.4 Reachablity on Graphs . . . 7
1.5 Restricting and extending the transition relation . . . 9
1.6 Products . . . 9

2 Nondeterministic Finite Automata **10**
2.1 Basic Definitions . . . 10
2.2 Constructing from a list representation . . . 13
2.3 Removing states . . . 15
2.4 Rename States / Combining . . . 18
2.5 Isomorphy . . . 23
2.6 Efficient Construction of NFAs . . . 29
2.7 Renaming letters (i.e. elements of the alphabet) . . . 38
2.8 Normalise states . . . 40
2.9 Product automata . . . 41
2.10 Reversal . . . 45
2.11 Right quotient . . . 46

3 Deterministic Finite Automata **48**
3.1 Basic Definitions . . . 48
3.2 The unique initial state . . . 49
3.3 The unique transition function . . . 49
3.4 Lemmas about deterministic automata . . . 50
3.5 Determinisation . . . 53
3.6 Right quotient . . . 58
3.7 Complement . . . 58
3.8 Boolean Combinations of NFAs . . . 60
3.9 Minimisation . . . 62
3.10 Brzozowski's Algorithm . . . 72
3.11 Abstract Minimisation Function . . . 75

4 Hopcroft's Minimisation Algorithm **76**
4.1 Main idea . . . 76
4.2 Basic notions . . . 78
4.2.1 Partitions . . . 78
4.2.2 Partitions and Equivalence Relations . . . 82

4.2.3 Weak Equivalence Partitions . . . 84
4.2.4 Initial partition . . . 89
4.2.5 Splitting Partitions . . . 89
4.3 Naive implementation . . . 96
4.4 Abstract implementation . . . 97
4.4.1 Splitting whole Partitions . . . 99
4.4.2 Updating the set of Splitters . . . 105
4.4.3 The abstract Algorithm . . . 110
4.5 Implementing step . . . 117
4.6 Precomputing Predecessors . . . 124
4.7 Data Refinement . . . 124
4.8 Code Generation . . . 141

5 Presburger Adaptation **150**
5.1 DFAs for Diophantine Equations and Inequations . . . 151
5.1.1 Definition . . . 151
5.1.2 Properties . . . 152
5.1.3 Efficiency . . . 157
5.1.4 Implementation . . . 157
5.2 Existential Quantification . . . 158
5.3 Universal Quantification . . . 162
5.4 Translation . . . 163
5.5 Code Generation . . . 164

1 Labeled transition systems

theory *LTS*
imports *Main*
begin

This theory defines labeled transition systems (LTS).

1.1 Auxiliary Definitions

lemma *lists-product* :
  $w \in lists\ (\Sigma 1 \times \Sigma 2) \longleftrightarrow (map\ fst\ w) \in lists\ \Sigma 1 \wedge (map\ snd\ w) \in lists\ \Sigma 2$
by (*induct w, auto*)

lemma *lists-inter* : $w \in lists\ (\Sigma 1 \cap \Sigma 2) = (w \in lists\ \Sigma 1 \wedge w \in lists\ \Sigma 2)$
  by (*simp add: lists-eq-set*)

lemma *lists-rev* [*simp*] :
  $(rev\ w) \in lists\ \Sigma \longleftrightarrow w \in lists\ \Sigma$
by *auto*

lemma *rev-image-lists* [*simp*] :
  rev ‘ $(lists\ A) = lists\ A$
proof (*rule set-eqI*)
  fix *xs*
  have $xs \in rev$ ‘ $(lists\ A) \longleftrightarrow rev\ xs \in lists\ A$
    apply (*simp add: image-iff Bex-def*)
    apply (*metis rev-swap lists-rev*)
  done
  thus $xs \in rev$ ‘ $lists\ A \longleftrightarrow xs \in lists\ A$
    by *simp*
qed

1.2 Basic Definitions

1.2.1 Transition Relations

Given a set of states $\mathcal{Q}$ and an alphabet Σ, a labeled transition system is a subset of $\mathcal{Q} \times \Sigma \times \mathcal{Q}$. Given such a relation $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$, a triple (q, σ, q') is an element of Δ iff starting in state q the state q' can be reached reading the label σ.

type-synonym $('q,'a)$ *LTS* = $('q * 'a * 'q)$ *set*

1.2.2 Paths

Given a word $w = w_1 \ldots w_n$ over Σ then a word $p = p_0 \ldots p_n$ over $\mathcal{Q}$ is called *a path of* w *through* Δ, iff $(p_i, w_{i+1}, p_{i+1}) \in \Delta$ holds for all $0 \leq i < n$. This definition of paths requires however to always keep track of the lengths of p and r. This leads to complicated simplication rules. Therefore, the following definition uses p_0 and the combined word $(w_1, r_1) \ldots (w_n, r_n)$ over $\Sigma \times \mathcal{Q}$ for the definition of paths.

fun *LTS-is-path* :: $('q,'a)$ *LTS* $\Rightarrow$ $'q$ $\Rightarrow$ $('a\ *\ 'q)$ *list* $\Rightarrow$ *bool* **where**
  LTS-is-path Δ *q* [] = *True*
| *LTS-is-path* Δ *q* $(\sigma q'$ # *xs*) = ((*q*, *fst* $\sigma q'$, *snd* $\sigma q'$) $\in \Delta \wedge$ *LTS-is-path* Δ (*snd* $\sigma q'$) *xs*)

Using this definition, it is easy to prove some properties of paths. For example lemmata about concatenating paths are easy to proof.

lemma *LTS-is-path-concat* :
  $\pi \neq [] \Longrightarrow$
   (*LTS-is-path* Δ *q* $(\pi$ @ $\pi')$ = (*LTS-is-path* Δ *q* $\pi \wedge$ *LTS-is-path* Δ (*snd* (*last* π)) π'))
by (*induct* π *arbitrary*: *q rule*: *list-nonempty-induct*, *simp-all*)

1.2.3 Reachability

Often it is enough to consider just the first and last state of a path. This leads to the following definition of reachability. Notice, that *LTS-is-reachable* Δ is the reflexive, transitive closure of Δ.

fun *LTS-is-reachable* :: $('q,\ 'a)$ *LTS* $\Rightarrow$ $'q$ $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'q$ $\Rightarrow$ *bool* **where**
   LTS-is-reachable Δ *q* [] q' = $(q = q')$
 | *LTS-is-reachable* Δ *q* $(\sigma$ # *w*) q' =
     $(\exists\, q''.\ (q,\ \sigma,\ q'') \in \Delta \wedge$ *LTS-is-reachable* Δ q'' *w* $q')$

Now let's show the connection with *LTS-is-path*. In order to establish the connection, a few auxiliary definitions are introduced first.

definition *LTS-path* :: $('a\ *\ 'q)$ *list* $\Rightarrow$ $'q$ *list* **where**
  LTS-path $\pi \equiv$ *map snd* π

definition *LTS-path-last-S* :: $'q \Rightarrow ('a\ *\ 'q)$ *list* $\Rightarrow$ $'q$ **where**
  LTS-path-last-S *q* $\pi \equiv$ *last* (*q* # (*LTS-path* π))

definition *LTS-path-L* :: $('a\ *\ 'q)$ *list* $\Rightarrow$ $'a$ *list* **where**
  LTS-path-L $\pi \equiv$ *map fst* π

lemma [*simp*] : *LTS-path* [] = [] **by** (*simp add*: *LTS-path-def*)
lemma [*simp*] : *LTS-path-L* [] = [] **by** (*simp add*: *LTS-path-L-def*)
lemma [*simp*] : *LTS-path* $(\sigma q$ # $\pi)$ = (*snd* σq # *LTS-path* π) **by** (*simp add*: *LTS-path-def*)
lemma [*simp*] : *LTS-path-L* $(\sigma q$ # $\pi)$ = (*fst* σq # *LTS-path-L* π) **by** (*simp add*: *LTS-path-L-def*)
lemma [*simp*] : *LTS-path* π = [] $\longleftrightarrow$ π = [] **by** (*simp add*: *LTS-path-def*)
lemma [*simp*] : *LTS-path-L* π = [] $\longleftrightarrow$ π = [] **by** (*simp add*: *LTS-path-L-def*)
lemma [*simp*] : *LTS-path-last-S* *q* [] = *q* **by** (*simp add*: *LTS-path-last-S-def LTS-path-def*)
lemma [*simp*] : *LTS-path-last-S* *q* (*x* # *xs*) = *LTS-path-last-S* (*snd x*) *xs*
   by (*simp add*: *LTS-path-last-S-def LTS-path-def*)

lemma *path-last-S-in* :
  LTS-path-last-S *q* $\pi \in$ (*insert q* (*set* (*LTS-path* π)))
by (*induct* π *arbitrary*: *q*, *auto*)

lemma *LTS-is-reachable-alt-def* :
LTS-is-reachable $\Delta\ q\ w\ q' = (\exists \pi.$ *LTS-is-path* $\Delta\ q\ \pi \wedge$ *LTS-path-last-S* $q\ \pi = q' \wedge$ *LTS-path-L* $\pi = w)$
(**is** - $= (\exists \pi.$ *?pathPred* $q\ w\ \pi))$
proof (*induct w arbitrary*: q)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* σ w) **note** *ind-hyp* = *this*
show *?case*
proof
assume *LTS-is-reachable* $\Delta\ q\ (\sigma \# w)\ q'$
then obtain q'' **where**
reach-w: *LTS-is-reachable* $\Delta\ q''\ w\ q'$ **and**
step-Δ: $(q, \sigma, q'') \in \Delta$
by *auto*

from *reach-w ind-hyp* **obtain** π **where**
?pathPred $q''\ w\ \pi$ **by** *auto*
with *step-*Δ **have** *?pathPred* $q\ (\sigma \# w)\ ((\sigma, q'') \# \pi)$ **by** *simp*
thus $\exists \pi.$ *?pathPred* $q\ (\sigma \# w)\ \pi$ **by** *metis*
next
assume $\exists \pi.$ *?pathPred* $q\ (\sigma \# w)\ \pi$
then obtain π **where** *path-*σw: *?pathPred* $q\ (\sigma \# w)\ \pi$ **by** *auto*
then obtain $q''\ \pi'$ **where** π*-eq*: $\pi = (\sigma, q'') \# \pi'$
by (*cases* π, *auto*, *metis*)

from π*-eq path-*σw **have**
path-w: *?pathPred* $q''\ w\ \pi'$ **and**
step-Δ: $(q, \sigma, q'') \in \Delta$
by (*simp-all*)
from *path-w ind-hyp* **have** *LTS-is-reachable* $\Delta\ q''\ w\ q'$ **by** *auto*
with *step-*Δ **show** *LTS-is-reachable* $\Delta\ q\ (\sigma \# w)\ q'$ **by** *auto*
qed
qed

lemma *LTS-is-reachable-concat* :
LTS-is-reachable $\Delta\ q\ (w\ @\ w')\ q' =$
$(\exists q''.$ *LTS-is-reachable* $\Delta\ q\ w\ q'' \wedge$ *LTS-is-reachable* $\Delta\ q''\ w'\ q')$
by (*induct w arbitrary*: q, *auto*)

lemma *LTS-is-reachable-snoc* [*simp*] :
LTS-is-reachable $\Delta\ q\ (w\ @\ [\sigma])\ q' =$
$(\exists q''.$ *LTS-is-reachable* $\Delta\ q\ w\ q'' \wedge (q'', \sigma, q') \in \Delta)$
by (*simp add*: *LTS-is-reachable-concat*)

Unreachability is often interesting as well.

definition *LTS-is-unreachable* **where**
LTS-is-unreachable $\Delta\ \Sigma\ q\ q' \longleftrightarrow \neg(\exists w \in$ *lists* $\Sigma.$ *LTS-is-reachable* $\Delta\ q\ w\ q')$

lemma *LTS-is-unreachable-not-refl* [*simp*] :
$\neg($*LTS-is-unreachable* $\Delta\ \Sigma\ q\ q)$
proof $-$
have *LTS-is-reachable* $\Delta\ q\ []\ q \wedge [] \in$ *lists* Σ **by** *simp*
thus $\neg($*LTS-is-unreachable* $\Delta\ \Sigma\ q\ q)$
by (*metis LTS-is-unreachable-def*)
qed

lemma *LTS-is-unreachable-reachable-start* :
assumes *unreach-q-q'*: *LTS-is-unreachable* $\Delta\ \Sigma\ q\ q'$
and *reach-q-q''*: *LTS-is-reachable* $\Delta\ q\ w\ q''$
and *w-in-*Σ: $w \in$ *lists* Σ
shows *LTS-is-unreachable* $\Delta\ \Sigma\ q''\ q'$

proof (*rule ccontr*)
  assume $\neg$ (*LTS-is-unreachable* Δ Σ q'' q')
  then obtain w' **where** w'*-in-*Σ: $w' \in$ *lists* Σ **and**
    reach-q''*-*q': *LTS-is-reachable* Δ q'' w' q'
    by (*auto simp add*: *LTS-is-unreachable-def*)

  from *reach-q-*q'' *reach-*q''*-*q' **have**
    reach-q-q' : *LTS-is-reachable* Δ q (w @ w') q' **by** (*auto simp add*: *LTS-is-reachable-concat*)
  from *w-in-*Σ w'*-in-*Σ **have** ww'*-in-*Σ: (w @ w') $\in$ *lists* Σ **by** *simp*

  from ww'*-in-*Σ *reach-q-*q' *unreach-q-*q' **show** *False*
    by (*metis LTS-is-unreachable-def*)
qed

lemma *LTS-is-unreachable-reachable-end* :
  $[\![$*LTS-is-unreachable* Δ Σ q q'; *LTS-is-reachable* Δ q'' w q'; $w \in$ *lists* $\Sigma]\!] \Longrightarrow$
    LTS-is-unreachable Δ Σ q q''
by (*metis LTS-is-unreachable-def LTS-is-unreachable-reachable-start*)

1.3 Deterministic Transition Relations

Often, one is interested in deterministic transition systems. A transition system Δ is deterministic iff given a start state q and a label σ there is at most one successor state q' such that $(q, \sigma, q') \in \Delta$ holds. In the following δ is used to denote deterministic transition systems. Therefore, these transition systems are functions.

definition *LTS-is-weak-deterministic* :: ($'q$, $'a$) *LTS* $\Rightarrow$ *bool* **where**
  LTS-is-weak-deterministic $\Delta = (\forall\, q\ \sigma\ q1'\ q2'.\ ((q,\ \sigma,\ q1') \in \Delta \wedge (q,\ \sigma,\ q2') \in \Delta) \dashrightarrow (q1' = q2'))$

definition *LTS-is-deterministic* :: $'q$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ ($'q$, $'a$) *LTS* $\Rightarrow$ *bool* **where**
  LTS-is-deterministic $\mathcal{Q}$ Σ $\Delta \equiv$ (*LTS-is-weak-deterministic* $\Delta \wedge (\forall\, q{\in}\mathcal{Q}.\ \forall\, \sigma{\in}\Sigma.\ \exists\, q'.\ (q,\ \sigma,\ q') \in \Delta)$)

type-synonym ($'q$,$'a$) *DLTS* = $'q$ $*$ $'a$ $\Rightarrow$ $'q$ *option*

definition *DLTS-to-LTS* :: ($'q$, $'a$) *DLTS* $\Rightarrow$ ($'q$, $'a$) *LTS*
where *DLTS-to-LTS* $\delta \equiv \{(q,\ \sigma,\ q') \mid q\ \sigma\ q'.\ \delta(q,\ \sigma) =$ *Some* $q'\}$

lemma *DLTS-to-LTS-alt-def* [*simp*]:
  $(q,\ \sigma,\ q') \in$ *DLTS-to-LTS* $\delta \longleftrightarrow (\delta(q,\ \sigma) =$ *Some* $q')$
by (*auto simp add*: *DLTS-to-LTS-def*)

lemma *DLTS-to-LTS---LTS-is-weak-deterministic* [*simp*] :
  LTS-is-weak-deterministic (*DLTS-to-LTS* δ)
by (*auto simp add*: *LTS-is-weak-deterministic-def*)

lemma *DLTS-to-LTS---LTS-is-deterministic* :
  $[\![\bigwedge q\ \sigma.\ [\![q \in \mathcal{Q};\ \sigma \in \Sigma]\!] \Longrightarrow \neg\ (\delta\ (q,\ \sigma) =$ *None*$)]\!] \Longrightarrow$
   LTS-is-deterministic $\mathcal{Q}$ Σ (*DLTS-to-LTS* δ)
by (*auto simp add*: *LTS-is-deterministic-def*)

definition *LTS-to-DLTS* :: ($'q$, $'a$) *LTS* $\Rightarrow$ ($'q$, $'a$) *DLTS* **where**
  LTS-to-DLTS $\Delta \equiv (\lambda(q,\sigma).$ *if* $(\exists\, q'.\ (q,\sigma,q') \in \Delta)$ *then Some* (*SOME* $q'.\ (q,\ \sigma,\ q') \in \Delta$) *else None*)

lemma *LTS-to-DLTS-in* :
  $(q,\ \sigma,\ q') \in \Delta \Longrightarrow (\exists\, q''.$ *LTS-to-DLTS* Δ $(q,\sigma) =$ *Some* $q'' \wedge (q,\ \sigma,\ q'') \in \Delta)$
by (*auto simp add*: *LTS-to-DLTS-def*,
    rule someI2, *simp-all*)

lemma *LTS-to-DLTS-is-some* :
  LTS-to-DLTS Δ $(q,\sigma) =$ *Some* $q' \Longrightarrow (q,\ \sigma,\ q') \in \Delta$
apply (*auto simp add*: *LTS-to-DLTS-def split-if-eq1*)
apply (*rule-tac someI-ex*)
apply *auto*

done

lemma *LTS-to-DLTS-is-none* :
  $(LTS\text{-}to\text{-}DLTS\ \Delta\ (q,\sigma) = None) \longleftrightarrow (\forall q'.\ (q,\ \sigma,\ q') \notin \Delta)$
by (*simp add*: *LTS-to-DLTS-def split-if-eq1*)

lemma *LTS-to-DLTS-is-some-det* :
  $LTS\text{-}is\text{-}weak\text{-}deterministic\ \Delta \Longrightarrow$
   $((LTS\text{-}to\text{-}DLTS\ \Delta\ (q,\ \sigma) = Some\ q') \longleftrightarrow (q,\ \sigma,\ q') \in \Delta)$
by (*metis LTS-to-DLTS-in LTS-is-weak-deterministic-def LTS-to-DLTS-is-some*)

lemma *DLTS-to-LTS-inv* [*simp*] :
   $LTS\text{-}to\text{-}DLTS\ (DLTS\text{-}to\text{-}LTS\ \delta) = \delta$
apply (*subst fun-eq-iff*, *clarify*)
apply (*auto simp add*: *DLTS-to-LTS-def LTS-to-DLTS-def*)
done

lemma *LTS-to-DLTS-inv* :
   $LTS\text{-}is\text{-}weak\text{-}deterministic\ \Delta \Longrightarrow$
    $DLTS\text{-}to\text{-}LTS\ (LTS\text{-}to\text{-}DLTS\ \Delta) = \Delta$
by (*simp add*: *DLTS-to-LTS-def LTS-to-DLTS-is-some-det*, *auto*)

lemma *LTS-is-reachable---LTS-is-weak-deterministic* :
assumes *det*-Δ: $LTS\text{-}is\text{-}weak\text{-}deterministic\ \Delta$
shows $[\![LTS\text{-}is\text{-}reachable\ \Delta\ q\ w\ q';\ LTS\text{-}is\text{-}reachable\ \Delta\ q\ w\ q'']\!] \Longrightarrow (q' = q'')$
using *assms*
apply (*induct w arbitrary*: *q*)
apply (*auto simp add*: *LTS-is-weak-deterministic-def*)
apply *metis*
done

1.3.1 Basic Definitions for DLTSs

fun *DLTS-path-of* :: $('q,'a)\ DLTS \Rightarrow 'q \Rightarrow 'a\ list \Rightarrow ('q\ list)\ option$ **where**
  $DLTS\text{-}path\text{-}of\ \delta\ q\ [] = Some\ []$
| $DLTS\text{-}path\text{-}of\ \delta\ q\ (\sigma\ \#\ w) =$
   $Option.bind\ (\delta\ (q,\ \sigma))\ (\lambda q'.\ Option.map\ (\lambda ql.\ q'\ \#\ ql)\ (DLTS\text{-}path\text{-}of\ \delta\ q'\ w))$

lemma *LTS-is-path---DLTS-path-of* :
shows $LTS\text{-}is\text{-}path\ (DLTS\text{-}to\text{-}LTS\ \delta)\ q\ wr =$
    $(DLTS\text{-}path\text{-}of\ \delta\ q\ (LTS\text{-}path\text{-}L\ wr) = Some\ (LTS\text{-}path\ wr))$
proof (*induct wr arbitrary*: *q*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons* $\sigma q'$ *wr*)
  note *ind-hyp* = *this*

  obtain $\sigma\ q'$ **where** $\sigma q'$*-eq*: $\sigma q' = (\sigma,\ q')$ **by** (*cases* $\sigma q'$, *blast*)

  thus *?case*
  proof (*cases* $\delta\ (q,\ \sigma)$)
    case *None* **thus** *?thesis* **by** (*simp add*: $\sigma q'$*-eq*)
  next
    case (*Some* q'')
    thus *?thesis*
      by (*auto simp add*: $\sigma q'$*-eq ind-hyp*)
  qed
qed

fun *DLTS-reach* :: $('q,\ 'a)\ DLTS \Rightarrow 'q \Rightarrow 'a\ list \Rightarrow 'q\ option$ **where**
   $DLTS\text{-}reach\ \delta\ q\ [] = Some\ q$

| *DLTS-reach* δ *q* (σ # *w*) =
Option.bind (δ (*q*, σ)) ($\lambda q'$. *DLTS-reach* δ *q'* *w*)

lemma *DLTS-reach-Concat* [*simp*] :
DLTS-reach δ *q* (*w* @ *w'*) = *Option.bind* (*DLTS-reach* δ *q* *w*) ($\lambda q'$. *DLTS-reach* δ *q'* *w'*)
by (*induct w arbitrary*: *q*, *auto*)

lemma *DLTS-reach-alt-def* :
DLTS-reach δ *q* *w* = *Option.map* (λwl. *last* (*q* # *wl*)) (*DLTS-path-of* δ *q* *w*)
proof (*induct w arbitrary*: *q*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* σ *w*)
note *ind-hyp* = *this*

show *?case*
proof (*cases* δ (*q*, σ))
case *None* **thus** *?thesis* **by** *simp*
next
case (*Some* *q'*)
thus *?thesis*
by (*simp add*: *ind-hyp option-map-comp o-def*)
qed
qed

lemma *LTS-is-reachable-weak-determistic* :
assumes *det-*Δ: *LTS-is-weak-deterministic* Δ
shows *LTS-is-reachable* Δ *q w q'* $\longleftrightarrow$
DLTS-reach (*LTS-to-DLTS* Δ) *q w* = *Some q'*
proof (*induct w arbitrary*: *q*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* σ *w*)
note *ind-hyp* = *this*
note *in-*Δ = *LTS-to-DLTS-is-some-det* [*OF det-*Δ, *symmetric*]

show *?case*
proof (*cases LTS-to-DLTS* Δ (*q*, σ))
case *None* **thus** *?thesis* **by** (*simp add*: *in-*Δ)
next
case (*Some q'*) **thus** *?thesis* **by** (*simp add*: *ind-hyp in-*Δ)
qed
qed

lemma *LTS-is-reachable-DLTS-to-LTS* [*simp*] :
LTS-is-reachable (*DLTS-to-LTS* δ) *q w q'* =
(*DLTS-reach* δ *q w* = *Some q'*)
by (*metis LTS-is-reachable-weak-determistic DLTS-to-LTS-inv*
DLTS-to-LTS---LTS-is-weak-deterministic)

1.4 Reachablity on Graphs

definition *LTS-forget-labels-pred* :: ($'a \Rightarrow$ *bool*) $\Rightarrow$ ($'q$, $'a$) *LTS* $\Rightarrow$ ($'q \times {'q}$) *set* **where**
LTS-forget-labels-pred P Δ = $\{(q, q')$. $\exists\sigma$. $(q, \sigma, q') \in \Delta \wedge P\ \sigma\}$

definition *LTS-forget-labels* :: ($'q$, $'a$) *LTS* $\Rightarrow$ ($'q \times {'q}$) *set* **where**
LTS-forget-labels Δ = $\{(q, q')$. $\exists\sigma$. $(q, \sigma, q') \in \Delta\}$

lemma *LTS-forget-labels-alt-def* :
LTS-forget-labels Δ = *LTS-forget-labels-pred* ($\lambda\sigma$. *True*) Δ
unfolding *LTS-forget-labels-def LTS-forget-labels-pred-def*
by *simp*

lemma *rtrancl-LTS-forget-labels-pred* :
  rtrancl (*LTS-forget-labels-pred P* Δ) =
    $\{(q,\ q')$. ($\exists$ *w*. *LTS-is-reachable* Δ *q w q'* $\wedge$ *w* $\in$ *lists* $\{\sigma.\ P\ \sigma\})\}$
  (**is** *?ls* = *?rs*)
proof (*intro set-eqI*)
  fix *qq'* :: ('*a* $\times$ '*a*)
  obtain *q q'* **where** *qq'-eq* : *qq'* = (*q*, *q'*) **by** (*cases qq'*, *auto*)

  have *imp1* : (*q*, *q'*) $\in$ *?ls* $\Longrightarrow$ (*q*,*q'*) $\in$ *?rs*
  proof (*induct rule*: *rtrancl-induct*)
    case *base* **thus** *?case*
      apply *simp*
      apply (*rule exI* [**where** *x* = []])
      apply *simp*
    done
  next
    case (*step q' q''*)
    then obtain *w* σ **where**
      qq'-ind : *LTS-is-reachable* Δ *q w q'* **and**
      w-in: *w* $\in$ *lists* $\{\sigma.\ P\ \sigma\}$ **and**
      q'q''-ind: (*q'*, σ, *q''*) $\in$ Δ **and**
      σ*-in*: *P* σ **unfolding** *LTS-forget-labels-pred-def* **by** *auto*
    hence *LTS-is-reachable* Δ *q* (*w* @ [σ]) *q''* $\wedge$ (*w* @ [σ]) $\in$ *lists* $\{\sigma.\ P\ \sigma\}$ **by** *auto*
    hence $\exists$ *w*. *LTS-is-reachable* Δ *q w q''* $\wedge$ *w* $\in$ *lists* $\{\sigma.\ P\ \sigma\}$ **by** *blast*
    thus *?case* **by** *simp*
  qed

  have *imp2* : $\bigwedge$*w*. $[\![$*LTS-is-reachable* Δ *q w q'*; *w* $\in$ *lists* $\{\sigma.\ P\ \sigma\}]\!] \Longrightarrow$ (*q*,*q'*) $\in$ *?ls*
  proof −
    fix *w* **show** $[\![$*LTS-is-reachable* Δ *q w q'*; *w* $\in$ *lists* $\{\sigma.\ P\ \sigma\}]\!] \Longrightarrow$ (*q*,*q'*) $\in$ *?ls*
    proof (*induct w arbitrary*: *q*)
      case *Nil* **thus** *?case* **by** *simp*
    next
      case (*Cons* σ *w*)
      then obtain *q''* **where**
        in-D: (*q*, σ, *q''*) $\in$ Δ **and**
        σ*-P*: *P* σ **and**
        in-trcl: (*q''*, *q'*) $\in$ *rtrancl* (*LTS-forget-labels-pred P* Δ)
       by *auto*

      from *in-D* σ*-P* **have** *in-D'*: (*q*, *q''*) $\in$ (*LTS-forget-labels-pred P* Δ)
        unfolding *LTS-forget-labels-pred-def* **by** *auto*

      note *converse-rtrancl-into-rtrancl* [*OF in-D' in-trcl*]
      thus *?case* **by** *auto*
    qed
  qed

  from *imp1 imp2 qq'-eq* **show** (*qq'* $\in$ *?ls*) = (*qq'* $\in$ *?rs*) **by** *auto*
qed

lemma *rtrancl-LTS-forget-labels* :
  rtrancl (*LTS-forget-labels* Δ) =
   $\{(q,\ q')$. ($\exists$ *w*. *LTS-is-reachable* Δ *q w q'*)$\}$
by (*simp add*: *LTS-forget-labels-alt-def*
             rtrancl-LTS-forget-labels-pred)

definition *LTS-rename-forget-labels* :: ('*q* $\Rightarrow$ '*q2*) $\Rightarrow$ ('*q*, '*a*) *LTS* $\Rightarrow$
  ('*q2* $\times$ '*q2*) *set* **where**
LTS-rename-forget-labels f Δ = $\{(f\ q,\ f\ q') \mid q\ \sigma\ q'.\ (q,\ \sigma,\ q') \in \Delta\}$

lemma *LTS-rename-forget-labels-alt-def* :
  LTS-rename-forget-labels f Δ =
   $\{(f\ q,\ f\ q') \mid q\ q'.\ (q,\ q') \in$ *LTS-forget-labels* $\Delta\}$
unfolding *LTS-forget-labels-def LTS-rename-forget-labels-def*
by *auto*

1.5 Restricting and extending the transition relation

lemma *LTS-is-path-mono* :
  $[\![\Delta \subseteq \Delta';$ *LTS-is-path* $\Delta\ q\ \pi]\!] \Longrightarrow$ *LTS-is-path* $\Delta'\ q\ \pi$
by (*induct* π *arbitrary*: q, *auto*)

lemma *LTS-is-reachable-mono* :
  $[\![\Delta \subseteq \Delta';$ *LTS-is-reachable* $\Delta\ q\ w\ q']\!] \Longrightarrow$ *LTS-is-reachable* $\Delta'\ q\ w\ q'$
by (*induct* w *arbitrary*: q, *auto*)

1.6 Products

definition *LTS-product* :: $('p,'a)$ *LTS* $\Rightarrow$ $('q,'a)$ *LTS* $\Rightarrow$ $('p*'q,'a)$ *LTS* **where**
  LTS-product $\Delta 1\ \Delta 2 = \{((p,q),\ \sigma,\ (p',q')) \mid p\ p'\ \sigma\ q\ q'.\ (p,\ \sigma,\ p') \in \Delta 1 \wedge (q,\ \sigma,\ q') \in \Delta 2\}$

lemma *LTS-product-elem* :
  $((p,q),\ \sigma,\ (p',q')) \in$ *LTS-product* $\Delta 1\ \Delta 2 = ((p,\sigma,p') \in \Delta 1 \wedge (q,\sigma,q') \in \Delta 2)$
by (*simp add*: *LTS-product-def*)

lemma *LTS-product-alt-def* [*simp*] :
  $x \in$ *LTS-product* $\Delta 1\ \Delta 2$ =
   ((*fst* (*fst* x),*fst* (*snd* x), *fst* (*snd* (*snd* x))) $\in \Delta 1 \wedge$
    (*snd* (*fst* x),*fst* (*snd* x), *snd* (*snd* (*snd* x))) $\in \Delta 2$)
by (*cases* x, *case-tac* a, *simp add*: *LTS-product-elem*)

definition *LTS-product-path1* **where** *LTS-product-path1* π = *map* (% x. (*fst* x, *fst* (*snd* x))) π
definition *LTS-product-path2* **where** *LTS-product-path2* π = *map* (% x. (*fst* x, *snd* (*snd* x))) π

lemma *LTS-is-path-product-iff*: *LTS-is-path* (*LTS-product* $\Delta 1\ \Delta 2$) *pq* $\pi \longleftrightarrow$
  LTS-is-path $\Delta 1$ (*fst pq*) (*LTS-product-path1* π) $\wedge$ *LTS-is-path* $\Delta 2$ (*snd pq*) (*LTS-product-path2* π)
unfolding *LTS-product-path1-def LTS-product-path2-def*
by (*induct* π *arbitrary*: *pq*, *auto*)

lemma *LTS-product-path1-L* [*simp*]: *LTS-path-L* (*LTS-product-path1* π) = *LTS-path-L* π
  by (*simp add*: *LTS-product-path1-def LTS-path-L-def*)

lemma *LTS-product-path2-L* [*simp*]: *LTS-path-L* (*LTS-product-path2* π) = *LTS-path-L* π
  by (*simp add*: *LTS-product-path2-def LTS-path-L-def*)

lemma *LTS-product-path1-S*: *LTS-path* (*LTS-product-path1* π) = *map fst* (*LTS-path* π)
  by (*simp add*: *LTS-product-path1-def LTS-path-def*)

lemma *LTS-product-path2-S*: *LTS-path* (*LTS-product-path2* π) = *map snd* (*LTS-path* π)
  by (*simp add*: *LTS-product-path2-def LTS-path-def*)

lemma *LTS-product-path1-last-S*: *LTS-path-last-S* (*fst pq*) (*LTS-product-path1* π) = *fst* (*LTS-path-last-S pq* π)
  by (*metis LTS-path-last-S-def LTS-product-path1-S last-map list.simps*(*3*) *map.simps*(*2*))

lemma *LTS-product-path2-last-S*: *LTS-path-last-S* (*snd pq*) (*LTS-product-path2* π) = *snd* (*LTS-path-last-S pq* π)
  by (*metis LTS-path-last-S-def LTS-product-path2-S last-map list.simps*(*3*) *map.simps*(*2*))

lemma *LTS-is-reachable-product* :
   LTS-is-reachable (*LTS-product* $\Delta 1\ \Delta 2$) *pq* w *pq'* $\longleftrightarrow$
    (*LTS-is-reachable* $\Delta 1$ (*fst pq*) w (*fst pq'*) $\wedge$
     LTS-is-reachable $\Delta 2$ (*snd pq*) w (*snd pq'*))

by (*induct w arbitrary*: *pq*, *auto*)

lemma *LTS-product-LTS-is-weak-deterministic* :
 $[\![$*LTS-is-weak-deterministic* $\Delta 1$; *LTS-is-weak-deterministic* $\Delta 2$ $]\!] \Longrightarrow$
  LTS-is-weak-deterministic (*LTS-product* $\Delta 1$ $\Delta 2$)
unfolding *LTS-is-weak-deterministic-def LTS-product-def* **by** *fast*

lemma *LTS-product-LTS-is-deterministic* :
 $[\![$*LTS-is-deterministic* $\mathcal{Q}1$ $\Sigma 1$ $\Delta 1$; *LTS-is-deterministic* $\mathcal{Q}2$ $\Sigma 2$ $\Delta 2$ $]\!] \Longrightarrow$
  LTS-is-deterministic ($\mathcal{Q}1 \times \mathcal{Q}2$) ($\Sigma 1 \cap \Sigma 2$) (*LTS-product* $\Delta 1$ $\Delta 2$)
unfolding *LTS-is-weak-deterministic-def LTS-is-deterministic-def LTS-product-def*
 by *blast*

end

2 Nondeterministic Finite Automata

theory *NFA*
imports *Main LTS ../General/Accessible*
  ~~/src/HOL/Library/Nat-Bijection
begin

2.1 Basic Definitions

class *NFA-states* =
 fixes *states-enumerate* :: *nat* ⇒ *'a*
 assumes *states-enumerate-inj*: *inj states-enumerate*
begin
 lemma *states-enumerate-eq*: *states-enumerate n* = *states-enumerate m* ⟷ *n* = *m*
  using *states-enumerate-inj*
  unfolding *inj-on-def* **by** *auto*

 lemma *not-finite-NFA-states-UNIV* : ~(*finite* (*UNIV*::*'a set*))
 proof
  assume *fin-UNIV*: *finite* (*UNIV*::*'a set*)
  hence *finite* (*states-enumerate* ‘ *UNIV*)
   by (*rule-tac finite-subset*[*of* - *UNIV*], *simp-all*)
  hence *finite* (*UNIV*::*nat set*)
   using *states-enumerate-inj*
   by (*rule finite-imageD*)
  thus *False* **using** *infinite-UNIV-nat* **by** *simp*
 qed
end

instantiation *nat* :: *NFA-states*
begin
 definition *states-enumerate q* = *q*

 instance proof
  show *inj* (*states-enumerate*::*nat* ⇒ *nat*)
   unfolding *states-enumerate-nat-def-raw*
   by (*simp add*: *inj-on-def*)
 qed
end

This theory defines nondetermistic finite automata. These automata are represented as records containing a transtion relation, a set of initial states and a set of final states.

record (*'q*,*'a*) *NFA-rec* =
 $\mathcal{Q}$ :: *'q set*   — The set of states

Σ :: $'a$ *set* — The set of labels
Δ :: $('q,'a)$ *LTS* — The transition relation
$\mathcal{I}$ :: $'q$ *set* — The set of initial states
$\mathcal{F}$ :: $'q$ *set* — The set of final states

Using notions for labelled transition systems, it is easy to define the languages accepted by automata.

definition *NFA-accept* **where**
NFA-accept $\mathcal{A}$ w = (($w \in$ *lists* (Σ $\mathcal{A}$)) $\wedge$ ($\exists$ $q \in (\mathcal{I}$ $\mathcal{A})$. $\exists$ $q' \in (\mathcal{F}$ $\mathcal{A})$. *LTS-is-reachable* (Δ $\mathcal{A}$) q w q'))

definition $\mathcal{L}$ **where** $\mathcal{L}$ $\mathcal{A}$ = {w. *NFA-accept* $\mathcal{A}$ w}

It is also useful to define the language accepted in a state.

definition $\mathcal{L}$*-in-state* **where**
$\mathcal{L}$*-in-state* $\mathcal{A}$ q = {w. ($w \in$ *lists* (Σ $\mathcal{A}$)) $\wedge$ ($\exists$ $q' \in$ ($\mathcal{F}$ $\mathcal{A}$). *LTS-is-reachable* (Δ $\mathcal{A}$) q w q')}

abbreviation $\mathcal{L}$*-right* == $\mathcal{L}$*-in-state*

lemma $\mathcal{L}$*-in-state-alt-def* :
$\mathcal{L}$*-in-state* $\mathcal{A}$ q = $\mathcal{L}$ (| $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$, Σ = Σ $\mathcal{A}$, Δ = Δ $\mathcal{A}$, $\mathcal{I}$ = {q}, $\mathcal{F}$ = $\mathcal{F}$ $\mathcal{A}$ |)
unfolding $\mathcal{L}$*-def* $\mathcal{L}$*-in-state-def*
by (*auto simp add*: *NFA-accept-def*)

definition $\mathcal{L}$*-left* **where**
$\mathcal{L}$*-left* $\mathcal{A}$ q = {w. ($w \in$ *lists* (Σ $\mathcal{A}$)) $\wedge$ ($\exists$ $i \in$ ($\mathcal{I}$ $\mathcal{A}$). *LTS-is-reachable* (Δ $\mathcal{A}$) i w q)}

lemma $\mathcal{L}$*-left-alt-def* :
$\mathcal{L}$*-left* $\mathcal{A}$ q = $\mathcal{L}$ (| $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$, Σ = Σ $\mathcal{A}$, Δ = Δ $\mathcal{A}$, $\mathcal{I}$ = $\mathcal{I}$ $\mathcal{A}$, $\mathcal{F}$ = {q} |)
unfolding $\mathcal{L}$*-def* $\mathcal{L}$*-left-def* **by** (*auto simp add*: *NFA-accept-def*)

lemma *NFA-accept-alt-def* : *NFA-accept* $\mathcal{A}$ w $\longleftrightarrow$ $w \in \mathcal{L}$ $\mathcal{A}$ **by** (*simp add*: $\mathcal{L}$*-def*)

lemma $\mathcal{L}$*-alt-def* :
$\mathcal{L}$ $\mathcal{A}$ = $\bigcup$ (($\mathcal{L}$*-in-state* $\mathcal{A}$) ' ($\mathcal{I}$ $\mathcal{A}$))
by (*auto simp add*: $\mathcal{L}$*-def* $\mathcal{L}$*-in-state-def* *NFA-accept-def*)

lemma *in-*$\mathcal{L}$*-in-state-Nil* [*simp*] : [] $\in$ $\mathcal{L}$*-in-state* $\mathcal{A}$ q $\longleftrightarrow$ ($q \in \mathcal{F}$ $\mathcal{A}$) **by** (*simp add*: $\mathcal{L}$*-in-state-def*)
lemma *in-*$\mathcal{L}$*-in-state-Cons* [*simp*] : (σ # w) $\in$ $\mathcal{L}$*-in-state* $\mathcal{A}$ q $\longleftrightarrow$
($\exists$ q'. $\sigma \in \Sigma$ $\mathcal{A}$ $\wedge$ (q, σ, q') $\in \Delta$ $\mathcal{A}$ $\wedge$ $w \in$ $\mathcal{L}$*-in-state* $\mathcal{A}$ q') **by** (*auto simp add*: $\mathcal{L}$*-in-state-def*)

definition *remove-prefix* :: $'a$ *list* $\Rightarrow$ ($'a$ *list*) *set* $\Rightarrow$ ($'a$ *list*) *set* **where**
remove-prefix pre L $\equiv$ *drop* (*length pre*) ' ($L \cap$ {*pre* @ w |w. *True*})

lemma *remove-prefix-alt-def* [*simp*] :
$w \in$ *remove-prefix pre L* $\longleftrightarrow$ (*pre* @ $w \in L$)
by (*auto simp add*: *remove-prefix-def image-iff Bex-def*)

lemma *remove-prefix-Nil* [*simp*] :
remove-prefix [] L = L **by** (*simp add*: *set-eq-iff*)

lemma *remove-prefix-Combine* [*simp*] :
remove-prefix p2 (*remove-prefix p1 L*) =
remove-prefix (*p1* @ *p2*) L
by (*rule set-eqI*, *simp*)

lemma $\mathcal{L}$*-in-state-remove-prefix* :
pre $\in$ *lists* (Σ $\mathcal{A}$) $\Longrightarrow$
(*remove-prefix pre* ($\mathcal{L}$*-in-state* $\mathcal{A}$ q) =
$\bigcup$ {$\mathcal{L}$*-in-state* $\mathcal{A}$ q' | q'. *LTS-is-reachable* (Δ $\mathcal{A}$) q *pre* q'})
by (*rule set-eqI*, *auto simp add*: $\mathcal{L}$*-in-state-def* *LTS-is-reachable-concat*)

lemma $\mathcal{L}$*-in-state---in-*$\mathcal{F}$:

assumes *L-eq*: $\mathcal{L}$*-in-state* $\mathcal{A}$ q = $\mathcal{L}$*-in-state* $\mathcal{A}$ q'
shows $(q \in \mathcal{F}\ \mathcal{A} \longleftrightarrow q' \in \mathcal{F}\ \mathcal{A})$
proof −
  from *L-eq* **have** ([] ∈ $\mathcal{L}$*-in-state* $\mathcal{A}$ q) = ([] ∈ $\mathcal{L}$*-in-state* $\mathcal{A}$ q') **by** *simp*
  thus *?thesis* **by** *simp*
qed

The following locale captures, whether a NFA is well-formed.

locale *NFA* =
  fixes $\mathcal{A}$::('*q*,'*a*, '*NFA-more*) *NFA-rec-scheme*
  assumes Δ*-consistent*: $\bigwedge q\ \sigma\ q'.\ (q,\sigma,q') \in \Delta\ \mathcal{A} \Longrightarrow (q \in \mathcal{Q}\ \mathcal{A}) \wedge (\sigma \in \Sigma\ \mathcal{A}) \wedge (q' \in \mathcal{Q}\ \mathcal{A})$
    and $\mathcal{I}$*-consistent*: $\mathcal{I}\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A}$
    and $\mathcal{F}$*-consistent*: $\mathcal{F}\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A}$
    and *finite-*$\mathcal{Q}$: *finite* ($\mathcal{Q}\ \mathcal{A}$)
    and *finite-*Σ: *finite* ($\Sigma\ \mathcal{A}$)

lemma *NFA-intro* [*intro*!] :
  $[\![\bigwedge q\ \sigma\ q'.\ (q,\sigma,q') \in \Delta\ \mathcal{A} \Longrightarrow (q \in \mathcal{Q}\ \mathcal{A}) \wedge (\sigma \in \Sigma\ \mathcal{A}) \wedge (q' \in \mathcal{Q}\ \mathcal{A});$
   $\mathcal{I}\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A};\ \mathcal{F}\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A};$ *finite* ($\mathcal{Q}\ \mathcal{A}$); *finite* ($\Sigma\ \mathcal{A}$)$]\!] \Longrightarrow$ *NFA* $\mathcal{A}$
by (*simp add*: *NFA-def*)

definition *dummy-NFA* **where**
dummy-NFA q a =
 $(\!|\mathcal{Q} = \{q\},\ \Sigma = \{a\},\ \Delta = \{(q,\ a,\ q)\},$
  $\mathcal{I} = \{q\},\ \mathcal{F} = \{q\}\ |\!)$

lemma *dummy-NFA---is-NFA* :
 NFA (*dummy-NFA* q a)
by (*simp add*: *NFA-def dummy-NFA-def*)

lemma (**in** *NFA*) *wf-NFA* : *NFA* $\mathcal{A}$
unfolding *NFA-def*
using Δ*-consistent finite-*$\mathcal{Q}$ *finite-*Σ $\mathcal{I}$*-consistent* $\mathcal{F}$*-consistent*
by *simp*

lemma (**in** *NFA*) *finite-*$\mathcal{I}$:
 finite ($\mathcal{I}\ \mathcal{A}$)
using $\mathcal{I}$*-consistent finite-*$\mathcal{Q}$
by (*metis finite-subset*)

lemma (**in** *NFA*) *finite-*$\mathcal{F}$:
 finite ($\mathcal{F}\ \mathcal{A}$)
using $\mathcal{F}$*-consistent finite-*$\mathcal{Q}$
by (*metis finite-subset*)

lemma (**in** *NFA*) Δ*-subset* :
$\Delta\ \mathcal{A} \subseteq \mathcal{Q}\ \mathcal{A} \times \Sigma\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
using Δ*-consistent*
by (*simp add*: *subset-iff*)

lemma (**in** *NFA*) *finite-*Δ :
 finite ($\Delta\ \mathcal{A}$)
proof −
  from *finite-*$\mathcal{Q}$ *finite-*Σ **have** *finite* ($\mathcal{Q}\ \mathcal{A} \times \Sigma\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$) **by** *simp*
  with Δ*-subset* **show** *finite* ($\Delta\ \mathcal{A}$) **by** (*metis finite-subset*)
qed

lemma (**in** *NFA*) *NFA-*Δ*-cons---is-path* :
shows $[\![q \in \mathcal{Q}\ \mathcal{A};$ *LTS-is-path* ($\Delta\ \mathcal{A}$) $q\ \pi]\!] \Longrightarrow$
   $\pi \in$ *lists* ($\Sigma\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$)
proof (*induct* π *arbitrary*: q)
  case *Nil* **thus** *?case* **by** *simp*

next
  case $(Cons\ \sigma q\ \pi)$
  note $ind\text{-}hyp = Cons(1)$
  note $q\text{-}in\text{-}Q = Cons(2)$
  note $path\text{-}\sigma q\text{-}\pi = Cons(3)$

  obtain $\sigma\ q''$ **where** $\sigma q\text{-}eq : \sigma q = (\sigma,\ q'')$ **by** $(cases\ \sigma q,\ auto)$

  from $path\text{-}\sigma q\text{-}\pi\ \sigma q\text{-}eq$ **have** $(q,\ \sigma,\ q'') \in (\Delta\ \mathcal{A})$ **by** *simp*
  hence $q''\text{-}in\text{-}Q$: $q'' \in \mathcal{Q}\ \mathcal{A}$ **and** $\sigma\text{-}in\text{-}\Sigma$: $\sigma \in \Sigma\ \mathcal{A}$
    using Δ-*consistent* **by** *blast+*
  from $path\text{-}\sigma q\text{-}\pi\ \sigma q\text{-}eq$ **have** $path\text{-}\pi$: $LTS\text{-}is\text{-}path\ (\Delta\ \mathcal{A})\ q''\ \pi$ **by** *simp*
  note $\pi\text{-}in = ind\text{-}hyp\ [OF\ q''\text{-}in\text{-}Q\ path\text{-}\pi]$

  from $\pi\text{-}in\ \sigma q\text{-}eq\ \sigma\text{-}in\text{-}\Sigma\ q''\text{-}in\text{-}Q$ **show** *?case* **by** *simp*
qed

lemma (**in** *NFA*) $NFA\text{-}\Delta\text{-}cons\text{---}LTS\text{-}is\text{-}reachable$:
  $[\![LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q']\!] \Longrightarrow (q \in \mathcal{Q}\ \mathcal{A} \longrightarrow q' \in \mathcal{Q}\ \mathcal{A}) \wedge w \in lists\ (\Sigma\ \mathcal{A})$
using Δ-*consistent*
apply $(induct\ w\ arbitrary\text{: } q)$
apply *auto*
done

lemma (**in** *NFA*) $LTS\text{-}is\text{-}reachable\text{---}labels$:
  $LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q' \Longrightarrow w \in lists\ (\Sigma\ \mathcal{A})$
by $(simp\ add\text{: } NFA\text{-}\Delta\text{-}cons\text{---}LTS\text{-}is\text{-}reachable)$

lemma (**in** *NFA*) $NFA\text{-}accept\text{-}wf\text{-}def$:
$NFA\text{-}accept\ \mathcal{A}\ w = (\exists\ q{\in}(\mathcal{I}\ \mathcal{A}).\ \exists\ q'{\in}(\mathcal{F}\ \mathcal{A}).\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q')$
by $(metis\ NFA\text{-}accept\text{-}def\ LTS\text{-}is\text{-}reachable\text{---}labels)$

lemma (**in** *NFA*) $LTS\text{-}is\text{-}reachable\text{-}from\text{-}initial\text{-}alt\text{-}def$:
  $accessible\ (LTS\text{-}forget\text{-}labels\ (\Delta\ \mathcal{A}))\ (\mathcal{I}\ \mathcal{A}) =$
    $\{q'.\ (\exists\ q{\in}\ (\mathcal{I}\ \mathcal{A}).\ \exists w.\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q')\}$
by $(simp\ add\text{: } accessible\text{-}def\ rtrancl\text{-}LTS\text{-}forget\text{-}labels)$

lemma (**in** *NFA*) $LTS\text{-}is\text{-}reachable\text{-}from\text{-}initial\text{-}subset$:
  $accessible\ (LTS\text{-}forget\text{-}labels\ (\Delta\ \mathcal{A}))\ (\mathcal{I}\ \mathcal{A}) \subseteq \mathcal{Q}\ \mathcal{A}$
using $NFA\text{-}\Delta\text{-}cons\text{---}LTS\text{-}is\text{-}reachable\ \mathcal{I}\text{-}consistent$
unfolding $LTS\text{-}is\text{-}reachable\text{-}from\text{-}initial\text{-}alt\text{-}def$
by *auto*

lemma (**in** *NFA*) $LTS\text{-}is\text{-}reachable\text{-}from\text{-}initial\text{-}finite$:
  $finite\ (accessible\ (LTS\text{-}forget\text{-}labels\ (\Delta\ \mathcal{A}))\ (\mathcal{I}\ \mathcal{A}))$
using $LTS\text{-}is\text{-}reachable\text{-}from\text{-}initial\text{-}subset\ finite\text{-}\mathcal{Q}$
by $(rule\ finite\text{-}subset)$

2.2 Constructing from a list representation

fun *construct-NFA-aux* **where**
   $construct\text{-}NFA\text{-}aux\ \mathcal{A}\ (q1,\ l,\ q2) =$
   $(\!|\ \mathcal{Q}{=}insert\ q1\ (insert\ q2\ (\mathcal{Q}\ \mathcal{A})),$
      $\Sigma{=}\Sigma\ \mathcal{A} \cup set\ l,$
      $\Delta{=}\Delta\ \mathcal{A} \cup \{\ (q1,a,q2)\ |\ a.\ a \in set\ l\},$
      $\mathcal{I} = \mathcal{I}\ \mathcal{A},\ \mathcal{F} = \mathcal{F}\ \mathcal{A}|\!)$

fun *NFA-construct* **where**
   $NFA\text{-}construct\ (Q,\ A,\ D,\ I,\ F) =$
   $foldl\ construct\text{-}NFA\text{-}aux$
   $(\!|\ \mathcal{Q}{=}set\ (Q\ @\ I\ @\ F),\ \Sigma{=}set\ A,\ \Delta{=}\{\},\ \mathcal{I}\ {=}set\ I,\ \mathcal{F} = set\ F|\!)\ D$
declare *NFA-construct.simps* [*simp del*]

```
lemma NFA-construct-alt-def :
  NFA-construct (Q, A, D, I, F) =
   (| 𝒬=set Q ∪ set I ∪ set F ∪
       set (map fst D) ∪
       set (map (snd ∘ snd) D), Σ=set A ∪ ⋃ set (map (set ∘ fst ∘ snd) D),
       Δ = { (q1,a,q2) . (∃ l. (q1, l, q2) ∈ set D ∧ a ∈ set l)}, ℐ = set I, ℱ = set F|)
proof (induct D)
  case Nil thus ?case by (auto simp add: NFA-construct.simps)
next
  case (Cons qlq D)
  have fold-lemma: ⋀𝒜. foldl construct-NFA-aux (construct-NFA-aux 𝒜 qlq) D =
           construct-NFA-aux (foldl construct-NFA-aux 𝒜 D) qlq
    by (rule-tac foldl-fun-comm [symmetric], auto)

  obtain q1 l q2 where qlq-eq : qlq = (q1, l, q2) by (cases qlq, auto)

  from Cons fold-lemma show ?case
    by (simp add: NFA-construct.simps, auto simp add: qlq-eq)
qed

fun NFA-construct-simple where
  NFA-construct-simple (Q, A, D, I, F) =
   NFA-construct (Q, A, map (λ(q1, a, q2). (q1, [a], q2)) D, I, F)

lemma NFA-construct---is-well-formed :
  NFA (NFA-construct l)
proof -
  obtain Q A D I F where l-eq[simp]: l = (Q, A, D, I, F) by (metis PairE)

  { fix q σ q'
    assume (q, σ, q') ∈ Δ (NFA-construct (Q, A, D, I, F))
    then obtain l where in-D: (q, l, q') ∈ set D and
                       in-l: σ ∈ set l by (auto simp add: NFA-construct-alt-def)

    from in-D have p1: q ∈ fst ` set D by (metis fst-conv imageI)
    from in-l have p2: σ ∈ set (fst (snd (q, l, q'))) by simp
    from in-D have p3: q' ∈ (snd ∘ snd) ` set D
      by (metis imageI image-compose snd-conv)

    note p1 p3
  }
  thus ?thesis
    by (auto simp add: NFA-construct-alt-def NFA-def Ball-def) metis
qed

lemma NFA-construct-exists :
fixes 𝒜 :: ('q, 'a) NFA-rec
assumes wf-A: NFA 𝒜
shows ∃ Q A D I F. 𝒜 = NFA-construct (Q, A, D, I, F)
proof -
  interpret NFA-A: NFA 𝒜 by (fact wf-A)

  from finite-list[OF NFA-A.finite-𝒬] guess Q .. note set-Q = this
  from finite-list[OF NFA-A.finite-ℐ] guess I .. note set-I = this
  from finite-list[OF NFA-A.finite-ℱ] guess F .. note set-F = this
  from finite-list[OF NFA-A.finite-Σ] guess A .. note set-A = this
  from finite-list[OF NFA-A.finite-Δ] guess DD .. note set-DD = this

  let ?D = map (λqaq. (fst qaq, [fst (snd qaq)], snd (snd qaq))) DD

  have 𝒜 = NFA-construct (Q, A, ?D, I, F)
```

apply (*rule NFA-rec.equality*)
apply (*simp-all add*: *NFA-construct-alt-def set-Q set-I set-F set-A set-DD o-def*)
apply (*insert NFA-A.*Δ*-consistent NFA-A.*$\mathcal{I}$*-consistent NFA-A.*$\mathcal{F}$*-consistent*) []
apply *auto*[]
apply (*insert NFA-A.*Δ*-consistent*) []
apply *auto* []
apply (*simp del*: *ex-simps add*: *image-iff Bex-def ex-simps*[*symmetric*])
done
thus *?thesis* **by** *blast*
qed

2.3 Removing states

definition *NFA-remove-states* :: $('q, 'a, 'x)$ *NFA-rec-scheme* $\Rightarrow$ $'q$ *set* $\Rightarrow$ $('q, 'a)$ *NFA-rec* **where**
NFA-remove-states $\mathcal{A}$ S == $(\!|$ $\mathcal{Q}$=$\mathcal{Q}$ $\mathcal{A}$ $-$ S, Σ=Σ $\mathcal{A}$, Δ = { $(s1,a,s2)$. $(s1,a,s2) \in \Delta$ $\mathcal{A}$ $\wedge$ $s1 \notin S$ $\wedge$ $s2 \notin S$}, $\mathcal{I}$ = $\mathcal{I}$ $\mathcal{A}$ $-$ S, $\mathcal{F}$ = $\mathcal{F}$ $\mathcal{A}$ $-$ $S|\!)$

lemma [*simp*] : $\mathcal{I}$ (*NFA-remove-states* $\mathcal{A}$ S) = $\mathcal{I}$ $\mathcal{A}$ $-$ S **by** (*simp add*: *NFA-remove-states-def*)
lemma [*simp*] : $\mathcal{Q}$ (*NFA-remove-states* $\mathcal{A}$ S) = $\mathcal{Q}$ $\mathcal{A}$ $-$ S **by** (*simp add*: *NFA-remove-states-def*)
lemma [*simp*] : $\mathcal{F}$ (*NFA-remove-states* $\mathcal{A}$ S) = $\mathcal{F}$ $\mathcal{A}$ $-$ S **by** (*simp add*: *NFA-remove-states-def*)
lemma [*simp*] : Σ (*NFA-remove-states* $\mathcal{A}$ S) = Σ $\mathcal{A}$ **by** (*simp add*: *NFA-remove-states-def*)
lemma [*simp*] : $x \in \Delta$ (*NFA-remove-states* $\mathcal{A}$ S) $\longleftrightarrow$
$x \in \Delta$ $\mathcal{A}$ $\wedge$ *fst* $x \notin S$ $\wedge$ *snd* (*snd* x) $\notin S$ **by** (*cases* x, *simp add*: *NFA-remove-states-def*)

lemma *NFA-remove-states-*Δ*-subset* : Δ (*NFA-remove-states* $\mathcal{A}$ S) $\subseteq$ Δ $\mathcal{A}$ **by** (*simp add*: *subset-iff*)
lemma *NFA-remove-states---is-well-formed* : *NFA* $\mathcal{A}$ $\Longrightarrow$ *NFA* (*NFA-remove-states* $\mathcal{A}$ S) **by** (*auto simp add*: *NFA-def*)

lemma *NFA-remove-states-empty* [*simp*] : *NFA-remove-states* $\mathcal{A}$ {} = $\mathcal{A}$
by (*rule NFA-rec.equality*, *simp-all add*: *NFA-remove-states-def*)

lemma *NFA-remove-states-NFA-remove-states* [*simp*] : *NFA-remove-states* (*NFA-remove-states* $\mathcal{A}$ *S1*) *S2* =
NFA-remove-states $\mathcal{A}$ (*S1* $\cup$ *S2*)
by (*rule NFA-rec.equality*, *auto simp add*: *NFA-remove-states-def*)

lemma *NFA-remove-states-*$\mathcal{L}$*-subset* : $\mathcal{L}$ (*NFA-remove-states* $\mathcal{A}$ S) $\subseteq$ $\mathcal{L}$ $\mathcal{A}$
by (*simp add*: $\mathcal{L}$*-def NFA-accept-def subset-iff Bex-def*,
metis LTS-is-reachable-mono NFA-remove-states-Δ*-subset*)

lemma *LTS-is-reachable-NFA-remove-states* :
assumes *Q-unrech*: $\bigwedge q'. \; q' \in Q \Longrightarrow$ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) q q'
and *w-in-*Σ: $w \in$ *lists* (Σ $\mathcal{A}$)
shows *LTS-is-reachable* (Δ $\mathcal{A}$) q w q' $\longleftrightarrow$ *LTS-is-reachable* (Δ (*NFA-remove-states* $\mathcal{A}$ Q)) q w q'
using *w-in-*Σ *Q-unrech*
proof (*induct w arbitrary*: *q*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* σ *w*)
note σ*-in-*Σ = *Cons*(*1*)
note *w-in-*Σ = *Cons*(*2*)
note *ind-hyp* = *Cons*(*3*)
note *Qq-cond* = *Cons*(*4*)

have *LTS-is-reachable* (Δ $\mathcal{A}$) q [] q $\wedge$ [] $\in$ *lists* (Σ $\mathcal{A}$) **by** *simp*
with *Qq-cond* **have** *q-nin-Q*: $q \notin Q$ **by** (*metis LTS-is-unreachable-def*)

have *Qq-cond'*: $\bigwedge q'. \; (q, \sigma, q') \in \Delta \; \mathcal{A} \Longrightarrow q' \notin Q \wedge (\forall q'' \in Q.$ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) q' q'')
proof $-$
fix q'
assume $(q, \sigma, q') \in \Delta$ $\mathcal{A}$
with σ*-in-*Σ **have** *LTS-is-reachable* (Δ $\mathcal{A}$) q [σ] q' $\wedge$ [σ] $\in$ *lists* (Σ $\mathcal{A}$) **by** *simp*
with *Qq-cond* **show** $q' \notin Q \wedge (\forall q'' \in Q.$ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) q' q'')
by (*metis LTS-is-unreachable-def LTS-is-unreachable-reachable-start*)

qed

from *ind-hyp q-nin-Q w-in-*Σ *Qq-cond'*
show *?case* **by** (*simp*, *metis*)
qed

lemma *NFA-remove-states-*$\mathcal{L}$*-in-state-iff* :
assumes *Q-unrech*: $\bigwedge q'.\ q'{\in}Q \Longrightarrow$ *LTS-is-unreachable* ($\Delta\ \mathcal{A}$) ($\Sigma\ \mathcal{A}$) $q\ q'$
shows $\mathcal{L}$*-in-state* (*NFA-remove-states* $\mathcal{A}$ Q) q = $\mathcal{L}$*-in-state* $\mathcal{A}$ q
(**is** *?L1* = *?L2*)
proof (*rule set-eqI*)
fix *w*
show $w \in$ *?L1* $\longleftrightarrow w \in$ *?L2*
proof (*cases* $w \in$ *lists* ($\Sigma\ \mathcal{A}$))
case *False* **thus** *?thesis* **by** (*simp add*: $\mathcal{L}$*-in-state-def*)
next
case *True*
note *w-in-*Σ = *this*

from *Q-unrech w-in-*Σ **have** *q'-nin-Q*: $\bigwedge q'.$ *LTS-is-reachable* ($\Delta\ \mathcal{A}$) $q\ w\ q' \Longrightarrow q' \notin Q$
by (*metis LTS-is-unreachable-def*)
note *reachable-remove-simp* = *LTS-is-reachable-NFA-remove-states* [*OF Q-unrech w-in-*Σ, *symmetric*]

with *w-in-*Σ *q'-nin-Q* **show** *?thesis*
by (*auto simp add*: $\mathcal{L}$*-in-state-def*)
qed
qed

lemma *NFA-remove-states-*$\mathcal{L}$*-iff* :
assumes *unreach-asm*: $\bigwedge q\ q'.\ [\![q \in \mathcal{I}\ \mathcal{A};\ q' \in Q]\!] \Longrightarrow$ *LTS-is-unreachable* ($\Delta\ \mathcal{A}$) ($\Sigma\ \mathcal{A}$) $q\ q'$
shows $\mathcal{L}$ (*NFA-remove-states* $\mathcal{A}$ Q) = $\mathcal{L}\ \mathcal{A}$
proof −
have *initial-not-Q*: $\forall\, q \in \mathcal{I}\ \mathcal{A}.\ q \notin Q$ **by** (*metis unreach-asm LTS-is-unreachable-not-refl*)
have *in-state-iff*: $\forall\, q \in \mathcal{I}\ \mathcal{A}.$ $\mathcal{L}$*-in-state* (*NFA-remove-states* $\mathcal{A}$ Q) q = $\mathcal{L}$*-in-state* $\mathcal{A}$ q
by (*metis unreach-asm NFA-remove-states-*$\mathcal{L}$*-in-state-iff*)

from *initial-not-Q in-state-iff*
show *?thesis* **by** (*simp add*: $\mathcal{L}$*-alt-def set-eq-iff Bex-def*, *metis*)
qed

lemma *NFA-remove-states-accept-iff* :
assumes *unreach-asm*: $\bigwedge q\ q'.\ [\![q \in \mathcal{I}\ \mathcal{A};\ q'{\in}\ Q]\!] \Longrightarrow$ *LTS-is-unreachable* ($\Delta\ \mathcal{A}$) ($\Sigma\ \mathcal{A}$) $q\ q'$
shows *NFA-accept* (*NFA-remove-states* $\mathcal{A}$ Q) w = *NFA-accept* $\mathcal{A}$ w
using *assms*
by (*simp add*: *NFA-accept-alt-def NFA-remove-states-*$\mathcal{L}$*-iff*)

definition *NFA-unreachable-states* **where**
NFA-unreachable-states $\mathcal{A}$ = $\{q'.\ \forall\, q{\in}\mathcal{I}\ \mathcal{A}.$ *LTS-is-unreachable* ($\Delta\ \mathcal{A}$) ($\Sigma\ \mathcal{A}$) $q\ q'\}$

lemma *NFA-unreachable-states-alt-def* :
NFA-unreachable-states $\mathcal{A}$ = $\{q.\ \mathcal{L}$*-left* $\mathcal{A}\ q = \{\}\}$
by (*simp add*: *NFA-unreachable-states-def* $\mathcal{L}$*-left-def LTS-is-unreachable-def*, *auto*)

lemma *NFA-unreachable-states-extend* :
$[\![x \notin$ *NFA-unreachable-states* $\mathcal{A}$; $(x,\ \sigma,\ q') \in \Delta\ \mathcal{A}$; $\sigma \in \Sigma\ \mathcal{A}]\!] \Longrightarrow q' \notin$ *NFA-unreachable-states* $\mathcal{A}$
proof
assume $x \notin$ *NFA-unreachable-states* $\mathcal{A}$
then obtain w q **where**
reach-x: *LTS-is-reachable* ($\Delta\ \mathcal{A}$) $q\ w\ x$ **and**
q-in-I: $q \in \mathcal{I}\ \mathcal{A}$ **and**
w-wf: $w \in$ *lists* (*NFA*.$\Sigma\ \mathcal{A}$)
by (*auto simp add*: *NFA-unreachable-states-def LTS-is-unreachable-def*)

```
  assume (x, σ, q′) ∈ Δ 𝒜
  with reach-x have reach-q′ : LTS-is-reachable (Δ 𝒜) q (w @ [σ]) q′ by auto

  assume σ ∈ NFA-rec.Σ 𝒜
  with w-wf have wσ-wf : w @ [σ] ∈ lists (Σ 𝒜) by simp

  assume q′ ∈ NFA-unreachable-states 𝒜
  thus False
    by (simp add: NFA-unreachable-states-def LTS-is-unreachable-def,
        metis wσ-wf q-in-I reach-q′)
qed

definition NFA-is-initially-connected where
  NFA-is-initially-connected 𝒜 ⟷ 𝒬 𝒜 ∩ NFA-unreachable-states 𝒜 = {}

lemma NFA-is-initially-connected-alt-def :
  NFA-is-initially-connected 𝒜 ⟷ (∀ q ∈ 𝒬 𝒜.
  ∃ i∈ℐ 𝒜. ∃ w∈lists (Σ 𝒜). LTS-is-reachable (Δ 𝒜) i w q)
unfolding NFA-is-initially-connected-def
by (auto simp add: set-eq-iff NFA-unreachable-states-def LTS-is-unreachable-def)

lemma dummy-NFA---NFA-is-initially-connected :
  NFA-is-initially-connected (dummy-NFA q a)
apply (simp add: NFA-is-initially-connected-alt-def dummy-NFA-def Bex-def)
apply (rule exI [where x = []])
apply simp
done

definition NFA-remove-unreachable-states where
  NFA-remove-unreachable-states 𝒜 = NFA-remove-states 𝒜 (NFA-unreachable-states 𝒜)

lemma NFA-remove-unreachable-states-ℒ [simp] :
  ℒ (NFA-remove-unreachable-states 𝒜) = ℒ 𝒜
apply (simp add: NFA-remove-unreachable-states-def)
apply (rule NFA-remove-states-ℒ-iff)
apply (simp add: NFA-unreachable-states-def)
done

lemma NFA-remove-unreachable-states-ℒ-in-state [simp] :
assumes q-in-Q: q ∈ 𝒬 (NFA-remove-unreachable-states 𝒜)
shows ℒ-in-state (NFA-remove-unreachable-states 𝒜) q = ℒ-in-state 𝒜 q
proof −
  from q-in-Q have q ∉ NFA-unreachable-states 𝒜
    by (simp add: NFA-remove-unreachable-states-def)
  then obtain q0 w where
    q0 ∈ ℐ 𝒜 ∧ LTS-is-reachable (Δ 𝒜) q0 w q ∧ w ∈ lists (Σ 𝒜)
    by (auto simp add: NFA-unreachable-states-def LTS-is-unreachable-def)
  then have Q-OK: ⋀q′. q′ ∈ NFA-unreachable-states 𝒜 ⟹ LTS-is-unreachable (Δ 𝒜) (Σ 𝒜) q q′
    by (simp add: NFA-unreachable-states-def,
        metis LTS-is-unreachable-reachable-start)

  note NFA-remove-states-ℒ-in-state-iff [where q = q and 𝒜 = 𝒜 and
     Q = NFA-unreachable-states 𝒜, OF Q-OK]
  thus ?thesis by (simp add: NFA-remove-unreachable-states-def)
qed

lemma NFA-remove-unreachable-states-accept-iff [simp] :
NFA-accept (NFA-remove-unreachable-states 𝒜) w = NFA-accept 𝒜 w
by (simp add: NFA-accept-alt-def)

lemma NFA-remove-unreachable-states---is-well-formed [simp] :
```

NFA $\mathcal{A} \Longrightarrow$ *NFA* (*NFA-remove-unreachable-states* $\mathcal{A}$)
by (*simp add*: *NFA-remove-unreachable-states-def NFA-remove-states---is-well-formed*)

lemma *NFA-unreachable-states-*$\mathcal{I}$:
$q \in \mathcal{I}\ \mathcal{A} \Longrightarrow q \notin$ *NFA-unreachable-states* $\mathcal{A}$
proof −
have *f1*: *LTS-is-reachable* ($\Delta\ \mathcal{A}$) q [] $q \wedge [] \in$ *lists* ($\Sigma\ \mathcal{A}$) **by** *simp*
show $q \in \mathcal{I}\ \mathcal{A} \Longrightarrow q \notin$ *NFA-unreachable-states* $\mathcal{A}$
by (*simp add*: *NFA-unreachable-states-def LTS-is-unreachable-def*, *metis f1*)
qed

lemma *NFA-remove-unreachable-states-*$\mathcal{I}$ [*simp*] :
$\mathcal{I}$ (*NFA-remove-unreachable-states* $\mathcal{A}$) = $\mathcal{I}\ \mathcal{A}$
by (*auto simp add*: *NFA-remove-unreachable-states-def NFA-unreachable-states-*$\mathcal{I}$)

lemma [*simp*] : Σ (*NFA-remove-unreachable-states* $\mathcal{A}$) = $\Sigma\ \mathcal{A}$
by (*simp add*: *NFA-remove-unreachable-states-def*)

lemma *NFA-unreachable-states-NFA-remove-unreachable-states* :
NFA-unreachable-states (*NFA-remove-unreachable-states* $\mathcal{A}$) =
NFA-unreachable-states $\mathcal{A}$
proof −
have $\bigwedge q\ q'\ w.\ [\![q \in \mathcal{I}\ \mathcal{A};\ w \in$ *lists* $(\Sigma\ \mathcal{A})]\!] \Longrightarrow$
LTS-is-reachable (Δ (*NFA-remove-unreachable-states* $\mathcal{A}$)) $q\ w\ q' \longleftrightarrow$
LTS-is-reachable ($\Delta\ \mathcal{A}$) $q\ w\ q'$
proof −
fix $q\ q'\ w$
assume *q-in-I*: $q \in \mathcal{I}\ \mathcal{A}$
assume *w-in-*Σ: $w \in$ *lists* ($\Sigma\ \mathcal{A}$)

with *q-in-I* **have**
Q-OK: $\bigwedge q'.\ q' \in$ *NFA-unreachable-states* $\mathcal{A} \Longrightarrow$ *LTS-is-unreachable* ($\Delta\ \mathcal{A}$) ($\Sigma\ \mathcal{A}$) $q\ q'$
by (*simp add*: *NFA-unreachable-states-def*)

note *LTS-is-reachable-NFA-remove-states* [**where** $\mathcal{A} = \mathcal{A}$ **and** $q = q$ **and** q'=q' **and** $w = w$
and Q = *NFA-unreachable-states* $\mathcal{A}$]
with *w-in-*Σ *Q-OK* **show** *LTS-is-reachable* (Δ (*NFA-remove-unreachable-states* $\mathcal{A}$)) $q\ w\ q' \longleftrightarrow$
LTS-is-reachable ($\Delta\ \mathcal{A}$) $q\ w\ q'$
by (*simp add*: *NFA-remove-unreachable-states-def*)
qed
thus *?thesis* **by** (*simp add*: *NFA-unreachable-states-def LTS-is-unreachable-def*)
qed

lemma *NFA-remove-unreachable-states-NFA-remove-unreachable-states* [*simp*] :
NFA-remove-unreachable-states (*NFA-remove-unreachable-states* $\mathcal{A}$) = *NFA-remove-unreachable-states* $\mathcal{A}$
apply (*simp add*: *NFA-remove-unreachable-states-def*)
apply (*simp add*: *NFA-remove-unreachable-states-def* [*symmetric*]
NFA-unreachable-states-NFA-remove-unreachable-states)
done

lemma *NFA-remove-unreachable-states---NFA-is-initially-connected* :
NFA-is-initially-connected (*NFA-remove-unreachable-states* $\mathcal{A}$)
apply (*simp add*: *NFA-is-initially-connected-def NFA-unreachable-states-NFA-remove-unreachable-states*)
apply (*simp add*: *NFA-remove-unreachable-states-def*)
apply *fastforce*
done

2.4 Rename States / Combining

definition *NFA-rename-states* ::
($'q1$, $'a$, $'x$) *NFA-rec-scheme* $\Rightarrow$ ($'q1 \Rightarrow 'q2$) $\Rightarrow$ ($'q2$, $'a$) *NFA-rec* **where**

```
NFA-rename-states 𝒜 f ≡
(| 𝒬=f ` (𝒬 𝒜), Σ=Σ 𝒜, Δ = { (f s1, a, f s2) | s1 a s2. (s1,a,s2) ∈ Δ 𝒜},
  ℐ=f ` (ℐ 𝒜), ℱ = f ` (ℱ 𝒜) |)

lemma [simp] : ℐ (NFA-rename-states 𝒜 f) = f ` ℐ 𝒜 by (simp add: NFA-rename-states-def)
lemma [simp] : 𝒬 (NFA-rename-states 𝒜 f) = f ` 𝒬 𝒜 by (simp add: NFA-rename-states-def)
lemma [simp] : ℱ (NFA-rename-states 𝒜 f) = f ` ℱ 𝒜 by (simp add: NFA-rename-states-def)
lemma [simp] : Σ (NFA-rename-states 𝒜 S) = Σ 𝒜 by (simp add: NFA-rename-states-def)
lemma [simp] : (fq, σ, fq') ∈ Δ (NFA-rename-states 𝒜 f) ⟷
                (∃ q q'. (q, σ, q') ∈ Δ 𝒜 ∧ (fq = f q) ∧ (fq' = f q'))
  by (auto simp add: NFA-rename-states-def)

lemma NFA-rename-states---is-well-formed :
 NFA 𝒜 ⟹ NFA (NFA-rename-states 𝒜 f)
by (auto simp add: NFA-def image-iff Bex-def)

lemma NFA-rename-states-id [simp] : NFA-rename-states 𝒜 id = 𝒜
  by (rule NFA-rec.equality, auto simp add: NFA-rename-states-def)

lemma NFA-rename-states-NFA-rename-states [simp] :
   NFA-rename-states (NFA-rename-states 𝒜 f1) f2 =
    NFA-rename-states 𝒜 (f2 ∘ f1)
by (auto simp add: NFA-rename-states-def image-compose, metis)

lemma (in NFA) NFA-rename-states-agree-on-Q :
assumes f12-agree: ⋀q. q ∈ 𝒬 𝒜 ⟹ f1 q = f2 q
shows NFA-rename-states 𝒜 f1 = NFA-rename-states 𝒜 f2
unfolding NFA-rename-states-def
proof (rule NFA-rec.equality, simp-all)
  have image-Q: ⋀Q. Q ⊆ 𝒬 𝒜 ⟹ f1 ` Q = f2 ` Q
    using f12-agree
    by (simp add: subset-iff set-eq-iff image-iff)

  from image-Q show f1 ` (𝒬 𝒜) = f2 ` (𝒬 𝒜) by simp
  from ℐ-consistent image-Q show f1 ` (ℐ 𝒜) = f2 ` (ℐ 𝒜) by simp
  from ℱ-consistent image-Q show f1 ` (ℱ 𝒜) = f2 ` (ℱ 𝒜) by simp
next
  from Δ-consistent f12-agree
  show {(f1 s1, a, f1 s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜} =
        {(f2 s1, a, f2 s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜}
    by (simp add: set-eq-iff) metis
qed

lemma LTS-is-reachable---NFA-rename-statesE :
 LTS-is-reachable (Δ 𝒜) q w q' ⟹
  LTS-is-reachable (Δ (NFA-rename-states 𝒜 f)) (f q) w (f q')
proof (induct w arbitrary: q)
  case Nil thus ?case by simp
next
  case (Cons σ w)
  note ind-hyp = Cons(1)
  note reach-σw = Cons(2)

  from reach-σw obtain q'' where
     q''-Δ: (q, σ, q'') ∈ Δ 𝒜 and
     reach-w: LTS-is-reachable (Δ 𝒜) q'' w q' by auto

  from ind-hyp reach-w
    have reach-w': LTS-is-reachable (Δ (NFA-rename-states 𝒜 f)) (f q'') w (f q')
    by blast

  from q''-Δ have q''-Δ' : (f q, σ, f q'') ∈ (Δ (NFA-rename-states 𝒜 f)) by auto
```

from q''-Δ' *reach-w'* **show** *?case* **by** *auto*
qed

lemma $\mathcal{L}$-*in-state-rename-subset1* :
$\mathcal{L}$-*in-state* $\mathcal{A}$ $q \subseteq$ $\mathcal{L}$-*in-state* (*NFA-rename-states* $\mathcal{A}$ f) (f q)
by (*simp add*: $\mathcal{L}$-*in-state-def subset-iff*, *metis LTS-is-reachable---NFA-rename-statesE*)

definition *NFA-is-equivalence-rename-fun* **where**
NFA-is-equivalence-rename-fun $\mathcal{A}$ f =
$(\forall q \in \mathcal{Q}\ \mathcal{A}.\ \forall q' \in \mathcal{Q}\ \mathcal{A}.\ (f\ q = f\ q') \longrightarrow (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q'))$

definition *NFA-is-strong-equivalence-rename-fun* **where**
NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ f =
$(\forall q \in \mathcal{Q}\ \mathcal{A}.\ \forall q' \in \mathcal{Q}\ \mathcal{A}.\ ((f\ q = f\ q') \longleftrightarrow (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q')))$

lemma *NFA-is-strong-equivalence-rename-funE* :
⟦*NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f;
$q \in \mathcal{Q}\ \mathcal{A}$; $q' \in \mathcal{Q}\ \mathcal{A}$⟧ $\Longrightarrow$
$f\ q = f\ q' \longleftrightarrow (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q')$
unfolding *NFA-is-strong-equivalence-rename-fun-def* **by** *metis*

lemma *NFA-is-strong-equivalence-rename-fun---weaken* :
NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ f $\Longrightarrow$
NFA-is-equivalence-rename-fun $\mathcal{A}$ f
by (*simp add*: *NFA-is-equivalence-rename-fun-def NFA-is-strong-equivalence-rename-fun-def*)

lemma *NFA-is-strong-equivalence-rename-fun-exists* :
$\exists f$::($'a \Rightarrow 'a$). (*NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f $\wedge$ $(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f\ q \in \mathcal{Q}\ \mathcal{A})$)
proof −
def *fe* $\equiv \lambda l.$ *SOME* $q.\ q \in \mathcal{Q}\ \mathcal{A} \wedge \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = l$
def $f \equiv \lambda q.$ *fe* ($\mathcal{L}$-*in-state* $\mathcal{A}$ q)

have *f-lang* : $\bigwedge q.\ q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow$
$f\ q \in \mathcal{Q}\ \mathcal{A}\ \wedge$
$\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ (f\ q) = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q$
by (*simp add*: *fe-def f-def*, *rule-tac someI-ex*, *auto*)

have *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f
by (*simp add*: *NFA-is-strong-equivalence-rename-fun-def*,
metis f-def f-lang)
with *f-lang* **show** *?thesis* **by** *metis*
qed

lemma *NFA-is-strong-equivalence-rename-fun---isomorph* :
fixes *f1* :: $'a \Rightarrow 'b$
and *f2* :: $'a$ => $'c$
and $\mathcal{A}$ :: ($'a$, $'l$, $'x$) *NFA-rec-scheme*
assumes *f1-OK*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ *f1*
and *f2-OK*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ *f2*
shows $\exists$*f12*. $(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f2\ q = (f12\ (f1\ q))) \wedge$ *inj-on f12* (*f1* ‘ ($\mathcal{Q}$ $\mathcal{A}$))
proof −
def *f12* $\equiv \lambda q.$ *f2* ((*inv-into* ($\mathcal{Q}$ $\mathcal{A}$) *f1*) q)

have *inj-f12*: *inj-on f12* (*f1* ‘ ($\mathcal{Q}$ $\mathcal{A}$))
unfolding *inj-on-def f12-def*
proof (*intro ballI impI*)
fix *q1 q2* :: $'b$
assume *q1-in-f1-Q*: *q1* $\in$ *f1* ‘ $\mathcal{Q}$ $\mathcal{A}$
assume *q2-in-f1-Q*: *q2* $\in$ *f1* ‘ $\mathcal{Q}$ $\mathcal{A}$
assume *f2-eq*: *f2* (*inv-into* ($\mathcal{Q}$ $\mathcal{A}$) *f1 q1*) = *f2* (*inv-into* ($\mathcal{Q}$ $\mathcal{A}$) *f1 q2*)

let *?q1 ′* = *inv-into* ($\mathcal{Q}$ $\mathcal{A}$) *f1 q1*
let *?q2 ′* = *inv-into* ($\mathcal{Q}$ $\mathcal{A}$) *f1 q2*

from *inv-into-into*[*OF q1-in-f1-Q*] **have** *q1 ′-in-Q*: *?q1 ′* ∈ $\mathcal{Q}$ $\mathcal{A}$ **.**
from *inv-into-into*[*OF q2-in-f1-Q*] **have** *q2 ′-in-Q*: *?q2 ′* ∈ $\mathcal{Q}$ $\mathcal{A}$ **.**

from *f-inv-into-f*[*OF q1-in-f1-Q*] **have** *q1-eq*: *f1 ?q1 ′* = *q1* **by** *simp*
from *f-inv-into-f*[*OF q2-in-f1-Q*] **have** *q2-eq*: *f1 ?q2 ′* = *q2* **by** *simp*

from *NFA-is-strong-equivalence-rename-funE* [*OF f2-OK q1 ′-in-Q q2 ′-in-Q*] *f2-eq*
have *q12 ′-equiv*: $\mathcal{L}$*-in-state* $\mathcal{A}$ *?q1 ′* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *?q2 ′*
unfolding *NFA-is-strong-equivalence-rename-fun-def*
by *simp*

with *NFA-is-strong-equivalence-rename-funE* [*OF f1-OK q1 ′-in-Q q2 ′-in-Q*]
show *q1* = *q2*
by (*simp add*: *q1-eq q2-eq*)
qed

have *f2-eq*: (∀ *q*∈$\mathcal{Q}$ $\mathcal{A}$. *f2 q* = *f12* (*f1 q*))
proof (*intro ballI*)
fix *q*
assume *q-in-Q*: *q*∈$\mathcal{Q}$ $\mathcal{A}$

let *?q′* = *inv-into* ($\mathcal{Q}$ $\mathcal{A}$) *f1* (*f1 q*)

from *q-in-Q* **have** *f1-q-in-f1-Q*: *f1 q* ∈ *f1* ‘ ($\mathcal{Q}$ $\mathcal{A}$) **by** *simp*

from *inv-into-into* [*OF f1-q-in-f1-Q*]
have *q′-in-Q*: *?q′* ∈ $\mathcal{Q}$ $\mathcal{A}$ **by** *simp*

from *f-inv-into-f*[*OF f1-q-in-f1-Q*] **have** *f1 ?q′* = *f1 q* **by** *simp*
with *NFA-is-strong-equivalence-rename-funE* [*OF f1-OK q′-in-Q q-in-Q*]
have *q-q′-equiv*: $\mathcal{L}$*-in-state* $\mathcal{A}$ *?q′* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q*
by *simp*

from *NFA-is-strong-equivalence-rename-funE* [*OF f2-OK q′-in-Q q-in-Q*] *q-q′-equiv*
have *f2 q* = *f2 ?q′* **by** *simp*

thus *f2 q* = *f12* (*f1 q*)
unfolding *f12-def*
by *simp*
qed

from *f2-eq inj-f12* **show** *?thesis* **by** *blast*
qed

lemma (**in** *NFA*) $\mathcal{L}$*-in-state-rename-subset2* :
assumes *equiv-f* : *NFA-is-equivalence-rename-fun* $\mathcal{A}$ *f*
assumes *q-in-Q* : *q* ∈ $\mathcal{Q}$ $\mathcal{A}$
shows $\mathcal{L}$*-in-state* (*NFA-rename-states* $\mathcal{A}$ *f*) (*f q*) ⊆ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q*
proof −
have $\bigwedge$*w q′*. ⟦*w* ∈ *lists* (Σ $\mathcal{A}$); *q′* ∈ $\mathcal{F}$ $\mathcal{A}$; *LTS-is-reachable* (Δ (*NFA-rename-states* $\mathcal{A}$ *f*)) (*f q*) *w* (*f q′*)⟧ ⟹
w ∈ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q*
proof −
fix *w q′*
assume *q′-in-F*: *q′* ∈ $\mathcal{F}$ $\mathcal{A}$
from *q′-in-F* **have** *q′-in-Q*: *q′* ∈ $\mathcal{Q}$ $\mathcal{A}$ **using** $\mathcal{F}$*-consistent* **by** (*auto simp add*: *NFA-def*)
show ⟦*q* ∈ $\mathcal{Q}$ $\mathcal{A}$; *w* ∈ *lists* (Σ $\mathcal{A}$); *LTS-is-reachable* (Δ (*NFA-rename-states* $\mathcal{A}$ *f*)) (*f q*) *w* (*f q′*)⟧ ⟹
w ∈ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q*

proof (*induct w arbitrary*: *q*)
case *Nil*
note *q-in-Q* = *Nil*(*1*)
from *Nil*(*3*) **have** $f\ q = f\ q'$ **by** *simp*
with *equiv-f q-in-Q q'-in-Q* **have** $\mathcal{L}$-*in-state* $\mathcal{A}$ *q* = $\mathcal{L}$-*in-state* $\mathcal{A}$ *q'*
by (*metis NFA-is-equivalence-rename-fun-def*)
with *q'-in-F* **have** $q \in \mathcal{F}\ \mathcal{A}$ **by** (*metis* $\mathcal{L}$-*in-state---in-*$\mathcal{F}$)
thus [] $\in$ $\mathcal{L}$-*in-state* $\mathcal{A}$ *q* **by** *simp*
next
case (*Cons* σ *w*)
note *ind-hyp* = *Cons*(*1*)
note *q-in-Q* = *Cons*(*2*)
note σ*w-lists* = *Cons*(*3*)
note *is-reach-*σ*w'* = *Cons*(*4*)

from *is-reach-*σ*w'* **obtain** q'' q''' **where**
f-q'''-eq: $f\ q''' = f\ q$ **and**
in-Δ: $(q''', \sigma, q'') \in \Delta\ \mathcal{A}$ **and**
in-rΔ: $(f\ q, \sigma, f\ q'') \in$ (Δ (*NFA-rename-states* $\mathcal{A}$ *f*)) **and**
is-reach-w': *LTS-is-reachable* (Δ (*NFA-rename-states* $\mathcal{A}$ *f*)) ($f\ q''$) *w* ($f\ q'$)
by (*auto*, *metis*)

from *in-*Δ **have** *q'''-in-Q* : $q''' \in \mathcal{Q}\ \mathcal{A}$ **and**
q''-in-Q: $q'' \in \mathcal{Q}\ \mathcal{A}$
using Δ-*consistent* **by** *simp-all*

from *q''-in-Q ind-hyp is-reach-w'* σ*w-lists* **have**
$w \in$ $\mathcal{L}$-*in-state* $\mathcal{A}$ q'' **by** *simp*
with *in-*Δ σ*w-lists* **have** $(\sigma\ \#\ w) \in$ $\mathcal{L}$-*in-state* $\mathcal{A}$ q''' **by** *auto*
with *q-in-Q q'''-in-Q f-q'''-eq equiv-f* **show** *?case*
by (*metis NFA-is-equivalence-rename-fun-def*)
qed
qed (*simp-all add*: *q-in-Q*)
thus *?thesis*
by (*auto simp add*: $\mathcal{L}$-*in-state-def subset-iff*)
qed

lemma (**in** *NFA*) $\mathcal{L}$-*in-state-rename-iff* :
assumes *equiv-f* : *NFA-is-equivalence-rename-fun* $\mathcal{A}$ *f*
assumes *q-in-Q* : $q \in \mathcal{Q}\ \mathcal{A}$
shows $\mathcal{L}$-*in-state* (*NFA-rename-states* $\mathcal{A}$ *f*) (*f q*) = $\mathcal{L}$-*in-state* $\mathcal{A}$ *q*
using *assms*
by (*metis* $\mathcal{L}$-*in-state-rename-subset1* $\mathcal{L}$-*in-state-rename-subset2 set-eq-subset*)

lemma (**in** *NFA*) $\mathcal{L}$-*rename-iff* :
assumes *equiv-f* : *NFA-is-equivalence-rename-fun* $\mathcal{A}$ *f*
shows $\mathcal{L}$ (*NFA-rename-states* $\mathcal{A}$ *f*) = $\mathcal{L}$ $\mathcal{A}$
proof −
have $\bigwedge q.\ q \in \mathcal{I}\ \mathcal{A} \Longrightarrow q \in \mathcal{Q}\ \mathcal{A}$ **using** $\mathcal{I}$-*consistent* **by** *auto*
thus *?thesis*
by (*simp add*: $\mathcal{L}$-*alt-def set-eq-iff Bex-def*,
metis $\mathcal{L}$-*in-state-rename-iff equiv-f*)
qed

lemma (**in** *NFA*) $\mathcal{L}$-*left-rename-iff* :
assumes *equiv-f* : *NFA-is-equivalence-rename-fun* $(\!|\mathcal{Q} = \mathcal{Q}\ \mathcal{A}, \Sigma = \Sigma\ \mathcal{A}, \Delta = \Delta\ \mathcal{A}, \mathcal{I} = \mathcal{I}\ \mathcal{A}, \mathcal{F} = \{q\}|\!)$ *f*
and *q-in*: $q \in \mathcal{Q}\ \mathcal{A}$
shows $\mathcal{L}$-*left* (*NFA-rename-states* $\mathcal{A}$ *f*) (*f q*) = $\mathcal{L}$-*left* $\mathcal{A}$ *q*
proof −
obtain $\mathcal{A}'$ **where** $\mathcal{A}'$-*def*: $\mathcal{A}' = (\!|\mathcal{Q} = \mathcal{Q}\ \mathcal{A}, \Sigma = \Sigma\ \mathcal{A}, \Delta = \Delta\ \mathcal{A}, \mathcal{I} = \mathcal{I}\ \mathcal{A}, \mathcal{F} = \{q\}|\!)$
by *blast*

  from *wf-NFA* $\mathcal{A}'$*-def q-in* **have** *wf-*$\mathcal{A}'$: *NFA* $\mathcal{A}'$
    unfolding *NFA-def* **by** *simp*

  have *left-A*: $\mathcal{L}$*-left* $\mathcal{A}$ $q = \mathcal{L}$ $\mathcal{A}'$
    unfolding $\mathcal{L}$*-left-alt-def* $\mathcal{A}'$*-def* **..**

  have *left-reA*: $\mathcal{L}$*-left* (*NFA-rename-states* $\mathcal{A}$ f) (f q) = $\mathcal{L}$ (*NFA-rename-states* $\mathcal{A}'$ f)
    unfolding $\mathcal{L}$*-left-alt-def* $\mathcal{A}'$*-def*
    by (*simp add*: *NFA-rename-states-def*)

  from *equiv-f* **have** *equiv-f* $'$: *NFA-is-equivalence-rename-fun* $\mathcal{A}'$ f
     unfolding $\mathcal{A}'$*-def*
     by *simp*

  from *NFA.*$\mathcal{L}$*-rename-iff* [*OF wf-*$\mathcal{A}'$ *equiv-f* $'$]
       left-A left-reA
  show *?thesis* **by** *simp*
qed

lemma (**in** *NFA*) $\mathcal{L}$*-left-rename-iff-inj* :
assumes *inj-f* : *inj-on* f ($\mathcal{Q}$ $\mathcal{A}$)
    and *q-in*: $q \in \mathcal{Q}$ $\mathcal{A}$
shows $\mathcal{L}$*-left* (*NFA-rename-states* $\mathcal{A}$ f) (f q) = $\mathcal{L}$*-left* $\mathcal{A}$ q
proof −
  from *inj-f* **have** *equiv-f* : *NFA-is-equivalence-rename-fun* $(\!|\mathcal{Q} = \mathcal{Q}\ \mathcal{A},\ \Sigma = \Sigma\ \mathcal{A},\ \Delta = \Delta\ \mathcal{A},\ \mathcal{I} = \mathcal{I}\ \mathcal{A},\ \mathcal{F} = \{q\}|\!)$ f
    unfolding *NFA-is-equivalence-rename-fun-def inj-on-def*
    by *auto*

  from $\mathcal{L}$*-left-rename-iff* [*OF equiv-f q-in*]
  show *?thesis* **.**
qed

lemma (**in** *NFA*) *NFA-rename-states---accept* :
assumes *equiv-f* : *NFA-is-equivalence-rename-fun* $\mathcal{A}$ f
shows *NFA-accept* (*NFA-rename-states* $\mathcal{A}$ f) w = *NFA-accept* $\mathcal{A}$ w
using *assms*
by (*simp add*: *NFA-accept-alt-def* $\mathcal{L}$*-rename-iff*)

Renaming without combining states is fine.

lemma *NFA-is-equivalence-rename-funI---inj-used* :
  inj-on f ($\mathcal{Q}$ $\mathcal{A}$) $\Longrightarrow$ *NFA-is-equivalence-rename-fun* $\mathcal{A}$ f
by (*auto simp add*: *inj-on-def NFA-is-equivalence-rename-fun-def*)

Combining without renaming is fine as well.

lemma *NFA-is-equivalence-rename-funI---intro2* :
$(\forall\, q \in \mathcal{Q}\ \mathcal{A}.\ ((f\ q \in \mathcal{Q}\ \mathcal{A}) \wedge (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ (f\ q) = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q))) \Longrightarrow$ *NFA-is-equivalence-rename-fun* $\mathcal{A}$ f
by (*simp add*: *inj-on-def NFA-is-equivalence-rename-fun-def*, *metis*)

2.5 Isomorphy

definition *NFA-isomorphic* **where**
  NFA-isomorphic $\mathcal{A}1$ $\mathcal{A}2 \equiv$
   ($\exists f$. *inj-on* f ($\mathcal{Q}$ $\mathcal{A}1$) $\wedge$ $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1$ f)

lemma *NFA-isomorphic-refl* :
  NFA-isomorphic $\mathcal{A}$ $\mathcal{A}$
proof −
  have *inj-on id* ($\mathcal{Q}$ $\mathcal{A}$) **by** *simp*
  moreover
  have *NFA-rename-states* $\mathcal{A}$ *id* = $\mathcal{A}$ **by** *simp*
  ultimately

```
  show ?thesis unfolding NFA-isomorphic-def by metis
qed

lemma NFA-isomorphic-sym-impl :
assumes wf-NFA1 : NFA 𝒜1
    and equiv-𝒜1-𝒜2: NFA-isomorphic 𝒜1 𝒜2
shows NFA-isomorphic 𝒜2 𝒜1
proof −
  from equiv-𝒜1-𝒜2
  obtain f where inj-f : inj-on f (𝒬 𝒜1) and
               𝒜2-eq: 𝒜2 = NFA-rename-states 𝒜1 f
    unfolding NFA-isomorphic-def by auto

  obtain f' where f'-def : f' = inv-into (𝒬 𝒜1) f by auto
  with inj-f have f'-f: ⋀q. q ∈ (𝒬 𝒜1) ⟹ f' (f q) = q by simp

  from 𝒜2-eq have Q-𝒜2-eq: 𝒬 𝒜2 = f ` (𝒬 𝒜1)
    by (simp add: NFA-rename-states-def)
  with f'-f have inj-f' : inj-on f' (𝒬 𝒜2)
    by (simp add: inj-on-def)

  have 𝒜1-eq : 𝒜1 = NFA-rename-states 𝒜2 f'
  proof (rule NFA-rec.equality)
    show Σ 𝒜1 = Σ (NFA-rename-states 𝒜2 f')
      by (simp add: 𝒜2-eq NFA-remove-states-def)
  next
    from f'-f
    show 𝒬 𝒜1 = 𝒬 (NFA-rename-states 𝒜2 f')
      by (auto simp add: 𝒜2-eq NFA-remove-states-def image-iff)
  next
    from wf-NFA1 have ℐ 𝒜1 ⊆ 𝒬 𝒜1 by (simp add: NFA-def)
    with f'-f have ⋀q. q ∈ (ℐ 𝒜1) ⟹ f' (f q) = q by auto
    thus ℐ 𝒜1 = ℐ (NFA-rename-states 𝒜2 f')
      by (auto simp add: 𝒜2-eq NFA-remove-states-def image-iff)
  next
    from wf-NFA1 have ℱ 𝒜1 ⊆ 𝒬 𝒜1 by (simp add: NFA-def)
    with f'-f have ⋀q. q ∈ (ℱ 𝒜1) ⟹ f' (f q) = q by auto
    thus ℱ 𝒜1 = ℱ (NFA-rename-states 𝒜2 f')
      by (auto simp add: 𝒜2-eq NFA-remove-states-def image-iff)
  next
    from wf-NFA1 f'-f
    have ⋀q1 a q2. (q1, a, q2) ∈ (Δ 𝒜1) ⟹ f' (f q1) = q1 ∧ f' (f q2) = q2
       unfolding NFA-def by auto
    thus Δ 𝒜1 = Δ (NFA-rename-states 𝒜2 f')
      by (auto simp add: 𝒜2-eq NFA-remove-states-def, metis)
  next
    show more 𝒜1 = more (NFA-rename-states 𝒜2 f')
      by (simp add: NFA-remove-states-def)
  qed

  from 𝒜1-eq inj-f' show NFA-isomorphic 𝒜2 𝒜1 unfolding NFA-isomorphic-def by auto
qed

lemma NFA-isomorphic-sym :
assumes wf-NFA1: NFA 𝒜1
    and wf-NFA2: NFA 𝒜2
shows NFA-isomorphic 𝒜1 𝒜2 ⟷ NFA-isomorphic 𝒜2 𝒜1
using assms
by (metis NFA-isomorphic-sym-impl)

lemma NFA-isomorphic---implies-well-formed :
assumes wf-NFA1: NFA 𝒜1
```

  and *equiv-𝒜1-𝒜2*: *NFA-isomorphic 𝒜1 𝒜2*
shows *NFA 𝒜2*
using *assms*
by (*metis NFA-rename-states---is-well-formed NFA-isomorphic-def*)

lemma *NFA-isomorphic-trans* :
assumes *equiv-𝒜1-𝒜2*: *NFA-isomorphic 𝒜1 𝒜2*
  and *equiv-𝒜2-𝒜3*: *NFA-isomorphic 𝒜2 𝒜3*
shows *NFA-isomorphic 𝒜1 𝒜3*
proof −
 from *equiv-𝒜1-𝒜2*
 obtain *f1* **where** *inj-f1* : *inj-on f1* (*𝒬 𝒜1*) **and**
        𝒜2-eq: *𝒜2* = *NFA-rename-states 𝒜1 f1*
  unfolding *NFA-isomorphic-def* **by** *auto*

 from *equiv-𝒜2-𝒜3*
 obtain *f2* **where** *inj-f2* : *inj-on f2* (*𝒬 𝒜2*) **and**
        𝒜3-eq: *𝒜3* = *NFA-rename-states 𝒜2 f2*
  unfolding *NFA-isomorphic-def* **by** *auto*

 from *𝒜2-eq 𝒜3-eq* **have** *𝒜3-eq'* : *𝒜3* = *NFA-rename-states 𝒜1* (*f2* ∘ *f1*) **by** *simp*

 from *𝒜2-eq* **have** *𝒬 𝒜2* = *f1* ‘ (*𝒬 𝒜1*) **by** (*simp add*: *NFA-remove-states-def*)
 with *inj-f1 inj-f2* **have** *f2-f1-inj*: *inj-on* (*f2* ∘ *f1*) (*𝒬 𝒜1*)
  by (*simp add*: *inj-on-def*)

 from *𝒜3-eq' f2-f1-inj* **show** *NFA-isomorphic 𝒜1 𝒜3* **unfolding** *NFA-isomorphic-def* **by** *auto*
qed

Normaly, one is interested in only well-formed automata. This simplifies reasoning about isomorphy.

definition *NFA-isomorphic-wf* **where**
 NFA-isomorphic-wf 𝒜1 𝒜2 ≡ *NFA-isomorphic 𝒜1 𝒜2* ∧ *NFA 𝒜1*

lemma *NFA-isomorphic-wf-ℒ* :
assumes *equiv*: *NFA-isomorphic-wf 𝒜1 𝒜2*
shows ℒ *𝒜1* = ℒ *𝒜2*
proof −
 from *equiv*
 obtain *f* **where** *inj-f* : *inj-on f* (*𝒬 𝒜1*) **and**
        𝒜2-eq: *𝒜2* = *NFA-rename-states 𝒜1 f* **and**
        wf-𝒜1: *NFA 𝒜1*
  unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

 from *inj-f* **have** *NFA-is-equivalence-rename-fun 𝒜1 f* **by**
  (*simp add*: *NFA-is-equivalence-rename-funI---inj-used*)

 note *NFA.ℒ-rename-iff* [*OF wf-𝒜1 this*]
 with *𝒜2-eq* **show** *?thesis* **by** *simp*
qed

lemma *NFA-isomorphic-wf-Σ* :
assumes *equiv*: *NFA-isomorphic-wf 𝒜1 𝒜2*
shows Σ *𝒜1* = Σ *𝒜2*
proof −
 from *equiv*
 obtain *f* **where** *𝒜2-eq*: *𝒜2* = *NFA-rename-states 𝒜1 f*
  unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

 with *𝒜2-eq* **show** *?thesis* **by** *simp*
qed

lemma *NFA-isomorphic-wf-accept* :

```
assumes equiv: NFA-isomorphic-wf 𝒜1 𝒜2
shows NFA-accept 𝒜1 w = NFA-accept 𝒜2 w
using NFA-isomorphic-wf-ℒ [OF equiv]
by (simp add: set-eq-iff ℒ-def)

lemma NFA-isomorphic-wf-alt-def :
  NFA-isomorphic-wf 𝒜1 𝒜2 ⟷
   NFA-isomorphic 𝒜1 𝒜2 ∧ NFA 𝒜1 ∧ NFA 𝒜2
unfolding NFA-isomorphic-wf-def
by (metis NFA-isomorphic---implies-well-formed)

lemma NFA-isomorphic-wf-sym :
  NFA-isomorphic-wf 𝒜1 𝒜2 = NFA-isomorphic-wf 𝒜2 𝒜1
unfolding NFA-isomorphic-wf-alt-def
by (metis NFA-isomorphic-sym)

lemma NFA-isomorphic-wf-trans :
  ⟦NFA-isomorphic-wf 𝒜1 𝒜2; NFA-isomorphic-wf 𝒜2 𝒜3⟧ ⟹
   NFA-isomorphic-wf 𝒜1 𝒜3
unfolding NFA-isomorphic-wf-alt-def
by (metis NFA-isomorphic-trans)

lemma NFA-isomorphic-wf-refl :
  NFA 𝒜1 ⟹ NFA-isomorphic-wf 𝒜1 𝒜1
unfolding NFA-isomorphic-wf-def
by (simp add: NFA-isomorphic-refl)

lemma NFA-isomorphic-wf-intro :
  ⟦NFA 𝒜1; NFA-isomorphic 𝒜1 𝒜2⟧ ⟹ NFA-isomorphic-wf 𝒜1 𝒜2
unfolding NFA-isomorphic-wf-def by simp

lemma NFA-isomorphic-wf-D :
  NFA-isomorphic-wf 𝒜1 𝒜2 ⟹ NFA 𝒜1
  NFA-isomorphic-wf 𝒜1 𝒜2 ⟹ NFA 𝒜2
  NFA-isomorphic-wf 𝒜1 𝒜2 ⟹ NFA-isomorphic 𝒜1 𝒜2
  NFA-isomorphic-wf 𝒜1 𝒜2 ⟹ NFA-isomorphic 𝒜2 𝒜1
unfolding NFA-isomorphic-wf-alt-def
by (simp-all, metis NFA-isomorphic-sym)

lemma NFA-isomorphic---NFA-rename-states :
inj-on f (𝒬 𝒜) ⟹ NFA-isomorphic 𝒜 (NFA-rename-states 𝒜 f)
unfolding NFA-isomorphic-def
by blast

lemma NFA-isomorphic-wf---NFA-rename-states :
⟦inj-on f (𝒬 𝒜); NFA 𝒜⟧ ⟹ NFA-isomorphic-wf 𝒜 (NFA-rename-states 𝒜 f)
unfolding NFA-isomorphic-def NFA-isomorphic-wf-def
by blast

lemma NFA-isomorphic-wf---rename-states-cong :
fixes 𝒜1 :: ('q1, 'a) NFA-rec
fixes 𝒜2 :: ('q2, 'a) NFA-rec
assumes inj-f1 : inj-on f1 (𝒬 𝒜1) and
        inj-f2 : inj-on f2 (𝒬 𝒜2) and
        equiv: NFA-isomorphic-wf 𝒜1 𝒜2
shows NFA-isomorphic-wf (NFA-rename-states 𝒜1 f1) (NFA-rename-states 𝒜2 f2)
proof −
  note wf-𝒜1 = NFA-isomorphic-wf-D(1)[OF equiv]
  note wf-𝒜2 = NFA-isomorphic-wf-D(2)[OF equiv]

  note eq-1 = NFA-isomorphic-wf---NFA-rename-states [OF inj-f1 wf-𝒜1]
  note eq-2 = NFA-isomorphic-wf---NFA-rename-states [OF inj-f2 wf-𝒜2]
```

```
  from eq-1 eq-2 equiv show ?thesis
    by (metis NFA-isomorphic-wf-trans NFA-isomorphic-wf-sym)
qed

lemma NFA-isomorphic-wf---NFA-remove-unreachable-states :
assumes equiv: NFA-isomorphic-wf 𝒜1 𝒜2
shows NFA-isomorphic-wf (NFA-remove-unreachable-states 𝒜1) (NFA-remove-unreachable-states 𝒜2)
proof −
  from equiv
  obtain f where inj-f : inj-on f (𝒬 𝒜1) and
                 𝒜2-eq: 𝒜2 = NFA-rename-states 𝒜1 f and
                 wf-𝒜1: NFA 𝒜1 and
                 wf-𝒜2: NFA 𝒜2
    unfolding NFA-isomorphic-wf-alt-def NFA-isomorphic-def by auto

  have 𝒬 (NFA-remove-unreachable-states 𝒜1) ⊆ 𝒬 𝒜1
    unfolding NFA-remove-unreachable-states-def by auto
  with inj-f have inj-f' : inj-on f (𝒬 (NFA-remove-unreachable-states 𝒜1))
    by (rule subset-inj-on)

  from NFA.ℒ-left-rename-iff-inj [OF wf-𝒜1 inj-f]
  have unreach-lem:
    ⋀q. q ∈ 𝒬 𝒜1 ⟹
     f q ∈ NFA-unreachable-states (NFA-rename-states 𝒜1 f) ⟷
     q ∈ (NFA-unreachable-states 𝒜1 ∩ 𝒬 𝒜1)
    by (simp add: NFA-unreachable-states-alt-def)

  have diff-lem :
   ⋀Q. Q ⊆ 𝒬 𝒜1 ⟹
    f ‘ Q − NFA-unreachable-states (NFA-rename-states 𝒜1 f) =
    f ‘ (Q − NFA-unreachable-states 𝒜1)
  apply (rule set-eqI)
  apply (insert unreach-lem)
  apply (auto)
  done

  have 𝒜2-eq' :
    NFA-remove-unreachable-states 𝒜2 = NFA-rename-states (NFA-remove-unreachable-states 𝒜1) f
    proof (rule  NFA-rec.equality)
      show Σ (NFA-remove-unreachable-states 𝒜2) =
            Σ (NFA-rename-states (NFA-remove-unreachable-states 𝒜1) f)
           NFA.more (NFA-remove-unreachable-states 𝒜2) =
            NFA.more (NFA-rename-states (NFA-remove-unreachable-states 𝒜1) f)

        unfolding NFA-remove-unreachable-states-def 𝒜2-eq
        by simp-all
    next
      from diff-lem [of 𝒬 𝒜1]
      show 𝒬 (NFA-remove-unreachable-states 𝒜2) =
            𝒬 (NFA-rename-states (NFA-remove-unreachable-states 𝒜1) f)
        unfolding NFA-remove-unreachable-states-def 𝒜2-eq
        by simp
    next
      from diff-lem [OF NFA.ℐ-consistent [OF wf-𝒜1]]
      show ℐ (NFA-remove-unreachable-states 𝒜2) =
            ℐ (NFA-rename-states (NFA-remove-unreachable-states 𝒜1) f)
        unfolding NFA-remove-unreachable-states-def 𝒜2-eq
        by simp
    next
      from diff-lem [OF NFA.ℱ-consistent [OF wf-𝒜1]]
      show ℱ (NFA-remove-unreachable-states 𝒜2) =
            ℱ (NFA-rename-states (NFA-remove-unreachable-states 𝒜1) f)
```

unfolding *NFA-remove-unreachable-states-def* $\mathcal{A}2$*-eq*
by *simp*
next
from *unreach-lem NFA.*Δ*-consistent* [*OF wf-*$\mathcal{A}1$]
have $\bigwedge q\ a\ q'.\ (q,\ a,\ q') \in \Delta\ \mathcal{A}1\ \wedge$
$f\ q \notin$ *NFA-unreachable-states* (*NFA-rename-states* $\mathcal{A}1\ f$) $\wedge$
$f\ q' \notin$ *NFA-unreachable-states* (*NFA-rename-states* $\mathcal{A}1\ f$) $\longleftrightarrow$
$(q,\ a,\ q') \in \Delta\ \mathcal{A}1\ \wedge$
$q \notin$ *NFA-unreachable-states* $\mathcal{A}1\ \wedge$
$q' \notin$ *NFA-unreachable-states* $\mathcal{A}1$
by *auto*
hence $\bigwedge q\ a\ q'.\ (q,\ a,\ q') \in \Delta$ (*NFA-remove-unreachable-states* $\mathcal{A}2$) $\longleftrightarrow$
$(q,\ a,\ q') \in \Delta$ (*NFA-rename-states* (*NFA-remove-unreachable-states* $\mathcal{A}1$) f)
unfolding *NFA-remove-unreachable-states-def* $\mathcal{A}2$*-eq*
by (*simp*, *blast*)
thus Δ (*NFA-remove-unreachable-states* $\mathcal{A}2$) =
Δ (*NFA-rename-states* (*NFA-remove-unreachable-states* $\mathcal{A}1$) f)
by *auto*
qed

from *inj-f'* $\mathcal{A}2$*-eq'* *NFA-remove-unreachable-states---is-well-formed*[*OF wf-*$\mathcal{A}1$]
show *?thesis*
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def*
by *blast*
qed

lemma *NFA-is-initially-connected---NFA-rename-states* :
assumes *connected*: *NFA-is-initially-connected* $\mathcal{A}$
shows *NFA-is-initially-connected* (*NFA-rename-states* $\mathcal{A}\ f$)
unfolding *NFA-is-initially-connected-alt-def*
proof (*intro ballI*)
fix *q2*
let *?A2* = *NFA-rename-states* $\mathcal{A}\ f$
assume *q2-in*: $q2 \in \mathcal{Q}$ *?A2*
from *q2-in* **obtain** *q1* **where** *q1-in*: $q1 \in \mathcal{Q}\ \mathcal{A}$ **and** *q2-eq*: $q2 = f\ q1$ **by** *auto*

from *connected q1-in* **obtain** *i1 w* **where**
i1-in: $i1 \in \mathcal{I}\ \mathcal{A}$ **and** *w-in1*: $w \in$ *lists* ($\Sigma\ \mathcal{A}$) **and**
reach1: *LTS-is-reachable* ($\Delta\ \mathcal{A}$) *i1 w q1*
unfolding *NFA-is-initially-connected-alt-def*
by *blast*

have *i2-in*: $f\ i1 \in \mathcal{I}$ *?A2*
using *i1-in* **by** *auto*

have *w-in2*: $w \in$ *lists* (Σ *?A2*)
using *w-in1* **by** *simp*

have *reach2* : *LTS-is-reachable* (Δ *?A2*) ($f\ i1$) w ($f\ q1$)
using *LTS-is-reachable---NFA-rename-statesE* [*OF reach1*, *of f*] **.**

from *i2-in w-in2 reach2*
show $\exists\, i{\in}\mathcal{I}$ *?A2*. $\exists\, w{\in}$*lists* (Σ *?A2*). *LTS-is-reachable* (Δ *?A2*) *i w q2*
by (*simp add*: *Bex-def q2-eq*) *blast*
qed

lemma *NFA-is-initially-connected---NFA-isomorphic* :
assumes *equiv*: *NFA-isomorphic* $\mathcal{A}1\ \mathcal{A}2$
and *connected-A1*: *NFA-is-initially-connected* $\mathcal{A}1$
shows *NFA-is-initially-connected* $\mathcal{A}2$
proof −
from *equiv* **obtain** f **where** $\mathcal{A}2$*-eq*: $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1\ f$

unfolding *NFA-isomorphic-def* **by** *auto*

from *NFA-is-initially-connected---NFA-rename-states* [*OF connected-A1*, *of f*, *folded* $\mathcal{A}2$*-eq*]
show *?thesis* **.**
qed

lemma *NFA-is-initially-connected---NFA-isomorphic-wf* :
fixes $\mathcal{A}1$ :: ('*q1*, '*a*) *NFA-rec*
fixes $\mathcal{A}2$ :: ('*q2*, '*a*) *NFA-rec*
assumes *iso*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-is-initially-connected* $\mathcal{A}1$ = *NFA-is-initially-connected* $\mathcal{A}2$
using *assms*
by (*metis NFA-is-initially-connected---NFA-isomorphic*
NFA-isomorphic-wf-sym NFA-isomorphic-wf-def)

2.6 Efficient Construction of NFAs

In the following automata are constructed by sequentially adding states together to an initially empty automaton. An *empty* automaton contains an alphabet of labels and a set of initial states. Adding states updates the set of states, the transition relation and the set of accepting states.

This construction is used to add only the reachable states to an automaton.

definition *NFA-initial-automaton* :: '*q set* $\Rightarrow$ '*a set* $\Rightarrow$ ('*q*, '*a*) *NFA-rec* **where**
NFA-initial-automaton I A $\equiv$ $(\!|\ \mathcal{Q}=\{\},\ \Sigma=A,\ \Delta = \{\},\ \mathcal{I}=I,\ \mathcal{F} = \{\}\ |\!)$

definition *NFA-insert-state* :: ('*q* $\Rightarrow$ *bool*) $\Rightarrow$ ('*q*, '*a*) *LTS* $\Rightarrow$ '*q* $\Rightarrow$ ('*q*, '*a*) *NFA-rec* $\Rightarrow$ ('*q*, '*a*) *NFA-rec* **where**
NFA-insert-state FP D q $\mathcal{A}$ $\equiv$
$(\!|\ \mathcal{Q}=insert\ q\ (\mathcal{Q}\ \mathcal{A}),\ \Sigma=\Sigma\ \mathcal{A},\ \Delta = \Delta\ \mathcal{A} \cup \{qsq\ .\ qsq \in D \wedge fst\ qsq = q\},$
$\mathcal{I}=\mathcal{I}\ \mathcal{A},\ \mathcal{F} = if\ (FP\ q)\ then\ insert\ q\ (\mathcal{F}\ \mathcal{A})\ else\ (\mathcal{F}\ \mathcal{A})|\!)$

definition *NFA-construct-reachable* **where**
NFA-construct-reachable I A FP D =
Finite-Set.fold (*NFA-insert-state FP D*) (*NFA-initial-automaton I A*)
(*accessible* (*LTS-forget-labels D*) *I*)

lemma *NFA-insert-state---comp-fun-commute* :
comp-fun-commute (*NFA-insert-state FP D*)
apply (*simp add*: *NFA-insert-state-def comp-fun-commute-def o-def*)
apply (*subst fun-eq-iff*)
apply *simp*
apply (*metis Un-commute Un-left-commute insert-commute*)
done

lemma *fold-NFA-insert-state* :
finite Q $\Longrightarrow$ *Finite-Set.fold* (*NFA-insert-state FP D*) $\mathcal{A}$ *Q* =
$(\!|\ \mathcal{Q}=Q \cup (\mathcal{Q}\ \mathcal{A}),\ \Sigma=\Sigma\ \mathcal{A},\ \Delta = \Delta\ \mathcal{A} \cup \{qsq.\ qsq \in D \wedge fst\ qsq \in Q\},$
$\mathcal{I}=\mathcal{I}\ \mathcal{A},\ \mathcal{F} = (\mathcal{F}\ \mathcal{A}) \cup \{q.\ q \in Q \wedge FP\ q\}\ |\!)$
apply (*induct rule*: *finite-induct*)
apply (*simp*)
apply (*simp add*: *comp-fun-commute.fold-insert* [*OF NFA-insert-state---comp-fun-commute*])
apply (*auto simp add*: *NFA-insert-state-def*)
done

lemma *NFA-construct-reachable-simp* :
finite (*accessible* (*LTS-forget-labels D*) *I*) $\Longrightarrow$
NFA-construct-reachable I A FP D =
$(\!|\mathcal{Q} = accessible\ (LTS\text{-}forget\text{-}labels\ D)\ I,\ \Sigma = A,\ \Delta = \{qsq.\ qsq \in D \wedge$
$fst\ qsq \in accessible\ (LTS\text{-}forget\text{-}labels\ D)\ I\},\ \mathcal{I} = I,$
$\mathcal{F} = \{q \in accessible\ (LTS\text{-}forget\text{-}labels\ D)\ I.\ FP\ q\}|\!)$
by (*simp add*: *NFA-construct-reachable-def*
fold-NFA-insert-state NFA-initial-automaton-def)

Now show that this can be used to remove unreachable states

lemma (**in** *NFA*) *NFA-remove-unreachable-states-implementation* :
assumes *I-OK*: $I = \mathcal{I}\ \mathcal{A}$
  and *A-OK*: $A = \Sigma\ \mathcal{A}$
  and *FP-OK*: $\bigwedge q.\ q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow FP\ q \longleftrightarrow q \in \mathcal{F}\ \mathcal{A}$
  and *D-OK*: $D = \Delta\ \mathcal{A}$
shows
 (*NFA-remove-unreachable-states* $\mathcal{A}$) = (*NFA-construct-reachable I A FP D*)
 (**is** *?ls* = *?rs*)
proof −
 let *?D* = *LTS-forget-labels* ($\Delta\ \mathcal{A}$)
 note *access-eq* = *LTS-is-reachable-from-initial-alt-def*
 have *access-eq'* : *accessible ?D* ($\mathcal{I}\ \mathcal{A}$) = $\mathcal{Q}\ \mathcal{A}$ − *NFA-unreachable-states* $\mathcal{A}$
   (**is** *?ls'* = *?rs'*)
 proof (*intro set-eqI iffI*)
  fix *q*
  assume $q \in$ *?rs'*
  thus $q \in$ *?ls'*
   by (*auto simp add*: *NFA-unreachable-states-def access-eq LTS-is-unreachable-def*)
 next
  fix *q*
  assume *ls*: $q \in$ *?ls'*
  with *LTS-is-reachable-from-initial-subset* **have** *q-in-Q*: $q \in \mathcal{Q}\ \mathcal{A}$ **by** *auto*
  from $\mathcal{I}$*-consistent* **have** *I-subset-Q*: $\bigwedge q.\ q \in \mathcal{I}\ \mathcal{A} \Longrightarrow q \in \mathcal{Q}\ \mathcal{A}$ **by** *auto*

  from *ls q-in-Q I-subset-Q* **show** $q \in$ *?rs'*
   by (*auto simp add*: *NFA-unreachable-states-def access-eq LTS-is-unreachable-def Bex-def*)
    (*metis LTS-is-reachable---labels*)
 qed

 have $\mathcal{I}$*-diff-eq*: $\mathcal{I}\ \mathcal{A}$ − *NFA-unreachable-states* $\mathcal{A} = \mathcal{I}\ \mathcal{A}$
  by (*auto simp add*: *NFA-unreachable-states-*$\mathcal{I}$)

 from $\mathcal{F}$*-consistent FP-OK* **have** $\mathcal{F}$*-eq*:
  $\{q \in \mathcal{Q}\ \mathcal{A}.\ q \notin$ *NFA-unreachable-states* $\mathcal{A} \wedge FP\ q\}$ =
  $\mathcal{F}\ \mathcal{A}$ − *NFA-unreachable-states* $\mathcal{A}$ **by** *auto*

 from *LTS-is-reachable-from-initial-finite*
 show *?ls* = *?rs*
  apply (*simp add*: *NFA-construct-reachable-simp A-OK I-OK D-OK*)
  apply (*simp add*: *NFA-remove-unreachable-states-def*
     NFA-remove-states-def access-eq')
  apply (*auto simp add*: $\mathcal{I}$*-diff-eq* $\mathcal{F}$*-eq* Δ*-consistent*)
  apply (*metis* Δ*-consistent NFA-unreachable-states-extend*)
 done
qed

Now let's implement efficiently constructing NFAs. During implementation, the states are renamend as well.

definition *NFA-construct-reachable-map-OK* **where**
 NFA-construct-reachable-map-OK S rm DD rm' $\longleftrightarrow$
  DD $\subseteq$ *dom rm'* $\wedge$
  ($\forall\ q\ r'.$ *rm q* = *Some r'* $\longrightarrow$ *rm' q* = *Some r'*) $\wedge$
  inj-on rm' ($S \cap$ *dom rm'*)

lemma *NFA-construct-reachable-map-OK-I* [*intro*!] :
⟦*DD* $\subseteq$ *dom rm'*; $\bigwedge q\ r'.$ *rm q* = *Some r'* $\Longrightarrow$ *rm' q* = *Some r'*; *inj-on rm'* ($S \cap$ *dom rm'*)⟧ $\Longrightarrow$
 NFA-construct-reachable-map-OK S rm DD rm'
unfolding *NFA-construct-reachable-map-OK-def* **by** *simp*

lemma *NFA-construct-reachable-map-OK-insert-DD* :
 NFA-construct-reachable-map-OK S rm (*insert q DD*) *rm'* $\longleftrightarrow$

$q \in dom\ rm' \wedge NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ S\ rm\ DD\ rm'$
unfolding *NFA-construct-reachable-map-OK-def* **by** *simp*

lemma *NFA-construct-reachable-map-OK-trans* :
$[\![NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ S\ rm\ DD\ rm';$
$NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ S\ rm'\ DD'\ rm'';$
$DD'' \subseteq DD \cup DD']\!] \Longrightarrow$
$NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ S\ rm\ DD''\ rm''$
unfolding *NFA-construct-reachable-map-OK-def*
by (*simp add*: *subset-iff dom-def*) *metis*

definition *NFA-construct-reachable-abstract-impl-invar* **where**
$NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}invar\ I\ A\ FP\ D \equiv (\lambda((rm, \mathcal{A}), wl).$
$(\exists s.\ NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ (accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I))\ empty$
$(s \cup set\ I \cup set\ wl \cup \{q'.\ \exists a\ q.\ q{\in}s \wedge (q,a,q'){\in}\ D\})\ rm \wedge$
$(accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I)\ =$
$accessible\text{-}restrict\ (LTS\text{-}forget\text{-}labels\ D)\ s\ (set\ wl)) \wedge$
$(\mathcal{A} = NFA\text{-}rename\text{-}states$
$(\!|\mathcal{Q} = s,\ \Sigma = A,\ \Delta = \{qsq.\ qsq \in D \wedge fst\ qsq \in s\},\ \mathcal{I} = set\ I,$
$\mathcal{F} = \{q \in s.\ FP\ q\}|\!)\ (the \circ rm))))$

definition *NFA-construct-reachable-abstract-impl-weak-invar* **where**
$NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}weak\text{-}invar\ I\ A\ FP\ D \equiv (\lambda(rm, \mathcal{A}).$
$(\exists s.\ NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ (accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I))\ empty$
$(s \cup set\ I \cup \{q'.\ \exists a\ q.\ q{\in}s \wedge (q,a,q'){\in}\ D\})\ rm \wedge$
$s \subseteq accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I) \wedge$
$(\mathcal{A} = NFA\text{-}rename\text{-}states$
$(\!|\mathcal{Q} = s,\ \Sigma = A,\ \Delta = \{qsq.\ qsq \in D \wedge fst\ qsq \in s\},\ \mathcal{I} = set\ I,$
$\mathcal{F} = \{q \in s.\ FP\ q\}|\!)\ (the \circ rm))))$

lemma *NFA-construct-reachable-abstract-impl-invar-weaken* :
assumes *invar*: $NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}invar\ I\ A\ FP\ D\ ((rm, \mathcal{A}), wl)$
shows $NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}weak\text{-}invar\ I\ A\ FP\ D\ (rm, \mathcal{A})$
using *assms*
unfolding *NFA-construct-reachable-abstract-impl-weak-invar-def*
NFA-construct-reachable-abstract-impl-invar-def
accessible-restrict-def NFA-construct-reachable-map-OK-def
apply *clarify*
apply (*rule-tac x=s* **in** *exI*)
apply *auto*
done

fun *NFA-construct-reachable-abstract-impl-foreach-invar* **where**
$NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}foreach\text{-}invar\ S\ D\ rm\ D0\ q\ it\ (rm', D', N) =$
$(let\ D'' = \{(a, q') .\ (q, a, q') \in D\} - it;\ qS = snd\ `\ D''\ in$
$(NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ S\ rm\ qS\ rm' \wedge$
$set\ N = qS \wedge$
$D' = D0 \cup \{(the\ (rm'\ q),\ a,\ the\ (rm'\ q')) \mid a\ q'.\ (a, q') \in D''\}))$
declare *NFA-construct-reachable-abstract-impl-foreach-invar.simps* [*simp del*]

definition *NFA-construct-reachable-abstract-impl-step* **where**
$NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}step\ S\ D\ rm\ D0\ q =$
FOREACHi
$(NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}foreach\text{-}invar\ S\ D\ rm\ D0\ q)$
$\{(a, q') .\ (q, a, q') \in D\}$
$(\lambda(a, q')\ (rm, D', N).\ do\ \{$
$(rm', r') \leftarrow SPEC\ (\lambda(rm', r').\ NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ S\ rm\ \{q'\}\ rm' \wedge rm'\ q' = Some\ r');$
$RETURN\ (rm', insert\ (the\ (rm\ q),\ a,\ r')\ D',\ q'\ \#\ N)$
$\})\ (rm, D0, [])$

lemma *NFA-construct-reachable-abstract-impl-step-correct* :
assumes *fin*: $finite\ \{(a, q').\ (q, a, q') \in D\}$

and *inj-rm*: *inj-on rm* $(S \cap dom\ rm)$
and *q-in-dom*: $q \in dom\ rm$
shows *NFA-construct-reachable-abstract-impl-step S D rm D0 q* $\leq$
SPEC (*NFA-construct-reachable-abstract-impl-foreach-invar S D rm D0 q* {})
unfolding *NFA-construct-reachable-abstract-impl-step-def*
apply (*intro refine-vcg*)
apply (*fact fin*)
apply (*simp add*: *NFA-construct-reachable-abstract-impl-foreach-invar.simps*
NFA-construct-reachable-map-OK-def inj-rm)
apply (*clarify*, *simp*)
apply (*rename-tac it a* q' rm' D' *N* rm'' *r*)
defer
apply *simp*
proof −
fix *it a* q' rm' D' *N* rm'' r'
assume *in-it*: $(a, q') \in it$
and *it-subset*: $it \subseteq \{(a, q').\ (q, a, q') \in D\}$
and *invar*: *NFA-construct-reachable-abstract-impl-foreach-invar S D rm D0 q it* (rm', D', N)
and *map-OK*: *NFA-construct-reachable-map-OK S* rm' $\{q'\}$ rm''
and rm''-q': $rm''\ q' = Some\ r'$
from *q-in-dom* **obtain** *r* **where** *rm-q*: $rm\ q = Some\ r$ **by** *auto*

def $D'' \equiv \{(a, q').\ (q, a, q') \in D\} - it$
have D''-*intro* : $(\{(a, q').\ (q, a, q') \in D\} - (it - \{(a, q')\})) = insert\ (a, q')\ D''$
using *in-it it-subset* D''-*def* **by** *auto*

from *invar* **have**
rm'-*OK*: *NFA-construct-reachable-map-OK S rm* (*snd* ‘ D'') rm' **and**
set-N-eq: *set N* = *snd* ‘ D'' **and**
D'-*eq*: $D' = D0 \cup \{(the\ (rm'\ q),\ a,\ the\ (rm'\ q'))\ |a\ q'.\ (a, q') \in D''\}$
unfolding *NFA-construct-reachable-abstract-impl-foreach-invar.simps*
by (*simp-all add*: *Let-def* D''-*def*)

have *prop1*: *NFA-construct-reachable-map-OK S rm* (*insert* q' (*snd* ‘ D'')) rm''
using *map-OK* rm'-*OK*
unfolding *NFA-construct-reachable-map-OK-def*
by *auto*

have *prop2*: $insert\ (the\ (rm'\ q),\ a,\ r')\ D' =$
$D0 \cup \{(the\ (rm''\ q),\ aa,\ the\ (rm''\ q'a))\ |aa\ q'a.\ aa = a \land q'a = q' \lor (aa, q'a) \in D''\}$
(**is** $?ls = D0 \cup ?rs$)
proof −
from rm'-*OK rm-q* **have** rm'-*q*: $rm'\ q = Some\ r$
unfolding *NFA-construct-reachable-map-OK-def* **by** *simp*
with *map-OK* **have** rm''-*q*: $rm''\ q = Some\ r$
unfolding *NFA-construct-reachable-map-OK-def* **by** *simp*

have *step1*: $?rs = insert\ (the\ (rm''\ q),\ a,\ the\ (rm''\ q'))\ \{(the\ (rm''\ q),\ a,\ the\ (rm''\ q'))\ |a\ q'.\ (a, q') \in D''\}$
(**is** - = (*insert* - $?rs'$))
by *auto*

from rm'-*OK map-OK* **have** rm''-*eq*: $\bigwedge a\ q'.\ (a, q') \in D'' \Longrightarrow rm''\ q' = rm'\ q'$
unfolding *NFA-construct-reachable-map-OK-def dom-def subset-iff*
by *auto*

from rm''-*eq* **have** D'-eq': $D' = D0 \cup ?rs'$ **unfolding** D'-*eq* rm'-*q* rm''-*q* **by** *auto metis*

show *?thesis* **unfolding** D'-eq' *step1* **by** (*simp add*: rm'-*q* rm''-*q* rm''-q')
qed

show *NFA-construct-reachable-abstract-impl-foreach-invar S D rm D0 q* $(it - \{(a, q')\})$
$(rm'',\ insert\ (the\ (rm'\ q),\ a,\ r')\ D',\ q' \# N)$

```
  unfolding NFA-construct-reachable-abstract-impl-foreach-invar.simps D''-intro
  by (simp add: Let-def set-N-eq prop1 prop2)
qed

definition NFA-construct-reachable-abstract-impl where
 NFA-construct-reachable-abstract-impl I A FP D =
  do {
    (rm, I') ← SPEC (λ(rm, I').
       NFA-construct-reachable-map-OK (accessible (LTS-forget-labels D) (set I)) empty (set I) rm ∧
       I' = (the ∘ rm) ` (set I));
   ((rm, 𝒜), -) ← WORKLISTIT (NFA-construct-reachable-abstract-impl-invar I A FP D)
    (λ-. True)
    (λ(rm, 𝒜) q. do {
       ASSERT (q ∈ dom rm ∧ q ∈ accessible (LTS-forget-labels D) (set I) ∧
              NFA-construct-reachable-abstract-impl-weak-invar I A FP D (rm, 𝒜));
       if (the (rm q) ∈ 𝒬 𝒜) then
         (RETURN ((rm, 𝒜), []))
       else
         do {
           (rm', D', N) ← SPEC (NFA-construct-reachable-abstract-impl-foreach-invar
               (accessible (LTS-forget-labels D) (set I)) D rm (Δ 𝒜) q {});
           RETURN ((rm', ⦇ 𝒬=insert (the (rm q)) (𝒬 𝒜), Σ=Σ 𝒜, Δ = D',
                        ℐ=ℐ 𝒜, ℱ = if (FP q) then (insert (the (rm q)) (ℱ 𝒜)) else (ℱ 𝒜)⦈), N)
         }
       }) ((rm, ⦇ 𝒬={}, Σ=A, Δ = {}, ℐ=I', ℱ={} ⦈), I);
   RETURN 𝒜
  }

lemma NFA-construct-reachable-abstract-impl-correct :
fixes D :: ('q × 'a × 'q) set and I
defines S ≡ (accessible (LTS-forget-labels D) (set I))
assumes fin-S: finite S
shows NFA-construct-reachable-abstract-impl I A FP D ≤
      SPEC (λ𝒜. NFA-isomorphic (NFA-construct-reachable (set I) A FP D) (𝒜::('q2, 'a) NFA-rec))
unfolding NFA-construct-reachable-abstract-impl-def S-def[symmetric]
apply (intro refine-vcg WORKLISTIT-rule)
apply (simp-all split: prod.splits split del: if-splits)

  apply (clarify, simp)
  apply (rename-tac rm)
defer
  apply (clarify, simp)
  apply (rename-tac rm)
  apply (subgoal-tac wf (inv-image (measure (λ𝒜. card S − card (𝒬 𝒜))) snd))
  apply assumption
defer
   apply (clarify)
   apply (intro refine-vcg)
   apply simp
   apply (rename-tac rm' wl q rm 𝒜)
   apply (intro conjI)
defer
defer
defer
   apply (clarify, simp)
   apply (rename-tac rm' wl q rm 𝒜 r)
defer
   apply clarify
   apply (simp split del: if-splits)
   apply (rename-tac rm'' wl q rm 𝒜 rm' D' N r)
   apply (intro conjI)
defer
```

defer
  apply (*clarify*, *simp*)
  apply (*rename-tac rm′ rm* $\mathcal{A}$)
defer
proof −
 fix *rm* :: ′*q* ⇒ ′*q2 option*
 assume *rm-OK*: *NFA-construct-reachable-map-OK S empty* (*set I*) *rm*

 thus *NFA-construct-reachable-abstract-impl-invar I A FP D*
   ((*rm*, (|$\mathcal{Q}$ = {}, Σ = *A*, Δ = {}, $\mathcal{I}$ = (*the* ∘ *rm*) ‘ (*set I*), $\mathcal{F}$ = {}|)), *I*)
  unfolding *NFA-construct-reachable-abstract-impl-invar-def*
  apply (*simp*)
  apply (*rule exI* [**where** *x* = {}])
  apply (*simp add*: *accessible-restrict-def NFA-rename-states-def S-def*)
 done
next
 show *wf* (*inv-image* (*measure* (λ$\mathcal{A}$. *card S* − *card* ($\mathcal{Q}$ $\mathcal{A}$))) *snd*)
  by (*intro wf-inv-image wf-measure*)
next
 fix *rm* :: ′*q* ⇒ ′*q2 option* **and** $\mathcal{A}$

 note *reach-simp* = *NFA-construct-reachable-simp* [*OF fin-S*[*unfolded S-def*]]
 assume *NFA-construct-reachable-abstract-impl-invar I A FP D* ((*rm*, $\mathcal{A}$), [])
 thus *NFA-isomorphic* (*NFA-construct-reachable* (*set I*) *A FP D*) $\mathcal{A}$
  unfolding *NFA-construct-reachable-abstract-impl-invar-def*
  apply (*simp add*: *reach-simp S-def*[*symmetric*])
  apply (*rule NFA-isomorphic---NFA-rename-states*)
  apply (*simp add*: *inj-on-def NFA-construct-reachable-map-OK-def dom-def subset-iff Ball-def*)
  apply (*metis the.simps*)
 done
next
 fix *rm* :: ′*q* ⇒ ′*q2 option*
 fix $\mathcal{A}$ *q wl*
 assume *invar*: *NFA-construct-reachable-abstract-impl-invar I A FP D* ((*rm*, $\mathcal{A}$), *q* # *wl*)

 from *invar* **obtain** *s* **where**
  S-eq: *S* = *accessible-restrict* (*LTS-forget-labels D*) *s* (*insert q* (*set wl*)) **and**
  rm-OK: *NFA-construct-reachable-map-OK S empty* (*insert q* (*s* ∪ *set I* ∪
    set wl ∪ {*q*′. ∃ *a q*. *q* ∈ *s* ∧ (*q*, *a*, *q*′) ∈ *D*})) *rm* **and**
  $\mathcal{A}$*-eq*: $\mathcal{A}$ = *NFA-rename-states*
   (|$\mathcal{Q}$ = *s*, Σ = *A*, Δ = {*qsq* ∈ *D*. *fst qsq* ∈ *s*}, $\mathcal{I}$ = *set I*, $\mathcal{F}$ = {*q* ∈ *s*. *FP q*}|)
   (*the* ∘ *rm*)
  unfolding *NFA-construct-reachable-abstract-impl-invar-def S-def*[*symmetric*]
 by *auto*

 from *rm-OK* **show** *q* ∈ *dom rm*
  unfolding *NFA-construct-reachable-map-OK-def* **by** *simp*

 from *S-eq* **show** *q* ∈ *S*
  using *accessible-restrict-subset-ws*[*of insert q* (*set wl*) *LTS-forget-labels D s*]
  by *simp*

 from *NFA-construct-reachable-abstract-impl-invar-weaken*[*OF invar*]
 show *NFA-construct-reachable-abstract-impl-weak-invar I A FP D* (*rm*, $\mathcal{A}$) **by** *simp*
next
 fix *rm* :: ′*q* ⇒ ′*q2 option* **and** $\mathcal{A}$ *q wl r*

 assume *invar*: *NFA-construct-reachable-abstract-impl-invar I A FP D* ((*rm*, $\mathcal{A}$), *q* # *wl*) **and**
   rm-q-eq: *rm q* = *Some r* **and**
   rm-q: *r* ∈ $\mathcal{Q}$ $\mathcal{A}$ **and**
   q-in-S: *q* ∈ *S*

show *NFA-construct-reachable-abstract-impl-invar I A FP D* $((rm, \mathcal{A}), wl)$
proof –
from *invar* **obtain** *s* **where**
rm-OK: *NFA-construct-reachable-map-OK S empty* $(insert\ q\ (s \cup set\ I \cup$
$set\ wl \cup \{q'.\ \exists\, a\ q.\ q \in s \wedge (q, a, q') \in D\}))$ *rm* **and**
S-eq: $S = accessible\text{-}restrict\ (LTS\text{-}forget\text{-}labels\ D)\ s\ (insert\ q\ (set\ wl))$ **and**
$\mathcal{A}$*-eq*: $\mathcal{A} = NFA\text{-}rename\text{-}states$
$(\!|\mathcal{Q} = s, \Sigma = A, \Delta = \{qsq.\ qsq \in D \wedge fst\ qsq \in s\}, \mathcal{I} = set\ I,$
$\mathcal{F} = \{q \in s.\ FP\ q\}|\!)\ (the \circ rm)$
unfolding *NFA-construct-reachable-abstract-impl-invar-def S-def*[*symmetric*] **by** *auto*
from *S-eq* **have** *s-subset*: $s \subseteq S$ **unfolding** *accessible-restrict-def* **by** *simp*

from *rm-q* $\mathcal{A}$*-eq* **have** $r \in (the \circ rm)$ ‘ s **by** *simp*
with *rm-OK rm-q-eq q-in-S s-subset* **have** $q \in s$
unfolding *NFA-construct-reachable-map-OK-def*
apply (*simp add*: *image-iff Bex-def dom-def subset-iff inj-on-def Ball-def*)
apply (*metis the.simps*)
done

from ‹$q \in s$› **have** $insert\ q\ (s \cup set\ I \cup set\ wl \cup \{q'.\ \exists\, a\ q.\ q \in s \wedge (q, a, q') \in D\}) =$
$s \cup\ set\ I \cup set\ wl \cup \{q'.\ \exists\, a\ q.\ q \in s \wedge (q, a, q') \in D\}$ **by** *auto*
with ‹$q \in s$› *rm-OK s-subset* **show** *?thesis*
unfolding *NFA-construct-reachable-abstract-impl-invar-def S-def*[*symmetric*]
apply *simp*
apply (*rule exI*[**where** $x = s$])
apply (*simp add*: $\mathcal{A}$*-eq S-eq accessible-restrict-insert-in*)
done
qed
next
fix *rm* :: $'q \Rightarrow\ 'q2\ option$
fix $\mathcal{A}$ *q wl rm′ D′ N r*
assume *invar*: *NFA-construct-reachable-abstract-impl-invar I A FP D* $((rm, \mathcal{A}), q \# wl)$
and *rm-q-eq*: $rm\ q = Some\ r$
and *nin-Q*: $r \notin \mathcal{Q}\ \mathcal{A}$
and *q-in-S*: $q \in S$
assume *foreach-invar*: *NFA-construct-reachable-abstract-impl-foreach-invar S D rm* $(\Delta\ \mathcal{A})\ q\ \{\}\ (rm', D', N)$
from *rm-q-eq* **have** *r-eq*: $r = the\ (rm\ q)$ **by** *simp*

from *invar* **obtain** *s* **where**
S-eq: $S = accessible\text{-}restrict\ (LTS\text{-}forget\text{-}labels\ D)\ s\ (insert\ q\ (set\ wl))$ **and**
rm-OK: *NFA-construct-reachable-map-OK S Map.empty*
$(insert\ q\ (s \cup set\ I \cup set\ wl \cup \{q'.\ \exists\, a\ q.\ q \in s \wedge (q, a, q') \in D\}))$ *rm* **and**
$\mathcal{A}$*-eq*: $\mathcal{A} = NFA\text{-}rename\text{-}states$
$(\!|\mathcal{Q} = s, \Sigma = A, \Delta = \{qsq.\ qsq \in D \wedge fst\ qsq \in s\}, \mathcal{I} = set\ I,$
$\mathcal{F} = \{q \in s.\ FP\ q\}|\!)\ (the \circ rm)$
unfolding *NFA-construct-reachable-abstract-impl-invar-def S-def*[*symmetric*] **by** *auto*

def $D'' \equiv \{(a, q').\ (q, a, q') \in D\}$
from *foreach-invar* **have**
rm′-OK: *NFA-construct-reachable-map-OK S rm* $(snd$ ‘ $D'')$ *rm′* **and**
set-N-eq: $set\ N = snd$ ‘ D'' **and**
D′-eq: $D' = \Delta\ \mathcal{A} \cup \{(the\ (rm'\ q), a, the\ (rm'\ q'))\ |a\ q'.\ (a, q') \in D''\}$
unfolding *NFA-construct-reachable-abstract-impl-foreach-invar.simps*
by (*simp-all add*: *Let-def D″-def*)

def $DD \equiv insert\ q\ (s \cup set\ I \cup set\ wl \cup \{q'.\ \exists\, a\ q.\ q \in s \wedge (q, a, q') \in D\})$
have *DD-intro*: $(insert\ q$
$(s \cup set\ I \cup (set\ N \cup set\ wl) \cup \{q'.\ \exists\, a\ qa.\ (qa = q \vee qa \in s) \wedge (qa, a, q') \in D\})) = DD \cup snd$ ‘ D''
unfolding *DD-def* **by** (*auto simp add*: *image-iff set-N-eq D″-def*)

from *nin-Q* $\mathcal{A}$*-eq* **have** *q-nin-s*: $q \notin s$ **by** (*auto simp add*: *image-iff r-eq*)

have (*set wl* ∪ {*y*. (*q*, *y*) ∈ *LTS-forget-labels D*}) = *set N* ∪ *set wl*
  unfolding *set-N-eq D′′-def LTS-forget-labels-def* **by** (*auto simp add*: *image-iff*)
hence *prop1*: *S* = *accessible-restrict* (*LTS-forget-labels D*) (*insert q s*) (*set N* ∪ *set wl*)
   by (*simp add*: *S-eq accessible-restrict-insert-nin q-nin-s*)

have *prop2*: *NFA-construct-reachable-map-OK S Map.empty*
   (*insert q* (*s* ∪ *set I* ∪ (*set N* ∪ *set wl*) ∪ {*q′*. ∃ *a qa*. (*qa* = *q* ∨ *qa* ∈ *s*) ∧ (*qa*, *a*, *q′*) ∈ *D*})) *rm′*
  unfolding *DD-intro NFA-construct-reachable-map-OK-def*
proof (*intro conjI allI impI Un-least*)
  from *rm′-OK* **show** *snd* ‘ *D′′* ⊆ *dom rm′ inj-on rm′* (*S* ∩ *dom rm′*)
    unfolding *NFA-construct-reachable-map-OK-def* **by** *simp-all*
next
  from *rm-OK* **have** *DD* ⊆ *dom rm*
    unfolding *NFA-construct-reachable-map-OK-def DD-def*[*symmetric*] **by** *simp*
  moreover from *rm′-OK* **have** *dom rm* ⊆ *dom rm′*
    unfolding *NFA-construct-reachable-map-OK-def dom-def* **by** *auto*
  finally show *DD* ⊆ *dom rm′* **.**
qed *simp*

from *rm′-OK* **have** ⋀*q*. *q* ∈ *dom rm* ⟹ *rm′ q* = *rm q*
  unfolding *NFA-construct-reachable-map-OK-def*
  by (*simp add*: *dom-def*) *metis*
with *rm-OK* **have** *rm′-eq*: ⋀*q′*. *q′* ∈ *DD* ⟹ *rm′ q′* = *rm q′*
  unfolding *NFA-construct-reachable-map-OK-def DD-def*[*symmetric*]
  by (*simp add*: *subset-iff*)

have *prop3*:
  ⦇𝒬 = *insert r* (𝒬 𝒜), Σ = Σ 𝒜, Δ = *D′*, ℐ = ℐ 𝒜,
     ℱ = *if FP q then insert* (*the* (*rm q*)) (ℱ 𝒜) *else* ℱ 𝒜⦈ =
  NFA-rename-states
   ⦇𝒬 = *insert q s*, Σ = *A*, Δ = {*qsq* ∈ *D*. *fst qsq* = *q* ∨ *fst qsq* ∈ *s*}, ℐ = *set I*,
     ℱ = {*qa*. (*qa* = *q* ∨ *qa* ∈ *s*) ∧ *FP qa*}⦈
   (*the* ∘ *rm′*)
proof −
  from *DD-def* **have** *set I* ⊆ *DD* **unfolding** *DD-def* **by** *auto*
  with *rm′-eq* **have** *rm′-eq-I*: (*the* ∘ *rm*) ‘ *set I* = (*the* ∘ *rm′*) ‘ *set I*
    apply (*rule-tac set-eqI*)
    apply (*auto simp add*: *image-iff Bex-def subset-iff*)
  done

  from *DD-def* **have** *insert q s* ⊆ *DD* **unfolding** *DD-def* **by** *auto*
  with *rm′-eq* **have** *rm′-eq-Q*: *insert r* ((*the* ∘ *rm*) ‘ *s*) = *insert* (*the* (*rm′ q*)) ((*the* ∘ *rm′*) ‘ *s*)
    apply (*rule-tac set-eqI*)
    apply (*auto simp add*: *image-iff Bex-def subset-iff r-eq*)
  done

  have *D′-eq′* : *D′* = {(*the* (*rm′ s1*), *a*, *the* (*rm′ s2*)) |*s1 a s2*. (*s1*, *a*, *s2*) ∈ *D* ∧ (*s1* = *q* ∨ *s1* ∈ *s*)}
    (**is** - = *?ls*)
  proof −
    have *?ls* = {(*the* (*rm′ s1*), *a*, *the* (*rm′ s2*)) |*s1 a s2*. (*s1*, *a*, *s2*) ∈ *D* ∧ *s1* ∈ *s*} ∪
              {(*the* (*rm′ q*), *a*, *the* (*rm′ q′*)) |*a q′*. (*a*, *q′*) ∈ *D′′*}
        (**is** - = *?ls′* ∪ -)
      unfolding *D′′-def* **by** *auto*
    moreover
      from *rm′-eq* **have** *?ls′* = Δ 𝒜 **unfolding** 𝒜*-eq DD-def*
        by (*auto simp add*: *NFA-rename-states-def*) *metis+*
    finally show *?thesis* **by** (*simp add*: *D′-eq*)
  qed

  from *rm′-eq* **have** *rm′-eq-F*:
    (*if FP q then insert* (*the* (*rm q*)) (ℱ 𝒜) *else* ℱ 𝒜) =
     (*the* ∘ *rm′*) ‘ {*qa*. (*qa* = *q* ∨ *qa* ∈ *s*) ∧ *FP qa*}

    apply (*rule-tac set-eqI*)
    apply (*simp add*: *image-iff* $\mathcal{A}$*-eq DD-def*)
    apply (*metis*)
  done

  show *?thesis* **by** (*simp add*: *NFA-rename-states-def* $\mathcal{A}$*-eq rm′-eq-F rm′-eq-I rm′-eq-Q D′-eq′*)
qed

from *S-eq* **have** *s-subset*: $s \subseteq S$ **unfolding** *accessible-restrict-def* **by** *simp*
show *NFA-construct-reachable-abstract-impl-invar I A FP D*
    $((rm', (\!| \mathcal{Q} = insert\ r\ (\mathcal{Q}\ \mathcal{A}),\ \Sigma = \Sigma\ \mathcal{A},\ \Delta = D',\ \mathcal{I} = \mathcal{I}\ \mathcal{A},$
      $\mathcal{F} = if\ FP\ q\ then\ insert\ (the\ (rm\ q))\ (\mathcal{F}\ \mathcal{A})\ else\ \mathcal{F}\ \mathcal{A} |\!)),\ N\ @\ wl)$
  unfolding *NFA-construct-reachable-abstract-impl-invar-def S-def* [*symmetric*]
  apply (*simp split del*: *if-splits*)
  apply (*rule exI* [**where** $x = insert\ q\ s$])
  apply (*simp split del*: *if-splits add*: *q-in-S s-subset*)
  apply (*simp add*: *prop3 prop2*)
  apply (*simp add*: *prop1*)
done

from *S-eq* **have** *s-subset*: $s \subseteq S$ **unfolding** *accessible-restrict-def* **by** *simp*
with *fin-S* **have** *fin-s*: *finite s* **by** (*metis finite-subset*)

from $\mathcal{A}$*-eq fin-s nin-Q* **have** *fin-Q*: *finite* $(\mathcal{Q}\ \mathcal{A})$ **and** *card-Q-le′*: $card\ (\mathcal{Q}\ \mathcal{A}) \leq card\ s$
  unfolding *NFA-rename-states-def* **by** (*simp-all add*: *card-image-le image-iff r-eq*)

from *S-eq s-subset accessible-restrict-subset-ws*[*of* (*insert q* (*set wl*)) *LTS-forget-labels D s*]
have $insert\ q\ s \subseteq S$ **unfolding** *accessible-restrict-def* **by** *simp*
hence $card\ (insert\ q\ s) \leq card\ S$ **by** (*rule card-mono*[*OF fin-S*])
hence $card\ s < card\ S$ **by** (*simp add*: *q-nin-s fin-s*)
with *card-Q-le′* **have** *card-Q-le*: $card\ (\mathcal{Q}\ \mathcal{A}) < card\ S$ **by** *simp*

hence $card\ S - card\ (insert\ r\ (\mathcal{Q}\ \mathcal{A})) < card\ S - card\ (\mathcal{Q}\ \mathcal{A})$ **by** (*simp add*: *nin-Q fin-Q*)
thus $card\ S - card\ (insert\ r\ (\mathcal{Q}\ \mathcal{A})) < card\ S - card\ (\mathcal{Q}\ \mathcal{A}) \vee$
    $rm' = rm \wedge (\!| \mathcal{Q} = insert\ r\ (\mathcal{Q}\ \mathcal{A}),\ \Sigma = \Sigma\ \mathcal{A},\ \Delta = D',\ \mathcal{I} = \mathcal{I}\ \mathcal{A},$
      $\mathcal{F} = if\ FP\ q\ then\ insert\ (the\ (rm\ q))\ (\mathcal{F}\ \mathcal{A})\ else\ \mathcal{F}\ \mathcal{A} |\!) = \mathcal{A} \wedge N = []$ **by** *simp*
qed

definition *NFA-construct-reachable-abstract2-impl* **where**
  NFA-construct-reachable-abstract2-impl I A FP D =
   do {
    $(rm, I') \leftarrow SPEC\ (\lambda(rm, I').$
      $NFA\text{-}construct\text{-}reachable\text{-}map\text{-}OK\ (accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I))\ empty\ (set\ I)\ rm \wedge$
      $I' = (the \circ rm)\ `\ (set\ I));$
    $((rm, \mathcal{A}), \text{-}) \leftarrow WORKLISTIT\ (NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}invar\ I\ A\ FP\ D)$
     $(\lambda\text{-}.\ True)$
     $(\lambda(rm, \mathcal{A})\ q.\ do\ \{$
       $ASSERT\ (q \in dom\ rm \wedge q \in accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I) \wedge$
          $NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}weak\text{-}invar\ I\ A\ FP\ D\ (rm, \mathcal{A}));$
       $if\ (the\ (rm\ q) \in \mathcal{Q}\ \mathcal{A})\ then$
        $(RETURN\ ((rm, \mathcal{A}), []))$
       $else$
        $do\ \{$
         $(rm', D', N) \leftarrow NFA\text{-}construct\text{-}reachable\text{-}abstract\text{-}impl\text{-}step$
           $(accessible\ (LTS\text{-}forget\text{-}labels\ D)\ (set\ I))\ D\ rm\ (\Delta\ \mathcal{A})\ q;$
         $RETURN\ ((rm', (\!| \mathcal{Q}{=}insert\ (the\ (rm\ q))\ (\mathcal{Q}\ \mathcal{A}), \Sigma{=}\Sigma\ \mathcal{A}, \Delta = D',$
              $\mathcal{I}{=}\mathcal{I}\ \mathcal{A}, \mathcal{F} = if\ (FP\ q)\ then\ (insert\ (the\ (rm\ q))\ (\mathcal{F}\ \mathcal{A}))\ else\ (\mathcal{F}\ \mathcal{A}) |\!)), N)$
        }
     $\})\ ((rm, (\!| \mathcal{Q}{=}\{\}, \Sigma{=}A, \Delta = \{\}, \mathcal{I}{=}I', \mathcal{F}{=}\{\} |\!)), I);$
    $RETURN\ \mathcal{A}$
   }

lemma *NFA-construct-reachable-abstract2-impl-correct* :
fixes D :: $('q \times 'a \times 'q)$ *set* **and** I
defines $S \equiv$ *accessible* (*LTS-forget-labels* D) (*set* I)
assumes *fin-S*: *finite* S
  and *fin-D*: $\bigwedge q.$ *finite* $\{(a, q'). (q, a, q') \in D\}$
shows *NFA-construct-reachable-abstract2-impl* I A *FP* D $\leq \Downarrow Id$ ((*NFA-construct-reachable-abstract-impl* I A *FP* D)::$('q2, 'a)$ *NFA-rec nres*)
unfolding *NFA-construct-reachable-abstract2-impl-def NFA-construct-reachable-abstract-impl-def S-def* [*symmetric*]
apply *refine-rcg*
apply (*simp*)
apply (*rule single-valued-Id*)
apply (*rule single-valued-Id*)
apply (*simp*)
apply (*simp-all add*: *list-all2-eq*[*symmetric*])

apply (*clarify*, *simp*)
apply (*rename-tac q rm* $\mathcal{A}$ *r*)
apply (*rule NFA-construct-reachable-abstract-impl-step-correct*)
proof −
  fix *rm* :: $'q \Rightarrow 'q2$ *option* **and** *q r*
  assume *rm q* = *Some r* **thus** $q \in$ *dom rm* **by** *blast*
next
  fix *rm* :: $'q \Rightarrow 'q2$ *option* **and**
    $\mathcal{A}$ :: $('q2, 'a)$ *NFA-rec*

  assume *NFA-construct-reachable-abstract-impl-weak-invar* I A *FP* D (*rm*, $\mathcal{A}$)
  thus *inj-on rm* ($S \cap$ *dom rm*)
    unfolding *NFA-construct-reachable-abstract-impl-weak-invar-def*
      NFA-construct-reachable-map-OK-def S-def [*symmetric*] **by** *auto*
next
  fix *q*
  have *rewr*: $\{(a, q'). (q, a, q') \in D\}$ =
    $(\lambda(q, a, q'). (a, q'))$ ‘ $\{qsq.\ qsq \in D \wedge$ *fst* $qsq = q\}$
  by (*simp add*: *set-eq-iff image-iff*)

  show *finite* $\{(a, q'). (q, a, q') \in D\}$ **by** (*rule fin-D*)
qed

2.7 Renaming letters (i.e. elements of the alphabet)

definition *NFA-rename-labels* :: $('q, 'a, 'x)$ *NFA-rec-scheme* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow$
  $('q, 'b)$ *NFA-rec* **where**
NFA-rename-labels $\mathcal{A}$ $f \equiv (\!|\ \mathcal{Q} = \mathcal{Q}\ \mathcal{A}$,
  $\Sigma = f$ ‘ $(\Sigma\ \mathcal{A})$,
  $\Delta = \{\ (p, f\ \sigma, q) \mid p\ \sigma\ q.\ (p,\sigma,q) \in \Delta\ \mathcal{A}\}$,
  $\mathcal{I} = \mathcal{I}\ \mathcal{A}$,
  $\mathcal{F} = \mathcal{F}\ \mathcal{A}\ |\!)$

lemma [*simp*] : $\mathcal{Q}$ (*NFA-rename-labels* $\mathcal{A}$ f) = $\mathcal{Q}$ $\mathcal{A}$ **by** (*simp add*: *NFA-rename-labels-def*)
lemma [*simp*] : Σ (*NFA-rename-labels* $\mathcal{A}$ f) = f ‘ $(\Sigma\ \mathcal{A})$ **by** (*simp add*: *NFA-rename-labels-def*)
lemma [*simp*] : $(p, f\sigma, q) \in \Delta$ (*NFA-rename-labels* $\mathcal{A}$ f) $\longleftrightarrow$
  $(\exists\ \sigma.\ (p, \sigma, q) \in \Delta\ \mathcal{A} \wedge (f\sigma = f\ \sigma))$
  by (*auto simp add*: *NFA-rename-labels-def*)
lemma [*simp*] : $\mathcal{I}$ (*NFA-rename-labels* $\mathcal{A}$ f) = $\mathcal{I}$ $\mathcal{A}$ **by** (*simp add*: *NFA-rename-labels-def*)
lemma [*simp*] : $\mathcal{F}$ (*NFA-rename-labels* $\mathcal{A}$ f) = $\mathcal{F}$ $\mathcal{A}$ **by** (*simp add*: *NFA-rename-labels-def*)

lemma (**in** *NFA*) *NFA-rename-labels---is-well-formed* :
NFA (*NFA-rename-labels* $\mathcal{A}$ f)
using *wf-NFA*
by (*auto simp add*: *NFA-def image-iff Bex-def*)

lemma *lists---NFA-rename-labels* :
$w \in lists\ (\Sigma\ \mathcal{A}) \Longrightarrow (map\ f\ w) \in lists\ (\Sigma\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f))$
by (*auto simp add*: *lists-def*)

lemma *NFA-rename-labels-id* [*simp*] : $NFA\text{-}rename\text{-}labels\ \mathcal{A}\ id = \mathcal{A}$
by (*rule NFA-rec.equality*, *auto simp add*: *NFA-rename-labels-def*)

lemma *LTS-is-reachable---NFA-rename-labels* :
$LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q' \Longrightarrow$
$LTS\text{-}is\text{-}reachable\ (\Delta\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f))\ q\ (map\ f\ w)\ q'$
proof (*induct w arbitrary*: *q*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons* σ *w*) **note** *ind-hyp* = *Cons*(*1*) **note** *reach-*σ*w* = *Cons*(*2*)
from *reach-*σ*w* **obtain** q'' **where**
q''-Δ: $(q, \sigma, q'') \in \Delta\ \mathcal{A}$ **and**
reach-w: $LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q''\ w\ q'$ **by** *auto*
from *ind-hyp reach-w*
have *reach-w'*: $LTS\text{-}is\text{-}reachable\ (\Delta\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f))\ q''\ (map\ f\ w)\ q'$ **by** *blast*
from q''-Δ **have** q''-Δ' : $(q, f\ \sigma, q'') \in (\Delta\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f))$ **by** *auto*
from q''-Δ' *reach-w'* **show** *?case* **by** *auto*
qed

lemma *LTS-is-reachable---NFA-rename-labelsE* :
$LTS\text{-}is\text{-}reachable\ (\Delta\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f))\ q\ w\ q' \Longrightarrow$
$\exists\ w'.\ w = (map\ f\ w') \wedge LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w'\ q'$
proof (*induct w arbitrary*: *q*)
case *Nil* **thus** *?case* **by** *simp*
next
let $?\mathcal{A}' = NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f$
case (*Cons* σ *w*) **note** *ind-hyp* = *Cons*(*1*)
from *Cons*(*2*) **obtain** q'' **where**
q''-Δ: $(q, \sigma, q'') \in \Delta\ ?\mathcal{A}'$ **and**
reach-w: $LTS\text{-}is\text{-}reachable\ (\Delta\ ?\mathcal{A}')\ q''\ w\ q'$ **by** *auto*
from *ind-hyp reach-w* **obtain** w' **where**
reach-w': $w = map\ f\ w' \wedge LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q''\ w'\ q'$ **by** *blast*
from q''-Δ **obtain** σ' **where** $f\ \sigma' = \sigma$ **and** $(q, \sigma', q'') \in \Delta\ \mathcal{A}$ **by** *auto*
with *reach-w'* **have** $\sigma\ \#\ w = map\ f\ (\sigma'\ \#\ w') \wedge LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ (\sigma'\ \#\ w')\ q'$ **by** *auto*
then show $\exists\, w'.\ \sigma\ \#\ w = map\ f\ w' \wedge LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w'\ q'$ **by** *metis*
qed

lemma *NFA-accept---NFA-rename-labels* :
assumes $NFA\text{-}accept\ \mathcal{A}\ w$
shows $NFA\text{-}accept\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f)\ (map\ f\ w)$
using *assms lists---NFA-rename-labels LTS-is-reachable---NFA-rename-labels* **unfolding** *NFA-accept-def*
by (*auto*, *fast*)

lemma (**in** *NFA*) *NFA-accept---NFA-rename-labelsE* :
$NFA\text{-}accept\ (NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f)\ w \Longrightarrow \exists\ w'.\ w = (map\ f\ w') \wedge NFA\text{-}accept\ \mathcal{A}\ w'$
proof −
let $?\mathcal{A} = NFA\text{-}rename\text{-}labels\ \mathcal{A}\ f$
assume $NFA\text{-}accept\ ?\mathcal{A}\ w$
then obtain *q* **and** q' **where**
a: $q \in \mathcal{I}\ ?\mathcal{A} \wedge q' \in \mathcal{F}\ ?\mathcal{A}$ **and** *b*: $LTS\text{-}is\text{-}reachable\ (\Delta\ ?\mathcal{A})\ q\ w\ q'$
unfolding *NFA-accept-def* **by** *auto*
from *a* **have** $q \in \mathcal{I}\ \mathcal{A} \wedge q' \in \mathcal{F}\ \mathcal{A}$ **by** *simp*
moreover
from *b* **obtain** w' **where** *c*: $w = map\ f\ w'$ **and** *d*: $LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w'\ q'$
using *LTS-is-reachable---NFA-rename-labelsE* **by** *metis*
moreover
from *d* **have** $w' \in lists\ (\Sigma\ \mathcal{A})$ **using** *LTS-is-reachable---labels* **by** *metis*
ultimately show *?thesis* **unfolding** *NFA-accept-def* **by** *metis*

qed

lemma (**in** *NFA*) *NFA-accept---NFA-rename-labels-iff* :
  NFA-accept (*NFA-rename-labels* $\mathcal{A}$ *f*) *w* $\longleftrightarrow$ ($\exists$ *w'*. *w* = (*map f w'*) $\wedge$ *NFA-accept* $\mathcal{A}$ *w'*)
by (*metis NFA-accept---NFA-rename-labels NFA-accept---NFA-rename-labelsE*)

lemma (**in** *NFA*) *$\mathcal{L}$-NFA-rename-labels* [*simp*] :
  $\mathcal{L}$ (*NFA-rename-labels* $\mathcal{A}$ *f*) = (*map f*) ‘ $\mathcal{L}$ $\mathcal{A}$
by (*simp add*: *set-eq-iff $\mathcal{L}$-def NFA-accept---NFA-rename-labels-iff image-iff*, *blast*)

lemma *NFA-isomorphic-wf---NFA-rename-labels-cong* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*NFA-rename-labels* $\mathcal{A}1$ *fl*) (*NFA-rename-labels* $\mathcal{A}2$ *fl*)
proof −
  from *equiv*
  obtain *f* **where** *inj-f* : *inj-on f* ($\mathcal{Q}$ $\mathcal{A}1$) **and**
      $\mathcal{A}2$-eq: $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1$ *f* **and**
      wf-$\mathcal{A}1$: *NFA* $\mathcal{A}1$
    unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

  note *wf-rename* = *NFA.NFA-rename-labels---is-well-formed* [*OF wf-$\mathcal{A}1$*, *of fl*]

  have *NFA-rename-labels* $\mathcal{A}2$ *fl* = *NFA-rename-states* (*NFA-rename-labels* $\mathcal{A}1$ *fl*) *f*
    unfolding *$\mathcal{A}2$-eq NFA-rename-labels-def NFA-rename-states-def*
    by *auto metis*
  with *wf-rename inj-f* **show** *?thesis*
    unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def*
    by *auto*
qed

2.8 Normalise states

Operations like building the product of two automata, naturally lead to a different type of automaton states. When combining several operations, one often needs to stay in the same type, however. The following definitions allows to transfer an automaton to an isomorphic one with a different type of states.

definition *NFA-normalise-states* :: ('*q*, '*a*, -) *NFA-rec-scheme* $\Rightarrow$ ('*q2*::{*NFA-states*}, '*a*) *NFA-rec* **where**
  NFA-normalise-states $\mathcal{A}$ = (*NFA-rename-states* $\mathcal{A}$ (*SOME* (*f*::'*q* $\Rightarrow$ '*q2*). *inj-on f* ($\mathcal{Q}$ $\mathcal{A}$)))

lemma *ex-inj-on-finite* :
assumes *inf-univ*: ~(*finite* (*UNIV*::'*b set*))
    and *fin-A*: *finite* (*A*::'*a set*)
shows $\exists$*f*::('*a* $\Rightarrow$ '*b*). *inj-on f A*
using *fin-A*
proof (*induct rule*: *finite-induct*)
  case *empty* **show** *?case* **by** *simp*
next
  case (*insert a A*)
  note *fin-A* = *insert*(*1*)
  note *a-nin* = *insert*(*2*)
  from *insert*(*3*) **obtain** *f* **where** *inj-f*: *inj-on* (*f*::'*a*$\Rightarrow$'*b*) *A* **by** *blast*

  from *fin-A* **have** *finite* (*f* ‘ *A*) **by** (*rule finite-imageI*)
  then obtain *b* **where** *b-nin*: *b* $\notin$ (*f* ‘ *A*) **by** (*metis ex-new-if-finite inf-univ*)

  let *?f'* = λ*x*. *if* (*x* = *a*) *then b else f x*
  from *inj-f a-nin b-nin* **have** *inj-on ?f'* (*insert a A*)
    unfolding *inj-on-def* **by** *auto*
  thus *?case* **by** *blast*
qed

lemma *NFA-isomorphic-wf-normalise-states* :

fixes $\mathcal{A}$:: ('q, 'a, -) *NFA-rec-scheme*
assumes *wf-A*: *NFA* $\mathcal{A}$
shows *NFA-isomorphic-wf* $\mathcal{A}$ ((*NFA-normalise-states* $\mathcal{A}$)::('q2::{*NFA-states*},'a) *NFA-rec*)
unfolding *NFA-normalise-states-def*
  apply (*rule NFA-isomorphic-wf---NFA-rename-states* [*OF* - *wf-A*])
  apply (*rule someI-ex* [**where** $P = \lambda f.$ *inj-on f* ($\mathcal{Q}$ $\mathcal{A}$)])
  apply (*rule ex-inj-on-finite*)
  apply (*simp-all add*: *NFA.finite-*$\mathcal{Q}$[*OF wf-A*] *not-finite-NFA-states-UNIV*)
done

lemma *NFA-isomorphic-wf---NFA-normalise-states* :
NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2$ $\Longrightarrow$ *NFA-isomorphic-wf* $\mathcal{A}1$ (*NFA-normalise-states* $\mathcal{A}2$)
by (*metis NFA-isomorphic-wf-alt-def NFA-isomorphic-wf-normalise-states NFA-isomorphic-wf-trans*)

lemma *NFA-isomorphic-wf---NFA-normalise-states-cong* :
fixes $\mathcal{A}1$::('q1, 'a) *NFA-rec*
  and $\mathcal{A}2$::('q2, 'a) *NFA-rec*
shows *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$ $\Longrightarrow$
 NFA-isomorphic-wf (*NFA-normalise-states* $\mathcal{A}1$) (*NFA-normalise-states* $\mathcal{A}2$)
unfolding *NFA-normalise-states-def*
  apply (*rule NFA.NFA-isomorphic-wf---rename-states-cong* [*of* - $\mathcal{A}1$])
  apply (*rule someI-ex* [**where** $P = \lambda f.$ *inj-on f* ($\mathcal{Q}$ $\mathcal{A}1$)])
  apply (*rule ex-inj-on-finite*)
  defer defer
  apply (*rule someI-ex* [**where** $P = \lambda f.$ *inj-on f* ($\mathcal{Q}$ $\mathcal{A}2$)])
  apply (*rule ex-inj-on-finite*)
  apply (*simp-all add*: *NFA.finite-*$\mathcal{Q}$ *not-finite-NFA-states-UNIV NFA-isomorphic-wf-alt-def*)
done

lemma *NFA-normalise-states-*Σ [*simp*] :
Σ (*NFA-normalise-states* $\mathcal{A}$) = Σ $\mathcal{A}$
unfolding *NFA-normalise-states-def* **by** *simp*

lemma *NFA-normalise-states-*$\mathcal{L}$ [*simp*] :
NFA $\mathcal{A}$ $\Longrightarrow$ $\mathcal{L}$ (*NFA-normalise-states* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
using *NFA-isomorphic-wf-normalise-states*[*of* $\mathcal{A}$]
by (*metis NFA-isomorphic-wf-*$\mathcal{L}$)

lemma *NFA-normalise-states-accept* [*simp*] :
NFA $\mathcal{A}$ $\Longrightarrow$ *NFA-accept* (*NFA-normalise-states* $\mathcal{A}$) *w* = *NFA-accept* $\mathcal{A}$ *w*
using *NFA-normalise-states-*$\mathcal{L}$ [*of* $\mathcal{A}$]
by (*auto simp add*: $\mathcal{L}$*-def*)

lemma *NFA-is-initially-connected---normalise-states* :
assumes *connected*: *NFA-is-initially-connected* $\mathcal{A}$
shows *NFA-is-initially-connected* (*NFA-normalise-states* $\mathcal{A}$)
unfolding *NFA-normalise-states-def*
by (*intro connected NFA-is-initially-connected---NFA-rename-states*)

2.9 Product automata

The following definition is an abstraction of product automata. It only becomes interesting for deterministic automata. For nondeterministic ones, only product automata are used.

definition *bool-comb-NFA* ::
  (*bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool*) $\Rightarrow$ ('q1, 'a, 'x1) *NFA-rec-scheme* $\Rightarrow$
   ('q2, 'a, 'x2) *NFA-rec-scheme* $\Rightarrow$ ('q1 $\times$ 'q2, 'a) *NFA-rec* **where**
bool-comb-NFA bc $\mathcal{A}1$ $\mathcal{A}2$ == (|
   $\mathcal{Q}$=$\mathcal{Q}$ $\mathcal{A}1$ $\times$ $\mathcal{Q}$ $\mathcal{A}2$,
   Σ=Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$,
   Δ = *LTS-product* (Δ $\mathcal{A}1$) (Δ $\mathcal{A}2$),
   $\mathcal{I}$ = $\mathcal{I}$ $\mathcal{A}1$ $\times$ $\mathcal{I}$ $\mathcal{A}2$,

$\mathcal{F} = \{q.\ q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2 \wedge bc\ (fst\ q \in \mathcal{F}\ \mathcal{A}1)\ (snd\ q \in \mathcal{F}\ \mathcal{A}2)\}\|)$

lemma [*simp*] : $\mathcal{I}$ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) = $\mathcal{I}$ $\mathcal{A}1$ $\times$ $\mathcal{I}$ $\mathcal{A}2$ **by** (*simp add*: *bool-comb-NFA-def*)
lemma [*simp*] : $\mathcal{Q}$ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) = $\mathcal{Q}$ $\mathcal{A}1$ $\times$ $\mathcal{Q}$ $\mathcal{A}2$ **by** (*simp add*: *bool-comb-NFA-def*)
lemma [*simp*] : $\mathcal{F}$ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) = $\{q.\ q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2 \wedge bc\ (fst\ q \in \mathcal{F}\ \mathcal{A}1)\ (snd\ q \in \mathcal{F}\ \mathcal{A}2)\}$ **by** (*simp add*: *bool-comb-NFA-def*)
lemma [*simp*] : Σ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) = Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$ **by** (*simp add*: *bool-comb-NFA-def*)
lemma [*simp*] : Δ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) = *LTS-product* (Δ $\mathcal{A}1$) (Δ $\mathcal{A}2$) **by** (*simp add*: *bool-comb-NFA-def*)

definition *product-NFA* **where**
product-NFA $\mathcal{A}1$ $\mathcal{A}2$ = *bool-comb-NFA op*$\wedge$ $\mathcal{A}1$ $\mathcal{A}2$

lemma *accept-product-NFA* :
assumes *wf1*: *NFA* $\mathcal{A}1$ **and** *wf2*: *NFA* $\mathcal{A}2$
shows *NFA-accept* (*product-NFA* $\mathcal{A}1$ $\mathcal{A}2$) *w* = ((*NFA-accept* $\mathcal{A}1$ *w*) $\wedge$ (*NFA-accept* $\mathcal{A}2$ *w*))
using *NFA*.$\mathcal{F}$*-consistent* [*OF wf1*] *NFA*.$\mathcal{F}$*-consistent* [*OF wf2*]
apply (*auto simp add*: *NFA-accept-def LTS-is-reachable-product product-NFA-def subset-iff Bex-def*)
apply *blast*
done

lemma $\mathcal{L}$*-product-NFA* :
$[\![$*NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2]\!]$ $\Longrightarrow$ $\mathcal{L}$ (*product-NFA* $\mathcal{A}1$ $\mathcal{A}2$) = $\mathcal{L}$ $\mathcal{A}1$ $\cap$ $\mathcal{L}$ $\mathcal{A}2$
unfolding $\mathcal{L}$*-def* **using** *accept-product-NFA* **by** *auto*

lemma *bool-comb-NFA---is-well-formed* :
$[\![$ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2]\!]$ $\Longrightarrow$ *NFA* (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$)
unfolding *NFA-def* **by** (*auto simp add*: *bool-comb-NFA-def*)

lemma *product-NFA---is-well-formed* :
$[\![$ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$ $]\!]$ $\Longrightarrow$ *NFA* (*product-NFA* $\mathcal{A}1$ $\mathcal{A}2$)
unfolding *NFA-def* **by** (*auto simp add*: *product-NFA-def*)

definition *efficient-bool-comb-NFA* **where**
efficient-bool-comb-NFA bc $\mathcal{A}1$ $\mathcal{A}2$ =
NFA-remove-unreachable-states (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$)

definition *efficient-product-NFA* **where**
efficient-product-NFA $\mathcal{A}1$ $\mathcal{A}2$ = *NFA-remove-unreachable-states* (*product-NFA* $\mathcal{A}1$ $\mathcal{A}2$)

lemma *efficient-product-NFA-alt-def* :
efficient-product-NFA $\mathcal{A}1$ $\mathcal{A}2$ = *efficient-bool-comb-NFA op*$\wedge$ $\mathcal{A}1$ $\mathcal{A}2$
unfolding *efficient-product-NFA-def efficient-bool-comb-NFA-def product-NFA-def*
..

lemma *efficient-bool-comb-NFA-*Σ [*simp*] :
Σ (*efficient-bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) = Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$
unfolding *efficient-bool-comb-NFA-def*
by *simp*

lemma *accept-efficient-product-NFA* :
$[\![$*NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2]\!]$ $\Longrightarrow$ *NFA-accept* (*efficient-product-NFA* $\mathcal{A}1$ $\mathcal{A}2$) *w* =
(*NFA-accept* $\mathcal{A}1$ *w* $\wedge$ *NFA-accept* $\mathcal{A}2$ *w*)
by (*simp add*: *efficient-product-NFA-def accept-product-NFA*)

lemma $\mathcal{L}$*-efficient-product-NFA* :
$[\![$ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2]\!]$ $\Longrightarrow$ $\mathcal{L}$ (*efficient-product-NFA* $\mathcal{A}1$ $\mathcal{A}2$) = $\mathcal{L}$ $\mathcal{A}1$ $\cap$ $\mathcal{L}$ $\mathcal{A}2$
unfolding $\mathcal{L}$*-def*
by (*auto simp add*: *accept-efficient-product-NFA*)

lemma *efficient-bool-comb-NFA---is-well-formed* :
$[\![$ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$ $]\!]$ $\Longrightarrow$ *NFA* (*efficient-bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$)
unfolding *efficient-bool-comb-NFA-def*

by (*simp add*: *bool-comb-NFA---is-well-formed*)

lemma *efficient-product-NFA---is-well-formed* :
  $[\![$ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$ $]\!] \Longrightarrow$ *NFA* (*efficient-product-NFA* $\mathcal{A}1$ $\mathcal{A}2$)
unfolding *efficient-product-NFA-alt-def*
by (*rule efficient-bool-comb-NFA---is-well-formed*)

lemma *efficient-bool-comb-NFA-compute* :
assumes *wf1*: *NFA* $\mathcal{A}1$ **and** *wf2*: *NFA* $\mathcal{A}2$
shows *efficient-bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$ =
  (*NFA-construct-reachable* ($\mathcal{I}$ $\mathcal{A}1$ $\times$ $\mathcal{I}$ $\mathcal{A}2$) (Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$)
    ($\lambda q.\ q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2 \wedge bc\ ((fst\ q) \in \mathcal{F}\ \mathcal{A}1)\ ((snd\ q) \in \mathcal{F}\ \mathcal{A}2)$)
  (*LTS-product* (Δ $\mathcal{A}1$) (Δ $\mathcal{A}2$)))
proof −
  note *bool-comb-NFA---is-well-formed* [*OF wf1 wf2*]
  hence *wf-bc*: *NFA* (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) **.**

  show *?thesis*
    unfolding *efficient-bool-comb-NFA-def*
    apply (*rule NFA.NFA-remove-unreachable-states-implementation*)
    apply (*simp-all add*: *wf-bc*)
  done
qed

lemma *efficient-product-NFA-compute* :
  $[\![$*NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$$]\!] \Longrightarrow$
   efficient-product-NFA $\mathcal{A}1$ $\mathcal{A}2$ =
  (*NFA-construct-reachable* ($\mathcal{I}$ $\mathcal{A}1$ $\times$ $\mathcal{I}$ $\mathcal{A}2$) (Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$)
   ($\lambda q.\ q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2 \wedge q \in (\mathcal{F}\ \mathcal{A}1 \times \mathcal{F}\ \mathcal{A}2)$)
    (*LTS-product* (Δ $\mathcal{A}1$) (Δ $\mathcal{A}2$)))
apply (*simp add*: *efficient-product-NFA-alt-def*
        efficient-bool-comb-NFA-compute)
apply (*subgoal-tac* ($\lambda q.\ q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2 \wedge q \in \mathcal{F}\ \mathcal{A}1 \times \mathcal{F}\ \mathcal{A}2$) =
        ($\lambda q.\ q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2 \wedge ((fst\ q) \in \mathcal{F}\ \mathcal{A}1) \wedge ((snd\ q) \in \mathcal{F}\ \mathcal{A}2)$)))
apply *simp*
apply (*simp add*: *fun-eq-iff*)
done

lemma *NFA-isomorphic-bool-comb-NFA* :
assumes *equiv-1*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}1'$
  and *equiv-2*: *NFA-isomorphic-wf* $\mathcal{A}2$ $\mathcal{A}2'$
shows *NFA-isomorphic-wf* (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$) (*bool-comb-NFA bc* $\mathcal{A}1'$ $\mathcal{A}2'$)
proof −
 from *equiv-1* **obtain** *f1* **where** *inj-f1* : *inj-on f1* ($\mathcal{Q}$ $\mathcal{A}1$) **and**
        $\mathcal{A}1'$*-eq*: $\mathcal{A}1'$ = *NFA-rename-states* $\mathcal{A}1$ *f1* **and**
        wf-$\mathcal{A}1$: *NFA* $\mathcal{A}1$
  unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

 from *equiv-2* **obtain** *f2* **where** *inj-f2* : *inj-on f2* ($\mathcal{Q}$ $\mathcal{A}2$) **and**
        $\mathcal{A}2'$*-eq*: $\mathcal{A}2'$ = *NFA-rename-states* $\mathcal{A}2$ *f2* **and**
        wf-$\mathcal{A}2$: *NFA* $\mathcal{A}2$
  unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

 def $f \equiv \lambda(q1,\ q2).\ (f1\ q1,\ f2\ q2)$
 with *inj-f1 inj-f2* **have** *inj-f* : *inj-on f* ($\mathcal{Q}$ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$))
  by (*simp add*: *inj-on-def*)

 have *f-image*: $\bigwedge S1\ S2.\ f\ `\ (S1 \times S2) = (f1\ `\ S1) \times (f2\ `\ S2)$
  unfolding *f-def* **by** *auto*

 have $\mathcal{F}$ (*bool-comb-NFA bc* $\mathcal{A}1'$ $\mathcal{A}2'$) = f ‘ ($\mathcal{F}$ (*bool-comb-NFA bc* $\mathcal{A}1$ $\mathcal{A}2$))
 proof −

{ fix *q*
have $q \in \{q \in f1 \text{ ` } \mathcal{Q}\ \mathcal{A}1 \times f2 \text{ ` } \mathcal{Q}\ \mathcal{A}2.\ bc\ (fst\ q \in f1 \text{ ` } \mathcal{F}\ \mathcal{A}1)\ (snd\ q \in f2 \text{ ` } \mathcal{F}\ \mathcal{A}2)\} \longleftrightarrow$
$q \in f \text{ ` } \{q \in \mathcal{Q}\ \mathcal{A}1 \times \mathcal{Q}\ \mathcal{A}2.\ bc\ (fst\ q \in \mathcal{F}\ \mathcal{A}1)\ (snd\ q \in \mathcal{F}\ \mathcal{A}2)\}$
proof −
obtain *q1 q2* **where** *q-eq*: $q = (q1,\ q2)$ **by** (*cases q*, *blast*)

have *fact1* : $\bigwedge y.\ y \in \mathcal{Q}\ \mathcal{A}1 \Longrightarrow (\exists x.\ x \in \mathcal{F}\ \mathcal{A}1 \wedge f1\ y = f1\ x) \longleftrightarrow (y \in \mathcal{F}\ \mathcal{A}1)$
by (*metis inj-f1* [*unfolded inj-on-def*]
NFA.$\mathcal{F}$*-consistent* [*OF wf-*$\mathcal{A}$*1*, *unfolded subset-iff*])

have *fact2* : $\bigwedge y.\ y \in \mathcal{Q}\ \mathcal{A}2 \Longrightarrow (\exists x.\ x \in \mathcal{F}\ \mathcal{A}2 \wedge f2\ y = f2\ x) \longleftrightarrow (y \in \mathcal{F}\ \mathcal{A}2)$
by (*metis inj-f2* [*unfolded inj-on-def*]
NFA.$\mathcal{F}$*-consistent* [*OF wf-*$\mathcal{A}$*2*, *unfolded subset-iff*])

show *?thesis*
apply (*simp del*: *ex-simps add*: *image-iff Bex-def ex-simps*[*symmetric*] *q-eq f-def*)
apply (*insert fact1 fact2*)
apply (*auto*, *metis*+)
done
qed
}
thus *?thesis*
unfolding $\mathcal{A}1'$*-eq* $\mathcal{A}2'$*-eq bool-comb-NFA-def*
by *simp*
qed

hence *prod-eq'*: *bool-comb-NFA bc* $\mathcal{A}1'\ \mathcal{A}2'$ = *NFA-rename-states* (*bool-comb-NFA bc* $\mathcal{A}1\ \mathcal{A}2$) *f*
unfolding $\mathcal{A}1'$*-eq* $\mathcal{A}2'$*-eq bool-comb-NFA-def*
apply (*rule-tac NFA-rec.equality*)
apply (*simp-all add*: *f-image*)
apply (*simp del*: *ex-simps add*: *NFA-rename-states-def set-eq-iff ex-simps*[*symmetric*] *f-def*)
apply (*blast*)
done

from *inj-f prod-eq'*
bool-comb-NFA---is-well-formed [*OF wf-*$\mathcal{A}$*1 wf-*$\mathcal{A}$*2*, *of bc*]
show *?thesis*
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *blast*
qed

lemma *NFA-isomorphic-efficient-bool-comb-NFA* :
assumes *equiv-1*: *NFA-isomorphic-wf* $\mathcal{A}1\ \mathcal{A}1'$
and *equiv-2*: *NFA-isomorphic-wf* $\mathcal{A}2\ \mathcal{A}2'$
shows *NFA-isomorphic-wf* (*efficient-bool-comb-NFA bc* $\mathcal{A}1\ \mathcal{A}2$) (*efficient-bool-comb-NFA bc* $\mathcal{A}1'\ \mathcal{A}2'$)
unfolding *efficient-bool-comb-NFA-def*
apply (*rule NFA-isomorphic-wf---NFA-remove-unreachable-states*)
apply (*rule-tac NFA-isomorphic-bool-comb-NFA*)
apply (*simp-all add*: *equiv-1 equiv-2*)
done

definition *NFA-bool-comb* :: (*bool* ⇒ *bool* ⇒ *bool*) ⇒
('*q*::{*NFA-states*}, '*a*, -) *NFA-rec-scheme* ⇒ ('*q*, '*a*, -) *NFA-rec-scheme* ⇒ ('*q*, '*a*) *NFA-rec* **where**
NFA-bool-comb bc $\mathcal{A}1\ \mathcal{A}2$ = *NFA-normalise-states* (*efficient-bool-comb-NFA bc* $\mathcal{A}1\ \mathcal{A}2$)

lemma *NFA-bool-comb---isomorphic-wf* :
NFA $\mathcal{A}1 \Longrightarrow$ *NFA* $\mathcal{A}2 \Longrightarrow$
NFA-isomorphic-wf (*efficient-bool-comb-NFA bc* $\mathcal{A}1\ \mathcal{A}2$)
(*NFA-bool-comb bc* $\mathcal{A}1\ \mathcal{A}2$)
unfolding *NFA-isomorphic-def NFA-bool-comb-def*
apply (*rule NFA-isomorphic-wf-normalise-states*)
apply (*simp add*: *efficient-bool-comb-NFA---is-well-formed*)
done

lemma *NFA-isomorphic-efficient-NFA-bool-comb* :
assumes *equiv-1*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}1'$
  and *equiv-2*: *NFA-isomorphic-wf* $\mathcal{A}2$ $\mathcal{A}2'$
shows *NFA-isomorphic-wf* (*NFA-bool-comb bc* $\mathcal{A}1$ $\mathcal{A}2$) (*NFA-bool-comb bc* $\mathcal{A}1'$ $\mathcal{A}2'$)
unfolding *NFA-bool-comb-def efficient-bool-comb-NFA-def*
apply (*rule NFA-isomorphic-wf---NFA-normalise-states-cong*)
apply (*rule NFA-isomorphic-wf---NFA-remove-unreachable-states*)
apply (*rule-tac NFA-isomorphic-bool-comb-NFA*)
apply (*simp-all add*: *equiv-1 equiv-2*)
done

lemma *NFA-bool-comb-NFA---*Σ [*simp*] :
 Σ (*NFA-bool-comb bc* $\mathcal{A}1$ $\mathcal{A}2$) = Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$
unfolding *NFA-bool-comb-def*
by *simp*

2.10 Reversal

definition *NFA-reverse* :: ('q, 'a, 'x) *NFA-rec-scheme* $\Rightarrow$ ('q, 'a) *NFA-rec* **where**
 NFA-reverse $\mathcal{A}$ = (| $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$, Σ = Σ $\mathcal{A}$, Δ = { (q,σ,p). $(p,\sigma,q) \in \Delta$ $\mathcal{A}$ }, $\mathcal{I}$ = $\mathcal{F}$ $\mathcal{A}$, $\mathcal{F}$ = $\mathcal{I}$ $\mathcal{A}$ |)

lemma [*simp*] : $\mathcal{Q}$ (*NFA-reverse* $\mathcal{A}$) = $\mathcal{Q}$ $\mathcal{A}$ **by** (*simp add*: *NFA-reverse-def*)
lemma [*simp*] : Σ (*NFA-reverse* $\mathcal{A}$) = Σ $\mathcal{A}$ **by** (*simp add*: *NFA-reverse-def*)
lemma [*simp*] : $\mathcal{I}$ (*NFA-reverse* $\mathcal{A}$) = $\mathcal{F}$ $\mathcal{A}$ **by** (*simp add*: *NFA-reverse-def*)
lemma [*simp*] : $\mathcal{F}$ (*NFA-reverse* $\mathcal{A}$) = $\mathcal{I}$ $\mathcal{A}$ **by** (*simp add*: *NFA-reverse-def*)

lemma [*simp*] : $p\sigma q \in \Delta$ (*NFA-reverse* $\mathcal{A}$) $\longleftrightarrow$ ((*snd* (*snd* $p\sigma q$)), (*fst* (*snd* $p\sigma q$)), (*fst* $p\sigma q$)) $\in \Delta$ $\mathcal{A}$
by (*cases* $p\sigma q$, *simp add*: *NFA-reverse-def*)

lemma *NFA-reverse---is-well-formed* :
 NFA $\mathcal{A}$ $\Longrightarrow$ *NFA* (*NFA-reverse* $\mathcal{A}$)
by (*simp add*: *NFA-def NFA-reverse-def*)

lemma *NFA-reverse-NFA-reverse* :
 NFA-reverse (*NFA-reverse* $\mathcal{A}$) = $\mathcal{A}$
proof −
 have Δ $\mathcal{A}$ = $\{(q, \sigma, p) \mid q\ \sigma\ p.\ (q, \sigma, p) \in \Delta\ \mathcal{A}\}$ **by** *auto*
 thus *?thesis* **by** (*simp add*: *NFA-reverse-def*)
qed

lemma *NFA-reverse---LTS-is-reachable* :
 LTS-is-reachable (Δ (*NFA-reverse* $\mathcal{A}$)) *p w q* $\longleftrightarrow$ *LTS-is-reachable* (Δ $\mathcal{A}$) *q* (*rev w*) *p*
by (*induct w arbitrary*: *p q*, *auto*)

lemma *NFA-reverse---accept* :
 NFA-accept (*NFA-reverse* $\mathcal{A}$) *w* $\longleftrightarrow$ *NFA-accept* $\mathcal{A}$ (*rev w*)
by (*simp add*: *NFA-accept-def NFA-reverse---LTS-is-reachable*, *auto*)

lemma *NFA-reverse---*$\mathcal{L}$:
 $\mathcal{L}$ (*NFA-reverse* $\mathcal{A}$) = {*rev w* | *w*. *w* $\in \mathcal{L}$ $\mathcal{A}$}
unfolding $\mathcal{L}$*-def* **using** *NFA-reverse---accept* **by** *force*

lemma *NFA-reverse---*$\mathcal{L}$*-in-state* :
 $\mathcal{L}$*-in-state* (*NFA-reverse* $\mathcal{A}$) *q* = {*rev w* | *w*. *w* $\in \mathcal{L}$*-left* $\mathcal{A}$ *q*}
by (*auto simp add*: $\mathcal{L}$*-in-state-def* $\mathcal{L}$*-left-def NFA-reverse---LTS-is-reachable lists-eq-set*,
  metis rev-rev-ident set-rev)

lemma *NFA-reverse---*$\mathcal{L}$*-left* :
 $\mathcal{L}$*-left* (*NFA-reverse* $\mathcal{A}$) *q* = {*rev w* | *w*. *w* $\in \mathcal{L}$*-in-state* $\mathcal{A}$ *q*}
by (*auto simp add*: $\mathcal{L}$*-in-state-def* $\mathcal{L}$*-left-def NFA-reverse---LTS-is-reachable lists-eq-set*,
  metis rev-rev-ident set-rev)

lemma *unreachable-states-NFA-reverse-def* :
NFA-unreachable-states $\mathcal{A}$ = {*q*. $\mathcal{L}$*-in-state* (*NFA-reverse* $\mathcal{A}$) *q* = {}}
by (*simp add*: *NFA-reverse---*$\mathcal{L}$*-in-state NFA-unreachable-states-alt-def*)

lemma *NFA-isomorphic-wf---NFA-reverse-cong* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}$*1* $\mathcal{A}$*2*
shows *NFA-isomorphic-wf* (*NFA-reverse* $\mathcal{A}$*1*) (*NFA-reverse* $\mathcal{A}$*2*)
proof −
from *equiv* **obtain** *f* **where** *inj-f* : *inj-on f* ($\mathcal{Q}$ $\mathcal{A}$*1*) **and**
$\mathcal{A}$*2-eq*: $\mathcal{A}$*2* = *NFA-rename-states* $\mathcal{A}$*1 f* **and**
wf-$\mathcal{A}$*1*: *NFA* $\mathcal{A}$*1*
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

with *inj-f* **have** *inj-f'* : *inj-on f* ($\mathcal{Q}$ (*NFA-reverse* $\mathcal{A}$*1*))
by *simp*

have $\mathcal{A}$*2-eq'*: *NFA-reverse* $\mathcal{A}$*2* = *NFA-rename-states* (*NFA-reverse* $\mathcal{A}$*1*) *f*
unfolding *NFA-reverse-def* $\mathcal{A}$*2-eq NFA-rename-states-def*
apply *simp*
apply *auto*
done

from *inj-f'* $\mathcal{A}$*2-eq' NFA-reverse---is-well-formed* [*OF wf-*$\mathcal{A}$*1*] **show** *?thesis*
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *blast*
qed

2.11 Right quotient

definition *NFA-right-quotient* :: ('*q*, '*a*, '*x*) *NFA-rec-scheme* ⇒ ('*a list*) *set* ⇒ ('*q*, '*a*) *NFA-rec* **where**
NFA-right-quotient $\mathcal{A}$ *L* =
(| $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$,
Σ = Σ $\mathcal{A}$,
Δ = Δ $\mathcal{A}$,
$\mathcal{I}$ = $\mathcal{I}$ $\mathcal{A}$,
$\mathcal{F}$ = {*q*. *q* ∈ $\mathcal{Q}$ $\mathcal{A}$ ∧ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q* ∩ *L* ≠ {}} |)

lemma [*simp*] : $\mathcal{Q}$ (*NFA-right-quotient* $\mathcal{A}$ *L*) = $\mathcal{Q}$ $\mathcal{A}$ **by** (*simp add*: *NFA-right-quotient-def*)
lemma [*simp*] : Σ (*NFA-right-quotient* $\mathcal{A}$ *L*) = Σ $\mathcal{A}$ **by** (*simp add*: *NFA-right-quotient-def*)
lemma [*simp*] : $\mathcal{I}$ (*NFA-right-quotient* $\mathcal{A}$ *L*) = $\mathcal{I}$ $\mathcal{A}$ **by** (*simp add*: *NFA-right-quotient-def*)
lemma [*simp*] : Δ (*NFA-right-quotient* $\mathcal{A}$ *L*) = Δ $\mathcal{A}$ **by** (*simp add*: *NFA-right-quotient-def*)
lemma [*simp*] : $\mathcal{F}$ (*NFA-right-quotient* $\mathcal{A}$ *L*) = {*q*. *q* ∈ $\mathcal{Q}$ $\mathcal{A}$ ∧ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q* ∩ *L* ≠ {}} **by** (*simp add*: *NFA-right-quotient-def*)

lemma *NFA-right-quotient---is-well-formed* :
NFA $\mathcal{A}$ ⟹ *NFA* (*NFA-right-quotient* $\mathcal{A}$ *L*)
unfolding *NFA-def NFA-right-quotient-def*
by *auto*

lemma (**in** *NFA*) *NFA-right-quotient---accepts* :
NFA-accept (*NFA-right-quotient* $\mathcal{A}$ *L*) *w* ⟷
(∃ *w2* ∈ *L*. *NFA-accept* $\mathcal{A}$ (*w* @ *w2*))
unfolding *NFA-def NFA-right-quotient-def*
using *NFA-*Δ*-cons---LTS-is-reachable* $\mathcal{I}$*-consistent*
by (*simp add*: *NFA-accept-def Bex-def LTS-is-reachable-concat subset-iff*
ex-simps[*symmetric*] $\mathcal{L}$*-in-state-def ex-in-conv* [*symmetric*]
del: *ex-simps*)
metis

lemma *NFA-right-quotient---alt-def* :
NFA-right-quotient $\mathcal{A}$ *L* =
(| $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$,

$\Sigma = \Sigma\ \mathcal{A}$,
$\Delta = \Delta\ \mathcal{A}$,
$\mathcal{I} = \mathcal{I}\ \mathcal{A}$,
$\mathcal{F} = \{q'\ .\ \exists\, q\ w.\ q' \in \mathcal{Q}\ \mathcal{A} \wedge q \in \mathcal{F}\ \mathcal{A} \wedge w \in \mathit{rev}\ `\ L \cap \mathit{lists}\ (\Sigma\ \mathcal{A}) \wedge$
$\quad \mathit{LTS\text{-}is\text{-}reachable}\ \{(q, \sigma, p)\ .\ (p, \sigma, q) \in \Delta\ \mathcal{A}\}\ q\ w\ q'\}\ \|)$

proof −
have $\bigwedge q.\ (\mathcal{L}\text{-}\mathit{in\text{-}state}\ \mathcal{A}\ q \cap L) = \mathit{rev}\ `\ (\mathcal{L}\text{-}\mathit{left}\ (\mathit{NFA\text{-}reverse}\ \mathcal{A})\ q \cap \mathit{rev}\ `\ L)$
apply (*simp add*: *NFA-reverse---ℒ-left image-Int*)
apply (*simp add*: *set-eq-iff image-iff ex-simps*[*symmetric*] *del*: *ex-simps*)
done
hence $\bigwedge q.\ \mathcal{L}\text{-}\mathit{in\text{-}state}\ \mathcal{A}\ q \cap L \neq \{\} \longleftrightarrow (\mathcal{L}\text{-}\mathit{left}\ (\mathit{NFA\text{-}reverse}\ \mathcal{A})\ q \cap \mathit{rev}\ `\ L) \neq \{\}$
by *simp*
thus *?thesis*
apply (*simp add*: *NFA-right-quotient-def ℒ-left-def set-eq-iff*)
apply (*simp del*: *ex-simps add*: *ex-simps*[*symmetric*] *image-iff Bex-def NFA-reverse-def*)
apply *blast*
done
qed

lemma *NFA-isomorphic-right-quotient* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1\ \mathcal{A}2$
shows *NFA-isomorphic-wf* (*NFA-right-quotient* $\mathcal{A}1$ *L*) (*NFA-right-quotient* $\mathcal{A}2$ *L*)
proof −
from *equiv* **obtain** *f* **where** *inj-f* : *inj-on f* $(\mathcal{Q}\ \mathcal{A}1)$ **and**
$\mathcal{A}2$-*eq*: $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1$ *f* **and**
wf-$\mathcal{A}1$: *NFA* $\mathcal{A}1$
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

with *inj-f* **have** *inj-f′* : *inj-on f* ($\mathcal{Q}$ (*NFA-right-quotient* $\mathcal{A}1$ *L*))
by *simp*
note *equiv-f* = *NFA-is-equivalence-rename-funI---inj-used* [*OF inj-f*]

from *NFA.ℒ-in-state-rename-iff* [*OF wf-*$\mathcal{A}1$ *equiv-f*]
have *F-eq*: $\{q \in f\ `\ \mathcal{Q}\ \mathcal{A}1.\ \mathcal{L}\text{-}\mathit{in\text{-}state}\ (\mathit{NFA\text{-}rename\text{-}states}\ \mathcal{A}1\ f)\ q \cap L \neq \{\}\} =$
$f\ `\ \{q \in \mathcal{Q}\ \mathcal{A}1.\ \mathcal{L}\text{-}\mathit{in\text{-}state}\ \mathcal{A}1\ q \cap L \neq \{\}\}$
by *auto*

have $\mathcal{A}2$-*eq′*: *NFA-right-quotient* $\mathcal{A}2$ *L* = *NFA-rename-states* (*NFA-right-quotient* $\mathcal{A}1$ *L*) *f*
unfolding *NFA-right-quotient-def* $\mathcal{A}2$-*eq*
apply (*rule NFA-rec.equality*)
apply (*simp-all*)
apply (*simp add*: *NFA-rename-states-def*)
apply (*simp add*: *F-eq*)
done

from *inj-f′* $\mathcal{A}2$-*eq′* *NFA-right-quotient---is-well-formed* [*OF wf-*$\mathcal{A}1$, *of L*]
show *?thesis*
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *blast*
qed

lemma *NFA-right-quotient---restrict-L*:
NFA-right-quotient $\mathcal{A}$ ($L \cap$ *lists* ($\Sigma\ \mathcal{A}$)) = *NFA-right-quotient* $\mathcal{A}$ *L*
by (*simp add*: *NFA-right-quotient-def ℒ-in-state-def Bex-def*
ex-in-conv [*symmetric*], *blast*)

Code will just be generated for a simpler version, which is very easy to implement:

abbreviation *NFA-right-quotient-lists* $\mathcal{A}$ *Q* ≡ *NFA-right-quotient* $\mathcal{A}$ (*lists Q*)

lemma *NFA-right-quotient-lists-inter* :
shows *NFA-right-quotient-lists* $\mathcal{A}$ *Q* = *NFA-right-quotient-lists* $\mathcal{A}$ ($Q \cap \Sigma\ \mathcal{A}$)
unfolding *NFA-right-quotient---alt-def*
by *auto*

lemma (**in** *NFA*) *NFA-right-quotient-lists-alt-def* :
shows *NFA-right-quotient-lists* $\mathcal{A}$ Q =
  (| $\mathcal{Q} = \mathcal{Q}\ \mathcal{A}$,
   $\Sigma = \Sigma\ \mathcal{A}$,
   $\Delta = \Delta\ \mathcal{A}$,
   $\mathcal{I} = \mathcal{I}\ \mathcal{A}$,
   $\mathcal{F} =$ *accessible* $\{(q,p) . \exists \sigma. (p, \sigma, q) \in \Delta\ \mathcal{A} \wedge \sigma \in Q\}$ $(\mathcal{F}\ \mathcal{A})$ |)
unfolding *NFA-right-quotient---alt-def*
proof (*clarify*, *intro conjI refl*)
 show $\{q'. \exists q\ w.\ q' \in \mathcal{Q}\ \mathcal{A} \wedge$
    $q \in \mathcal{F}\ \mathcal{A} \wedge$
    $w \in$ *rev* ‘ *lists* $Q \cap$ *lists* $(\Sigma\ \mathcal{A}) \wedge$ *LTS-is-reachable* $\{(q, \sigma, p). (p, \sigma, q) \in \Delta\ \mathcal{A}\}\ q\ w\ q'\}$ =
  accessible $\{(q, p). \exists \sigma. (p, \sigma, q) \in \Delta\ \mathcal{A} \wedge \sigma \in Q\}$ $(\mathcal{F}\ \mathcal{A})$
  (**is** *?ls* = *?rs*)
 proof −
  have $\{(q, p). \exists \sigma. (p, \sigma, q) \in \Delta\ \mathcal{A} \wedge \sigma \in Q\}$ =
    LTS-forget-labels-pred $(\lambda\sigma.\ \sigma \in Q)$ $\{(q, \sigma, p). (p, \sigma, q) \in \Delta\ \mathcal{A}\}$
   unfolding *LTS-forget-labels-pred-def* **by** *simp*
  hence *rs-eq*: *?rs* = $\{q'. \exists q{\in}\mathcal{F}\ \mathcal{A}. \exists w.$ *LTS-is-reachable* $(\Delta$ (*NFA-reverse* $\mathcal{A}$)) $q\ w\ q' \wedge w \in$ *lists* $Q\}$
   by (*simp add*: *accessible-def rtrancl-LTS-forget-labels-pred NFA-reverse-def*)

  interpret *revA* : *NFA NFA-reverse* $\mathcal{A}$ **by** (*rule NFA-reverse---is-well-formed* [*OF NFA-axioms*])

  show *?thesis*
  proof (*intro iffI set-eqI*)
   fix q'
   assume $q' \in$ *?ls*
   thus $q' \in$ *?rs* **unfolding** *rs-eq NFA-reverse-def* **by** *auto*
  next
   fix q'
   assume $q' \in$ *?rs*
   with *rs-eq* **obtain** q w **where**
    q-in-F: $q{\in}\mathcal{F}\ \mathcal{A}$ **and** *reach*: *LTS-is-reachable* (Δ (*NFA-reverse* $\mathcal{A}$)) $q\ w\ q'$ **and** *w-in-Q*: $w \in$ *lists* Q **by** *blast*

   from *q-in-F* $\mathcal{F}$*-consistent* **have** *q-in-Q*: $q \in \mathcal{Q}\ \mathcal{A}$ **by** *auto*
   with *reach revA.NFA-*Δ*-cons---LTS-is-reachable* **have** q'*-in-Q*: $q' \in \mathcal{Q}\ \mathcal{A}$ **and** *w-in-A*: $w \in$ *lists* $(\Sigma\ \mathcal{A})$ **by** *simp-all*

   show $q' \in$ *?ls*
    apply (*simp add*: q'*-in-Q del*: *ex-simps add*: *ex-simps*[*symmetric*])
    apply (*rule exI* [**where** $x{=}q$])
    apply (*rule exI* [**where** $x{=}w$])
    apply (*insert reach*)
    apply (*simp add*: *q-in-F w-in-Q w-in-A NFA-reverse-def*)
   done
  qed
 qed
qed

end

3 Deterministic Finite Automata

theory *DFA*
imports *Main LTS NFA*
begin

3.1 Basic Definitions

definition *NFA-is-deterministic* **where**

NFA-is-deterministic $\mathcal{A} \equiv$
  (*LTS-is-deterministic* ($\mathcal{Q}$ $\mathcal{A}$) (Σ $\mathcal{A}$) (Δ $\mathcal{A}$) $\wedge$ ($\exists$ *q0*. $\mathcal{I}$ $\mathcal{A}$ = {*q0*}))

lemma *dummy-NFA---NFA-is-deterministic* :
  NFA-is-deterministic (*dummy-NFA q a*)
by (*simp add*: *NFA-is-deterministic-def LTS-is-deterministic-def*
  LTS-is-weak-deterministic-def dummy-NFA-def)

locale *detNFA* =
  fixes $\mathcal{A}$::('*q*,'*a*,'*DFA-more*) *NFA-rec-scheme*
  assumes *deterministic-NFA*: *NFA-is-deterministic* $\mathcal{A}$

locale *DFA* = *detNFA* $\mathcal{A}$ + *NFA* $\mathcal{A}$
  for $\mathcal{A}$::('*q*,'*a*, '*DFA-more*) *NFA-rec-scheme*

lemma (**in** *DFA*) *det-NFA* : *detNFA* $\mathcal{A}$ **by** *unfold-locales*
lemma (**in** *DFA*) *wf-DFA* : *DFA* $\mathcal{A}$ **by** *unfold-locales*

lemma *DFA-alt-def* :
  DFA $\mathcal{A}$ $\equiv$ *NFA-is-deterministic* $\mathcal{A}$ $\wedge$ *NFA* $\mathcal{A}$
by (*simp add*: *DFA-def detNFA-def*)

lemma *DFA-implies-NFA*[*simp*] :
  DFA $\mathcal{A}$ $\Longrightarrow$ *NFA* $\mathcal{A}$ **unfolding** *DFA-alt-def* **by** *simp*

lemma *dummy-NFA---is-DFA* :
  DFA (*dummy-NFA q a*)
by (*simp add*: *DFA-alt-def dummy-NFA---is-NFA dummy-NFA---NFA-is-deterministic*)

lemma (**in** *detNFA*) *deterministic* : *LTS-is-deterministic* ($\mathcal{Q}$ $\mathcal{A}$) (Σ $\mathcal{A}$) (Δ $\mathcal{A}$) $\wedge$ ($\exists$ *q0*. $\mathcal{I}$ $\mathcal{A}$ = {*q0*})
using *deterministic-NFA* **by** (*simp add*: *NFA-is-deterministic-def*)

lemma (**in** *detNFA*) *weak-deterministic* : *LTS-is-weak-deterministic* (Δ $\mathcal{A}$)
using *deterministic* **by** (*simp add*: *LTS-is-deterministic-def*)

3.2 The unique initial state

lemma (**in** *detNFA*) *unique-initial* : $\exists$! *x*. *x* $\in$ $\mathcal{I}$ $\mathcal{A}$
using *deterministic* **by** (*auto*)

definition i **where** i $\mathcal{A}$ $\equiv$ *SOME q*. *q* $\in$ $\mathcal{I}$ $\mathcal{A}$

lemma (**in** *detNFA*) $\mathcal{I}$-*is-set*-i [*simp*] : $\mathcal{I}$ $\mathcal{A}$ = {i $\mathcal{A}$}
proof −
  obtain *x* **where** *x*: $\mathcal{I}$ $\mathcal{A}$ = {*x*} **using** *unique-initial* **by** *blast*
  then show $\mathcal{I}$ $\mathcal{A}$ = {i $\mathcal{A}$} **unfolding** i-*def* **by** *simp*
qed

lemma (**in** *DFA*) i-*is-state* : i $\mathcal{A}$ $\in$ $\mathcal{Q}$ $\mathcal{A}$ **using** $\mathcal{I}$-*consistent* **by** *simp*
lemma (**in** *DFA*) $\mathcal{Q}$-*not-Emp*: $\mathcal{Q}$ $\mathcal{A}$ $\neq$ {} **using** i-*is-state* **by** *auto*

lemma (**in** *detNFA*) $\mathcal{L}$-*in-state*-i : $\mathcal{L}$-*in-state* $\mathcal{A}$ (i $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$ **using** $\mathcal{L}$-*alt-def* **by** *force*

3.3 The unique transition function

definition δ **where** δ $\mathcal{A}$ $\equiv$ *LTS-to-DLTS* (Δ $\mathcal{A}$)

lemma (**in** *DFA*) δ-*to*-Δ [*simp*] : *DLTS-to-LTS* (δ $\mathcal{A}$) = Δ $\mathcal{A}$
  by (*simp add*: δ-*def LTS-to-DLTS-inv weak-deterministic*)

lemma (**in** *detNFA*) δ-*in*-Δ-*iff* :
  (q, σ, q') $\in$ Δ $\mathcal{A}$ $\longleftrightarrow$ (δ $\mathcal{A}$) (q, σ) = *Some* q'

unfolding *δ-def*
by (*simp add*: *LTS-to-DLTS-is-some-det weak-deterministic*)

lemma (**in** *DFA*) *δ-wf* :
  assumes $\delta\ \mathcal{A}\ (q,\sigma) = Some\ q'$
  shows $q \in \mathcal{Q}\ \mathcal{A} \wedge (\sigma \in \Sigma\ \mathcal{A}) \wedge (q' \in \mathcal{Q}\ \mathcal{A})$
using *assms*
by (*simp add*: *δ-in-Δ-iff* [*symmetric*] *Δ-consistent*)

3.4 Lemmas about deterministic automata

lemma (**in** *DFA*) *DFA-LTS-is-reachable-DLTS-reach-simp* :
  LTS-is-reachable $(\Delta\ \mathcal{A})\ q\ w\ q' \longleftrightarrow$ (*DLTS-reach* $(\delta\ \mathcal{A})\ q\ w = Some\ q'$)
by (*simp add*: *LTS-is-reachable-weak-determistic weak-deterministic δ-def*)

lemma (**in** *DFA*) *DFA-δ-is-none-iff* :
  $(\delta\ \mathcal{A}\ (q, \sigma) = None) \longleftrightarrow \neg (q \in \mathcal{Q}\ \mathcal{A} \wedge \sigma \in \Sigma\ \mathcal{A})$
apply (*insert deterministic*)
apply (*simp add*: *δ-def LTS-to-DLTS-def LTS-is-deterministic-def*)
apply (*auto simp add*: *Δ-consistent*)
done

lemma (**in** *DFA*) *DFA-δ-is-some* :
  $[\![q \in \mathcal{Q}\ \mathcal{A};\ \sigma \in \Sigma\ \mathcal{A}]\!] \Longrightarrow \neg (\delta\ \mathcal{A}\ (q, \sigma) = None)$
by (*simp add*: *DFA-δ-is-none-iff*)

lemma (**in** *DFA*) *DFA-reach-is-none-iff* :
  DLTS-reach $(\delta\ \mathcal{A})\ q\ w = None \longleftrightarrow \neg((w \neq []\ {-}{-}{>}\ q \in \mathcal{Q}\ \mathcal{A}) \wedge w \in lists\ (\Sigma\ \mathcal{A}))$
proof (*induct w arbitrary*: *q*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons σ w*)
  note *ind-hyp* = *this*

  show *?case*
  proof (*cases* $\delta\ \mathcal{A}\ (q, \sigma)$)
    case *None* **thus** *?thesis* **by** (*simp*, *auto simp add*: *DFA-δ-is-none-iff*)
  next
    case (*Some q′*) **note** *δ-eq* = *this*
    hence $q \in \mathcal{Q}\ \mathcal{A} \wedge \sigma \in \Sigma\ \mathcal{A} \wedge q' \in \mathcal{Q}\ \mathcal{A}$ **by** (*simp add*: *δ-wf*)
    with *δ-eq* **show** *?thesis* **by** (*simp add*: *ind-hyp*)
  qed
qed

lemma (**in** *DFA*) *DFA-DLTS-reach-is-some* :
  $[\![q \in \mathcal{Q}\ \mathcal{A};\ w \in lists\ (\Sigma\ \mathcal{A})]\!] \Longrightarrow \neg$ (*DLTS-reach* $(\delta\ \mathcal{A})\ q\ w = None$)
by (*simp add*: *DFA-reach-is-none-iff*)

lemma (**in** *DFA*) *DFA-DLTS-reach-wf* : $[\![\ q \in \mathcal{Q}\ \mathcal{A};$ *DLTS-reach* $(\delta\ \mathcal{A})\ q\ w = Some\ q']\!] \Longrightarrow\ q' \in \mathcal{Q}\ \mathcal{A}\ \wedge$
$w \in lists\ (\Sigma\ \mathcal{A})$
by (*simp add*: *DFA-LTS-is-reachable-DLTS-reach-simp* [*symmetric*] *NFA-Δ-cons---LTS-is-reachable*)

lemma (**in** *DFA*) *DFA-left-languages---pairwise-disjoint* :
assumes *p-in-Q* : $p \in \mathcal{Q}\ \mathcal{A}$
    and *q-in-Q* : $q \in \mathcal{Q}\ \mathcal{A}$
    and *p-neq-Q*: $p \neq q$
shows $\mathcal{L}$-*left* $\mathcal{A}\ p \cap \mathcal{L}$-*left* $\mathcal{A}\ q = \{\}$
proof (*rule ccontr*)
  assume $\mathcal{L}$-*left* $\mathcal{A}\ p \cap \mathcal{L}$-*left* $\mathcal{A}\ q \neq \{\}$
  then obtain *w* **where** $w \in \mathcal{L}$-*left* $\mathcal{A}\ p$ **and** $w \in \mathcal{L}$-*left* $\mathcal{A}\ q$ **by** *auto*
  then have *LTS-is-reachable* $(\Delta\ \mathcal{A})$ (i $\mathcal{A}$) *w p* **and** *LTS-is-reachable* $(\Delta\ \mathcal{A})$ (i $\mathcal{A}$) *w q*
    unfolding $\mathcal{L}$-*left-def* **by** *auto*

with ⟨$p \neq q$⟩ **show** *False* **by** (*simp add*: *DFA-LTS-is-reachable-DLTS-reach-simp*)
qed

lemma (**in** *DFA*) *DFA-accept-alt-def* :
NFA-accept $\mathcal{A}$ *w* =
(*case* (*DLTS-reach* (δ $\mathcal{A}$) (i $\mathcal{A}$) *w*) *of None* $\Rightarrow$ *False* | *Some* q' $\Rightarrow$ $q' \in \mathcal{F}$ $\mathcal{A}$)
by (*simp add*: *NFA-accept-wf-def wf-NFA Bex-def DFA-LTS-is-reachable-DLTS-reach-simp*
split: *option.split*)

lemma $\mathcal{L}$*-in-state---DFA---eq-reachable-step* :
assumes *DFA-*$\mathcal{A}$: *DFA* $\mathcal{A}$
and *DFA-*$\mathcal{A}'$: *DFA* $\mathcal{A}'$
and *in-D1*: ($q1$, σ, $q1'$) $\in \Delta$ $\mathcal{A}$
and *in-D2*: ($q2$, σ, $q2'$) $\in \Delta$ $\mathcal{A}'$
and *lang-eq* : $\mathcal{L}$*-in-state* $\mathcal{A}$ $q1$ = $\mathcal{L}$*-in-state* $\mathcal{A}'$ $q2$
shows $\mathcal{L}$*-in-state* $\mathcal{A}$ $q1'$ = $\mathcal{L}$*-in-state* $\mathcal{A}'$ $q2'$
proof −
interpret *DFA1* : *DFA* $\mathcal{A}$ **by** (*fact DFA-*$\mathcal{A}$)
interpret *DFA2* : *DFA* $\mathcal{A}'$ **by** (*fact DFA-*$\mathcal{A}'$)

from *DFA1.weak-deterministic in-D1* **have** *q1'-unique*:
$\bigwedge q1''$. ($q1$, σ, $q1''$) $\in \Delta$ $\mathcal{A}$ = ($q1''$ = $q1'$)
by (*auto simp add*: *LTS-is-weak-deterministic-def*)

from *DFA2.weak-deterministic in-D2* **have** *q2'-unique*:
$\bigwedge q2''$. ($q2$, σ, $q2''$) $\in \Delta$ $\mathcal{A}'$ = ($q2''$ = $q2'$)
by (*auto simp add*: *LTS-is-weak-deterministic-def*)

from *DFA1.*Δ*-consistent DFA2.*Δ*-consistent in-D1 in-D2* **have**
σ*-in-*Σ: $\sigma \in$ (Σ $\mathcal{A}$ $\cap$ Σ $\mathcal{A}'$) **by** *simp*

from *lang-eq*
have *remove-prefix* [σ] ($\mathcal{L}$*-in-state* $\mathcal{A}$ $q1$) = *remove-prefix* [σ] ($\mathcal{L}$*-in-state* $\mathcal{A}'$ $q2$) **by** *simp*
with σ*-in-*Σ
have $\bigcup\{\mathcal{L}$*-in-state* $\mathcal{A}$ q' $|q'$. ($q1$, σ, q') $\in \Delta$ $\mathcal{A}\}$ =
$\bigcup\{\mathcal{L}$*-in-state* $\mathcal{A}'$ q' $|q'$. ($q2$, σ, q') $\in \Delta$ $\mathcal{A}'\}$
by (*simp add*: $\mathcal{L}$*-in-state-remove-prefix*)
with *q1'-unique q2'-unique* **show** $\mathcal{L}$*-in-state* $\mathcal{A}$ $q1'$ = $\mathcal{L}$*-in-state* $\mathcal{A}'$ $q2'$ **by** *simp*
qed

lemma $\mathcal{L}$*-in-state---DFA---eq-DLTS-reachable* :
assumes *DFA-*$\mathcal{A}$: *DFA* $\mathcal{A}$
and *DFA-*$\mathcal{A}'$: *DFA* $\mathcal{A}'$
and *DLTS-reach-1*: *LTS-is-reachable* (Δ $\mathcal{A}$) $q1$ *w* $q1'$
and *DLTS-reach-2*: *LTS-is-reachable* (Δ $\mathcal{A}'$) $q2$ *w* $q2'$
and *lang-eq* : $\mathcal{L}$*-in-state* $\mathcal{A}$ $q1$ = $\mathcal{L}$*-in-state* $\mathcal{A}'$ $q2$
shows $\mathcal{L}$*-in-state* $\mathcal{A}$ $q1'$ = $\mathcal{L}$*-in-state* $\mathcal{A}'$ $q2'$
using *DLTS-reach-1 DLTS-reach-2 lang-eq*
proof (*induct w arbitrary*: *q1 q2*)
case *Nil* **thus** *?thesis* **by** *simp*
next
case (*Cons* σ *w*)
note *ind-hyp* = *Cons* (*1*)
note *DLTS-reach-1* = *Cons* (*2*)
note *DLTS-reach-2* = *Cons* (*3*)
note *lang-eq* = *Cons* (*4*)

from *DLTS-reach-1* **obtain** $q1''$ **where** *in-D1* : ($q1$, σ, $q1''$) $\in$ (Δ $\mathcal{A}$) **and**
DLTS-reach-1' : *LTS-is-reachable* (Δ $\mathcal{A}$) $q1''$ *w* $q1'$
by *auto*
from *DLTS-reach-2* **obtain** $q2''$ **where** *in-D2* : ($q2$, σ, $q2''$) $\in$ (Δ $\mathcal{A}'$) **and**
DLTS-reach-2' : *LTS-is-reachable* (Δ $\mathcal{A}'$) $q2''$ *w* $q2'$

by *auto*

have *lang-eq′* : *ℒ-in-state* $\mathcal{A}$ *q1′′* = *ℒ-in-state* $\mathcal{A}'$ *q2′′*
by (*fact ℒ-in-state---DFA---eq-reachable-step*
[*OF DFA-*$\mathcal{A}$ *DFA-*$\mathcal{A}'$ *in-D1 in-D2 lang-eq*])

note *ind-hyp* [*OF DLTS-reach-1′ DLTS-reach-2′ lang-eq′*]
thus *?thesis* **by** *assumption*
qed

lemma (**in** *DFA*) *NFA-is-deterministic---inj-rename* :
assumes *inj-f*: *inj-on f* ($\mathcal{Q}$ $\mathcal{A}$)
shows *NFA-is-deterministic* (*NFA-rename-states* $\mathcal{A}$ *f*)
proof −
have *step1*:
$\bigwedge\sigma$ *s1 s1′ s2 s2′*.
$[\![f\ s1 = f\ s1'; (s1, \sigma, s2) \in \Delta\ \mathcal{A}; (s1', \sigma, s2') \in \Delta\ \mathcal{A}]\!] \Longrightarrow f\ s2 = f\ s2'$
proof −
fix σ *s1 s1′ s2 s2′*
assume *in-D*: $(s1, \sigma, s2) \in \Delta\ \mathcal{A}$ **and** *in-D′*: $(s1', \sigma, s2') \in \Delta\ \mathcal{A}$

from Δ*-consistent in-D* **have** *s1-in*: $s1 \in \mathcal{Q}\ \mathcal{A}$ **by** *fast*
from Δ*-consistent in-D′* **have** *s1′-in*: $s1' \in \mathcal{Q}\ \mathcal{A}$ **by** *fast*

assume $f\ s1 = f\ s1'$
with *s1-in s1′-in inj-f* **have** *s1-eq*: $s1 = s1'$
unfolding *inj-on-def* **by** *simp*

from *s1-eq in-D in-D′ weak-deterministic*
have $s2 = s2'$
by (*auto simp add*: *LTS-is-weak-deterministic-def*)
thus $f\ s2 = f\ s2'$ **by** *simp*
qed

show *NFA-is-deterministic* (*NFA-rename-states* $\mathcal{A}$ *f*)
apply (*simp add*: *NFA-is-deterministic-def LTS-is-deterministic-def*
NFA-rename-states-def LTS-is-weak-deterministic-def)
apply (*rule conjI*)
apply (*metis step1*)
apply (*simp del*: *ex-simps add*: *ex-simps*[*symmetric*])
apply (*metis deterministic*[*unfolded LTS-is-deterministic-def*])
done
qed

lemma *DFA---inj-rename* :
assumes *DFA-*$\mathcal{A}$: *DFA* $\mathcal{A}$
and *inj-f*: *inj-on f* ($\mathcal{Q}$ $\mathcal{A}$)
shows *DFA* (*NFA-rename-states* $\mathcal{A}$ *f*)
using *DFA-*$\mathcal{A}$ *DFA.NFA-is-deterministic---inj-rename* [*OF DFA-*$\mathcal{A}$ *inj-f*]
unfolding *DFA-alt-def*
by (*simp add*: *NFA-rename-states---is-well-formed*)

lemma *NFA-isomorphic---is-well-formed-DFA* :
assumes *wf-A1*: *DFA* $\mathcal{A}1$
and *eq-A12*: *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$
shows *DFA* $\mathcal{A}2$
proof −
from *eq-A12* **obtain** *f* **where**
inj-f: *inj-on f* ($\mathcal{Q}$ $\mathcal{A}1$) **and**
A2-eq: $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1$ *f*
unfolding *NFA-isomorphic-def* **by** *blast*

from *A2-eq DFA---inj-rename* [*OF wf-A1 inj-f*]
show *DFA* $\mathcal{A}2$ **by** *simp*
qed

lemma *NFA-isomorphic-wf---DFA* :
fixes $\mathcal{A}1$ :: ($'q1$, $'a$) *NFA-rec* **and** $\mathcal{A}2$ :: ($'q2$, $'a$) *NFA-rec*
assumes *iso*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *DFA* $\mathcal{A}1 \longleftrightarrow$ *DFA* $\mathcal{A}2$
using *assms*
by (*metis NFA-isomorphic-wf-sym NFA-isomorphic-wf-def NFA-isomorphic---is-well-formed-DFA*)

lemma *NFA-normalise-states-DFA* [*simp*] :
assumes *wf-A*: *DFA* $\mathcal{A}$
shows *DFA* (*NFA-normalise-states* $\mathcal{A}$)
by (*metis NFA-isomorphic-wf-normalise-states*[*unfolded NFA-isomorphic-wf-def*]
NFA-isomorphic---is-well-formed-DFA DFA-alt-def assms)

3.5 Determinisation

definition *determinise-NFA* **where**
determinise-NFA $\mathcal{A}$ =
$(\!|\ \mathcal{Q} = \{q.\ q \subseteq (\mathcal{Q}\ \mathcal{A})\},$
$\Sigma = (\Sigma\ \mathcal{A}),$
$\Delta = \{(Q, \sigma, \{q'.\ (\exists q{\in}Q.\ (q, \sigma, q') \in \Delta\ \mathcal{A})\}) \mid Q\ \sigma.\ Q \subseteq \mathcal{Q}\ \mathcal{A} \wedge \sigma \in \Sigma\ \mathcal{A}\},$
$\mathcal{I} = \{\mathcal{I}\ \mathcal{A}\},$
$\mathcal{F} = \{fs.\ (fs \subseteq (\mathcal{Q}\ \mathcal{A})) \wedge (fs \cap \mathcal{F}\ \mathcal{A}) \neq \{\}\}\ |\!)$

lemma [*simp*] : $\mathcal{I}$ (*determinise-NFA* $\mathcal{A}$) = $\{\mathcal{I}\ \mathcal{A}\}$ **by** (*simp add*: *determinise-NFA-def*)
lemma [*simp*] : ($q \in \mathcal{Q}$ (*determinise-NFA* $\mathcal{A}$)) $\longleftrightarrow q \subseteq \mathcal{Q}\ \mathcal{A}$ **by** (*simp add*: *determinise-NFA-def*)
lemma [*simp*] : Σ (*determinise-NFA* $\mathcal{A}$) = $\Sigma\ \mathcal{A}$ **by** (*simp add*: *determinise-NFA-def*)
lemma [*simp*] : $q \in \mathcal{F}$ (*determinise-NFA* $\mathcal{A}$) $\longleftrightarrow (q \subseteq (\mathcal{Q}\ \mathcal{A})) \wedge q \cap \mathcal{F}\ \mathcal{A} \neq \{\}$ **by** (*simp add*: *determinise-NFA-def*)

lemma [*simp*] : $Q\sigma Q' \in \Delta$ (*determinise-NFA* $\mathcal{A}$) $\longleftrightarrow$
$((snd\ (snd\ Q\sigma Q') = \{q'.\ \exists q{\in}(fst\ Q\sigma Q').\ (q,\ fst\ (snd\ Q\sigma Q'),\ q') \in \Delta\ \mathcal{A}\}) \wedge$
$(fst\ Q\sigma Q' \subseteq \mathcal{Q}\ \mathcal{A}) \wedge ((fst\ (snd\ Q\sigma Q')) \in \Sigma\ \mathcal{A}))$
by (*cases* $Q\sigma Q'$, *simp add*: *determinise-NFA-def*)

lemma *determinise-NFA-is-detNFA* :
NFA-is-deterministic (*determinise-NFA* $\mathcal{A}$)
proof −
let $?\mathcal{A}$ = *determinise-NFA* $\mathcal{A}$
have $\exists q0.\ \mathcal{I}\ ?\mathcal{A} = \{q0\}$ **by** *simp*
moreover
have *LTS-is-deterministic* ($\mathcal{Q}\ ?\mathcal{A}$) ($\Sigma\ ?\mathcal{A}$) ($\Delta\ ?\mathcal{A}$)
unfolding *LTS-is-deterministic-def LTS-is-weak-deterministic-def determinise-NFA-def* **by** *force*
ultimately show *NFA-is-deterministic* $?\mathcal{A}$ **unfolding** *NFA-is-deterministic-def* **by** *simp*
qed

interpretation *det*: *detNFA determinise-NFA* $\mathcal{A}$ **by** (*simp add*: *detNFA-def determinise-NFA-is-detNFA*)

lemma *determinised*-i [*simp*] : i (*determinise-NFA* $\mathcal{A}$) = $\mathcal{I}\ \mathcal{A}$ **using** *det*.$\mathcal{I}$*-is-set*-i **by** *force*
lemma *determinised*-δ :
δ (*determinise-NFA* $\mathcal{A}$) = $(\lambda(Q,\sigma).$
if $(Q \subseteq (\mathcal{Q}\ \mathcal{A}) \wedge \sigma \in \Sigma\ \mathcal{A})$ *then Some* $\{q' \mid q\ q'.\ q \in Q \wedge (q, \sigma, q') \in \Delta\ \mathcal{A}\}$ *else None*)
apply (*subst fun-eq-iff* , *clarify*)
apply (*simp add*: *determinise-NFA-def* δ*-def LTS-to-DLTS-def Bex-def*)
done

lemma *determinise-NFA---is-well-formed* :
NFA $\mathcal{A} \Longrightarrow$ *NFA* (*determinise-NFA* $\mathcal{A}$)
by (*auto simp add*: *NFA-def determinise-NFA-def*)

lemma *determinise-NFA---DFA* :
$NFA\ \mathcal{A} \Longrightarrow DFA\ (determinise\text{-}NFA\ \mathcal{A})$
by (*simp add*: *determinise-NFA---is-well-formed DFA-alt-def*
determinise-NFA-is-detNFA)

lemma (**in** *NFA*) *determinise-NFA---DLTS-reach* :
shows $Q \subseteq \mathcal{Q}\ \mathcal{A} \Longrightarrow$
$DLTS\text{-}reach\ (\delta\ (determinise\text{-}NFA\ \mathcal{A}))\ Q\ w =$
$(if\ (w \in lists\ (\Sigma\ \mathcal{A}))\ then\ Some\ \{q'.\ \exists\, q{\in}Q.\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q'\}\ else\ None)$
proof (*induct w arbitrary*: *Q*)
case *Nil* **thus** *?case* **by** *simp*
next
case ($Cons\ \sigma\ w$)
note *ind-hyp* = *Cons*(*1*)
note *Q-subset* = *Cons*(*2*)

interpret *DFA-detA*: $DFA\ (determinise\text{-}NFA\ \mathcal{A})$ **using** *determinise-NFA---DFA* [*OF wf-NFA*]
by *assumption*

let $?Q' = \{q' \mid q\ q'.\ q \in Q \wedge (q, \sigma, q') \in \Delta\ \mathcal{A}\}$
from Δ-*consistent* **have** Q'-*subset* : $?Q' \subseteq \mathcal{Q}\ \mathcal{A}$ **by** *auto*

note *ind-hyp'* = *ind-hyp* [*OF Q'-subset*]
thus *?case*
by (*auto simp add*: *determinised-*δ *Q-subset*)
qed

lemma (**in** *NFA*) *determinise-NFA---*$\mathcal{L}$*-in-state* :
assumes *Q-subset*: $Q \subseteq \mathcal{Q}\ \mathcal{A}$
shows $\mathcal{L}\text{-}in\text{-}state\ (determinise\text{-}NFA\ \mathcal{A})\ Q = \bigcup\ \{\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q \mid q.\ q \in Q\}$
(**is** *?s1* = *?s2*)
proof (*intro set-eqI*)
fix *w*
show $w \in ?s1 \longleftrightarrow w \in ?s2$
proof ($cases\ w \in lists\ (\Sigma\ \mathcal{A})$)
case *False* **thus** *?thesis* **by** (*auto simp add*: $\mathcal{L}$*-in-state-def*)
next
case *True* **note** *w-wf* = *this*

interpret *DFA-detA*: $DFA\ (determinise\text{-}NFA\ \mathcal{A})$ **using** *determinise-NFA---DFA* [*OF wf-NFA*]
by *assumption*

from *w-wf*
have *in-s1-eq*: $w \in ?s1 =$
$(\{q'.\ \exists\, q{\in}Q.\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q'\} \subseteq \mathcal{Q}\ \mathcal{A} \wedge$
$\{q'.\ \exists\, q{\in}Q.\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q'\} \cap \mathcal{F}\ \mathcal{A} \neq \{\})$
apply (*simp add*: $\mathcal{L}$*-in-state-def DFA-detA.DFA-LTS-is-reachable-DLTS-reach-simp*)
apply (*simp add*: *determinise-NFA---DLTS-reach Q-subset*)
done

have *in-s1-eq2*: $(\{q'.\ \exists\, q{\in}Q.\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q'\} \cap \mathcal{F}\ \mathcal{A} \neq \{\}) =$
$(\exists\, q\ q'.\ q \in Q \wedge q' \in (\mathcal{F}\ \mathcal{A}) \wedge LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q')$
by *auto*

from *Q-subset NFA-*Δ*-cons---LTS-is-reachable* [**where** $w = w$]
have *in-s1-eq1*: $\{q'.\ \exists\, q{\in}Q.\ LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q'\} \subseteq \mathcal{Q}\ \mathcal{A}$ **by** *auto*

from *w-wf* **have** *in-s2-eq*:
$w \in ?s2 = (\exists\, q\ q'.\ q \in Q \wedge q' \in (\mathcal{F}\ \mathcal{A}) \wedge LTS\text{-}is\text{-}reachable\ (\Delta\ \mathcal{A})\ q\ w\ q')$
by (*auto simp add*: $\mathcal{L}$*-in-state-def*)

show *?thesis* **unfolding** *in-s2-eq in-s1-eq in-s1-eq2*

by (*simp add*: *in-s1-eq1*)
qed
qed

lemma (**in** *NFA*) *determinise-NFA-$\mathcal{L}$* [*simp*] :
$\mathcal{L}$ (*determinise-NFA* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
proof −
from *$\mathcal{I}$-consistent* **have** $\mathcal{I}$ $\mathcal{A}$ ∈ $\mathcal{Q}$ (*determinise-NFA* $\mathcal{A}$) **by** (*simp add*: *NFA-def*)

thus *?thesis*
by (*auto simp add*: *$\mathcal{L}$-alt-def determinise-NFA---$\mathcal{L}$-in-state*)
qed

lemma (**in** *NFA*) *determinise-NFA-accept* [*simp*] :
NFA-accept (*determinise-NFA* $\mathcal{A}$) *w* = *NFA-accept* $\mathcal{A}$ *w*
by (*simp add*: *NFA-accept-alt-def*)

definition *efficient-determinise-NFA* **where**
efficient-determinise-NFA $\mathcal{A}$ = *NFA-remove-unreachable-states* (*determinise-NFA* $\mathcal{A}$)

lemma *NFA-is-deterministic---NFA-remove-unreachable-states* :
NFA-is-deterministic $\mathcal{A}$ $\Longrightarrow$
NFA-is-deterministic (*NFA-remove-unreachable-states* $\mathcal{A}$)
proof −
have *det-impl*:
LTS-is-deterministic ($\mathcal{Q}$ $\mathcal{A}$) (Σ $\mathcal{A}$) (Δ $\mathcal{A}$) $\Longrightarrow$
LTS-is-deterministic ($\mathcal{Q}$ (*NFA-remove-unreachable-states* $\mathcal{A}$)) (Σ $\mathcal{A}$)
(Δ (*NFA-remove-unreachable-states* $\mathcal{A}$))
by (*simp add*: *LTS-is-deterministic-def LTS-is-weak-deterministic-def*
NFA-remove-unreachable-states-def Ball-def,
metis NFA-unreachable-states-extend)

assume *NFA-is-deterministic* $\mathcal{A}$
with *det-impl* **show** *?thesis* **by** (*simp add*: *NFA-is-deterministic-def*)
qed

lemma *DFA---NFA-remove-unreachable-states* :
DFA $\mathcal{A}$ $\Longrightarrow$ *DFA* (*NFA-remove-unreachable-states* $\mathcal{A}$)
unfolding *DFA-alt-def*
by (*metis NFA-is-deterministic---NFA-remove-unreachable-states*
NFA-remove-unreachable-states---is-well-formed)

lemma *efficient-determinise-NFA-is-detNFA* :
NFA-is-deterministic (*efficient-determinise-NFA* $\mathcal{A}$)
unfolding *efficient-determinise-NFA-def*
apply (*rule NFA-is-deterministic---NFA-remove-unreachable-states*)
apply (*rule determinise-NFA-is-detNFA*)
done

lemma *efficient-determinise-NFA-is-DFA* :
NFA $\mathcal{A}$ $\Longrightarrow$ *DFA* (*efficient-determinise-NFA* $\mathcal{A}$)
unfolding *DFA-def detNFA-def*
proof
show *NFA-is-deterministic* (*efficient-determinise-NFA* $\mathcal{A}$)
by (*fact efficient-determinise-NFA-is-detNFA*)
next
assume *wf-A*: *NFA* $\mathcal{A}$
show *NFA* (*efficient-determinise-NFA* $\mathcal{A}$)
unfolding *efficient-determinise-NFA-def*
apply (*rule NFA-remove-unreachable-states---is-well-formed*)
apply (*rule determinise-NFA---is-well-formed*)
apply (*fact wf-A*)

done
qed

lemma *efficient-determinise-NFA-*Σ [*simp*] :
Σ (*efficient-determinise-NFA* $\mathcal{A}$) = Σ $\mathcal{A}$
unfolding *efficient-determinise-NFA-def* **by** *simp*

lemma (**in** *NFA*) *efficient-determinise-NFA-accept* [*simp*] :
NFA-accept (*efficient-determinise-NFA* $\mathcal{A}$) *w* = *NFA-accept* $\mathcal{A}$ *w*
unfolding *efficient-determinise-NFA-def*
by *simp*

lemma (**in** *NFA*) *efficient-determinise-NFA-*$\mathcal{L}$ [*simp*] :
$\mathcal{L}$ (*efficient-determinise-NFA* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
unfolding *efficient-determinise-NFA-def*
by *simp*

lemma *efficient-determinise-NFA---is-initially-connected* :
NFA-is-initially-connected (*efficient-determinise-NFA* $\mathcal{A}$)
unfolding *efficient-determinise-NFA-def*
by (*simp add*: *NFA-remove-unreachable-states---NFA-is-initially-connected*)

lemma (**in** *NFA*) *efficient-determinise-NFA-compute* :
efficient-determinise-NFA $\mathcal{A}$ =
(*NFA-construct-reachable* $\{\mathcal{I}\ \mathcal{A}\}$ (Σ $\mathcal{A}$)
$(\lambda Q.\ (Q \subseteq \mathcal{Q}\ \mathcal{A}) \wedge Q \cap (\mathcal{F}\ \mathcal{A}) \neq \{\})\ \{(Q, \sigma, \{q'.\ \exists q{\in}Q.\ (q, \sigma, q') \in \Delta\ \mathcal{A}\})\ |Q\ \sigma.\ Q \subseteq \mathcal{Q}\ \mathcal{A} \wedge \sigma \in \Sigma\ \mathcal{A}\})$
proof −
note *determinise-NFA---is-well-formed* [*OF wf-NFA*]
hence *wf-det*: *NFA* (*determinise-NFA* $\mathcal{A}$) **.**

thus *?thesis*
unfolding *efficient-determinise-NFA-def determinise-NFA-def*
by (*rule NFA.NFA-remove-unreachable-states-implementation*) *simp-all*
qed

lemma *NFA-isomorphic-wf---NFA-determinise-cong* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*determinise-NFA* $\mathcal{A}1$) (*determinise-NFA* $\mathcal{A}2$)
proof −
from *equiv* **obtain** *f* **where** *inj-f* : *inj-on f* ($\mathcal{Q}$ $\mathcal{A}1$) **and**
$\mathcal{A}2$*-eq*: $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1$ *f* **and**
wf-$\mathcal{A}1$: *NFA* $\mathcal{A}1$
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

obtain *F* **where** *F-def* : $F = (\lambda S.\ f\ `\ S)$ **by** *blast*

have *F-elim*: $\bigwedge Q1\ Q2.\ [\![Q1 \subseteq \mathcal{Q}\ \mathcal{A}1;\ Q2 \subseteq \mathcal{Q}\ \mathcal{A}1]\!] \Longrightarrow (F\ Q1 = F\ Q2) \longleftrightarrow Q1 = Q2$
proof −
fix *Q1 Q2*
assume $Q1 \subseteq \mathcal{Q}\ \mathcal{A}1$ $Q2 \subseteq \mathcal{Q}\ \mathcal{A}1$
hence $Q1 \cup Q2 \subseteq \mathcal{Q}\ \mathcal{A}1$ **by** *simp*
with *inj-f* **have** *inj-on f* ($Q1 \cup Q2$) **by** (*rule subset-inj-on*)

thus $(F\ Q1 = F\ Q2) \longleftrightarrow Q1 = Q2$
using *inj-on-Un-image-eq-iff* [*of f Q1 Q2*] *F-def*
by *simp*
qed

from *F-elim* **have** *inj-F* : *inj-on F* ($\mathcal{Q}$ (*determinise-NFA* $\mathcal{A}1$))
unfolding *determinise-NFA-def inj-on-def*
by *simp*

have $\mathcal{A}2\text{-}eq'$: *determinise-NFA* $\mathcal{A}2$ = *NFA-rename-states* (*determinise-NFA* $\mathcal{A}1$) *F*
unfolding *determinise-NFA-def* $\mathcal{A}2$*-eq NFA-rename-states-def*
apply *simp*
proof (*intro conjI*)
show f ' $\mathcal{I}$ $\mathcal{A}1$ = F ($\mathcal{I}$ $\mathcal{A}1$) **unfolding** *F-def* **by** *simp*
next
show $\{q.\ q \subseteq f$ ' $\mathcal{Q}\ \mathcal{A}1\} = F$ ' $\{q.\ q \subseteq \mathcal{Q}\ \mathcal{A}1\}$
apply (*rule set-eqI*)
apply (*simp add*: *subset-image-iff F-def image-iff*)
done
next
have $\bigwedge Q.\ Q \subseteq (\mathcal{Q}\ \mathcal{A}1) \Longrightarrow (f$ ' $Q \cap (f$ ' $(\mathcal{F}\ \mathcal{A}1)) \neq \{\}) = (Q \cap \mathcal{F}\ \mathcal{A}1 \neq \{\})$
proof −
fix Q
assume *Q-sub*: $Q \subseteq (\mathcal{Q}\ \mathcal{A}1)$

from *NFA.*$\mathcal{F}$*-consistent* [*OF wf-*$\mathcal{A}1$]
have *F-sub*: $\mathcal{F}\ \mathcal{A}1 \subseteq \mathcal{Q}\ \mathcal{A}1$ **.**

from *inj-on-image-Int* [*OF inj-f Q-sub F-sub*, *symmetric*]
show $(f$ ' $Q \cap (f$ ' $(\mathcal{F}\ \mathcal{A}1)) \neq \{\}) = (Q \cap \mathcal{F}\ \mathcal{A}1 \neq \{\})$
by *simp*
qed
thus $\{fs.\ fs \subseteq f$ ' $\mathcal{Q}\ \mathcal{A}1 \wedge fs \cap f$ ' $\mathcal{F}\ \mathcal{A}1 \neq \{\}\} = F$ ' $\{fs.\ fs \subseteq \mathcal{Q}\ \mathcal{A}1 \wedge fs \cap \mathcal{F}\ \mathcal{A}1 \neq \{\}\}$
apply (*rule-tac set-eqI*)
apply (*simp add*: *subset-image-iff F-def image-iff*)
apply *metis*
done
next
from *inj-f NFA.*Δ*-consistent* [*OF wf-*$\mathcal{A}1$]
have *lem*: $\bigwedge\sigma\ Q.\ [\![Q \subseteq \mathcal{Q}\ \mathcal{A}1;\ \sigma \in \Sigma\ \mathcal{A}1]\!] \Longrightarrow$
$\{q'.\ \exists\, q{\in}Q.\ \exists\, s1\ s2.\ f\ q = f\ s1 \wedge q' = f\ s2 \wedge (s1,\ \sigma,\ s2) \in \Delta\ \mathcal{A}1\}$ =
f ' $\{q'.\ \exists\, q{\in}Q.\ (q,\ \sigma,\ q') \in \Delta\ \mathcal{A}1\}$
apply (*rule-tac set-eqI*)
apply (*simp add*: *image-iff inj-on-def subset-iff Bex-def*)
apply *auto*
apply *metis*
done

show $\{(Q,\ \sigma,\ \{q'.\ \exists\, q{\in}Q.\ \exists\, s1.\ q = f\ s1 \wedge (\exists\, s2.\ q' = f\ s2 \wedge (s1,\ \sigma,\ s2) \in \Delta\ \mathcal{A}1)\})\ |Q\ \sigma.$
$Q \subseteq f$ ' $\mathcal{Q}\ \mathcal{A}1 \wedge \sigma \in \Sigma\ \mathcal{A}1\}$ =
$\{(F\ s1,\ a,\ F\ \{q'.\ \exists\, q{\in}s1.\ (q,\ a,\ q') \in \Delta\ \mathcal{A}1\})\ |s1\ a.\ s1 \subseteq \mathcal{Q}\ \mathcal{A}1 \wedge a \in \Sigma\ \mathcal{A}1\}$

apply (*rule-tac set-eqI*)
apply (*simp add*: *subset-image-iff F-def image-iff*)
apply (*simp del*: *ex-simps add*: *ex-simps*[*symmetric*])
apply *auto*
apply (*insert lem*, *auto*)
apply (*insert lem*[*symmetric*], *auto*)
done
qed

from *inj-F* $\mathcal{A}2\text{-}eq'$ *determinise-NFA---is-well-formed* [*OF wf-*$\mathcal{A}1$]
show *?thesis*
unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *blast*
qed

lemma *NFA-isomorphic-efficient-determinise* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*efficient-determinise-NFA* $\mathcal{A}1$) (*efficient-determinise-NFA* $\mathcal{A}2$)
unfolding *efficient-determinise-NFA-def*

by (*intro NFA-isomorphic-wf---NFA-remove-unreachable-states*
NFA-isomorphic-wf---NFA-determinise-cong equiv)

definition *NFA-determinise* :: ($'q$::{*NFA-states*}, $'a$, -) *NFA-rec-scheme* $\Rightarrow$ ($'q$, $'a$) *NFA-rec* **where**
NFA-determinise $\mathcal{A}$ = *NFA-normalise-states* (*efficient-determinise-NFA* $\mathcal{A}$)

lemma *NFA-determinise-isomorphic-wf* :
assumes *wf-A*: *NFA* $\mathcal{A}$
shows *NFA-isomorphic-wf* (*efficient-determinise-NFA* $\mathcal{A}$)
(*NFA-determinise* $\mathcal{A}$)
unfolding *NFA-determinise-def*
apply (*rule NFA-isomorphic-wf-normalise-states*)
apply (*rule efficient-determinise-NFA-is-DFA* [*OF wf-A*, *THEN DFA-implies-NFA*])
done

lemma *NFA-determinise-*Σ [*simp*] :
Σ (*NFA-determinise* $\mathcal{A}$) = Σ $\mathcal{A}$
unfolding *NFA-determinise-def* **by** *simp*

lemma *NFA-determinise-is-DFA* :
assumes *wf-A* : *NFA* $\mathcal{A}$
shows *DFA* (*NFA-determinise* $\mathcal{A}$)
proof −
note *step1* = *efficient-determinise-NFA-is-DFA* [*OF wf-A*]
note *step2* = *NFA-determinise-isomorphic-wf* [*OF wf-A*]
note *step3* = *NFA-isomorphic---is-well-formed-DFA* [*OF step1 NFA-isomorphic-wf-D*(*3*)[*OF step2*]]
thus *?thesis* **.**
qed

lemma *NFA-determinise-NFA-accept* :
assumes *wf-A* : *NFA* $\mathcal{A}$
shows *NFA-accept* (*NFA-determinise* $\mathcal{A}$) *w* = *NFA-accept* $\mathcal{A}$ *w*
proof −
note *iso* = *NFA-determinise-isomorphic-wf* [*OF wf-A*]
from *NFA-isomorphic-wf-accept* [*OF iso*] *NFA.efficient-determinise-NFA-accept* [*OF wf-A*]
show *?thesis* **by** *auto*
qed

lemma *NFA-determinise-*$\mathcal{L}$:
NFA $\mathcal{A}$ $\Longrightarrow$ $\mathcal{L}$ (*NFA-determinise* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
by (*simp add*: $\mathcal{L}$*-def NFA-determinise-NFA-accept*)

3.6 Right quotient

lemma *NFA-right-quotient---is-well-formed-DFA* :
DFA $\mathcal{A}$ $\Longrightarrow$ *DFA* (*NFA-right-quotient* $\mathcal{A}$ *L*)
unfolding *DFA-alt-def NFA-is-deterministic-def*
by (*simp add*: *NFA-right-quotient---is-well-formed*)

3.7 Complement

definition *DFA-complement* :: ($'q$, $'a$, $'x$) *NFA-rec-scheme* $\Rightarrow$ ($'q$, $'a$) *NFA-rec* **where**
DFA-complement $\mathcal{A}$ = $(\!|$ $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$, Σ = Σ $\mathcal{A}$, Δ = Δ $\mathcal{A}$, $\mathcal{I}$ = $\mathcal{I}$ $\mathcal{A}$, $\mathcal{F}$ = $\mathcal{Q}$ $\mathcal{A}$ − $\mathcal{F}$ $\mathcal{A}$ $|\!)$

lemma [*simp*] : $\mathcal{Q}$ (*DFA-complement* $\mathcal{A}$) = $\mathcal{Q}$ $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def*)
lemma [*simp*] : Σ (*DFA-complement* $\mathcal{A}$) = Σ $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def*)
lemma [*simp*] : Δ (*DFA-complement* $\mathcal{A}$) = Δ $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def*)
lemma [*simp*] : $\mathcal{I}$ (*DFA-complement* $\mathcal{A}$) = $\mathcal{I}$ $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def*)
lemma [*simp*] : $\mathcal{F}$ (*DFA-complement* $\mathcal{A}$) = $\mathcal{Q}$ $\mathcal{A}$ − $\mathcal{F}$ $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def*)

lemma [*simp*] : δ (*DFA-complement* $\mathcal{A}$) = δ $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def* δ*-def*)
lemma [*simp*] : i (*DFA-complement* $\mathcal{A}$) = i $\mathcal{A}$ **by** (*simp add*: *DFA-complement-def* i*-def*)

lemma *DFA-complement---is-well-formed* : *NFA* $\mathcal{A} \Longrightarrow$ *NFA* (*DFA-complement* $\mathcal{A}$)
unfolding *NFA-def* **by** *auto*

lemma *DFA-complement-of-DFA-is-DFA* :
DFA $\mathcal{A} \Longrightarrow$ *DFA* (*DFA-complement* $\mathcal{A}$)
unfolding *DFA-alt-def detNFA-def NFA-def NFA-is-deterministic-def* **by** *auto*

lemma *DFA-complement---DLTS-reach* :
DLTS-reach (δ (*DFA-complement* $\mathcal{A}$)) q w = *DLTS-reach* (δ $\mathcal{A}$) q w **by** *simp*

lemma *DFA-complement-DFA-complement* [*simp*] :
NFA $\mathcal{A} \Longrightarrow$ *DFA-complement* (*DFA-complement* $\mathcal{A}$) = $\mathcal{A}$
proof −
assume *a*: *NFA* $\mathcal{A}$
then have $\mathcal{Q}\ \mathcal{A} - (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) = \mathcal{F}\ \mathcal{A}$ **using** *NFA.$\mathcal{F}$-consistent* **by** *auto*
thus *?thesis* **unfolding** *DFA-complement-def* **by** *simp*
qed

lemma *DFA-complement-word* :
assumes *wf-A*: *DFA* $\mathcal{A}$ **and** *w-in-Σ*: $w \in$ *lists* (Σ $\mathcal{A}$)
shows *NFA-accept* (*DFA-complement* $\mathcal{A}$) $w \longleftrightarrow \neg$ *NFA-accept* $\mathcal{A}$ w
proof −
let *?c$\mathcal{A}$* = *DFA-complement* $\mathcal{A}$

interpret *DFA-A*: *DFA* $\mathcal{A}$ **by** (*fact wf-A*)
interpret *DFA-cA*: *DFA ?c$\mathcal{A}$* **by** (*simp add*: *DFA-complement-of-DFA-is-DFA wf-A*)

from *w-in-Σ DFA-A.*i*-is-state*
have ~(*DLTS-reach* (δ $\mathcal{A}$) (i $\mathcal{A}$) w = *None*)
by (*simp add*: *DFA-A.DFA-DLTS-reach-is-some*)
then obtain q' **where** *DLTS-reach-eq-q'*: *DLTS-reach* (δ $\mathcal{A}$) (i $\mathcal{A}$) w = *Some* q' **by** *auto*

from *DLTS-reach-eq-q' DFA-A.*i*-is-state* **have**
$q' \in \mathcal{Q}\ \mathcal{A}$ **by** (*metis DFA-A.DFA-DLTS-reach-wf*)

with *DLTS-reach-eq-q'* **show** *?thesis*
by (*simp add*: *NFA-accept-def Bex-def w-in-Σ*
DFA-A.DFA-LTS-is-reachable-DLTS-reach-simp
DFA-cA.DFA-LTS-is-reachable-DLTS-reach-simp)
qed

lemma *DFA-complement-$\mathcal{L}$---lists-Σ* :
$\mathcal{L}$ (*DFA-complement* $\mathcal{A}$) $\subseteq$ *lists* (Σ $\mathcal{A}$)
by (*simp add*: *DFA-complement-def $\mathcal{L}$-def NFA-accept-def*, *auto*)

lemma (**in** *DFA*) *DFA-complement-$\mathcal{L}$* [*simp*] :
shows $\mathcal{L}$ (*DFA-complement* $\mathcal{A}$) = *lists* (Σ $\mathcal{A}$) − $\mathcal{L}$ $\mathcal{A}$
proof (*intro set-eqI*)
fix w
show ($w \in \mathcal{L}$ (*DFA-complement* $\mathcal{A}$)) $\longleftrightarrow w \in$ (*lists* (Σ $\mathcal{A}$) − $\mathcal{L}$ $\mathcal{A}$)
proof (*cases* $w \in$ *lists* (Σ $\mathcal{A}$))
case *False* **with** *DFA-complement-$\mathcal{L}$---lists-Σ* **show** *?thesis*
by *auto*
next
case *True*
note *w-in-Σ* = *this*
note *DFA-complement-w* = *DFA-complement-word* [*OF wf-DFA w-in-Σ*]

from *DFA-complement-w w-in-Σ* **show** *?thesis* **by** (*simp add*: *$\mathcal{L}$-def*)
qed

qed

lemma *NFA-isomorphic-wf---DFA-complement-cong* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*DFA-complement* $\mathcal{A}1$) (*DFA-complement* $\mathcal{A}2$)
proof −
  from *equiv* **obtain** *f* **where** *inj-f* : *inj-on f* ($\mathcal{Q}$ $\mathcal{A}1$) **and**
        $\mathcal{A}2$*-eq*: $\mathcal{A}2$ = *NFA-rename-states* $\mathcal{A}1$ *f* **and**
        wf-$\mathcal{A}1$: *NFA* $\mathcal{A}1$
    unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *auto*

  with *inj-f* **have** *inj-f'* : *inj-on f* ($\mathcal{Q}$ (*DFA-complement* $\mathcal{A}1$))
    by *simp*

  have $\mathcal{A}2$*-eq'*: *DFA-complement* $\mathcal{A}2$ = *NFA-rename-states* (*DFA-complement* $\mathcal{A}1$) *f*
    unfolding *DFA-complement-def* $\mathcal{A}2$*-eq*
    apply (*rule NFA-rec.equality*)
    apply (*simp-all*)
    apply (*simp add*: *NFA-rename-states-def*)
    apply (*insert inj-on-image-set-diff* [*of f* $\mathcal{Q}$ $\mathcal{A}1$ $\mathcal{Q}$ $\mathcal{A}1$ $\mathcal{F}$ $\mathcal{A}1$])
    apply (*insert NFA.*$\mathcal{F}$*-consistent*[*OF wf-*$\mathcal{A}1$])
    apply (*simp add*: *inj-f*)
  done

  from *inj-f'* $\mathcal{A}2$*-eq'* *DFA-complement---is-well-formed* [*OF wf-*$\mathcal{A}1$]
  show *?thesis*
    unfolding *NFA-isomorphic-wf-def NFA-isomorphic-def* **by** *blast*
qed

3.8 Boolean Combinations of NFAs

lemma *bool-comb-DFA---is-well-formed* :
assumes *DFA-*$\mathcal{A}1$: *DFA* $\mathcal{A}1$
    and *DFA-*$\mathcal{A}2$: *DFA* $\mathcal{A}2$
shows *DFA* (*bool-comb-NFA f* $\mathcal{A}1$ $\mathcal{A}2$)
proof −
  from *DFA-*$\mathcal{A}1$ *DFA-*$\mathcal{A}2$ *bool-comb-NFA---is-well-formed* [*of* $\mathcal{A}1$ $\mathcal{A}2$]
  have *NFA-ok*: *NFA* (*bool-comb-NFA f* $\mathcal{A}1$ $\mathcal{A}2$)
    unfolding *DFA-def* **by** *auto*

  from *DFA-*$\mathcal{A}1$ *DFA-*$\mathcal{A}2$
  have *NFA-is-deterministic* $\mathcal{A}1$ ∧ *NFA-is-deterministic* $\mathcal{A}2$
    unfolding *DFA-alt-def* **by** *simp*
  hence *det-ok*: *NFA-is-deterministic* (*bool-comb-NFA f* $\mathcal{A}1$ $\mathcal{A}2$)
    unfolding *NFA-is-deterministic-def LTS-is-deterministic-def*
      LTS-is-weak-deterministic-def Ball-def
    by (*auto*, *blast*)

  from *NFA-ok det-ok* **show** *?thesis*
    unfolding *DFA-alt-def* **by** *simp*
qed

lemma *efficient-bool-comb-DFA---is-well-formed* :
assumes *DFA-*$\mathcal{A}1$: *DFA* $\mathcal{A}1$
    and *DFA-*$\mathcal{A}2$: *DFA* $\mathcal{A}2$
shows *DFA* (*efficient-bool-comb-NFA f* $\mathcal{A}1$ $\mathcal{A}2$)
using *bool-comb-DFA---is-well-formed* [*OF DFA-*$\mathcal{A}1$ *DFA-*$\mathcal{A}2$, *of f*]
unfolding *efficient-bool-comb-NFA-def*
by (*rule DFA---NFA-remove-unreachable-states*)

lemma *bool-comb-DFA-NFA-accept* :
assumes *DFA-*$\mathcal{A}1$: *DFA* $\mathcal{A}1$

and *DFA-𝒜2* : *DFA 𝒜2*
shows *NFA-accept* (*bool-comb-NFA f 𝒜1 𝒜2*) *w* =
(*w* ∈ *lists* (Σ *𝒜1*) ∩ *lists* (Σ *𝒜2*) ∧ *f* (*NFA-accept 𝒜1 w*) (*NFA-accept 𝒜2 w*))
proof −
interpret *DFA1* : *DFA 𝒜1* **by** (*fact DFA-𝒜1*)
interpret *DFA2* : *DFA 𝒜2* **by** (*fact DFA-𝒜2*)

thus *?thesis*
proof (*cases w* ∈ *lists* (Σ *𝒜1*) ∩ *lists* (Σ *𝒜2*))
case *False* **thus** *?thesis*
by (*simp add*: *bool-comb-NFA-def NFA-accept-def*, *auto*)
next
case *True* **note** *w-in-lists* = *this*
then obtain *q1 q2* **where** *DLTS-reach-is-some*:
(*DLTS-reach* (δ *𝒜1*) (i *𝒜1*) *w* = *Some q1*) ∧
(*DLTS-reach* (δ *𝒜2*) (i *𝒜2*) *w* = *Some q2*)
using *DFA1.DFA-reach-is-none-iff* [**where** *w* = *w* **and** *q* = i *𝒜1*]
DFA2.DFA-reach-is-none-iff [**where** *w* = *w* **and** *q* = i *𝒜2*]
by (*simp add*: *DFA1*.i-*is-state DFA2*.i-*is-state*, *auto*)

from *DLTS-reach-is-some*
have *q12-in-Q*: *q1* ∈ *𝒬 𝒜1* ∧ *q2* ∈ *𝒬 𝒜2*
using *DFA1.DFA-DLTS-reach-wf* [**where** *w* = *w* **and** *q* = i *𝒜1* **and** *q'* = *q1*]
DFA2.DFA-DLTS-reach-wf [**where** *w* = *w* **and** *q* = i *𝒜2* **and** *q'* = *q2*]
DFA1.i-*is-state DFA2*.i-*is-state*
by *simp*

from *q12-in-Q DLTS-reach-is-some w-in-lists* **show** *?thesis*
by (*simp add*: *bool-comb-NFA-def NFA-accept-def*
LTS-is-reachable-product Bex-def lists-inter
DFA1.DFA-LTS-is-reachable-DLTS-reach-simp DFA1.i-*is-state*
DFA2.DFA-LTS-is-reachable-DLTS-reach-simp DFA2.i-*is-state*)
qed
qed

lemma *efficient-bool-comb-DFA-accept* :
assumes *DFA-𝒜1* : *DFA 𝒜1*
and *DFA-𝒜2* : *DFA 𝒜2*
shows *NFA-accept* (*efficient-bool-comb-NFA f 𝒜1 𝒜2*) *w* =
(*w* ∈ *lists* (Σ *𝒜1*) ∩ *lists* (Σ *𝒜2*) ∧ *f* (*NFA-accept 𝒜1 w*) (*NFA-accept 𝒜2 w*))
using *assms*
unfolding *efficient-bool-comb-NFA-def*
by (*simp add*: *bool-comb-DFA-NFA-accept*)

lemma *NFA-bool-comb-DFA---is-well-formed* :
assumes *DFA-𝒜1* : *DFA 𝒜1*
and *DFA-𝒜2* : *DFA 𝒜2*
shows *DFA* (*NFA-bool-comb f 𝒜1 𝒜2*)
proof −
from *DFA-𝒜1 DFA-𝒜2* **have** *NFA-𝒜12*: *NFA 𝒜1 NFA 𝒜2*
unfolding *DFA-alt-def* **by** *simp-all*

note *step1* = *efficient-bool-comb-DFA---is-well-formed* [*OF DFA-𝒜1 DFA-𝒜2*, *of f*]
note *iso* = *NFA-isomorphic-wf-D*(*3*) [*OF NFA-bool-comb---isomorphic-wf* [*OF NFA-𝒜12*]]
note *step3* = *NFA-isomorphic---is-well-formed-DFA* [*OF step1 iso*]
thus *?thesis* **.**
qed

lemma *NFA-bool-comb-DFA---NFA-accept* :
assumes *DFA-𝒜1* : *DFA 𝒜1*
and *DFA-𝒜2* : *DFA 𝒜2*

shows *NFA-accept* (*NFA-bool-comb f* $\mathcal{A}1$ $\mathcal{A}2$) *w* =
(*w* ∈ *lists* (Σ $\mathcal{A}1$) ∩ *lists* (Σ $\mathcal{A}2$) ∧ *f* (*NFA-accept* $\mathcal{A}1$ *w*) (*NFA-accept* $\mathcal{A}2$ *w*))
proof −
from *DFA-*$\mathcal{A}1$ *DFA-*$\mathcal{A}2$ **have** *NFA-*$\mathcal{A}12$: *NFA* $\mathcal{A}1$ *NFA* $\mathcal{A}2$
unfolding *DFA-alt-def* **by** *simp-all*

note *step1* = *efficient-bool-comb-DFA-accept* [*OF DFA-*$\mathcal{A}1$ *DFA-*$\mathcal{A}2$, *of f*]
note *iso* = *NFA-bool-comb---isomorphic-wf* [*OF NFA-*$\mathcal{A}12$]
note *step2* = *NFA-isomorphic-wf-*$\mathcal{L}$ [*OF iso*, *of f*]

from *step1 step2*
show *?thesis* **by** (*auto simp add*: *NFA-accept-alt-def*)
qed

3.9 Minimisation

definition *DFA-is-smaller-equiv* **where**
DFA-is-smaller-equiv $\mathcal{A}$ $\mathcal{A}'$ ≡
DFA $\mathcal{A}'$ ∧ (Σ $\mathcal{A}$ = Σ $\mathcal{A}'$) ∧ ($\mathcal{L}$ $\mathcal{A}$ = $\mathcal{L}$ $\mathcal{A}'$) ∧ (*card* ($\mathcal{Q}$ $\mathcal{A}'$) < *card* ($\mathcal{Q}$ $\mathcal{A}$))

definition *DFA-is-minimal* **where**
DFA-is-minimal $\mathcal{A}$ ≡ *DFA* ($\mathcal{A}$::(′*q*, ′*a*, ′*X*) *NFA-rec-scheme*) ∧
(∀ $\mathcal{A}'$::(′*q*, ′*a*) *NFA-rec*. ¬(*DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$))

The definition of minimal automata considers only automata that use the same type of states and no extra information. This is useful, because otherwise the right hand side would contain type variables that do not occur on the left side of the definition. However, the property holds in general.

lemma *NFA-accept-truncate* [*simp*] :
NFA-accept (*NFA-rec.truncate* $\mathcal{A}$) = *NFA-accept* $\mathcal{A}$
by (*simp add*: *fun-eq-iff NFA-accept-def NFA-rec.truncate-def*)

lemma $\mathcal{L}$*-truncate* [*simp*] :
$\mathcal{L}$ (*NFA-rec.truncate* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
by (*simp add*: $\mathcal{L}$*-def*)

lemma *NFA-truncate* [*simp*] :
NFA (*NFA-rec.truncate* $\mathcal{A}$) = *NFA* $\mathcal{A}$
unfolding *NFA-def*
by (*simp add*: *NFA-rec.truncate-def*)

lemma *NFA-is-deterministic-truncate* [*simp*] :
NFA-is-deterministic (*NFA-rec.truncate* $\mathcal{A}$) = *NFA-is-deterministic* $\mathcal{A}$
unfolding *NFA-is-deterministic-def*
by (*simp add*: *NFA-rec.truncate-def*)

lemma *detNFA-truncate* [*simp*] :
detNFA (*NFA-rec.truncate* $\mathcal{A}$) = *detNFA* $\mathcal{A}$
unfolding *detNFA-def*
by *simp*

lemma *DFA-truncate* [*simp*] :
DFA (*NFA-rec.truncate* $\mathcal{A}$) = *DFA* $\mathcal{A}$
unfolding *DFA-def*
by *simp*

lemma *DFA-is-smaller-equiv-truncate* [*simp*] : *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rec.truncate* $\mathcal{A}'$) ⟷ *DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$
unfolding *DFA-is-smaller-equiv-def*
by (*simp*, *simp add*: *NFA-rec.truncate-def*)

lemma *DFA-is-minimal-gen-def* :

fixes $\mathcal{A}$:: ('q, 'a, 'X) *NFA-rec-scheme*
shows *DFA-is-minimal* $\mathcal{A} \longleftrightarrow$ *DFA* $\mathcal{A} \wedge (\forall \mathcal{A}'$::('q, 'a, 'X2) *NFA-rec-scheme*. $\neg$(*DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$))
unfolding *DFA-is-minimal-def*
proof (*intro iffI allI conjI*)
  fix $\mathcal{A}'$ :: ('q, 'a, 'X2) *NFA-rec-scheme*
  assume *DFA* $\mathcal{A} \wedge$
        ($\forall \mathcal{A}'$::('q, 'a) *NFA-rec*. $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$)
  hence $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rec.truncate* $\mathcal{A}'$)
    by *blast*
  thus $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$ **by** *simp*
next
  fix $\mathcal{A}'$ :: ('q, 'a) *NFA-rec*
  assume *DFA* $\mathcal{A} \wedge$
        ($\forall \mathcal{A}'$::('q, 'a, 'X2) *NFA-rec-scheme*. $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$)
  hence $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rec.extend* $\mathcal{A}'$ (*ARB*::'X2))
    by *blast*
  hence $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rec.truncate* (*NFA-rec.extend* $\mathcal{A}'$ (*ARB*::'X2))) **by** *simp*
  hence $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rec.truncate* $\mathcal{A}'$)
    by (*simp add*: *NFA-rec.defs*)
  thus $\neg$ *DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$ **by** *simp*
qed *simp-all*

locale *minDFA* = *DFA* $\mathcal{A}$ **for** $\mathcal{A}$ +
  assumes *minimal-DFA*: *DFA-is-minimal* $\mathcal{A}$

lemma *minDFA-alt-def* :
  minDFA $\mathcal{A}$ = *DFA-is-minimal* $\mathcal{A}$
by (*simp add*: *minDFA-def minDFA-axioms-def DFA-is-minimal-def*)

lemma (**in** *minDFA*) *wf-minDFA* :
  minDFA $\mathcal{A}$ **by** (*simp add*: *minDFA-alt-def minimal-DFA*)

lemma (**in** *minDFA*) *DFA-is-minimal---DLTS-reachable* :
  $q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow q \notin$ *NFA-unreachable-states* $\mathcal{A}$
proof
  let *?*$\mathcal{A}'$ = *NFA-remove-unreachable-states* $\mathcal{A}$
  from *deterministic-NFA* **have** *det-A'*: *NFA-is-deterministic ?*$\mathcal{A}'$
    by (*simp add*: *NFA-is-deterministic---NFA-remove-unreachable-states*)

  have Σ-*A'*: Σ *?*$\mathcal{A}'$ = Σ $\mathcal{A}$ **by** (*simp add*: *NFA-remove-unreachable-states-def*)
  have *L-A'*: $\mathcal{L}$ *?*$\mathcal{A}'$ = $\mathcal{L}$ $\mathcal{A}$ **by** *simp*
  from *wf-NFA* **have** *wf-A'*: *NFA ?*$\mathcal{A}'$ **using** *NFA-remove-states---is-well-formed*
    unfolding *NFA-remove-unreachable-states-def* **by** *metis*
  with *det-A'* **have** *DFA ?*$\mathcal{A}'$ **unfolding** *DFA-alt-def* **by** *fast*

  assume $q \in \mathcal{Q}\ \mathcal{A}$ $q \in$ *NFA-unreachable-states* $\mathcal{A}$
  hence $\mathcal{Q}$ *?*$\mathcal{A}' \subset \mathcal{Q}\ \mathcal{A}$ **by** (*auto simp add*: *NFA-remove-unreachable-states-def*)
  hence *card-A'*: *card* ($\mathcal{Q}$ *?*$\mathcal{A}'$) < *card* ($\mathcal{Q}$ $\mathcal{A}$)
    by (*metis psubset-card-mono NFA-def wf-NFA*)

  from ‹*DFA ?*$\mathcal{A}'$› Σ-*A' L-A' card-A'*
  have *DFA-is-smaller-equiv* $\mathcal{A}$ *?*$\mathcal{A}'$
    unfolding *DFA-is-smaller-equiv-def*
    by *simp*
  hence *not-min-A*: $\neg$ *DFA-is-minimal* $\mathcal{A}$
    by (*metis DFA-is-minimal-def*)

  from *not-min-A minimal-DFA* **show** *False* **by** *auto*
qed

lemma (**in** *minDFA*) *DFA-is-minimal---initially-connected* :
  NFA-is-initially-connected $\mathcal{A}$

using *DFA-is-minimal---DLTS-reachable*
unfolding *NFA-is-initially-connected-def*
by *auto*

lemma (**in** *DFA*) *NFA-is-deterministic---combine-equiv-states* :
assumes *equiv-f*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ *f*
shows *NFA-is-deterministic* (*NFA-rename-states* $\mathcal{A}$ *f*)
proof −
  have *det*-Δ: *LTS-is-deterministic* ($\mathcal{Q}$ $\mathcal{A}$) (Σ $\mathcal{A}$) (Δ $\mathcal{A}$)
    by (*simp add*: *deterministic*)
  hence *weak-det*-Δ: *LTS-is-weak-deterministic* (Δ $\mathcal{A}$)
    by (*simp add*: *LTS-is-deterministic-def*)

  have *part1*:
  $\bigwedge q1\ q2\ q1'\ q2'\ \sigma.$
     $[\![(q1, \sigma, q1') \in \Delta\ \mathcal{A}; (q2, \sigma, q2') \in \Delta\ \mathcal{A}; f\ q1 = f\ q2]\!] \Longrightarrow (f\ q1' = f\ q2')$
  proof −
    fix $q1\ q2\ q1'\ q2'\ \sigma$
    assume *in*-Δ*1* : $(q1, \sigma, q1') \in \Delta\ \mathcal{A}$
    assume *in*-Δ*2* : $(q2, \sigma, q2') \in \Delta\ \mathcal{A}$
    assume *f-eq* : $f\ q1 = f\ q2$

    from *in*-Δ*1* *in*-Δ*2* Δ-*consistent* **have** $q1 \in \mathcal{Q}\ \mathcal{A} \wedge q2 \in \mathcal{Q}\ \mathcal{A}$ **by** *simp*
    with *f-eq equiv-f* **have** $\mathcal{L}$-*eq*: $\mathcal{L}$-*in-state* $\mathcal{A}$ *q1* = $\mathcal{L}$-*in-state* $\mathcal{A}$ *q2*
      by (*simp add*: *NFA-is-strong-equivalence-rename-fun-def*)

    have $\mathcal{L}'$-*eq*: $\mathcal{L}$-*in-state* $\mathcal{A}$ *q1'* = $\mathcal{L}$-*in-state* $\mathcal{A}$ *q2'*
      by (*fact* $\mathcal{L}$-*in-state---DFA---eq-reachable-step* [*OF wf-DFA wf-DFA in*-Δ*1 in*-Δ*2* $\mathcal{L}$-*eq*])

    from *in*-Δ*1* *in*-Δ*2* Δ-*consistent* **have** $q1' \in \mathcal{Q}\ \mathcal{A} \wedge q2' \in \mathcal{Q}\ \mathcal{A}$ **by** *simp*
    with $\mathcal{L}'$-*eq equiv-f* **show** *f'-eq*: $f\ q1' = f\ q2'$
      by (*simp add*: *NFA-is-strong-equivalence-rename-fun-def*)
  qed

  have *part2*:
   $\bigwedge q\ \sigma.\ [\![q \in \mathcal{Q}\ \mathcal{A}; \sigma \in \Sigma\ \mathcal{A}]\!] \Longrightarrow \exists\, q1\ q1'.\ f\ q = f\ q1 \wedge (q1, \sigma, q1') \in \Delta\ \mathcal{A}$
  proof −
    fix $q\ \sigma$
    assume $q \in \mathcal{Q}\ \mathcal{A}\ \sigma \in \Sigma\ \mathcal{A}$
    with *det*-Δ **obtain** q' **where** *q'-intro*: $(q, \sigma, q') \in \Delta\ \mathcal{A}$
      by (*auto simp add*: *LTS-is-deterministic-def*)

    from *q'-intro* **show** $\exists\, q1\ q1'.\ f\ q = f\ q1 \wedge (q1, \sigma, q1') \in \Delta\ \mathcal{A}$ **by** *blast*
  qed

  from *part1 part2* **show** *?thesis*
    by (*simp add*: *NFA-is-deterministic-def LTS-is-deterministic-def LTS-is-weak-deterministic-def*,
      metis)
qed

lemma *DFA---combine-equiv-states* :
assumes *DFA*-$\mathcal{A}$: *DFA* $\mathcal{A}$
  and *equiv-f*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ *f*
shows *DFA* (*NFA-rename-states* $\mathcal{A}$ *f*)
using *DFA*-$\mathcal{A}$ *DFA.NFA-is-deterministic---combine-equiv-states* [*OF DFA*-$\mathcal{A}$ *equiv-f*]
unfolding *DFA-alt-def*
by (*simp add*: *NFA-rename-states---is-well-formed*)

lemma (**in** *minDFA*) $\mathcal{L}$-*in-state-inj* :
  assumes *q*: $q \in \mathcal{Q}\ \mathcal{A}$ **and** *q'*: $q' \in \mathcal{Q}\ \mathcal{A}$ **and** *q-not-q'*: $q \neq q'$

shows $\mathcal{L}$*-in-state* $\mathcal{A}\ q \neq \mathcal{L}$*-in-state* $\mathcal{A}\ q'$
proof
obtain *f* **where** *f-is-serf*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ $(f{::}'a \Rightarrow 'a)$
using *NFA-is-strong-equivalence-rename-fun-exists* **by** *metis*
let $?\mathcal{A}'$ = *NFA-rename-states* $\mathcal{A}$ *f*

have *DFA* $?\mathcal{A}'$ **by**
(*fact DFA---combine-equiv-states* [*OF wf-DFA f-is-serf*])
moreover
note $\mathcal{L}$*-rename-iff* [*OF NFA-is-strong-equivalence-rename-fun---weaken* [*OF f-is-serf*]]
hence $\mathcal{L}\ \mathcal{A} = \mathcal{L}\ ?\mathcal{A}'$ **by** *simp*
moreover
assume $\mathcal{L}$*-in-state* $\mathcal{A}\ q = \mathcal{L}$*-in-state* $\mathcal{A}\ q'$
with *f-is-serf* $q\ q'$ *q-not-q'* **have** *not-inj*: $\neg$ *inj-on f* $(\mathcal{Q}\ \mathcal{A})$
unfolding *inj-on-def NFA-is-strong-equivalence-rename-fun-def* **by** *auto*
from *finite-*$\mathcal{Q}$ *not-inj* **have** *not-eq*: *card* $(\mathcal{Q}\ ?\mathcal{A}') \neq$ *card* $(\mathcal{Q}\ \mathcal{A})$ **using** *eq-card-imp-inj-on*
unfolding *NFA-rename-states-def* **by** *auto*
from *finite-*$\mathcal{Q}$ **have** *card* $(\mathcal{Q}\ ?\mathcal{A}') \leq$ *card* $(\mathcal{Q}\ \mathcal{A})$
using *card-image-le* **unfolding** *NFA-rename-states-def* **by** *auto*
with *not-eq* **have** *card* $(\mathcal{Q}\ ?\mathcal{A}') <$ *card* $(\mathcal{Q}\ \mathcal{A})$ **by** *simp*
ultimately have *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rename-states* $\mathcal{A}$ *f*)
unfolding *DFA-is-smaller-equiv-def* **by** *simp*
hence $\neg$ *DFA-is-minimal* $\mathcal{A}$ **by** (*metis DFA-is-minimal-def*)
with *minimal-DFA* **show** *False* **by** *simp*
qed

lemma (**in** *DFA*) *DFA-is-minimal-intro* :
assumes
connected: *NFA-is-initially-connected* $\mathcal{A}$ **and**
no-equivalant-states: *inj-on* ($\mathcal{L}$*-in-state* $\mathcal{A}$) $(\mathcal{Q}\ \mathcal{A})$
shows *DFA-is-minimal* $\mathcal{A}$
proof (*rule ccontr*)
assume $\neg$ *DFA-is-minimal* $\mathcal{A}$
then obtain $\mathcal{A}'{::}('q,\ 'a)$ *NFA-rec* **where**
dfa-A': *DFA* $\mathcal{A}'$ **and**
same-Σ: $\Sigma\ \mathcal{A} = \Sigma\ \mathcal{A}'$ **and**
same-$\mathcal{L}$: $\mathcal{L}\ \mathcal{A} = \mathcal{L}\ \mathcal{A}'$ **and**
smaller: *card* $(\mathcal{Q}\ \mathcal{A}') <$ *card* $(\mathcal{Q}\ \mathcal{A})$
unfolding *DFA-is-minimal-def DFA-is-smaller-equiv-def* **by** (*auto simp add*: *wf-DFA*)

interpret *DFA-A'* : *DFA* $\mathcal{A}'$ **by** (*fact dfa-A'*)

have $\neg$ ($\mathcal{L}$*-in-state* $\mathcal{A}$) ‘ $(\mathcal{Q}\ \mathcal{A}) \subseteq$ ($\mathcal{L}$*-in-state* $\mathcal{A}'$) ‘ $(\mathcal{Q}\ \mathcal{A}')$
proof –
note *DFA-A'.finite-*$\mathcal{Q}$
hence *fin-L-A'*: *finite* (($\mathcal{L}$*-in-state* $\mathcal{A}'$) ‘ $(\mathcal{Q}\ \mathcal{A}')$) **by** (*metis finite-imageI*)
from ⟨*finite* $(\mathcal{Q}\ \mathcal{A}')$⟩ **have** *card* (($\mathcal{L}$*-in-state* $\mathcal{A}'$) ‘ $(\mathcal{Q}\ \mathcal{A}')$) $\leq$ *card* $(\mathcal{Q}\ \mathcal{A}')$ **by** (*metis card-image-le*)
moreover
from *no-equivalant-states* **have** *card* (($\mathcal{L}$*-in-state* $\mathcal{A}$) ‘ $(\mathcal{Q}\ \mathcal{A})$) = *card* $(\mathcal{Q}\ \mathcal{A})$ **by** (*metis card-image*)
ultimately
have $\neg$ *card* (($\mathcal{L}$*-in-state* $\mathcal{A}$) ‘ $(\mathcal{Q}\ \mathcal{A})$) $\leq$ *card* (($\mathcal{L}$*-in-state* $\mathcal{A}'$) ‘ $(\mathcal{Q}\ \mathcal{A}')$) **using** *smaller* **by** *simp*
then show *?thesis* **by** (*metis card-mono fin-L-A'*)
qed
then obtain *q* **where** *q-in-Q*: $q \in \mathcal{Q}\ \mathcal{A}$ **and**
$\mathcal{L}$*-q*: $\mathcal{L}$*-in-state* $\mathcal{A}\ q \notin$ ($\mathcal{L}$*-in-state* $\mathcal{A}'$) ‘ $(\mathcal{Q}\ \mathcal{A}')$ **by** *auto*
from *q-in-Q connected* **obtain** *w* **where** *DLTS-reach-*$\mathcal{A}$: *LTS-is-reachable* $(\Delta\ \mathcal{A})$ $(\mathrm{i}\ \mathcal{A})$ *w q*
unfolding *NFA-is-initially-connected-alt-def*
by *auto*
hence *w-word*: $w \in$ *lists* $(\Sigma\ \mathcal{A})$ **using** *LTS-is-reachable---labels* **by** *fast*
with *same-*Σ **have** $w \in$ *lists* $(\Sigma\ \mathcal{A}')$ **by** *simp*
hence $\neg$(*DLTS-reach* $(\delta\ \mathcal{A}')$ $(\mathrm{i}\ \mathcal{A}')$ *w* = *None*) **unfolding** *DFA-A'.DFA-reach-is-none-iff*
by (*simp add*: *DFA-A'*.i*-is-state*)

```
  then obtain q' where DLTS-reach-𝒜': LTS-is-reachable (Δ 𝒜') (i 𝒜') w q'
    by (auto simp add: DFA-A'.DFA-LTS-is-reachable-DLTS-reach-simp)
  hence q' ∈ 𝒬 𝒜'
    using DFA-A'.NFA-Δ-cons---LTS-is-reachable DFA-A'.i-is-state
    by simp
  with ℒ-q have ℒ-q-q'-neq: ℒ-in-state 𝒜' q' ≠ ℒ-in-state 𝒜 q by blast
  show False
    using same-ℒ ℒ-q-q'-neq ℒ-in-state-i DFA-A'.ℒ-in-state-i
          ℒ-in-state---DFA---eq-DLTS-reachable [OF wf-DFA dfa-A'
             DLTS-reach-𝒜 DLTS-reach-𝒜']
    by simp
qed

theorem (in DFA) DFA-is-minimal-alt-def :
  DFA-is-minimal 𝒜 ⟷
    (NFA-is-initially-connected 𝒜 ∧
     inj-on (ℒ-in-state 𝒜) (𝒬 𝒜))
   (is - = (?P1 ∧ ?P2))
proof
  assume DFA-is-minimal 𝒜
  then interpret minDFA-A: minDFA 𝒜 by (simp add: minDFA-alt-def)

  show ?P1 ∧ ?P2
  proof
    show ?P1 by (auto simp add: minDFA-A.DFA-is-minimal---initially-connected)
  next
    show ?P2 by (auto simp add: inj-on-def, metis minDFA-A.ℒ-in-state-inj)
  qed
next
  assume ?P1 ∧ ?P2
  with wf-DFA show DFA-is-minimal 𝒜
    by (simp add: DFA-is-minimal-intro)
qed

lemma DFA-is-minimal-alt-DFA-def :
  DFA-is-minimal 𝒜 ⟷
    (DFA 𝒜 ∧ NFA-is-initially-connected 𝒜 ∧
     inj-on (ℒ-in-state 𝒜) (𝒬 𝒜))
   (is - = (?P1 ∧ ?P2))
apply (cases DFA 𝒜)
  apply (simp add: DFA.DFA-is-minimal-alt-def)
apply (simp add: DFA-is-minimal-def)
done

lemma NFA-isomorphic---DFA-is-minimal :
fixes 𝒜1 :: ('q1, 'a, 'X) NFA-rec-scheme
fixes 𝒜2 :: ('q2, 'a) NFA-rec
assumes eq-A12: NFA-isomorphic 𝒜1 𝒜2
    and min-A1: DFA-is-minimal 𝒜1
shows DFA-is-minimal 𝒜2
proof (rule ccontr)
  assume not-min-A2: ¬(DFA-is-minimal 𝒜2)

  from eq-A12 obtain f where
    inj-f: inj-on f (𝒬 𝒜1) and
    A2-eq: 𝒜2 = NFA-rename-states 𝒜1 f
    unfolding NFA-isomorphic-def by blast

  from min-A1 have wf-A1: DFA 𝒜1
    unfolding DFA-is-minimal-def by fast

  from NFA-isomorphic---is-well-formed-DFA [OF wf-A1 eq-A12]
```

have *wf-A2*: *DFA* $\mathcal{A}2$ **.**

from *not-min-A2* **obtain** $\mathcal{A}2'$ :: ('*q2*, '*a*) *NFA-rec*
  where *A2'-smaller*: *DFA-is-smaller-equiv* $\mathcal{A}2$ $\mathcal{A}2'$
  by (*auto simp add*: *DFA-is-minimal-def wf-A2*)

have *wf-A2'*: *DFA* $\mathcal{A}2'$
  using *A2'-smaller*
  unfolding *DFA-is-smaller-equiv-def* **by** *simp*

have *fin-QA2'* : *finite* ($\mathcal{Q}$ $\mathcal{A}2'$)
  using *wf-A2'* [*unfolded DFA-alt-def*] *NFA.finite-*$\mathcal{Q}$
  by *blast*
have *fin-QA1* : *finite* ($\mathcal{Q}$ $\mathcal{A}1$)
  using *wf-A1* [*unfolded DFA-alt-def*] *NFA.finite-*$\mathcal{Q}$
  by *blast*

from *A2'-smaller* **have** *card* ($\mathcal{Q}$ $\mathcal{A}2'$) < *card* ($\mathcal{Q}$ $\mathcal{A}2$)
  unfolding *DFA-is-smaller-equiv-def* **by** *fast*
hence *card* ($\mathcal{Q}$ $\mathcal{A}2'$) < *card* ($\mathcal{Q}$ $\mathcal{A}1$)
  unfolding *A2-eq*
  by (*simp add*: *inj-f card-image*)
then obtain *f'* :: '*q2* ⇒ '*q1* **where** *inj-f'*: *inj-on f'* ($\mathcal{Q}$ $\mathcal{A}2'$)
  using *inj-on-iff-card-le* [*OF fin-QA2' fin-QA1*]
  by *auto*

have *equiv-f'*: *NFA-is-equivalence-rename-fun* $\mathcal{A}2'$ *f'*
  using *inj-f'*
  unfolding *NFA-is-equivalence-rename-fun-def inj-on-def*
  by *auto*

have *equiv-f*: *NFA-is-equivalence-rename-fun* $\mathcal{A}1$ *f*
  using *inj-f*
  unfolding *NFA-is-equivalence-rename-fun-def inj-on-def*
  by *auto*

from *A2'-smaller wf-A1* **have**
  A1-smaller: *DFA-is-smaller-equiv* $\mathcal{A}1$ (*NFA-rename-states* $\mathcal{A}2'$ *f'*)
  unfolding *DFA-is-smaller-equiv-def A2-eq*
  apply (*simp add*: *inj-f inj-f' card-image DFA---inj-rename*)
  apply (*simp add*: *DFA-alt-def equiv-f' NFA.*$\mathcal{L}$*-rename-iff equiv-f*)
done

from *A1-smaller* **have** ¬(*DFA-is-minimal* $\mathcal{A}1$)
  unfolding *DFA-is-minimal-def* **by** *auto*
with *min-A1* **show** *False* **by** *fast*
qed

lemma *DFA-is-minimal---equiv-states-injection-exists* :
fixes $\mathcal{A}1$ :: ('*q1*, '*a*, '*X1*) *NFA-rec-scheme*
fixes $\mathcal{A}2$ :: ('*q2*, '*a*, '*X2*) *NFA-rec-scheme*
assumes *L-eq*: $\mathcal{L}$ $\mathcal{A}1$ = $\mathcal{L}$ $\mathcal{A}2$
  and Σ*-eq*: Σ $\mathcal{A}1$ = Σ $\mathcal{A}2$
  and *min-A1*: *DFA-is-minimal* $\mathcal{A}1$
  and *min-A2*: *DFA-is-minimal* $\mathcal{A}2$
shows ∃ *f*. *inj-on f* ($\mathcal{Q}$ $\mathcal{A}1$) ∧
    (∀ *q*∈$\mathcal{Q}$ $\mathcal{A}1$. (*f q*) ∈ $\mathcal{Q}$ $\mathcal{A}2$ ∧ $\mathcal{L}$*-in-state* $\mathcal{A}2$ (*f q*) = $\mathcal{L}$*-in-state* $\mathcal{A}1$ *q*)
proof −
  from *min-A1* **have** *wf-A1*: *DFA* $\mathcal{A}1$
    by (*simp add*: *DFA-is-minimal-alt-DFA-def*)
  from *min-A2* **have** *wf-A2*: *DFA* $\mathcal{A}2$
    by (*simp add*: *DFA-is-minimal-alt-DFA-def*)

```
interpret DFA1 : DFA 𝒜1 by (fact wf-A1)
interpret DFA2 : DFA 𝒜2 by (fact wf-A2)

let ?states-equiv = λq1 q2. q2 ∈ 𝒬 𝒜2 ∧ ℒ-in-state 𝒜2 q2 = ℒ-in-state 𝒜1 q1
have ∀ q1 ∈ 𝒬 𝒜1. ∃ q2. ?states-equiv q1 q2
proof
  fix q1
  assume q1-in: q1 ∈ 𝒬 𝒜1

  from min-A1 have NFA-is-initially-connected 𝒜1 by (simp add: DFA-is-minimal-alt-DFA-def)
  then obtain w where w-in-Σ1: w ∈ lists (Σ 𝒜1) and
                  DLTS-reach-q1: LTS-is-reachable (Δ 𝒜1) (i 𝒜1) w q1
    unfolding NFA-is-initially-connected-alt-def
    using q1-in by auto

  from  w-in-Σ1 Σ-eq have w-in-Σ2: w ∈ lists (Σ 𝒜2) by fast

  from DFA2.DFA-DLTS-reach-is-some [OF DFA2.i-is-state w-in-Σ2]
  obtain q2 where DLTS-reach (δ 𝒜2) (i 𝒜2) w = Some q2 by auto
  hence DLTS-reach-q2: LTS-is-reachable (Δ 𝒜2) (i 𝒜2) w q2
    by (simp add: DFA2.DFA-LTS-is-reachable-DLTS-reach-simp)

  from DFA2.NFA-Δ-cons---LTS-is-reachable [OF DLTS-reach-q2]
  have q2-in: q2 ∈ 𝒬 𝒜2 by (simp add: DFA2.i-is-state)

  from L-eq have initial-equiv: ℒ-in-state 𝒜1 (i 𝒜1) = ℒ-in-state 𝒜2 (i 𝒜2)
    unfolding ℒ-alt-def by simp

  from ℒ-in-state---DFA---eq-DLTS-reachable [OF wf-A1 wf-A2 DLTS-reach-q1 DLTS-reach-q2 initial-equiv]
  have q12-equiv: ℒ-in-state 𝒜1 q1 = ℒ-in-state 𝒜2 q2 .

  from q12-equiv q2-in show ∃ q2. ?states-equiv q1 q2 by blast
qed
from bchoice[OF this] guess f .. note f-prop = this

have inj-f : inj-on f (𝒬 𝒜1)
unfolding inj-on-def
proof (intro ballI impI)
  fix q q′
  assume q1-in: q ∈ 𝒬 𝒜1
     and q2-in: q′ ∈ 𝒬 𝒜1
     and f-qq′-eq: f q = f q′
  with f-prop have L-qq′-eq: ℒ-in-state 𝒜1 q = ℒ-in-state 𝒜1 q′ by metis

  from q1-in q2-in L-qq′-eq min-A1
  show q = q′
    unfolding DFA-is-minimal-alt-DFA-def inj-on-def
    by simp
qed

from f-prop inj-f show ?thesis by blast
qed

lemma DFA-is-minimal---equiv-states-bijection-exists :
fixes 𝒜1 :: ('q1, 'a, 'X1) NFA-rec-scheme
fixes 𝒜2 :: ('q2, 'a, 'X2) NFA-rec-scheme
assumes L-eq: ℒ 𝒜1 = ℒ 𝒜2
  and Σ-eq: Σ 𝒜1 = Σ 𝒜2
  and min-A1: DFA-is-minimal 𝒜1
  and min-A2: DFA-is-minimal 𝒜2
shows ∃f. bij-betw f (𝒬 𝒜1) (𝒬 𝒜2) ∧
```

```
        (∀ q∈𝒬 𝒜1. ℒ-in-state 𝒜2 (f q) = ℒ-in-state 𝒜1 q)
proof -
  from DFA-is-minimal---equiv-states-injection-exists
    [OF L-eq Σ-eq min-A1 min-A2]
  guess f1 .. note f1-props = this

  from DFA-is-minimal---equiv-states-injection-exists
    [OF L-eq[symmetric] Σ-eq[symmetric] min-A2 min-A1]
  guess f2 .. note f2-props = this

  have A2-eq: f1 ‘ 𝒬 𝒜1 = 𝒬 𝒜2
  proof
    from f1-props show f1 ‘ 𝒬 𝒜1 ⊆ 𝒬 𝒜2 by auto
  next
    show 𝒬 𝒜2 ⊆ f1 ‘ 𝒬 𝒜1
    proof
      fix q2
      assume q2-in: q2 ∈ 𝒬 𝒜2
      let ?q1 = f2 q2
      let ?q2' = f1 ?q1

      from q2-in f2-props
      have q1-in: ?q1 ∈ 𝒬 𝒜1 and
           q1-q2-equiv: ℒ-in-state 𝒜1 ?q1 = ℒ-in-state 𝒜2 q2
        by simp-all

      from f1-props q1-in
      have q2'-in: ?q2' ∈ 𝒬 𝒜2 and
           q2'-q1-equiv: ℒ-in-state 𝒜2 ?q2' = ℒ-in-state 𝒜1 ?q1
        by simp-all

      from q1-q2-equiv q2'-q1-equiv have
        q2-q2'-equiv: ℒ-in-state 𝒜2 ?q2' = ℒ-in-state 𝒜2 q2
        by simp

      from min-A2 q2-in q2'-in q2-q2'-equiv
      have q2'-eq: ?q2' = q2
        unfolding DFA-is-minimal-alt-DFA-def inj-on-def by simp

      from q2'-eq q1-in show q2 ∈ f1 ‘ 𝒬 𝒜1
        by (simp add: image-iff) metis
    qed
  qed
  from f1-props A2-eq show ?thesis
    apply (rule-tac exI [where x = f1])
    apply (simp add: bij-betw-def)
  done
qed

lemma DFA-is-minimal---isomorph-wf :
fixes 𝒜1 :: ('q1, 'a, 'X) NFA-rec-scheme
fixes 𝒜2 :: ('q2, 'a) NFA-rec
assumes min-A1: DFA-is-minimal 𝒜1
    and min-A2: DFA-is-minimal 𝒜2
    and L-eq: ℒ 𝒜1 = ℒ 𝒜2
    and Σ-eq: Σ 𝒜1 = Σ 𝒜2
shows NFA-isomorphic-wf 𝒜1 𝒜2
proof -
  from min-A1 have wf-A1: DFA 𝒜1
    by (simp add: DFA-is-minimal-alt-DFA-def)
  from min-A2 have wf-A2: DFA 𝒜2
    by (simp add: DFA-is-minimal-alt-DFA-def)
```

```
interpret DFA1 : DFA 𝒜1 by (fact wf-A1)
interpret DFA2 : DFA 𝒜2 by (fact wf-A2)

from DFA-is-minimal---equiv-states-bijection-exists
  [OF L-eq Σ-eq min-A1 min-A2]
guess f .. note f-props = this
hence bij-f: bij-betw f (𝒬 𝒜1) (𝒬 𝒜2)
  and f-equiv: ⋀q1. q1∈𝒬 𝒜1 ⟹ ℒ-in-state 𝒜2 (f q1) = ℒ-in-state 𝒜1 q1
  by simp-all

from bij-f have f-is-state: ⋀q1. q1∈𝒬 𝒜1 ⟹ f q1 ∈ 𝒬 𝒜2
   unfolding bij-betw-def by auto

from bij-f have inj-f: inj-on f (𝒬 𝒜1)
   unfolding bij-betw-def by fast

have A2-eq: 𝒜2 = NFA-rename-states 𝒜1 f
unfolding NFA-rename-states-def
proof (intro NFA-rec.equality, simp-all)
  show Σ 𝒜2 = Σ 𝒜1 by (simp add: Σ-eq)
next
  show 𝒬 𝒜2 = f ‘ 𝒬 𝒜1
    using bij-f unfolding bij-betw-def
    by simp
next
  from f-equiv [OF DFA1.i-is-state] L-eq
  have ℒ-in-state 𝒜2 (f (i 𝒜1)) = ℒ-in-state 𝒜2 (i 𝒜2)
    unfolding ℒ-alt-def by simp
  with min-A2 DFA2.i-is-state f-is-state [OF DFA1.i-is-state]
  show i 𝒜2 = f (i 𝒜1)
    unfolding DFA-is-minimal-alt-DFA-def inj-on-def
    by simp
next
  show ℱ 𝒜2 = f ‘ ℱ 𝒜1
  proof (intro set-eqI iffI)
    fix q2
    assume q2 ∈ f ‘ (ℱ 𝒜1)
    then obtain q1 where q1-in-F: q1 ∈ ℱ 𝒜1 and
                         q2-eq: q2 = f q1 by auto

    from DFA1.ℱ-consistent q1-in-F
    have q1-in-Q: q1 ∈ 𝒬 𝒜1 by blast

    from f-equiv[OF q1-in-Q] q2-eq q1-in-F
    show q2 ∈ ℱ 𝒜2
    by (simp add: in-ℒ-in-state-Nil [symmetric]
             del: in-ℒ-in-state-Nil)
  next
    fix q2
    assume q2-in-F: q2 ∈ ℱ 𝒜2

    from DFA2.ℱ-consistent q2-in-F
    have q2-in-Q: q2 ∈ 𝒬 𝒜2 by blast

    with bij-f obtain q1 where q1-in-Q: q1 ∈ 𝒬 𝒜1 and
                       q2-eq: q2 = f q1
      unfolding bij-betw-def
      by auto

    from f-equiv[OF q1-in-Q] q2-eq q2-in-F
    have q1-in-F: q1 ∈ ℱ 𝒜1
```

```
      by (simp add: in-ℒ-in-state-Nil [symmetric]
              del: in-ℒ-in-state-Nil)

    show q2 ∈ f ‘ (ℱ 𝒜1)
      unfolding q2-eq
      using q1-in-F
      by auto
  qed
next
  show Δ 𝒜2 = {(f s1, a, f s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜1}
  proof (intro set-eqI)
    fix q2aq2' :: 'q2 × 'a × 'q2
    obtain q2 a q2' where q2aq2'-eq [simp]: q2aq2' = (q2, a, q2')
      by (cases q2aq2', blast)

    show q2aq2' ∈ Δ 𝒜2 ⟷ q2aq2' ∈ {(f s1, a, f s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜1}
    proof (cases q2 ∈ 𝒬 𝒜2 ∧ a ∈ Σ 𝒜2 ∧ q2' ∈ 𝒬 𝒜2)
      case False note q2aq2'-not-wf = this

      from q2aq2'-not-wf have q2aq2'-nin-ls: q2aq2' ∉ Δ 𝒜2
        using DFA2.Δ-consistent
        by auto

      have q2aq2'-nin-rs: q2aq2' ∉ {(f s1, a, f s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜1}
      proof (rule notI)
        assume q2aq2' ∈ {(f s1, a, f s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜1}
        then obtain q1 q1' where
          q2-eq:  q2 = f q1 and
          q2'-eq: q2' = f q1' and
          q1aq1'-in: (q1, a, q1') ∈ Δ 𝒜1
          by auto
        with DFA1.Δ-consistent f-is-state q2aq2'-not-wf Σ-eq
        show False by metis
      qed
      from q2aq2'-nin-rs q2aq2'-nin-ls show ?thesis by simp
    next
      case True
      hence q2-in: q2 ∈ 𝒬 𝒜2
        and a-in2: a ∈ Σ 𝒜2
        and q2'-in: q2' ∈ 𝒬 𝒜2
      by simp-all
      from a-in2 Σ-eq have a-in1: a ∈ Σ 𝒜1 by fast

      from bij-f q2-in
        obtain q1 where q1-in: q1 ∈ 𝒬 𝒜1
                  and q2-eq: q2 = f q1
        unfolding bij-betw-def by auto

      from DFA1.DFA-δ-is-some [OF q1-in a-in1]
      obtain q1'' where δ-q1a: (δ 𝒜1) (q1, a) = Some q1'' by auto
      from DFA1.δ-wf[OF δ-q1a] have q1''-in: q1'' ∈ 𝒬 𝒜1 by fast

      from DFA2.DFA-δ-is-some [OF q2-in a-in2]
      obtain q2'' where δ-q2a: (δ 𝒜2) (q2, a) = Some q2'' by auto
      from DFA2.δ-wf[OF δ-q2a] have q2''-in: q2'' ∈ 𝒬 𝒜2 by fast

      from ℒ-in-state---DFA---eq-reachable-step [OF wf-A1 wf-A2,
        of q1 a q1'' q2 q2''] δ-q1a δ-q2a f-equiv[OF q1-in]
      have q1''-q2''-equiv: ℒ-in-state 𝒜1 q1'' = ℒ-in-state 𝒜2 q2''
        by (simp add: q2-eq DFA1.δ-in-Δ-iff [symmetric]
                  DFA2.δ-in-Δ-iff [symmetric])
```

```
      from q1''-q2''-equiv f-equiv[OF q1''-in]
      have ℒ-in-state 𝒜2 q2'' = ℒ-in-state 𝒜2 (f q1'') by simp
      hence q2''-eq: q2'' = f q1''
        using min-A2 q2''-in f-is-state[OF q1''-in]
        unfolding DFA-is-minimal-alt-DFA-def inj-on-def
        by simp

      from DFA1.δ-wf inj-f δ-q1a
      have q2aq2'-in-rs-eq: q2aq2' ∈ {(f s1, a, f s2) |s1 a s2. (s1, a, s2) ∈ Δ 𝒜1} ⟷ q2' = f q1''
         unfolding inj-on-def
         by (simp add: q2-eq DFA1.δ-in-Δ-iff)
            (metis option.inject)
      show ?thesis
        unfolding q2aq2'-in-rs-eq
        by (simp add: q2''-eq DFA2.δ-in-Δ-iff δ-q2a) blast
    qed
  qed
qed

  from wf-A1 inj-f A2-eq
  show ?thesis
    unfolding NFA-isomorphic-wf-def NFA-isomorphic-def
    apply (simp add: DFA-alt-def)
    apply (rule exI [where x = f])
    apply simp
  done
qed
```

3.10 Brzozowski's Algorithm

Brzozowski's algorithm for minimisation applies powerset construction to the reverse and repeats this to obtain a minimal automaton.

```
definition Brzozowski-halfway
where Brzozowski-halfway 𝒜 ≡ efficient-determinise-NFA (NFA-reverse 𝒜)

definition Brzozowski
where Brzozowski 𝒜 ≡ Brzozowski-halfway (Brzozowski-halfway 𝒜)

lemma Brzozowski-halfway---well-formed :
  NFA 𝒜 ⟹ NFA (Brzozowski-halfway 𝒜)
unfolding Brzozowski-halfway-def efficient-determinise-NFA-def
by (metis determinise-NFA---is-well-formed NFA-reverse---is-well-formed
          NFA-remove-unreachable-states---is-well-formed)

lemma Brzozowski---well-formed :
  NFA 𝒜 ⟹ NFA (Brzozowski 𝒜)
unfolding Brzozowski-def
by (metis Brzozowski-halfway---well-formed)

lemma Brzozowski-Σ [simp] :
  Σ (Brzozowski 𝒜) = Σ 𝒜
unfolding Brzozowski-def Brzozowski-halfway-def efficient-determinise-NFA-def
by simp

lemma Brzozowski-halfway-yields-DFA :
  assumes NFA 𝒜
  shows DFA (Brzozowski-halfway 𝒜)
unfolding DFA-alt-def
using assms
by (simp add: Brzozowski-halfway---well-formed,
    simp add: NFA-is-deterministic---NFA-remove-unreachable-states
```

efficient-determinise-NFA-def
Brzozowski-halfway-def det.deterministic-NFA)

lemma *Brzozowski-yields-DFA* :
assumes *NFA* $\mathcal{A}$
shows *DFA* (*Brzozowski* $\mathcal{A}$)
unfolding *Brzozowski-def*
using *assms*
by (*metis DFA-alt-def Brzozowski-halfway-yields-DFA*)

lemma (**in** *NFA*) *Brzozowski-halfway---*$\mathcal{L}$:
shows $\mathcal{L}$ (*Brzozowski-halfway* $\mathcal{A}$) = {*rev w* |*w*. *w* ∈ $\mathcal{L}$ $\mathcal{A}$}
using *NFA-reverse---*$\mathcal{L}$ *NFA.determinise-NFA-*$\mathcal{L}$ [*OF NFA-reverse---is-well-formed* [*OF wf-NFA*]]
by (*simp add*: *Brzozowski-halfway-def efficient-determinise-NFA-def*)

Finally we show that Brzozowski's algorithm preserves the language of the automaton.

theorem (**in** *NFA*) *Brzozowski---*$\mathcal{L}$:
$\mathcal{L}$ (*Brzozowski* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
unfolding *Brzozowski-def*
using *Brzozowski-halfway---*$\mathcal{L}$
using *NFA.Brzozowski-halfway---*$\mathcal{L}$ [*OF Brzozowski-halfway---well-formed* [*OF wf-NFA*]]
by (*simp add*: *set-eq-iff del*: *ex-simps add*: *ex-simps*[*symmetric*])

theorem (**in** *NFA*) *Brzozowski---NFA-accept* :
NFA-accept (*Brzozowski* $\mathcal{A}$) *w* = *NFA-accept* $\mathcal{A}$ *w*
using *Brzozowski---*$\mathcal{L}$
by (*simp add*: $\mathcal{L}$*-def set-eq-iff*)

lemma (**in** *DFA*) *Brzozowski-halfway---minimal* :
assumes *connected*: *NFA-is-initially-connected* $\mathcal{A}$
shows *DFA-is-minimal* (*Brzozowski-halfway* $\mathcal{A}$)
proof −
let *?BH* = *Brzozowski-halfway* $\mathcal{A}$
interpret *DFA-BH*: *DFA ?BH*
using *Brzozowski-halfway-yields-DFA* [*OF wf-NFA*] **.**

show *DFA-is-minimal ?BH*
proof (*rule DFA-BH.DFA-is-minimal-intro*)
show *NFA-is-initially-connected ?BH*
by (*simp add*: *Brzozowski-halfway-def efficient-determinise-NFA-def*
NFA-remove-unreachable-states---NFA-is-initially-connected)
next
let *?DR* = *determinise-NFA* (*NFA-reverse* $\mathcal{A}$)

have *wf-reverse-A*: *NFA* (*NFA-reverse* $\mathcal{A}$)
using *NFA-reverse---is-well-formed* [*OF wf-NFA*] **.**

have $\bigwedge$*Q1 Q2*. ⟦*Q1* ∈ $\mathcal{Q}$ *?DR*; *Q2* ∈ $\mathcal{Q}$ *?DR*;
$\mathcal{L}$*-in-state ?DR Q1* = $\mathcal{L}$*-in-state ?DR Q2*⟧ ⟹ (*Q1* = *Q2*)
proof −
fix *Q1 Q2*
assume *Q1* ∈ $\mathcal{Q}$ *?DR* **hence** *Q1-Q* : *Q1* ⊆ $\mathcal{Q}$ $\mathcal{A}$ **by** *simp*
assume *Q2* ∈ $\mathcal{Q}$ *?DR* **hence** *Q2-Q* : *Q2* ⊆ $\mathcal{Q}$ $\mathcal{A}$ **by** *simp*
assume $\mathcal{L}$*-eq* : $\mathcal{L}$*-in-state ?DR Q1* = $\mathcal{L}$*-in-state ?DR Q2*

have $\mathcal{L}$*-left-not-empty* :
$\bigwedge$*q*. *q* ∈ *Q1* ∪ *Q2* ⟹ ∃ *w*. *w* ∈ $\mathcal{L}$*-left* $\mathcal{A}$ *q*
proof −
fix *q*
assume *q* ∈ *Q1* ∪ *Q2*
with *Q1-Q Q2-Q* **have** *q* ∈ $\mathcal{Q}$ $\mathcal{A}$ **by** *auto*

with *connected* **have** $q \notin$ *NFA-unreachable-states* $\mathcal{A}$ **unfolding** *NFA-is-initially-connected-def* **by** *auto*
hence $\mathcal{L}$*-left* $\mathcal{A}$ $q \neq \{\}$ **by** (*simp add*: *NFA-unreachable-states-alt-def*)
thus $\exists\, w.\ w \in \mathcal{L}$*-left* $\mathcal{A}$ q
by (*metis ex-in-conv*)
qed

note *DFA-left-languages---pairwise-disjoint*
with *Q1-Q Q2-Q* **have** $\mathcal{L}$*-left-unique*: $\bigwedge q1\ q2\ w.\ [\![q1 \in Q1;\ q2 \in Q2;$
$w \in \mathcal{L}$*-left* $\mathcal{A}\ q1;\ w \in \mathcal{L}$*-left* $\mathcal{A}\ q2]\!] \Longrightarrow (q1 = q2)$
by (*simp add*: *set-eq-iff subset-iff*, *metis*)

obtain *QQ* **where** *QQ-def*: $QQ = (\lambda Q.\ \bigcup\{\{rev\ w\ |w.\ w \in \mathcal{L}\text{-}left\ \mathcal{A}\ q\}\ |q.\ q \in Q\})$ **by** *auto*
have *QQ-in*: $\bigwedge w\ Q.\ rev\ w \in QQ\ Q = (\exists\, q \in Q.\ w \in \mathcal{L}\text{-}left\ \mathcal{A}\ q)$
by (*auto simp add*: *QQ-def*)

note *NFA.determinise-NFA---*$\mathcal{L}$*-in-state* [*OF wf-reverse-A*]
with *Q1-Q Q2-Q* $\mathcal{L}$*-eq* **have** $QQ\ Q1 = QQ\ Q2$
by (*simp add*: *NFA-reverse---*$\mathcal{L}$*-in-state QQ-def*)
hence $\bigwedge w.\ (rev\ w \in QQ\ Q1) = (rev\ w \in QQ\ Q2)$ **by** *simp*
hence $\bigwedge w.\ (\exists\, q \in Q1.\ w \in \mathcal{L}\text{-}left\ \mathcal{A}\ q) \longleftrightarrow (\exists\, q \in Q2.\ w \in \mathcal{L}\text{-}left\ \mathcal{A}\ q)$
by (*simp add*: *QQ-in*)
with $\mathcal{L}$*-left-unique* $\mathcal{L}$*-left-not-empty* **have** $\bigwedge q.\ q \in Q1 \longleftrightarrow q \in Q2$
by (*simp*, *metis*)
thus $Q1 = Q2$ **by** *auto*
qed
hence $\bigwedge q1\ q2.\ [\![q1 \in \mathcal{Q}\ ?BH;\ q2 \in \mathcal{Q}\ ?BH;$
$\mathcal{L}\text{-}in\text{-}state\ ?BH\ q1 = \mathcal{L}\text{-}in\text{-}state\ ?BH\ q2]\!] \Longrightarrow (q1 = q2)$
apply (*simp add*: *Brzozowski-halfway-def efficient-determinise-NFA-def*)
apply (*simp add*: *NFA-remove-unreachable-states-def*)
done

thus *inj-on* ($\mathcal{L}$*-in-state ?BH*) ($\mathcal{Q}$ *?BH*)
by (*simp add*: *inj-on-def*)
qed
qed

theorem (**in** *NFA*) *Brzozowski---minimal* :
DFA-is-minimal (*Brzozowski* $\mathcal{A}$)
unfolding *Brzozowski-def*
proof (*rule DFA.Brzozowski-halfway---minimal*)
let *?BH* = *Brzozowski-halfway* $\mathcal{A}$

show *DFA ?BH*
using *Brzozowski-halfway-yields-DFA* [*OF wf-NFA*] **.**

show *NFA-is-initially-connected ?BH*
by (*simp add*: *Brzozowski-halfway-def efficient-determinise-NFA-def*
NFA-remove-unreachable-states---NFA-is-initially-connected)
qed

lemma *NFA-isomorphic-Brzozowski-halfway* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*Brzozowski-halfway* $\mathcal{A}1$) (*Brzozowski-halfway* $\mathcal{A}2$)
proof −
note *equiv-1* = *NFA-isomorphic-wf---NFA-reverse-cong* [*OF equiv*]
note *equiv-2* = *NFA-isomorphic-wf---NFA-determinise-cong* [*OF equiv-1*]
note *equiv-3* = *NFA-isomorphic-wf---NFA-remove-unreachable-states* [*OF equiv-2*]
thus *?thesis* **unfolding** *Brzozowski-halfway-def efficient-determinise-NFA-def* **.**
qed

lemma *NFA-isomorphic-Brzozowski* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$

shows *NFA-isomorphic-wf* (*Brzozowski* $\mathcal{A}1$) (*Brzozowski* $\mathcal{A}2$)
proof −
  note *equiv-1* = *NFA-isomorphic-Brzozowski-halfway* [*OF equiv*]
  note *equiv-2* = *NFA-isomorphic-Brzozowski-halfway* [*OF equiv-1*]
  thus *?thesis* **unfolding** *Brzozowski-def* **.**
qed

3.11 Abstract Minimisation Function

Above, it is shown that all minimal automata that accept the same language are isomorphic. Let's use Brzozwski minisation and this property in order to define a minisation function that is later used for specification.

consts *NFA-minimise* :: ('*q*::{*NFA-states*}, '*a*, '*X*) *NFA-rec-scheme* ⇒ ('*q*, '*a*) *NFA-rec*

specification (*NFA-minimise*) *NFA-minimise-spec-aux*:
∀$\mathcal{A}$. *NFA* $\mathcal{A}$ ⟶
  $\mathcal{L}$(*NFA-minimise* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$ ∧
  Σ(*NFA-minimise* $\mathcal{A}$) = Σ $\mathcal{A}$ ∧
  DFA-is-minimal (*NFA-minimise* $\mathcal{A}$)
apply (*rule exI* [**where** *x* = λ$\mathcal{A}$. *NFA-normalise-states* (*Brzozowski* $\mathcal{A}$)])
apply (*simp add*: *Brzozowski---well-formed NFA.Brzozowski---*$\mathcal{L}$)
apply (*metis Brzozowski---well-formed NFA.Brzozowski---minimal NFA-isomorphic-wf-D*(*3*)
    NFA-isomorphic-wf-normalise-states NFA-isomorphic---DFA-is-minimal)
done

lemma *NFA-minimise-spec*:
  NFA $\mathcal{A}$ ⟹ $\mathcal{L}$(*NFA-minimise* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
  NFA $\mathcal{A}$ ⟹ Σ(*NFA-minimise* $\mathcal{A}$) = Σ $\mathcal{A}$
  NFA $\mathcal{A}$ ⟹ *DFA-is-minimal* (*NFA-minimise* $\mathcal{A}$)
by (*simp-all add*: *NFA-minimise-spec-aux*)

lemma *NFA-isomorphic-wf---minimise* :
fixes $\mathcal{A}1$ :: ('*q1*::{*NFA-states*}, '*a*, -) *NFA-rec-scheme*
  and $\mathcal{A}2$ :: ('*q2*, '*a*) *NFA-rec*
assumes *wf-*$\mathcal{A}1$: *NFA* $\mathcal{A}1$
shows *NFA-isomorphic-wf* $\mathcal{A}2$ (*NFA-minimise* $\mathcal{A}1$) ⟷
    $\mathcal{L}$ $\mathcal{A}2$ = $\mathcal{L}$ $\mathcal{A}1$ ∧ Σ $\mathcal{A}2$ = Σ $\mathcal{A}1$ ∧ *DFA-is-minimal* $\mathcal{A}2$
using *NFA-minimise-spec*[*OF wf-*$\mathcal{A}1$]
   DFA-is-minimal---isomorph-wf [*of* $\mathcal{A}2$ *NFA-minimise* $\mathcal{A}1$]
   NFA-isomorphic---DFA-is-minimal [*of NFA-minimise* $\mathcal{A}1$ $\mathcal{A}2$]
by (*simp*, *metis NFA-isomorphic-wf-D*(*4*) *NFA-isomorphic-wf-*$\mathcal{L}$ *NFA-isomorphic-wf-*Σ)

lemma *NFA-isomorphic-wf---minimise-cong* :
assumes *pre*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*NFA-minimise* $\mathcal{A}1$) (*NFA-minimise* $\mathcal{A}2$)
proof −
  from *pre* **have** *NFA* $\mathcal{A}1$ *NFA* $\mathcal{A}2$
   by (*simp-all add*: *NFA-isomorphic-wf-alt-def*)

  from ⟨*NFA* $\mathcal{A}2$⟩ *NFA-minimise-spec*[*OF* ⟨*NFA* $\mathcal{A}1$⟩] *pre*
  show *?thesis*
   by (*simp add*: *NFA-isomorphic-wf---minimise NFA-isomorphic-wf-*$\mathcal{L}$ *NFA-isomorphic-wf-*Σ)
qed

lemma *Brzozowski---minimise* :
  NFA $\mathcal{A}$ ⟹ *NFA-isomorphic-wf* (*Brzozowski* $\mathcal{A}$) (*NFA-minimise* $\mathcal{A}$)
by (*simp add*: *NFA-isomorphic-wf---minimise NFA.Brzozowski---*$\mathcal{L}$ *NFA.Brzozowski---minimal*)

end

4 Hopcroft's Minimisation Algorithm

theory *Hopcroft-Minimisation*
imports *Main DFA Collections*
begin

In this theory, Hopcroft's minimisation algorithm [see Hopcroft, J.E.: An n log n algorithm for minimizing the states in a finite automaton. In Kohavi, Z. (ed.) The Theory of Machines and Computations. Academic Press 189–196 (1971)] is verified.

4.1 Main idea

A determinsitic automaton with no unreachable states can be minimised by merging equivalent states.

lemma *merge-is-minimal* :
assumes *wf-A*: *DFA* $\mathcal{A}$
  and *connected*: *NFA-is-initially-connected* $\mathcal{A}$
  and *equiv*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ *f*
shows *DFA-is-minimal* (*NFA-rename-states* $\mathcal{A}$ *f*)
 (**is** *DFA-is-minimal ?*$\mathcal{A}$*-min*)
proof (*rule DFA.DFA-is-minimal-intro*)
 show *DFA ?*$\mathcal{A}$*-min*
  by (*fact DFA---combine-equiv-states*[*OF wf-A equiv*])
next
 have $\bigwedge q'.\ q' \in \mathcal{Q}\ \mathcal{A} \Longrightarrow (\exists\ q \in \mathcal{I}\ \mathcal{A}.\ \exists\, w \in \mathit{lists}\ (\Sigma\ \mathcal{A}).$
    LTS-is-reachable (Δ (*NFA-rename-states* $\mathcal{A}$ *f*)) (*f q*) *w* (*f q'*))
 proof –
  fix *q'*
  assume *q'-in-Q*: $q' \in \mathcal{Q}\ \mathcal{A}$
  with *connected* **obtain** *q w* **where**
   q-in: $q \in \mathcal{I}\ \mathcal{A}$ **and**
   w-in: $w \in \mathit{lists}\ (\Sigma\ \mathcal{A})$ **and**
   reach: *LTS-is-reachable* (Δ $\mathcal{A}$) *q w q'*
  unfolding *NFA-is-initially-connected-alt-def*
  by (*simp add*: *Bex-def Ball-def*) *metis*

  from *LTS-is-reachable---NFA-rename-statesE*[*OF reach*, *of f*] *q-in w-in*
  show *?thesis q'* **by** *blast*
 qed

 thus *NFA-is-initially-connected ?*$\mathcal{A}$*-min*
  unfolding *NFA-is-initially-connected-alt-def*
  by *simp*
next
 have *wf-NFA* : *NFA* $\mathcal{A}$ **using** *wf-A* **unfolding** *DFA-alt-def* **by** *simp*
 from *NFA.*$\mathcal{L}$*-in-state-rename-iff* [*OF wf-NFA*
  NFA-is-strong-equivalence-rename-fun---weaken [*OF equiv*]]
 show *inj-on* ($\mathcal{L}$*-in-state ?*$\mathcal{A}$*-min*) ($\mathcal{Q}$ *?*$\mathcal{A}$*-min*)
  using *equiv*
  unfolding *NFA-is-strong-equivalence-rename-fun-def inj-on-def*
  by *simp*
qed

lemma *merge-NFA-minimise* :
assumes *wf-A*: *DFA* $\mathcal{A}$
  and *connected*: *NFA-is-initially-connected* $\mathcal{A}$
  and *equiv*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ *f*
shows *NFA-isomorphic-wf* (*NFA-rename-states* $\mathcal{A}$ *f*) (*NFA-minimise* $\mathcal{A}$)
proof –
 from *wf-A* **have** *wf-A'*: *NFA* $\mathcal{A}$ **by** (*simp add*: *DFA-alt-def*)
 note *min* = *merge-is-minimal*[*OF wf-A connected equiv*]

show *?thesis*
apply (*simp add*: *NFA-isomorphic-wf---minimise* [*OF wf-A'*] *min*)
apply (*rule NFA.$\mathcal{L}$-rename-iff* [*OF wf-A'*])
apply (*rule NFA-is-strong-equivalence-rename-fun---weaken*)
apply (*rule equiv*)
done
qed

This allows to define a high level, non-executable version of an minimisation algorithm. These definitions and lemmata are later used as an abstract interface to an executable implementation.

definition *Hopcroft-minimise* :: $('q, 'a, 'x)$ *NFA-rec-scheme* $\Rightarrow$ $('q, 'a)$ *NFA-rec* **where**
Hopcroft-minimise $\mathcal{A} \equiv$ *NFA-rename-states* $\mathcal{A}$ (*SOME f*.
NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ f $\wedge$ $(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f\ q \in \mathcal{Q}\ \mathcal{A})$)

lemma *Hopcroft-minimise-correct* :
fixes $\mathcal{A}$ :: $('q, 'a, 'x)$ *NFA-rec-scheme*
assumes *wf-A*: *DFA* $\mathcal{A}$
and *connected*: *NFA-is-initially-connected* $\mathcal{A}$
shows *DFA-is-minimal* (*Hopcroft-minimise* $\mathcal{A}$) $\mathcal{L}$ (*Hopcroft-minimise* $\mathcal{A}$) $= \mathcal{L}\ \mathcal{A}$
proof −
def $f \equiv$ (*SOME f*. *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f $\wedge$ $(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f\ q \in \mathcal{Q}\ \mathcal{A})$)

from *Hilbert-Choice.someI-ex* [**where** *?P* $= \lambda f$. *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f $\wedge$
$(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f\ q \in \mathcal{Q}\ \mathcal{A})$,
OF NFA-is-strong-equivalence-rename-fun-exists]
have *f-OK*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f **and** *f-subset*: $\bigwedge q.\ q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow f\ q \in \mathcal{Q}\ \mathcal{A}$
by (*simp-all add*: *f-def*)

from *wf-A* **have** *wf-NFA*: *NFA* $\mathcal{A}$ **unfolding** *DFA-alt-def* **by** *simp*

from *NFA.$\mathcal{L}$-rename-iff* [*OF wf-NFA NFA-is-strong-equivalence-rename-fun---weaken*[*OF f-OK*]]
show $\mathcal{L}$ (*Hopcroft-minimise* $\mathcal{A}$) $= \mathcal{L}\ \mathcal{A}$
unfolding *Hopcroft-minimise-def f-def*[*symmetric*]
by *simp*

from *merge-is-minimal* [*OF wf-A connected f-OK*]
show *DFA-is-minimal* (*Hopcroft-minimise* $\mathcal{A}$)
unfolding *Hopcroft-minimise-def f-def*[*symmetric*]
by *simp*
qed

lemma *Hopcroft-minimise-$\mathcal{Q}$* :
fixes $\mathcal{A}$ :: $('q, 'a, 'x)$ *NFA-rec-scheme*
shows $\mathcal{Q}$ (*Hopcroft-minimise* $\mathcal{A}$) $\subseteq \mathcal{Q}\ \mathcal{A}$
proof −
def $f \equiv$ (*SOME f*. *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f $\wedge$ $(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f\ q \in \mathcal{Q}\ \mathcal{A})$)

from *Hilbert-Choice.someI-ex* [**where** *?P* $= \lambda f$. *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f $\wedge$
$(\forall q \in \mathcal{Q}\ \mathcal{A}.\ f\ q \in \mathcal{Q}\ \mathcal{A})$,
OF NFA-is-strong-equivalence-rename-fun-exists]
have *f-OK*: *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f **and** *f-subset*: $\bigwedge q.\ q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow f\ q \in \mathcal{Q}\ \mathcal{A}$
by (*simp-all add*: *f-def*)

show $\mathcal{Q}$ (*Hopcroft-minimise* $\mathcal{A}$) $\subseteq \mathcal{Q}\ \mathcal{A}$
unfolding *Hopcroft-minimise-def f-def*[*symmetric*]
using *f-subset*
by (*simp add*: *subset-iff image-iff*) *metis*
qed

definition *Hopcroft-minimise-NFA* **where**
Hopcroft-minimise-NFA $\mathcal{A}$ = *Hopcroft-minimise* (*NFA-determinise* $\mathcal{A}$)

lemma *Hopcroft-minimise-NFA-correct* :
fixes $\mathcal{A}$ :: (*'q*::{*NFA-states*}, *'a*, *'x*) *NFA-rec-scheme*
assumes *wf-A*: *NFA* $\mathcal{A}$
shows *DFA-is-minimal* (*Hopcroft-minimise-NFA* $\mathcal{A}$) $\mathcal{L}$ (*Hopcroft-minimise-NFA* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
proof −
  let *?A'* = *NFA-determinise* $\mathcal{A}$

  from *NFA-determinise-is-DFA* [*OF wf-A*]
  have *wf-A'*: *DFA ?A'* **.**

  have *connected*: *NFA-is-initially-connected ?A'*
    unfolding *NFA-determinise-def efficient-determinise-NFA-def*
    apply (*rule NFA-is-initially-connected---normalise-states*)
    apply (*simp add*: *NFA-remove-unreachable-states---NFA-is-initially-connected*)
  done
  note *min-correct* = *Hopcroft-minimise-correct* [*OF wf-A' connected*]

  from *min-correct*(*1*) **show** *DFA-is-minimal* (*Hopcroft-minimise-NFA* $\mathcal{A}$)
    unfolding *Hopcroft-minimise-NFA-def* **.**

  from *min-correct*(*2*) *NFA-determinise-*$\mathcal{L}$[*OF wf-A*]
  show $\mathcal{L}$ (*Hopcroft-minimise-NFA* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
    unfolding *Hopcroft-minimise-NFA-def*
    by *auto*
qed

Now, we can consider the essence of Hopcroft's algorithm: finding a suitable renaming function. Hopcroft's algorithm computes the Myhill-Nerode equivalence relation in form of a partition. From this partition, a renaming function can be easily derived.

4.2 Basic notions

Before considering Hopcrofts algorithm, some basic notions need to be introduced.

4.2.1 Partitions

definition *is-partition* :: *'a set* $\Rightarrow$ (*'a set*) *set* $\Rightarrow$ *bool* **where**
  is-partition Q P $\longleftrightarrow$
    $(\bigcup P = Q) \wedge (\{\} \notin P) \wedge (\forall p1\ p2.\ p1 \in P \wedge p2 \in P \wedge p1 \neq p2 \longrightarrow p1 \cap p2 = \{\})$

lemma *is-partitionI* [*intro*!]:
  $[\![\bigwedge q.\ q \in Q \Longrightarrow q \in \bigcup P;$
    $\bigwedge p.\ p \in P \Longrightarrow p \subseteq Q;$
    $\bigwedge p.\ p \in P \Longrightarrow p \neq \{\};$
    $\bigwedge q\ p1\ p2.\ [\![p1 \in P;\ p2 \in P;\ q \in p1;\ q \in p2]\!] \Longrightarrow p1 = p2$
  $]\!] \Longrightarrow$ *is-partition Q P*
unfolding *is-partition-def*
by *auto metis+*

lemma *is-partition-Nil-Q* :
  is-partition {} *P* $\longleftrightarrow$ *P* = {}
unfolding *is-partition-def*
by *auto*

lemma *is-partition-Nil* [*simp*] :
  is-partition Q {} $\longleftrightarrow$ *Q* = {}
unfolding *is-partition-def*
by *auto*

lemma *is-partition-Insert* :

assumes *p-nin*: $p \notin P$
shows *is-partition* Q (*insert* p P) $\longleftrightarrow$
    $(p \neq \{\}) \wedge (p \subseteq Q) \wedge$ *is-partition* $(Q - p)$ P
proof (*cases* $p \neq \{\} \wedge (\{\} \notin P) \wedge p \subseteq Q$)
  case *False* **thus** *?thesis*
    unfolding *is-partition-def*
    by (*rule-tac iffI*, *auto*)
next
  case *True*
  note *p-not-Emp* = *conjunct1* [*OF True*]
  note *P-not-Emp* = *conjunct1* [*OF conjunct2* [*OF True*]]
  note *p-subset* = *conjunct2* [*OF conjunct2* [*OF True*]]

  have $(p \cup \bigcup P = Q \wedge$
      $(\forall p1\ p2.\ (p1 = p \vee p1 \in P) \wedge (p2 = p \vee p2 \in P) \wedge p1 \neq p2 \longrightarrow p1 \cap p2 = \{\})) =$
      $(\bigcup P = Q - p \wedge (\forall p1\ p2.\ p1 \in P \wedge p2 \in P \wedge p1 \neq p2 \longrightarrow p1 \cap p2 = \{\}))$
    (**is** $?l1 \wedge ?l2 \longleftrightarrow ?r1 \wedge ?r2$)
  proof (*intro iffI conjI*)
    assume *r12*: $?r1 \wedge ?r2$
    note *r1* = *conjunct1* [*OF r12*]
    note *r2* = *conjunct2* [*OF r12*]

    from *r1* **show** *?l1* **using** *p-subset* **by** *auto*

    have *p-disjoint* : $\bigwedge p'.\ p' \in P \Longrightarrow p \cap p' = \{\}$
    proof −
      fix p'
      assume $p' \in P$
      with *r1* **have** $p' \subseteq Q - p$
        by *auto*
      thus $p \cap p' = \{\}$ **by** *auto*
    qed

    with *r2* **show** *?l2* **by** *blast*
  next
    assume *l12*: $?l1 \wedge ?l2$
    note *l1* = *conjunct1* [*OF l12*]
    note *l2* = *conjunct2* [*OF l12*]

    from *l2* **show** *?r2* **by** *blast*

    from *l2 p-nin*
    have *p-disjoint* : $\bigwedge p'.\ p' \in P \Longrightarrow p \cap p' = \{\}$ **by** *blast*
    with *l1* **show** *?r1* **by** *auto*
  qed
  with *p-not-Emp P-not-Emp p-subset* **show** *?thesis*
    unfolding *is-partition-def*
    by *simp*
qed

lemma *is-partition-in-subset* :
assumes *is-part*: *is-partition* Q P
    and *p-in*: $p \in P$
shows $p \subseteq Q$
using *assms*
unfolding *is-partition-def*
by *auto*

lemma *is-partition-memb-finite* :
assumes *fin-Q*: *finite* Q
    and *is-part*: *is-partition* Q P
    and *p-in*: $p \in P$

```
shows finite p
by (metis finite-subset[OF - fin-Q] is-partition-in-subset[OF is-part p-in])

lemma is-partition-distinct-subset :
assumes is-part: is-partition Q P
    and p-in: p ∈ P
    and p'-sub: p' ⊆ p
shows p' ∉ P − {p}
proof (rule notI)
  assume p'-in: p' ∈ P − {p}

  with is-part have p' ≠ {}
    unfolding is-partition-def by auto
  hence p' ∩ p ≠ {} using p'-sub by auto

  with is-part show False
    using p'-in p-in
    unfolding is-partition-def
    by blast
qed

lemma is-partition-finite :
assumes fin-Q: finite Q
    and is-part: is-partition Q P
shows finite P
proof −
  from is-part have P ⊆ Pow Q
    unfolding is-partition-def by auto
  moreover
  from fin-Q have finite (Pow Q) by blast
  ultimately show finite P
    by (metis finite-subset)
qed

lemma is-partition-card :
assumes fin-Q: finite Q
    and is-part: is-partition Q P
shows card Q = setsum card P
proof −
  have fin-P: finite P
    using is-partition-finite [OF fin-Q is-part]
    .

  have finite P ⟹ finite Q ⟹ is-partition Q P ⟹ card Q = setsum card P
  proof (induct arbitrary: Q rule: finite-induct)
    case empty thus ?case by simp
  next
    case (insert p P Q)
    note fin-P = insert(1)
    note p-nin-P = insert(2)
    note ind-hyp = insert(3)
    note fin-Q = insert(4)
    note is-part-pP = insert(5)

    from is-part-pP p-nin-P have is-part-P: is-partition (Q − p) P
      and p-neq-Nil: p ≠ {} and p-subset: p ⊆ Q
      by (simp-all add: is-partition-Insert)

    from p-subset fin-Q have fin-p: finite p by (metis finite-subset)
    from fin-Q have fin-Q-p: finite (Q − p) by simp
    from p-subset have card-p: card p ≤ card Q
      using card-mono [OF fin-Q] by simp
```

```
    show ?case
      using setsum-insert [OF fin-P p-nin-P, of card]
            card-Diff-subset [OF fin-p p-subset]
            ind-hyp[OF fin-Q-p is-part-P, symmetric]
            card-p
      by simp
  qed
  with fin-Q fin-P is-part
  show ?thesis by simp
qed

lemma is-partition-card-P :
assumes fin-Q: finite Q
    and is-part: is-partition Q P
shows card P ≤ card Q
proof −
  from is-partition-card [OF fin-Q is-part]
  have card Q = setsum card P .
  moreover
  have setsum (λp. 1) P ≤ setsum card P
  proof (rule setsum-mono)
    fix p
    assume p ∈ P
    with is-part have p-neq-Nil: p ≠ {} and p-sub: p ⊆ Q
      unfolding is-partition-def
      by auto
    from p-sub fin-Q have finite p by (metis finite-subset)
    with p-neq-Nil have card p ≠ 0
      by (simp add: card-eq-0-iff)
    thus 1 ≤ card p by auto
  qed
  moreover
  have setsum (λp. 1) P = card P
    by (metis card-eq-setsum)
  ultimately show ?thesis by simp
qed


lemma is-partition-find :
assumes is-part: is-partition Q P
    and q-in-Q: q ∈ Q
shows ∃!p. p ∈ P ∧ q ∈ p
using assms
unfolding is-partition-def
by auto

lemma is-partition-refine :
assumes fin-Q:          finite Q
    and p-nin:          p ∉ P
    and is-part:        is-partition Q (insert p P)
    and p1-p2-neq-emp: p1 ≠ {} p2 ≠ {}
    and p1-p2-disj: p1 ∩ p2 = {}
    and p1-p2-union: p = p1 ∪ p2
shows is-partition Q (insert p1 (insert p2 P)) ∧
       card (insert p1 (insert p2 P)) = Suc (card (insert p P))
proof
  from p1-p2-neq-emp p1-p2-disj
  have p1-neq-p2: p1 ≠ p2 by auto

  have q-disj: ⋀p'. p' ⊆ p ⟹ p' ∉ P
  proof (rule notI)
```

fix p'
assume p'-*sub*: $p' \subseteq p$
assume p'-*in*: $p' \in P$

from *is-part* p'-*in* *p-nin* **have** $p' \cap p = \{\} \wedge p' \neq \{\}$
unfolding *is-partition-def* **by** *blast*
with p'-*sub* **show** *False* **by** *auto*
qed
hence *p1-nin*: $p1 \notin$ *insert p2 P* **and**
p2-nin: $p2 \notin P$
unfolding *p1-p2-union* **using** *p1-neq-p2* **by** *auto*

have $Q - p = Q - p1 - p2$
unfolding *p1-p2-union* **by** *auto*
with *is-part p1-p2-disj*
show *is-partition Q* (*insert p1* (*insert p2 P*))
using *p-nin p1-nin p2-nin*
apply (*simp add*: *is-partition-Insert p1-p2-neq-emp*)
apply (*simp add*: *p1-p2-union*)
apply *auto*
done

have *finite P* **using** *is-partition-finite*[*OF fin-Q is-part*] **by** *simp*
thus *card* (*insert p1* (*insert p2 P*)) = *Suc* (*card* (*insert p P*))
using *p-nin p2-nin p1-nin*
by (*simp add*: *card-insert-if*)
qed

4.2.2 Partitions and Equivalence Relations

lemma *quotient-of-equiv-relation-is-partition* :
assumes *eq-r*: *equiv Q r*
shows *is-partition Q* ($Q//r$)
proof −
note *Q-eq* = *Union-quotient*[*OF eq-r*]
note *quot-disj* = *quotient-disj* [*OF eq-r*]

from *eq-r* **have** $\bigwedge q.\ q \in Q \Longrightarrow (q, q) \in r$
unfolding *equiv-def refl-on-def* **by** *simp*
hence *emp-nin*: $\{\} \notin Q \mathbin{//} r$
by (*simp add*: *quotient-def Image-def*) *blast*

from *Q-eq quot-disj emp-nin*
show *?thesis* **by** *blast*
qed

definition *relation-of-partition* **where**
relation-of-partition $P = \{(q1, q2).\ \exists\, Q \in P.\ q1 \in Q \wedge q2 \in Q\}$

lemma *relation-of-partition-is-equiv* :
assumes *part-P*: *is-partition Q P*
shows *equiv Q* (*relation-of-partition P*)
using *is-partition-in-subset*[*OF part-P*] *is-partition-find*[*OF part-P*]
unfolding *equiv-def relation-of-partition-def refl-on-def sym-def trans-def*
by (*auto simp add*: *subset-iff Ball-def Bex-def*) *metis+*

definition *partition-less-eq* **where**
partition-less-eq P1 P2 $\longleftrightarrow$ *relation-of-partition P1* $\subseteq$ *relation-of-partition P2*

lemma *relation-of-partition-inverse* :
assumes *part-P*: *is-partition Q P*
shows $Q \mathbin{//}$ (*relation-of-partition P*) = P

```
  (is ?ls = ?rs)
proof
  {
    fix q
    assume q-in: q ∈ Q
    from is-partition-find[OF part-P q-in]
    obtain p where p-props: p ∈ P q ∈ p ⋀p'. ⟦p' ∈ P; q ∈ p'⟧ ⟹ p' = p by auto

    have {q2. ∃ Q∈P. q ∈ Q ∧ q2 ∈ Q} = p
      apply (intro set-eqI)
      apply (simp add: Bex-def)
      apply (metis p-props)
    done
    with p-props(1)
    have {q2. ∃ Q∈P. q ∈ Q ∧ q2 ∈ Q} ∈ P by simp
  }
  thus ?ls ⊆ ?rs
    unfolding relation-of-partition-def quotient-def
    by auto
next
  {
    fix p
    assume p-in: p ∈ P
    from p-in part-P have p ≠ {} unfolding is-partition-def by blast
    then obtain q where q-in-p: q ∈ p by auto

    with is-partition-in-subset[OF part-P p-in] have q-in-Q: q ∈ Q by blast

    from is-partition-find[OF part-P q-in-Q] p-in q-in-p
    have p-dist: ⋀p'. ⟦p' ∈ P; q ∈ p'⟧ ⟹ p' = p by auto

    have {q2. ∃ Q∈P. q ∈ Q ∧ q2 ∈ Q} = p
      apply (intro set-eqI)
      apply (simp add: Bex-def)
      apply (metis p-dist p-in q-in-p)
    done
    hence ∃ q∈Q. p = {q2. ∃ Q∈P. q ∈ Q ∧ q2 ∈ Q}
      unfolding Bex-def
      apply (rule-tac exI [where x = q])
      apply (simp add: q-in-Q)
    done
  }
  thus ?rs ⊆ ?ls
    unfolding relation-of-partition-def quotient-def
    by auto
qed

lemma quotient-inverse :
assumes eq-r: equiv Q r
shows (relation-of-partition (Q // r)) = r
  (is ?ls = ?rs)
using eq-r
unfolding relation-of-partition-def quotient-def
apply (rule-tac set-eqI)
apply auto
apply (metis equiv-def sym-def trans-def)
apply (simp add: equiv-def refl-on-def subset-iff sym-def)
apply metis
done
```

4.2.3 Weak Equivalence Partitions

For Hopcroft's algorithm, we consider special partitions. They have to satisfy two properties: First, if two states are equivalent, they have to be in the same set of the partition. This property will later allow an induction argument when splitting partitions. However, for the base case, we need a stronger property. All sets of the considered partitions either contain only accepting states or only non-accepting states.

```
definition is-weak-language-equiv-set where
  is-weak-language-equiv-set 𝒜 p ≡
   (p ⊆ 𝒬 𝒜) ∧
   ((p ⊆ ℱ 𝒜) ∨ (p ∩ ℱ 𝒜 = {})) ∧
   (∀ q1 ∈ 𝒬 𝒜. ∀ q2 ∈ 𝒬 𝒜.
      ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2 ∧
      q1 ∈ p ⟶ q2 ∈ p)

lemma is-weak-language-equiv-setI [intro!] :
  ⟦p ⊆ 𝒬 𝒜;
   (p ⊆ ℱ 𝒜) ∨ (p ∩ ℱ 𝒜 = {});
   ⋀q1 q2. ⟦q1 ∈ 𝒬 𝒜; q2 ∈ 𝒬 𝒜; q1 ∈ p;
          ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2⟧ ⟹
          q2 ∈ p⟧ ⟹
   is-weak-language-equiv-set 𝒜 p
unfolding is-weak-language-equiv-set-def
by blast

lemma is-weak-language-equiv-setD :
  ⟦is-weak-language-equiv-set 𝒜 p;
   q1 ∈ 𝒬 𝒜; q2 ∈ 𝒬 𝒜; ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2⟧ ⟹
   (q1 ∈ p) ⟷ (q2 ∈ p)
unfolding is-weak-language-equiv-set-def by blast

definition is-weak-language-equiv-partition where
  is-weak-language-equiv-partition 𝒜 P ⟷
   is-partition (𝒬 𝒜) P ∧
   (∀ p ∈ P. is-weak-language-equiv-set 𝒜 p)

lemma is-weak-language-equiv-partitionI [intro!] :
  ⟦is-partition (𝒬 𝒜) P;
   ⋀p. p ∈ P ⟹ is-weak-language-equiv-set 𝒜 p⟧ ⟹
   is-weak-language-equiv-partition 𝒜 P
  unfolding is-weak-language-equiv-partition-def Ball-def
  by blast

lemma is-weak-language-equiv-partitionD1 :
  ⟦is-weak-language-equiv-partition 𝒜 P;
   p ∈ P⟧ ⟹ p ⊆ ℱ 𝒜 ∨ (p ∩ ℱ 𝒜 = {})
  unfolding is-weak-language-equiv-partition-def
          is-weak-language-equiv-set-def Ball-def
  by blast

lemma is-weak-language-equiv-partitionD2 :
  ⟦is-weak-language-equiv-partition 𝒜 P;
   q1 ∈ 𝒬 𝒜; q2 ∈ 𝒬 𝒜; p ∈ P;
   ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2;
   q1 ∈ p⟧ ⟹ q2 ∈ p
  unfolding is-weak-language-equiv-partition-def
    is-weak-language-equiv-set-def Ball-def
  by blast

lemma is-weak-language-equiv-partitionD2-iff :
  ⟦is-weak-language-equiv-partition 𝒜 P;
   q1 ∈ 𝒬 𝒜; q2 ∈ 𝒬 𝒜; p ∈ P;
```

$\mathcal{L}$-*in-state* $\mathcal{A}$ $q1$ = $\mathcal{L}$-*in-state* $\mathcal{A}$ $q2$⟧ $\Longrightarrow$
$q1 \in p \longleftrightarrow q2 \in p$
by (*metis is-weak-language-equiv-partitionD2*)

lemma *is-weak-language-equiv-partitionD3* :
is-weak-language-equiv-partition $\mathcal{A}$ $P \Longrightarrow$
is-partition ($\mathcal{Q}$ $\mathcal{A}$) P
unfolding *is-weak-language-equiv-partition-def*
by *simp*

An alternative definition of *is-weak-language-equiv-partition* (that is often used in literature about Hopcroft's algorithm) can be given using the connection between partitions and equivalence relations.

definition *Hopcroft-accepting-relation* **where**
Hopcroft-accepting-relation $\mathcal{A} \equiv \{(q1, q2) . q1 \in \mathcal{Q}\ \mathcal{A} \wedge q2 \in \mathcal{Q}\ \mathcal{A} \wedge (q1 \in \mathcal{F}\ \mathcal{A} \longleftrightarrow q2 \in \mathcal{F}\ \mathcal{A})\}$

lemma *equiv-Hopcroft-accepting-relation* :
equiv ($\mathcal{Q}$ $\mathcal{A}$) (*Hopcroft-accepting-relation* $\mathcal{A}$)
unfolding *Hopcroft-accepting-relation-def equiv-def refl-on-def sym-def trans-def*
by *auto*

definition *Hopcroft-accepting-partition* **where**
Hopcroft-accepting-partition $\mathcal{A} \equiv (\mathcal{Q}\ \mathcal{A})$ // (*Hopcroft-accepting-relation* $\mathcal{A}$)

lemma *Hopcroft-accepting-partition-alt-def* :
assumes *wf-A*: *NFA* $\mathcal{A}$
shows *Hopcroft-accepting-partition* $\mathcal{A} = \{\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}, \mathcal{F}\ \mathcal{A}\} \cap \{s.\ s \neq \{\}\}$
unfolding *Hopcroft-accepting-partition-def quotient-def Hopcroft-accepting-relation-def Bex-def*
using *NFA.*$\mathcal{F}$*-consistent* [*OF wf-A*]
by *auto*

definition *Myhill-Nerode-relation* **where**
Myhill-Nerode-relation $\mathcal{A} \equiv \{(q1, q2) . q1 \in \mathcal{Q}\ \mathcal{A} \wedge q2 \in \mathcal{Q}\ \mathcal{A} \wedge (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2)\}$

lemma *equiv-Myhill-Nerode-relation* :
equiv ($\mathcal{Q}$ $\mathcal{A}$) (*Myhill-Nerode-relation* $\mathcal{A}$)
unfolding *Myhill-Nerode-relation-def equiv-def refl-on-def sym-def trans-def*
by *auto*

definition *Myhill-Nerode-partition* **where**
Myhill-Nerode-partition $\mathcal{A} \equiv (\mathcal{Q}\ \mathcal{A})$ // (*Myhill-Nerode-relation* $\mathcal{A}$)

lemma *Myhill-Nerode-partition-alt-def* :
Myhill-Nerode-partition $\mathcal{A}$ =
$\{\{q'.\ q' \in \mathcal{Q}\ \mathcal{A} \wedge \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q' = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q\} \mid q.\ q \in \mathcal{Q}\ \mathcal{A}\}$
unfolding *Myhill-Nerode-partition-def quotient-def Myhill-Nerode-relation-def Bex-def*
by *auto*

lemma *Myhill-Nerode-partition---*$\mathcal{F}$*-emp* :
assumes $\mathcal{F}$*-eq*: $\mathcal{F}\ \mathcal{A} = \{\}$
and $\mathcal{Q}$*-neq*: $\mathcal{Q}\ \mathcal{A} \neq \{\}$
shows *Myhill-Nerode-partition* $\mathcal{A} = \{\mathcal{Q}\ \mathcal{A}\}$
proof −
from $\mathcal{F}$*-eq* **have** $\bigwedge q.\ q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = \{\}$
unfolding $\mathcal{L}$*-in-state-def* **by** *simp*

with $\mathcal{Q}$*-neq* **show** *?thesis*
unfolding *Myhill-Nerode-partition-alt-def*
by *auto*
qed

lemma *Myhill-Nerode-partition---*$\mathcal{Q}$*-emp* :

$\mathcal{Q}\ \mathcal{A} = \{\} \Longrightarrow$ *Myhill-Nerode-partition* $\mathcal{A} = \{\}$
unfolding *Myhill-Nerode-partition-def*
by *auto*

lemma *is-weak-language-equiv-partition-alt-def* :
is-weak-language-equiv-partition $\mathcal{A}$ $P \longleftrightarrow$
is-partition ($\mathcal{Q}\ \mathcal{A}$) $P \wedge$
partition-less-eq (*Myhill-Nerode-partition* $\mathcal{A}$) $P \wedge$
partition-less-eq P (*Hopcroft-accepting-partition* $\mathcal{A}$)
proof (*cases is-partition* ($\mathcal{Q}\ \mathcal{A}$) P)
  case *False* **thus** *?thesis* **unfolding** *is-weak-language-equiv-partition-def* **by** *simp*
next
  case *True* **note** *part-P* = *this*

  from *is-partition-find*[*OF part-P*] *is-partition-in-subset*[*OF part-P*]
  show *?thesis*
  unfolding *is-weak-language-equiv-partition-def partition-less-eq-def*
      Myhill-Nerode-partition-def Hopcroft-accepting-partition-def
    apply (*simp add*: *equiv-Myhill-Nerode-relation quotient-inverse equiv-Hopcroft-accepting-relation*)
    apply (*simp add*: *part-P is-weak-language-equiv-set-def relation-of-partition-def subset-iff*
        set-eq-iff Ball-def Bex-def Myhill-Nerode-relation-def Hopcroft-accepting-relation-def)
    apply (*intro iffI conjI allI impI*)
    apply *metis+*
  done
qed

Hopcroft's algorithm is interested in finding a partition such that two states are in the same set of the partition, if and only if they are equivalent. The concept of weak language equivalence partitions above guarentees that two states that are equivalent are in the same partition.

In the following the missing property that all the states in one partition are equivalent is formalised.

definition *is-weak2-language-equiv-set* **where**
  is-weak2-language-equiv-set $\mathcal{A}$ $p \equiv$
   ($p \subseteq \mathcal{Q}\ \mathcal{A}$) $\wedge$
   (($p \subseteq \mathcal{F}\ \mathcal{A}$) $\vee$ ($p \cap \mathcal{F}\ \mathcal{A} = \{\}$)) $\wedge$
   ($\forall$ *q1* $\in \mathcal{Q}\ \mathcal{A}$. $\forall$ *q2* $\in \mathcal{Q}\ \mathcal{A}$.
     q1 $\in p \wedge$ *q2* $\in p \longrightarrow$ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q2*)

lemma *is-weak2-language-equiv-set-alt-def* :
  is-weak2-language-equiv-set $\mathcal{A}$ p =
   (($p \subseteq \mathcal{Q}\ \mathcal{A}$) $\wedge$
   ($\forall$ *q1* $\in \mathcal{Q}\ \mathcal{A}$. $\forall$ *q2* $\in \mathcal{Q}\ \mathcal{A}$.
     q1 $\in p \wedge$ *q2* $\in p \longrightarrow$ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q2*))
unfolding *is-weak2-language-equiv-set-def Ball-def*
by *auto* (*metis* $\mathcal{L}$*-in-state---in-*$\mathcal{F}$ *in-mono*)

definition *is-weak2-language-equiv-partition* **where**
  is-weak2-language-equiv-partition $\mathcal{A}$ $P \longleftrightarrow$
   is-partition ($\mathcal{Q}\ \mathcal{A}$) $P \wedge$
   ($\forall$ $p \in P$. *is-weak2-language-equiv-set* $\mathcal{A}$ p)

lemma *is-weak2-language-equiv-partition-alt-def* :
is-weak2-language-equiv-partition $\mathcal{A}$ $P \longleftrightarrow$
is-partition ($\mathcal{Q}\ \mathcal{A}$) $P \wedge$ *partition-less-eq* P (*Myhill-Nerode-partition* $\mathcal{A}$)
proof (*cases is-partition* ($\mathcal{Q}\ \mathcal{A}$) P)
  case *False* **thus** *?thesis* **unfolding** *is-weak2-language-equiv-partition-def* **by** *simp*
next
  case *True* **note** *part-P* = *this*

  from *is-partition-find*[*OF part-P*] *is-partition-in-subset*[*OF part-P*]
  show *?thesis*

```
  unfolding is-weak2-language-equiv-partition-def partition-less-eq-def Myhill-Nerode-partition-def
    apply (simp add: equiv-Myhill-Nerode-relation quotient-inverse)
    apply (simp add: part-P is-weak2-language-equiv-set-alt-def relation-of-partition-def
                     subset-iff set-eq-iff Ball-def Bex-def Myhill-Nerode-relation-def)
    apply (intro iffI conjI allI impI)
    apply metis+
  done
qed

definition is-strong-language-equiv-set where
  is-strong-language-equiv-set 𝒜 p ≡
   is-weak-language-equiv-set 𝒜 p ∧
   is-weak2-language-equiv-set 𝒜 p

lemma is-strong-language-equiv-set-alt-def :
  is-strong-language-equiv-set 𝒜 p =
   ((p ⊆ 𝒬 𝒜) ∧
    (∀ q1 ∈ p. ∀ q2 ∈ 𝒬 𝒜.
      ((q2 ∈ p)) ⟷ (ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2)))
proof (cases p ⊆ 𝒬 𝒜)
  case False thus ?thesis
    unfolding is-strong-language-equiv-set-def
          is-weak2-language-equiv-set-def
          is-weak-language-equiv-set-def
    by simp
next
  case True note p-subset = this
  thus ?thesis
  proof (cases (p ⊆ ℱ 𝒜) ∨ (p ∩ ℱ 𝒜 = {}))
    case False note not-F-prop = this
    then obtain q1 q2 where
      q12-neq-in-F: (q1 ∈ ℱ 𝒜) ≠ (q2 ∈ ℱ 𝒜) and
      q1-in: q1 ∈ p and q2-in: q2 ∈ p by auto

    thus ?thesis
      unfolding is-strong-language-equiv-set-def
                is-weak2-language-equiv-set-def
                is-weak-language-equiv-set-def
      apply (simp add: not-F-prop Bex-def)
      apply (rule impI)
      apply (rule-tac exI [where x = q1])
      apply (simp add: subset-iff)
      apply (rule-tac exI [where x = q2])
      apply (simp only: in-ℒ-in-state-Nil[symmetric] subset-iff set-eq-iff Bex-def)
      apply metis
    done
  next
    case True note F-prop = this
    have ((∀ q1∈𝒬 𝒜. ∀ q2∈𝒬 𝒜. ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2 ∧ q1 ∈ p ⟶ q2 ∈ p) ∧
           (∀ q1∈𝒬 𝒜. ∀ q2∈𝒬 𝒜. ((q1 ∈ p) ∧ (q2 ∈ p)) ⟶ (ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2))) =
          (∀ q1∈p. ∀ q2∈𝒬 𝒜. (q2 ∈ p) = (ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2))
          (is (?P11 ∧ ?P12) = ?P2)
    proof (intro iffI)
      assume ?P2 hence
        ∀ q1∈p. ∀ q2∈𝒬 𝒜. ((ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2) = (q2 ∈ p))
        by metis
      thus ?P11 ∧ ?P12 by simp metis
    next
      assume pre: ?P11 ∧ ?P12
      thus ?P2
      proof (intro ballI)
        fix q1 q2
```

```
      assume q1-in-p: q1 ∈ p and q2-in-Q: q2 ∈ 𝒬 𝒜
      from q1-in-p p-subset have q1-in-Q: q1 ∈ 𝒬 𝒜 by auto

      show (q2 ∈ p) = (ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2)
        by (metis q1-in-p q1-in-Q q2-in-Q pre)
    qed
  qed
  thus ?thesis
    unfolding is-strong-language-equiv-set-def
              is-weak2-language-equiv-set-def
              is-weak-language-equiv-set-def
    by (simp add: p-subset F-prop)
 qed
qed

lemma is-strong-language-equiv-partition-fun-alt-def :
 (P = Myhill-Nerode-partition 𝒜) ⟷
  (is-partition (𝒬 𝒜) P ∧ (∀ p∈P. is-strong-language-equiv-set 𝒜 p))
proof
 assume P-eq: P = Myhill-Nerode-partition 𝒜
 show is-partition (𝒬 𝒜) P ∧ (∀ p∈P. is-strong-language-equiv-set 𝒜 p)
  unfolding is-partition-def P-eq Myhill-Nerode-partition-alt-def
           Ball-def is-strong-language-equiv-set-alt-def
  by auto
next
 assume pre: is-partition (𝒬 𝒜) P ∧ (∀ p∈P. is-strong-language-equiv-set 𝒜 p)
 note is-part = conjunct1[OF pre]
 have strong-set : ⋀p. p ∈ P ⟹ is-strong-language-equiv-set 𝒜 p using pre by simp

 show P = Myhill-Nerode-partition 𝒜
 proof (intro set-eqI iffI)
  fix p
  assume p-in-P: p ∈ P

  with is-part have p ≠ {} unfolding is-partition-def by blast
  then obtain q where q-in: q ∈ p by auto

  from strong-set [OF p-in-P] q-in
  show p ∈ Myhill-Nerode-partition 𝒜
    unfolding Myhill-Nerode-partition-alt-def is-strong-language-equiv-set-alt-def
    apply (simp)
    apply (rule-tac exI [where x = q])
    apply (auto)
  done
 next
  fix p
  assume p-in: p ∈ Myhill-Nerode-partition 𝒜
  then obtain q where
    p-eq: p = {q' ∈ 𝒬 𝒜. ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q} and q-in-Q: q ∈ 𝒬 𝒜
    unfolding Myhill-Nerode-partition-alt-def
    by auto

  from is-partition-find [OF is-part q-in-Q]
  obtain p' where p'-in-P: p' ∈ P and q-in-p': q ∈ p' by auto

  from strong-set [OF p'-in-P] q-in-p'
  have p' = p
    unfolding p-eq is-strong-language-equiv-set-alt-def
    by auto
  with p'-in-P show p ∈ P by simp
 qed
qed
```

4.2.4 Initial partition

By now, the essential concepts of different partitions have been introduced. Hopcrofts algorithm operates by splitting weak language partions. If no further split is possible, the searched partition has been found. For this algorithm a suitable initial partition is needed:

lemma (**in** *NFA*) *is-weak-language-equiv-partition-init* :
is-weak-language-equiv-partition $\mathcal{A}$
(*Hopcroft-accepting-partition* $\mathcal{A}$)
unfolding *is-weak-language-equiv-partition-alt-def*
proof (*intro conjI*)
show *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) (*Hopcroft-accepting-partition* $\mathcal{A}$)
unfolding *Hopcroft-accepting-partition-def*
using *equiv-Hopcroft-accepting-relation*
by (*rule quotient-of-equiv-relation-is-partition*)
next
show *partition-less-eq* (*Hopcroft-accepting-partition* $\mathcal{A}$) (*Hopcroft-accepting-partition* $\mathcal{A}$)
unfolding *partition-less-eq-def* **by** *simp*
next
have *Myhill-Nerode-relation* $\mathcal{A} \subseteq$ *Hopcroft-accepting-relation* $\mathcal{A}$
unfolding *Myhill-Nerode-relation-def Hopcroft-accepting-relation-def*
by (*simp add*: *subset-iff*) (*metis* $\mathcal{L}$*-in-state---in-*$\mathcal{F}$)
thus *partition-less-eq* (*Myhill-Nerode-partition* $\mathcal{A}$) (*Hopcroft-accepting-partition* $\mathcal{A}$)
unfolding *partition-less-eq-def Myhill-Nerode-partition-def Hopcroft-accepting-partition-def*
by (*simp add*: *quotient-inverse equiv-Myhill-Nerode-relation equiv-Hopcroft-accepting-relation*)
qed

4.2.5 Splitting Partitions

Next, we need to define how partitions are splitted.

definition *split-set* **where**
split-set $P\ S = (\{s \in S.\ P\ s\}, \{s \in S.\ \neg P\ s\})$

lemma *split-set-empty* [*simp*] :
split-set $P\ \{\} = (\{\}, \{\})$
unfolding *split-set-def* **by** *simp*

lemma *split-set-insert* :
split-set P (*insert* s S) =
(*let* (ST, SF) = *split-set* P S *in*
(*if* $P\ s$ *then* (*insert* s ST, SF) *else* (ST, *insert* s SF)))
unfolding *split-set-def* **by** *auto*

lemma *split-set-union-distint*:
split-set $P\ S = (S1, S2) \Longrightarrow$
$(S = S1 \cup S2) \wedge (S1 \cap S2 = \{\})$
unfolding *split-set-def* **by** *auto*

Given two sets of states *p1*, *p2* of an automaton $\mathcal{A}$ and a label *a*. The set *p1* is splitted according to whether a state in *p2* is reachable by *a*.

definition *split-language-equiv-partition* **where**
split-language-equiv-partition $\mathcal{A}$ *p1 a p2* =
split-set $(\lambda q.\ \exists\, q' \in p2.\ (q,\ a,\ q') \in \Delta\ \mathcal{A})$ *p1*

Hopcroft's algorithm operates on determistic automata. Exploiting the property, that the automaton is determistic, the definition of splitting a partition becomes much simpler.

lemma (**in** *DFA*) *split-language-equiv-partition-alt-def* :
assumes *p1-subset*: $p1 \subseteq \mathcal{Q}\ \mathcal{A}$
and *a-in*: $a \in \Sigma\ \mathcal{A}$
shows *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* =
$(\{q\ .\ \exists\, q'.\ q \in p1 \wedge (\delta\ \mathcal{A})\ (q,\ a) = Some\ q' \wedge q' \in p2\},$

$\{q \,.\, \exists\, q'.\ q \in p1 \wedge (\delta\ \mathcal{A})\ (q,\ a) = \mathit{Some}\ q' \wedge q' \notin p2\})$
unfolding *split-language-equiv-partition-def split-set-def*
using *assms*
apply (*auto simp add*: *δ-in-Δ-iff*)
by (*metis DFA-δ-is-some LTS-to-DLTS-is-none δ-def δ-in-Δ-iff subsetD*)

lemma *split-language-equiv-partition-disjoint* :
  $[\![$*split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)$]\!] \Longrightarrow$
   p1a ∩ *p1b* = {}
unfolding *split-language-equiv-partition-def*
by (*simp add*: *split-set-union-distint*)

lemma *split-language-equiv-partition-union* :
  split-language-equiv-partition $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*) $\Longrightarrow$
   p1 = *p1a* ∪ *p1b*
unfolding *split-language-equiv-partition-def*
by (*simp add*: *split-set-union-distint*)

lemma *split-language-equiv-partition-subset* :
assumes *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
shows *p1a* ⊆ *p1* ∧ *p1b* ⊆ *p1*
using *split-language-equiv-partition-union*[*OF assms*]
by *simp*

Splitting only makes sense if one of the resulting sets is non-empty. This property is very important. Therefore, a special predicate is introduced.

definition *split-language-equiv-partition-pred* ::
  (*'q*, *'a*, *'x*) *NFA-rec-scheme* ⇒ *'q set* ⇒ *'a* ⇒ *'q set* ⇒ *bool* **where**
  split-language-equiv-partition-pred $\mathcal{A}$ *p1 a p2* ≡
   (*fst* (*split-language-equiv-partition* $\mathcal{A}$ *p1 a p2*) ≠ {}) ∧
   (*snd* (*split-language-equiv-partition* $\mathcal{A}$ *p1 a p2*) ≠ {})

Splitting according to this definition preserves the property that the partion is a weak language equivalence partition.

lemma (**in** *DFA*) *split-language-equiv-partition---weak-language-equiv-set* :
assumes *split*: *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
   and *a-in*: *a* ∈ Σ $\mathcal{A}$
   and *equiv-p1*: *is-weak-language-equiv-set* $\mathcal{A}$ *p1*
   and *equiv-p2*: *is-weak-language-equiv-set* $\mathcal{A}$ *p2*
shows *is-weak-language-equiv-set* $\mathcal{A}$ *p1a* ∧ *is-weak-language-equiv-set* $\mathcal{A}$ *p1b*
proof −
  from *equiv-p1* **have** *p1-subset*: *p1* ⊆ $\mathcal{Q}$ $\mathcal{A}$
   unfolding *is-weak-language-equiv-set-def*
   by *fast*

  from *split-language-equiv-partition-union* [*OF split*]
  have *p1-eq*: *p1* = *p1a* ∪ *p1b* **.**

  have $\bigwedge$*p*. *p* = *p1a* ∨ *p* = *p1b* $\Longrightarrow$ *is-weak-language-equiv-set* $\mathcal{A}$ *p*
  proof −
   fix *p*
   assume *p-eq*: *p* = *p1a* ∨ *p* = *p1b*

   from *p-eq p1-eq* **have** *p-subset*: *p* ⊆ *p1* **by** *blast*
   show *is-weak-language-equiv-set* $\mathcal{A}$ *p*
   proof
    from *p1-subset p-subset*
    show *p* ⊆ $\mathcal{Q}$ $\mathcal{A}$ **by** *blast*
   next
    from *equiv-p1*
    have *p1* ⊆ $\mathcal{F}$ $\mathcal{A}$ ∨ *p1* ∩ $\mathcal{F}$ $\mathcal{A}$ = {}

unfolding *is-weak-language-equiv-set-def*
by *fast*
with *p-subset*
show $p \subseteq \mathcal{F}\ \mathcal{A} \vee p \cap \mathcal{F}\ \mathcal{A} = \{\}$
by *blast*
next
fix *q1 q2*
assume *q1-in-Q*: $q1 \in \mathcal{Q}\ \mathcal{A}$
and *q2-in-Q*: $q2 \in \mathcal{Q}\ \mathcal{A}$
and *q1-q2-equiv*: $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q2*
and *q1-in-p*: $q1 \in p$

from *p1-eq p-eq q1-in-p* **have** *q1-in-p1*: $q1 \in p1$ **by** *auto*
with *equiv-p1* **have** $q2 \in p1$
unfolding *is-weak-language-equiv-set-def*
using *q1-in-Q q2-in-Q q1-q2-equiv q1-in-p1*
by *blast*
with *p1-eq* **have** *q2-in*: $q2 \in p1a \vee q2 \in p1b$ **by** *simp*

obtain *q1' q2'*
where *q1-delta*: $(q1, a, q1') \in \Delta\ \mathcal{A}$
and *q2-delta*: $(q2, a, q2') \in \Delta\ \mathcal{A}$
using *deterministic q1-in-Q q2-in-Q a-in*
unfolding *LTS-is-deterministic-def* **by** *blast*
from $\mathcal{L}$*-in-state---DFA---eq-reachable-step* [*OF wf-DFA wf-DFA q1-delta q2-delta q1-q2-equiv*]
have *q1'-q2'-equiv*: $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1'* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q2'* **.**

from *q1-delta q2-delta* Δ*-consistent* **have**
$q1' \in \mathcal{Q}\ \mathcal{A}\ q2' \in \mathcal{Q}\ \mathcal{A}$ **by** *simp-all*
hence *in-p2* : $q1' \in p2 \longleftrightarrow q2' \in p2$
using *is-weak-language-equiv-setD*
[*OF equiv-p2 - - q1'-q2'-equiv*]
by *simp*

from *in-p2 p-eq q1-in-p q2-in split*[*symmetric*]
show $q2 \in p$
using *q1-delta* [*unfolded* δ*-in-*Δ*-iff*] *q2-delta* [*unfolded* δ*-in-*Δ*-iff*]
unfolding *split-language-equiv-partition-alt-def* [*OF p1-subset a-in*]
by *auto*
qed
qed
thus *?thesis* **by** *simp*
qed

lemma (**in** *DFA*) *split-language-equiv-partition-step* :
assumes *is-part*: *is-weak-language-equiv-partition* $\mathcal{A}$ *P*
and *p1-in* : $p1 \in P$
and *a-in*: $a \in \Sigma\ \mathcal{A}$
and *p2-equiv-set*: *is-weak-language-equiv-set* $\mathcal{A}$ *p2*
and *split-pred*: *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2*
and *split*: *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
shows *is-weak-language-equiv-partition* $\mathcal{A}$ $((P - \{p1\}) \cup \{p1a, p1b\})$
(*card* $((P - \{p1\}) \cup \{p1a, p1b\})$ = *Suc* (*card P*))
proof
from *split-pred split* **have**
p1ab-neq-Nil: $p1a \neq \{\}\ p1b \neq \{\}$
unfolding *split-language-equiv-partition-pred-def* **by** *simp-all*

from *is-part* **have**
$P \subseteq Pow\ (\mathcal{Q}\ \mathcal{A})$
unfolding *is-weak-language-equiv-partition-def is-partition-def* **by** *auto*

with *p1-in* **have** *p1-sub*: $p1 \subseteq \mathcal{Q}\ \mathcal{A}$ **by** *auto*

note *p1ab-disj* = *split-language-equiv-partition-disjoint* [*OF split*]
note *p1ab-union* = *split-language-equiv-partition-union* [*OF split*]

let $?P' = ((P - \{p1\}) \cup \{p1a,\ p1b\})$
show *is-part-P'*: *is-partition* $(\mathcal{Q}\ \mathcal{A})$ $?P'$ **and** *card-P'*: *card* $?P' = Suc\ (card\ P)$
proof −
  note *is-partition-refine* [*OF finite-$\mathcal{Q}$ - - p1ab-neq-Nil p1ab-disj p1ab-union*,
                where $P = P - \{p1\}$]
      is-part
  moreover
  have *insert p1 P = P* **using** *p1-in* **by** *auto*
  ultimately show *is-partition* $(\mathcal{Q}\ \mathcal{A})$ $?P'$ *card* $?P' = Suc\ (card\ P)$
   by (*simp-all add*: *is-weak-language-equiv-partition-def*)
qed

{
  fix *p*
  assume *p-in-P'*: $p \in ?P'$
  show *is-weak-language-equiv-set* $\mathcal{A}$ *p*
  proof (*cases* $p \in P$)
   case *True*
   with *is-part* **show** *?thesis*
    unfolding *is-weak-language-equiv-partition-def*
    by *simp*
  next
   case *False*
   hence *p-eq*: $p = p1a \lor p = p1b$ **using** *p-in-P'* **by** *simp*
   show *?thesis*
   proof −
    from *p1-in is-part*
    have *p1-equiv-set*: *is-weak-language-equiv-set* $\mathcal{A}$ *p1*
     unfolding *is-weak-language-equiv-partition-def* **by** *simp*

    note *split-language-equiv-partition---weak-language-equiv-set*
     [*OF split a-in p1-equiv-set p2-equiv-set*]
    with *p-eq* **show** *?thesis* **by** *metis*
   qed
  qed
}
qed

If no more splitting is possible, the desired strong language equivalence partion has been found.

lemma (**in** *DFA*) *split-language-equiv-partition-final---weak2* :
assumes *is-part*: *is-partition* $(\mathcal{Q}\ \mathcal{A})$ *P*
  and *accept-P*: $\bigwedge p.\ p \in P \Longrightarrow (p \subseteq \mathcal{F}\ \mathcal{A}) \lor (p \cap \mathcal{F}\ \mathcal{A} = \{\})$
  and *no-split*: $\bigwedge p1\ a\ p2.\ [\![p1 \in P;\ a \in \Sigma\ \mathcal{A};\ p2 \in P]\!] \Longrightarrow$
    $\neg$(*split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2*)
  and *p0-in*: $p0 \in P$
shows *is-weak2-language-equiv-set* $\mathcal{A}$ *p0*
proof (*rule ccontr*)
  assume $\neg$ *is-weak2-language-equiv-set* $\mathcal{A}$ *p0*
  then obtain *q1-0 q2-0* **where** *q12-0-in-p0*: $q1\text{-}0 \in p0$ $q2\text{-}0 \in p0$ **and**
   not-$\mathcal{L}$-eq-q12-0: $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1-0* $\neq$ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q2-0*
  using *p0-in is-partition-in-subset*[*OF is-part p0-in*] *accept-P*[*OF p0-in*]
  unfolding *is-weak2-language-equiv-set-def*
   by *metis*

  def *counter-P* $\equiv \lambda(q1,\ q2,\ p,\ w).$
    $(p \in P \land q1 \in p \land q2 \in p \land$
    $((w \in \mathcal{L}\text{-in-state}\ \mathcal{A}\ q1) \neq (w \in \mathcal{L}\text{-in-state}\ \mathcal{A}\ q2)))$

have $\exists$ *q1 q2 p w. counter-P* (*q1*, *q2*, *p*, *w*)
proof −
  from *not-$\mathcal{L}$-eq-q12-0* **obtain** *w* **where**
    ($w \in$ *$\mathcal{L}$-in-state* $\mathcal{A}$ *q1-0*) $\neq$ ($w \in$ *$\mathcal{L}$-in-state* $\mathcal{A}$ *q2-0*)
    by (*simp add*: *set-eq-iff*) *metis*

  thus *?thesis*
    apply (*rule-tac exI* [**where** x = *q1-0*])
    apply (*rule-tac exI* [**where** x = *q2-0*])
    apply (*rule-tac exI* [**where** x = *p0*])
    apply (*rule-tac exI* [**where** x = *w*])
    apply (*simp add*: *counter-P-def q12-0-in-p0 p0-in*)
  done
qed

with *ex-has-least-nat* [**where** P = *counter-P* **and** $m = \lambda$(*q1*, *q2*, *p*, *w*). *length w*]
obtain *q1 q2 p w* **where**
  counter-ex: *counter-P* (*q1*, *q2*, *p*, *w*) **and**
  min-counter: $\bigwedge$*q1′ q2′ p′ w′*. *counter-P* (*q1′*, *q2′*, *p′*, *w′*) $\Longrightarrow$ *length w* $\leq$ *length w′*
  by *auto metis*

from *counter-ex* **have**
  p-in: $p \in P$ **and**
  q12-in-p: *q1* $\in p$ *q2* $\in p$ **and**
  w-equiv: ($w \in$ *$\mathcal{L}$-in-state* $\mathcal{A}$ *q1*) $\neq$ ($w \in$ *$\mathcal{L}$-in-state* $\mathcal{A}$ *q2*)
  unfolding *counter-P-def* **by** *simp-all*

from *p-in is-part* **have** *p-sub*: $p \subseteq \mathcal{Q}\ \mathcal{A}$
  unfolding *is-partition-def* **by** *auto*
with *q12-in-p* **have** *q12-in-Q*: *q1* $\in \mathcal{Q}\ \mathcal{A}$ *q2* $\in \mathcal{Q}\ \mathcal{A}$ **by** *auto*

have $w \neq []$
  using *accept-P* [*OF p-in*]
      w-equiv q12-in-p
  by *auto*
then obtain *a w′* **where** *w-eq*: $w = a\ \#\ w'$ **by** (*cases w*, *blast*)
from *w-equiv w-eq* **have** *a-in*: $a \in \Sigma\ \mathcal{A}$
  unfolding *$\mathcal{L}$-in-state-def* **by** *auto*

obtain *q1′ q2′* **where**
  δ-q1a: (δ $\mathcal{A}$) (*q1*, *a*) = *Some q1′* **and**
  δ-q2a: (δ $\mathcal{A}$) (*q2*, *a*) = *Some q2′*
  using *DFA-δ-is-some* [*OF q12-in-Q*(*1*) *a-in*]
  using *DFA-δ-is-some* [*OF q12-in-Q*(*2*) *a-in*]
  by *blast*

from *is-partition-find*[*of* $\mathcal{Q}$ $\mathcal{A}$ *P q1′*] *is-part*
    δ-wf[*OF δ-q1a*]
obtain *p′* **where** *p′-in*: $p' \in P$ **and** *q1′-in*: *q1′* $\in p'$
  unfolding *is-weak-language-equiv-partition-def* **by** *metis*

from *no-split* [*OF p-in a-in p′-in*]
have *q2′-in*: *q2′* $\in p'$
  using *q12-in-p q1′-in δ-q1a δ-q2a a-in p-in p′-in*
  unfolding
    split-language-equiv-partition-pred-def
    split-language-equiv-partition-def split-set-def
  by (*auto simp add*: *δ-in-Δ-iff*)

from *w-equiv* **have**
  w′-equiv: ($w' \in$ *$\mathcal{L}$-in-state* $\mathcal{A}$ *q1′*) $\neq$ ($w' \in$ *$\mathcal{L}$-in-state* $\mathcal{A}$ *q2′*)

```
    unfolding w-eq
    by (simp add: δ-in-Δ-iff δ-q1a δ-q2a a-in)

  from min-counter [of q1' q2' p' w']
  show False
    unfolding counter-P-def
    using w'-equiv
    by (simp add: p'-in q1'-in q2'-in w-eq)
qed

lemma (in DFA) split-language-equiv-partition-final :
assumes is-part: is-weak-language-equiv-partition 𝒜 P
    and no-split: ⋀p1 a p2. ⟦p1 ∈ P; a ∈ Σ 𝒜; p2 ∈ P⟧ ⟹
       ¬(split-language-equiv-partition-pred 𝒜 p1 a p2)
shows P = Myhill-Nerode-partition 𝒜
proof (rule ccontr)
  assume not-strong-part: P ≠ Myhill-Nerode-partition 𝒜

  def counter-P ≡ λ(q1, q2, p, w).
        (p ∈ P ∧ q1 ∈ p ∧ q2 ∈ p ∧
        ((w ∈ ℒ-in-state 𝒜 q1) ≠ (w ∈ ℒ-in-state 𝒜 q2)))

  have ∃ q1 q2 p w. counter-P (q1, q2, p, w)
  proof (rule ccontr)
    assume ¬ (∃ q1 q2 p w. counter-P (q1, q2, p, w))
    hence p-equiv: ⋀q1 q2 p. ⟦q2 ∈ p; q1 ∈ p; p ∈ P⟧ ⟹ ℒ-in-state 𝒜 q1 = ℒ-in-state 𝒜 q2
      unfolding counter-P-def by auto

    have sub1: P ⊆ Myhill-Nerode-partition 𝒜
    proof
      fix p
      assume p-in-P: p ∈ P
      from p-in-P is-part have p-subset: p ⊆ 𝒬 𝒜
        unfolding is-weak-language-equiv-partition-def is-partition-def by auto
      from p-in-P is-part have p ≠ {}
        unfolding is-weak-language-equiv-partition-def is-partition-def by auto
      with p-subset obtain q where q-in: q ∈ p q ∈ 𝒬 𝒜 by auto

      from p-in-P is-part have p-equiv': is-weak-language-equiv-set 𝒜 p
        unfolding is-weak-language-equiv-partition-def by auto

      have p = {q' ∈ 𝒬 𝒜. ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q}
      proof
        from p-equiv' q-in
        have ⋀q'. ⟦q' ∈ 𝒬 𝒜 ; ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q⟧ ⟹ q' ∈ p
          unfolding is-weak-language-equiv-set-def by metis
        thus {q' ∈ 𝒬 𝒜. ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q} ⊆ p
          by auto
      next
        from p-equiv [OF q-in(1) - p-in-P] p-subset
        show p ⊆ {q' ∈ 𝒬 𝒜. ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q}
          by auto
      qed
      with q-in(2) show p ∈ Myhill-Nerode-partition 𝒜
        unfolding Myhill-Nerode-partition-alt-def
        by blast
    qed

    have sub2: Myhill-Nerode-partition 𝒜 ⊆ P
    proof
      fix p
      assume p ∈ Myhill-Nerode-partition 𝒜
```

```
    then obtain q
    where q-in-Q: q ∈ 𝒬 𝒜 and
          p-eq: p = {q' ∈ 𝒬 𝒜. ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q}
      unfolding Myhill-Nerode-partition-alt-def
      by blast
    from p-eq q-in-Q have q-in-p: q ∈ p
      unfolding p-eq by simp

    from is-part have p-part: is-partition (𝒬 𝒜) P
      unfolding is-weak-language-equiv-partition-def by fast

    from p-part q-in-Q obtain p'
    where p'-in: p' ∈ P
      and q-in-p': q ∈ p'
      unfolding is-partition-def by blast

    from p'-in is-part have p-equiv': is-weak-language-equiv-set 𝒜 p'
      unfolding is-weak-language-equiv-partition-def by auto
    have p'-subset: p' ⊆ 𝒬 𝒜
      using is-partition-in-subset [OF p-part p'-in] .

    have p' = p
    proof
      from p-equiv [OF q-in-p' - p'-in] p'-subset
      show p' ⊆ p
        unfolding p-eq
        by auto
    next
      from p-equiv' q-in-Q q-in-p'
      have ⋀q'. ⟦q' ∈ 𝒬 𝒜; ℒ-in-state 𝒜 q' = ℒ-in-state 𝒜 q⟧ ⟹ q' ∈ p'
        unfolding p-eq is-weak-language-equiv-set-def by metis
      thus p ⊆ p'
        unfolding p-eq
        by auto
    qed
    with p'-in show p ∈ P by fast
  qed

  from not-strong-part sub1 sub2 show False by simp
qed

with ex-has-least-nat [where P = counter-P and m = λ(q1, q2, p, w). length w]
obtain q1 q2 p w where
  counter-ex: counter-P (q1, q2, p, w) and
  min-counter: ⋀q1' q2' p' w'. counter-P (q1', q2', p', w') ⟹ length w ≤ length w'
  by auto metis

from counter-ex have
  p-in: p ∈ P and
  q12-in-p: q1 ∈ p q2 ∈ p and
  w-equiv: (w ∈ ℒ-in-state 𝒜 q1) ≠ (w ∈ ℒ-in-state 𝒜 q2)
  unfolding counter-P-def by simp-all

from p-in is-part have p-sub: p ⊆ 𝒬 𝒜
  unfolding is-weak-language-equiv-partition-def is-partition-def by auto
with q12-in-p have q12-in-Q: q1 ∈ 𝒬 𝒜 q2 ∈ 𝒬 𝒜 by auto

have w ≠ []
  using is-weak-language-equiv-partitionD1 [OF is-part p-in]
        w-equiv q12-in-p
  by auto
then obtain a w' where w-eq: w = a # w' by (cases w, blast)
```

```
  from w-equiv w-eq have a-in: a ∈ Σ 𝒜
    unfolding ℒ-in-state-def by auto

  obtain q1' q2' where
    δ-q1a: (δ 𝒜) (q1, a) = Some q1' and
    δ-q2a: (δ 𝒜) (q2, a) = Some q2'
    using DFA-δ-is-some [OF q12-in-Q(1) a-in]
    using DFA-δ-is-some [OF q12-in-Q(2) a-in]
    by blast

  from is-partition-find[of 𝒬 𝒜 P q1'] is-part
       δ-wf[OF δ-q1a]
  obtain p' where p'-in: p' ∈ P and q1'-in: q1' ∈ p'
     unfolding is-weak-language-equiv-partition-def by metis

  from no-split [OF p-in a-in p'-in]
  have q2'-in: q2' ∈ p'
    using q12-in-p q1'-in δ-q1a δ-q2a a-in p-in p'-in
    unfolding
      split-language-equiv-partition-pred-def
      split-language-equiv-partition-def split-set-def
    by (auto simp add: δ-in-Δ-iff)

  from w-equiv have
    w'-equiv: (w' ∈ ℒ-in-state 𝒜 q1') ≠ (w' ∈ ℒ-in-state 𝒜 q2')
    unfolding w-eq
    by (simp add: δ-in-Δ-iff δ-q1a δ-q2a a-in)

  from min-counter [of q1' q2' p' w']
  show False
    unfolding counter-P-def
    using w'-equiv
    by (simp add: p'-in q1'-in q2'-in w-eq)
qed
```

4.3 Naive implementation

```
definition Hopcroft-naive where
  Hopcroft-naive 𝒜 =
    WHILEIT (is-weak-language-equiv-partition 𝒜)

          (λP. ∃p1 a p2. (p1 ∈ P ∧ a ∈ Σ 𝒜 ∧ p2 ∈ P ∧ split-language-equiv-partition-pred 𝒜 p1 a p2))

          (λP. do {
             (p1, a, p2) ← SPEC (λ(p1,a,p2). p1 ∈ P ∧ a ∈ Σ 𝒜 ∧ p2 ∈ P ∧
                 split-language-equiv-partition-pred 𝒜 p1 a p2);
             let (p1a, p1b) = split-language-equiv-partition 𝒜 p1 a p2;
             RETURN ((P − {p1}) ∪ {p1a, p1b})}) (Hopcroft-accepting-partition 𝒜)

lemma (in DFA) Hopcroft-naive-correct :
  Hopcroft-naive 𝒜 ≤ SPEC (λP. P = Myhill-Nerode-partition 𝒜)
unfolding Hopcroft-naive-def
proof (rule WHILEIT-rule [where R = measure (λP. card (𝒬 𝒜) − card P)])
  show wf (measure (λP. card (𝒬 𝒜) − card P)) by simp
next
  show is-weak-language-equiv-partition 𝒜 (Hopcroft-accepting-partition 𝒜)
    by (rule is-weak-language-equiv-partition-init)
next
  case (goal3 P)
  note weak-part-P = goal3(1)
  show ?case
    apply (intro refine-vcg)
```

```
    apply (simp)
    apply (clarify)
  proof -
    fix p1 a p2 p1a p1b
    assume p1 ∈ P a ∈ Σ 𝒜  p2 ∈ P and
           split-pred: split-language-equiv-partition-pred 𝒜 p1 a p2 and
           eval-part: split-language-equiv-partition 𝒜 p1 a p2 = (p1a, p1b)

    from ‹p2 ∈ P› weak-part-P have weak-set-p2: is-weak-language-equiv-set 𝒜 p2
      unfolding is-weak-language-equiv-partition-def by simp

    note step = split-language-equiv-partition-step  [OF weak-part-P ‹p1 ∈ P› ‹a ∈ Σ 𝒜› weak-set-p2
      split-pred eval-part]

    from is-partition-card-P[OF finite-𝒬 is-weak-language-equiv-partitionD3[OF step(1)]] step(2)
    have card P < card (𝒬 𝒜) by simp

    with step
    show is-weak-language-equiv-partition 𝒜 (insert p1a (insert p1b (P − {p1}))) ∧
          card (𝒬 𝒜) − card (insert p1a (insert p1b (P − {p1}))) < card (𝒬 𝒜) − card P
      by simp
  qed
next
  case goal4 thus ?case
    by (metis split-language-equiv-partition-final)
qed
```

4.4 Abstract implementation

The naive implementation captures the main ideas. However, one would like to optimise for sets *p1*, *p2* and a label *a*. In the following an explicit set of possible choices for *p2* and *a* is maintained. An element from this set is chosen, all elements of the current partition processed and the set of possible choices (the splitter set) updated.

For efficiency reasons, the splitter set should be as small as possible. The following lemma guarentees that a possible choice that has been splitted can be replaced by both splitted subsets.

```
lemma split-language-equiv-partition-pred-split :
assumes p2ab-union: p2a ∪ p2b = p2
    and part-pred-p2: split-language-equiv-partition-pred 𝒜 p1 a p2
shows split-language-equiv-partition-pred 𝒜 p1 a p2a ∨
      split-language-equiv-partition-pred 𝒜 p1 a p2b
proof -
  obtain p1a p1b where p1ab-eq:
    split-language-equiv-partition 𝒜 p1 a p2 = (p1a, p1b)
    by (rule PairE)
  with part-pred-p2 have p1ab-neq-emp: p1a ≠ {} p1b ≠ {}
    unfolding split-language-equiv-partition-pred-def
    by simp-all
  let ?P = λp2 q. ∃ q'∈p2. (q, a, q') ∈ Δ 𝒜

  from p1ab-neq-emp p1ab-eq
    obtain qa qb
    where qa-in: qa ∈ p1
      and qb-in: qb ∈ p1
      and qa-P-p2: ?P p2 qa
      and qb-neg-P-p2: ¬ (?P p2 qb)
    unfolding split-language-equiv-partition-def
              split-set-def
    by auto

  from qb-neg-P-p2 p2ab-union
```

```
  have qb-nin-P-p2ab : ¬ (?P p2a qb) ¬ (?P p2b qb)
    by auto

  show ?thesis
  proof (cases ?P p2a qa)
    case True
    hence split-language-equiv-partition-pred 𝒜 p1 a p2a
      unfolding split-language-equiv-partition-pred-def
                split-language-equiv-partition-def split-set-def
      using qb-nin-P-p2ab(1) qa-in qb-in
      by auto
    thus ?thesis ..
  next
    case False
    with p2ab-union qa-P-p2 have ?P p2b qa
      by auto
    hence split-language-equiv-partition-pred 𝒜 p1 a p2b
      unfolding split-language-equiv-partition-pred-def
                split-language-equiv-partition-def split-set-def
      using qb-nin-P-p2ab(2) qa-in qb-in
      by auto
    thus ?thesis ..
  qed
qed
```

More interestingly, if one already knows that there is no split according to a set *p2* (as it is for example not in the set of splitters), then it is sufficient to consider only one of its splitted components.

```
lemma (in DFA) split-language-equiv-partition-pred-split-neg :
assumes p2ab-union: p2a ∪ p2b = p2
    and p2ab-dist: p2a ∩ p2b = {}
    and part-pred-p2: ¬ (split-language-equiv-partition-pred 𝒜 p1 a p2)
shows (split-language-equiv-partition-pred 𝒜 p1 a p2a) ⟷
       (split-language-equiv-partition-pred 𝒜 p1 a p2b)
proof -
  obtain p1a p1b where p1ab-eq:
    split-language-equiv-partition 𝒜 p1 a p2 = (p1a, p1b)
    by (rule PairE)
  with part-pred-p2 have p1ab-eq-emp: p1a = {} ∨ p1b = {}
    unfolding split-language-equiv-partition-pred-def
    by simp

  obtain p1aa p1ba where p1aba-eq:
    split-language-equiv-partition 𝒜 p1 a p2a = (p1aa, p1ba)
    by (rule PairE)

  obtain p1ab p1bb where p1abb-eq:
    split-language-equiv-partition 𝒜 p1 a p2b = (p1ab, p1bb)
    by (rule PairE)

  def P ≡ λp2 q. ∃ q'∈p2. (q, a, q') ∈ Δ 𝒜
  from p1aba-eq [symmetric] p1abb-eq[symmetric] p1ab-eq[symmetric]
  have p1-eval:
     p1aa = {q ∈ p1. P p2a q} ∧
     p1ba = {q ∈ p1. ¬P p2a q} ∧
     p1ab = {q ∈ p1. P p2b q} ∧
     p1bb = {q ∈ p1. ¬P p2b q} ∧
     p1a = {q ∈ p1. P p2 q} ∧
     p1b = {q ∈ p1. ¬P p2 q}
  unfolding split-language-equiv-partition-def split-set-def P-def
  by simp
```

have *P-p2*: $\bigwedge q.\ P\ p2\ q = (P\ p2a\ q \lor P\ p2b\ q)$
  by (*simp add*: *P-def p2ab-union*[*symmetric*])
    blast

from *p2ab-dist* **have** $\bigwedge q.\ \neg(P\ p2a\ q) \lor \neg(P\ p2b\ q)$
  unfolding *P-def*
  by (*auto simp add*: δ*-in-*Δ*-iff*)
with *p1ab-eq-emp*
have *p1abb-eq-emp*: $p1aa = \{\} \lor p1ba = \{\} \longleftrightarrow p1ab = \{\} \lor p1bb = \{\}$
  by (*simp add*: *p1-eval P-p2*) *auto*
thus *?thesis*
  unfolding *split-language-equiv-partition-pred-def*
    p1abb-eq p1aba-eq
  by *simp blast*
qed

If a set @textp1 can be split, then each superset can be split as well.

lemma *split-language-equiv-partition-pred---superset-p1*:
assumes *split-pred*: *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2*
  and *p1-sub*: $p1 \subseteq p1'$
shows *split-language-equiv-partition-pred* $\mathcal{A}$ $p1'$ *a p2*
proof −
  obtain *p1a p1b* **where** *p1ab-eq*:
    split-language-equiv-partition $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
    by (*rule PairE*)
  obtain $p1a'$ $p1b'$ **where** *p1ab-eq*$'$:
    split-language-equiv-partition $\mathcal{A}$ $p1'$ *a p2* = ($p1a'$, $p1b'$)
    by (*rule PairE*)

  have $p1a \subseteq p1a' \land p1b \subseteq p1b'$
    using *p1ab-eq p1ab-eq*$'$ *p1-sub*
    unfolding *split-language-equiv-partition-def split-set-def*
    by *auto*
  with *split-pred*
  show *?thesis*
    unfolding *split-language-equiv-partition-pred-def*
      p1ab-eq p1ab-eq$'$
    by *auto*
qed

4.4.1 Splitting whole Partitions

definition *split-language-equiv-partition-set* **where**
  split-language-equiv-partition-set $\mathcal{A}$ *a p* p' =
  (*let* ($p1'$, $p2'$) = *split-language-equiv-partition* $\mathcal{A}$ p' *a p in*
    $\{p1', p2'\} \cap \{p.\ p \neq \{\}\}$)

definition *Hopcroft-split* **where**
  Hopcroft-split $\mathcal{A}$ *p a res P* =
  ($res \cup \bigcup$((*split-language-equiv-partition-set* $\mathcal{A}$ *a p*) ‘ *P*))

lemmas *Hopcroft-split-full-def* =
  Hopcroft-split-def [*unfolded split-language-equiv-partition-set-def-raw*]

lemma *Hopcroft-split-Nil* [*simp*] :
  Hopcroft-split $\mathcal{A}$ *p a res* {} = *res*
  unfolding *Hopcroft-split-def*
  by *simp*

definition *Hopcroft-split-aux* **where**
  Hopcroft-split-aux $\mathcal{A}$ *p2 a res p1* =
  (*let* (*p1a*, *p1b*) = *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2 in*

(*if* (*p1a* = {} ∧ *p1b* = {}) *then res else*
(*if p1a* = {} *then* (*insert p1b res*) *else*
(*if p1b* = {} *then* (*insert p1a res*) *else*
(*insert p1a* (*insert p1b res*))))))

lemma *Hopcroft-split-aux-alt-def* :
assumes *p1-neq-Emp*: *p1* ≠ {}
shows *Hopcroft-split-aux* 𝒜 *p2 a res p1* =
(*let* (*p1a*, *p1b*) = *split-language-equiv-partition* 𝒜 *p1 a p2 in*
(*if* (*p1a* = {} ∨ *p1b* = {}) *then* (*insert p1 res*) *else*
(*insert p1a* (*insert p1b res*))))
proof −
obtain *p1a p1b* **where** *p1ab-eq*: *split-language-equiv-partition* 𝒜 *p1 a p2* = (*p1a*, *p1b*)
by (*rule PairE*)

note *p1ab-union* = *split-language-equiv-partition-union* [*OF p1ab-eq*]
note *p1ab-disj* = *split-language-equiv-partition-disjoint* [*OF p1ab-eq*]

from *p1ab-union p1ab-disj p1ab-eq p1-neq-Emp* **show** *?thesis*
by (*simp add*: *Hopcroft-split-aux-def p1ab-eq*)
qed

lemma *Hopcroft-split-Insert* [*simp*] :
Hopcroft-split 𝒜 *p2 a res* (*insert p1 P*) =
Hopcroft-split 𝒜 *p2 a* (*Hopcroft-split-aux* 𝒜 *p2 a res p1*) *P*
by (*cases split-language-equiv-partition* 𝒜 *p1 a p2*,
simp add: *Hopcroft-split-def split-language-equiv-partition-set-def*
Hopcroft-split-aux-def Let-def)

lemma (**in** *DFA*) *Hopcroft-split-correct* :
assumes *is-part*: *is-weak-language-equiv-partition* 𝒜 (*res* ∪ *P*)
and *p2-equiv-set*: *is-weak-language-equiv-set* 𝒜 *p2*
and *a-in*: *a* ∈ Σ 𝒜
and *res-P-disjoint*: *res* ∩ *P* = {}
shows *is-weak-language-equiv-partition* 𝒜
(*Hopcroft-split* 𝒜 *p2 a res P*) (**is** *?P1 res P*)
(*Hopcroft-split* 𝒜 *p2 a res P*) ≠ (*res* ∪ *P*) ⟶
card (*res* ∪ *P*) < *card* (*Hopcroft-split* 𝒜 *p2 a res P*) (**is** *?P2 res P*)
proof −
from *is-part finite-𝒬* **have** *finite* (*res* ∪ *P*)
unfolding *is-weak-language-equiv-partition-def*
by (*metis is-partition-finite*)
hence *fin-P*: *finite P* **by** *simp*

have *finite P* ⟹ *res* ∩ *P* = {} ⟹ *is-weak-language-equiv-partition* 𝒜 (*res* ∪ *P*) ⟹ *?P1 res P* ∧ *?P2 res P*
proof (*induct arbitrary*: *res rule*: *finite-induct*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert p1 P res*)
note *p1-nin-P* = *insert*(*2*)
note *ind-hyp* = *insert*(*3*)
note *res-P-disjoint* = *insert*(*4*)
note *is-part-p1* = *insert*(*5*)

from *res-P-disjoint* **have** *p1-nin-res*: *p1* ∉ *res* **by** *auto*
from *is-part-p1*
have *p1-neq-Emp*: *p1* ≠ {} **and** *p1-sub*: *p1* ⊆ 𝒬 𝒜
unfolding *is-weak-language-equiv-partition-def*
is-partition-def
by *auto*

obtain *p1a p1b* **where** *p1ab-eq*: *split-language-equiv-partition* 𝒜 *p1 a p2* = (*p1a*, *p1b*)

  by (*cases split-language-equiv-partition* $\mathcal{A}$ *p1 a p2*, *blast*)
note *p1ab-union* = *split-language-equiv-partition-union* [*OF p1ab-eq*]

let *?res′* = *Hopcroft-split-aux* $\mathcal{A}$ *p2 a res p1*
have *pre*: *is-weak-language-equiv-partition* $\mathcal{A}$ (*?res′* ∪ *P*) ∧
          (*?res′* ∩ *P* = {}) ∧
          (((*?res′* ∪ *P*) ≠ *insert p1* (*res* ∪ *P*)) ⟶
           card (*insert p1* (*res* ∪ *P*)) < *card* (*?res′* ∪ *P*))
proof (*cases p1a* = {})
  case *True* **note** *p1a-eq* = *True*
  hence *p1b-eq*: *p1b* = *p1* **using** *p1ab-union* **by** *simp*
  show *?thesis*
    unfolding *Hopcroft-split-aux-def p1ab-eq p1a-eq p1b-eq*
    using *p1-neq-Emp is-part-p1 res-P-disjoint p1-nin-P*
    by *simp*
next
  case *False* **note** *p1a-neq-Emp* = *False*
  show *?thesis*
  proof (*cases p1b* = {})
    case *True* **note** *p1b-eq* = *True*
    hence *p1a-eq*: *p1a* = *p1* **using** *p1ab-union* **by** *simp*
    show *?thesis*
      unfolding *Hopcroft-split-aux-def p1ab-eq p1a-eq p1b-eq*
      using *p1-neq-Emp is-part-p1 res-P-disjoint p1-nin-P*
      by *simp*
  next
    case *False* **note** *p1b-neq-Emp* = *False*
    have *split-pred*: *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2*
      unfolding *split-language-equiv-partition-pred-def*
      by (*simp add*: *p1ab-eq p1a-neq-Emp p1b-neq-Emp*)

    have ⋀*p1′*. *p1′* ⊆ *p1* ⟹ *p1′* ∉ (*res* ∪ *P*)
      using *is-partition-distinct-subset*
        [*OF is-weak-language-equiv-partitionD3* [*OF is-part-p1*],
         where *p* = *p1*] *p1-nin-P p1-nin-res*
      by *simp*
    hence *p1ab-nin-P*: *p1a* ∉ *P* ∧ *p1b* ∉ *P* **using** *p1ab-union* **by** *simp*

    have (*res* ∪ *P* − {*p1*}) = *res* ∪ *P*
      using *p1-nin-res p1-nin-P* **by** *auto*
    with *split-language-equiv-partition-step* [*OF is-part-p1 - a-in
        p2-equiv-set split-pred p1ab-eq*]
    show *?thesis*
      unfolding *Hopcroft-split-aux-def p1ab-eq*
      using *p1a-neq-Emp p1b-neq-Emp p1-nin-res res-P-disjoint p1ab-nin-P*
      by *simp*
  qed
qed
note *pre1* = *conjunct1* [*OF pre*]
note *pre2* = *conjunct1* [*OF conjunct2* [*OF pre*]]
note *pre3* = *conjunct2* [*OF conjunct2* [*OF pre*]]

note *ind-hyp′* = *ind-hyp*[*OF pre2 pre1*]

from *conjunct1* [*OF ind-hyp′*]
have *P1*: *?P1 res* (*insert p1 P*) **by** *simp*

from *conjunct2* [*OF ind-hyp′*] *pre3*
have *P2*: *?P2 res* (*insert p1 P*)
  apply (*cases Hopcroft-split* $\mathcal{A}$ *p2 a* (*Hopcroft-split-aux* $\mathcal{A}$ *p2 a res p1*) *P* =
 Hopcroft-split-aux $\mathcal{A}$ *p2 a res p1* ∪ *P*)
    apply *simp*

apply (*cases Hopcroft-split-aux* $\mathcal{A}$ *p2 a res p1* $\cup$ *P* $\neq$ *insert p1* (*res* $\cup$ *P*))
apply *simp-all*
done

from *P1 P2* **show** *?case* **..**
qed
with *fin-P res-P-disjoint is-part*
show *?P1 res P ?P2 res P* **by** *simp-all*
qed

lemma (**in** *DFA*) *Hopcroft-split-correct-simple* :
assumes *is-part*: *is-weak-language-equiv-partition* $\mathcal{A}$ *P*
and *p2-in*: *p2* $\in$ *P*
and *a-in*: *a* $\in$ Σ $\mathcal{A}$
shows *is-weak-language-equiv-partition* $\mathcal{A}$ (*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
(*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*) $\neq$ *P* $\Longrightarrow$
card P < *card* (*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
proof −
from *is-part p2-in* **have**
p2-equiv-set: *is-weak-language-equiv-set* $\mathcal{A}$ *p2*
unfolding *is-weak-language-equiv-partition-def* **by** *simp*

note *Hopcroft-split-correct* [**where** *?res* = {} **and** *?P*=*P*, *OF* - *p2-equiv-set a-in*]
with *is-part* **show**
is-weak-language-equiv-partition $\mathcal{A}$
(*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
(*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*) $\neq$ *P* $\Longrightarrow$
card P < *card* (*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
by *simp-all*
qed

lemma (**in** *DFA*) *split-language-equiv-partition-pred---split-not-eq*:
assumes *p1-in-split*: *p1* $\in$ (*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
and *split-pred*: *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2'*
shows (*aa*, *p2'*) $\neq$ (*a*, *p2*)
proof −
from *p1-in-split*
obtain *p pa pb* **where**
pab-eq: *split-language-equiv-partition* $\mathcal{A}$ *p a p2* = (*pa*, *pb*) **and**
p1-eq: *p1* = *pa* $\vee$ *p1* = *pb*
unfolding *Hopcroft-split-def split-language-equiv-partition-set-def-raw*
by *auto*

obtain *p1a p1b* **where**
p1ab-eq: *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
by (*rule PairE*)

have *p1a* = *p1* $\vee$ *p1b* = *p1*
using *p1ab-eq p1-eq pab-eq*
unfolding *split-language-equiv-partition-def split-set-def*
by *auto*
with *split-language-equiv-partition-disjoint* [*OF p1ab-eq*]
split-language-equiv-partition-subset [*OF p1ab-eq*]
have *p1a* = {} $\vee$ *p1b* = {} **by** *auto*
thus *?thesis*
using *split-pred*
by (*rule-tac notI*)
(*simp add*: *split-language-equiv-partition-pred-def*
p1ab-eq)
qed

lemma *Hopcroft-split-in* :

$p \in$ *Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P* $\longleftrightarrow$
 (($p \neq$ {}) $\wedge$
  ($\exists$ *p1* $\in$ *P*. *let* (*p1a*, *p1b*) = *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2 in*
      (*p* = *p1a* $\vee$ *p* = *p1b*)))
unfolding *Hopcroft-split-def split-language-equiv-partition-set-def-raw*
apply (*simp del*: *ex-simps add*: *Bex-def ex-simps*[*symmetric*])
apply (*rule iff-exI*)
apply (*case-tac split-language-equiv-partition* $\mathcal{A}$ *x a p2*)
apply *auto*
done

definition *Hopcroft-splitted* **where**
 Hopcroft-splitted $\mathcal{A}$ *p a res P* =
 (*res* $\cup$ {(*p′*, *p1′*, *p2′*)| *p′ p1′ p2′*. *p′* $\in$ *P* $\wedge$ *p1′* $\neq$ {} $\wedge$ *p2′* $\neq$ {} $\wedge$
  (*p1′*, *p2′*) = *split-language-equiv-partition* $\mathcal{A}$ *p′ a p*})

lemma *Hopcroft-splitted-Nil* [*simp*] :
 Hopcroft-splitted $\mathcal{A}$ *p a res* {} = *res*
 unfolding *Hopcroft-splitted-def*
 by *simp*

definition *Hopcroft-splitted-aux* **where**
 Hopcroft-splitted-aux $\mathcal{A}$ *p2 a res p1* =
 (*let* (*p1a*, *p1b*) = *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2 in*
  (*if p1a* $\neq$ {} $\wedge$ *p1b* $\neq$ {} *then* (*insert* (*p1*, *p1a*, *p1b*) *res*) *else res*))

lemma *Hopcroft-splitted-Insert* [*simp*] :
Hopcroft-splitted $\mathcal{A}$ *p2 a res* (*insert p1 P*) =
 Hopcroft-splitted $\mathcal{A}$ *p2 a* (*Hopcroft-splitted-aux* $\mathcal{A}$ *p2 a res p1*) *P*
by (*cases split-language-equiv-partition* $\mathcal{A}$ *p1 a p2*,
 auto simp add: *Hopcroft-splitted-def Hopcroft-splitted-aux-def Let-def*)

lemma *Hopcroft-splitted-Insert-res* :
Hopcroft-splitted $\mathcal{A}$ *p2 a* (*insert ppp res*) *P* =
 insert ppp (*Hopcroft-splitted* $\mathcal{A}$ *p2 a res P*)
unfolding *Hopcroft-splitted-def*
by *simp*

lemma (**in** *DFA*) *Hopcroft-split-in2* :
assumes *is-part*: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) *P* **and**
  a-in: $a \in \Sigma$ $\mathcal{A}$
shows
 $p \in$ *Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P* $\longleftrightarrow$
 (($p \in P$ $\wedge$ ($\forall$ *pa pb*. (*p*, *pa*, *pb*) $\notin$ (*Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*))) $\vee$
  ($\exists$ *p1 p1a p1b*. (*p1*, *p1a*, *p1b*) $\in$ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P* $\wedge$
   (*p* = *p1a* $\vee$ *p* = *p1b*)))
 (**is** *?ls* = (*?rs1* $\vee$ *?rs2*))
proof
 assume *?ls*
 then obtain *p1* **where**
  p-not-emp: $p \neq$ {} **and**
  p1-in-P: *p1* $\in$ *P* **and**
  p-eq′: *let* (*p1a*, *p1b*) = *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2 in*
    (*p* = *p1a* $\vee$ *p* = *p1b*)
  by (*simp add*: *Hopcroft-split-in Bex-def*, *metis*)
 obtain *p1a p1b* **where** *p1ab-eq*:
  split-language-equiv-partition $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
  by (*rule PairE*)
 with *p-eq′* **have** *p-eq*: *p* = *p1a* $\vee$ *p* = *p1b* **by** *simp*

```
  from is-partition-in-subset [OF is-part p1-in-P]
  have p1-sub: p1 ⊆ 𝒬 𝒜 .

  show ?rs1 ∨ ?rs2
  proof (cases (p1, p1a, p1b) ∈ Hopcroft-splitted 𝒜 p2 a {} P)
    case True
    with p-eq have ?rs2 by blast
    thus ?thesis ..
  next
    case False note not-split = False
    hence p1ab-eq-Nil: p1a = {} ∨ p1b = {}
      unfolding Hopcroft-splitted-def
      by (simp add: p1-in-P p1ab-eq)
    with split-language-equiv-partition-union [OF p1ab-eq]
         p-not-emp p-eq
    have p1-eq: p1 = p by auto

    from p1-in-P p1-eq p1ab-eq p1ab-eq-Nil
    have ?rs1
      by (simp add: Hopcroft-splitted-def, metis)
    thus ?thesis ..
  qed
next
  assume rs12 : ?rs1 ∨ ?rs2
  show ?ls
  proof (cases ?rs1)
    case True note rs1 = this
    obtain pa pb where pab-eq:
      split-language-equiv-partition 𝒜 p a p2 = (pa, pb)
      by (rule PairE)
    with rs1
    have pab-eq-emp: pa = {} ∨ pb = {}
      unfolding Hopcroft-splitted-def
      by auto

    from rs1 is-partition-in-subset [OF is-part]
    have p-sub: p ⊆ 𝒬 𝒜 by simp

    from rs1 is-part
    have p-not-emp: p ≠ {}
      unfolding is-partition-def by auto

    from split-language-equiv-partition-union [OF pab-eq]
         pab-eq-emp p-not-emp rs1 pab-eq
    show ?ls
      apply (simp add: Hopcroft-splitted-def Hopcroft-split-in Bex-def)
      apply (rule exI [where x = p])
      apply auto
    done
  next
    case False
    with rs12 have ?rs2 by fast
    then obtain p1 p1a p1b where
       in-splitted: (p1, p1a, p1b) ∈ Hopcroft-splitted 𝒜 p2 a {} P and
       p-eq: (p = p1a ∨ p = p1b) by blast
    thus ?ls
      unfolding Hopcroft-splitted-def
      by (auto simp add: Hopcroft-split-in Bex-def)
  qed
qed

lemma (in DFA) Hopcroft-split-in2-E [consumes 3,
```

case-names in-P in-splitted]:
assumes *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) *P* $a \in \Sigma$ $\mathcal{A}$ $p \in$ *Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*
obtains (*in-P*) $p \in P$ $\bigwedge$*pa pb*. (*p*, *pa*, *pb*) $\notin$ (*Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*)
| (*in-splitted*) *p1 p1a p1b* **where**
(*p1*, *p1a*, *p1b*) $\in$ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*
$p = p1a \lor p = p1b$
using *Hopcroft-split-in2 assms*
by *blast*

lemma (**in** *DFA*) *Hopcroft-split-eq* :
assumes *split-eq*: *Hopcroft-split* $\mathcal{A}$ *p a* {} *P* = *P*
and *a-in*: $a \in \Sigma$ $\mathcal{A}$
and *is-part*: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) *P*
shows *Hopcroft-splitted* $\mathcal{A}$ *p a* {} *P* = {}
proof (*rule ccontr*)
assume *Hopcroft-splitted* $\mathcal{A}$ *p a* {} *P* $\neq$ {}
then obtain *p1 p1a p1b* **where** *in-splitted*: (*p1*, *p1a*, *p1b*) $\in$ *Hopcroft-splitted* $\mathcal{A}$ *p a* {} *P*
by *auto*
hence *p1ab-neq-Emp*: *p1a* $\neq$ {} *p1b* $\neq$ {} **and**
p1-in-P: $p1 \in P$ **and**
p1ab-eq: *split-language-equiv-partition* $\mathcal{A}$ *p1 a p* = (*p1a*, *p1b*)
unfolding *Hopcroft-splitted-def* **by** *simp-all*

from *is-partition-in-subset* [*OF is-part p1-in-P*]
have *p1-sub*: $p1 \subseteq \mathcal{Q}$ $\mathcal{A}$ **.**

note *p1ab-union* = *split-language-equiv-partition-union* [*OF p1ab-eq*]
note *p1ab-disjoint* = *split-language-equiv-partition-disjoint* [*OF p1ab-eq*]

from *p1ab-union p1ab-disjoint p1ab-neq-Emp*
have $p1a \cap p1 \neq \{\} \land p1a \neq p1$ **by** *auto*

hence *p1a-nin-P*: $p1a \notin P$
using *p1-in-P is-part*
unfolding *is-partition-def*
by *blast*

have *p1a-in-split*: *p1a* $\in$ *Hopcroft-split* $\mathcal{A}$ *p a* {} *P*
unfolding *Hopcroft-split-full-def*
apply (*simp add*: *Bex-def*)
apply (*rule exI* [**where** *x* = *p1*])
apply (*simp add*: *p1ab-eq p1-in-P p1ab-neq-Emp*(*1*))
done

from *p1a-nin-P p1a-in-split split-eq*
show *False*
by *simp*
qed

4.4.2 Updating the set of Splitters

definition *Hopcroft-update-splitters-pred-aux-upper* ::
$'a$ *set* $\Rightarrow$ ($'q$ *set* $\times$ $'q$ *set* $\times$ $'q$ *set*) *set* $\Rightarrow$ ($'q$ *set*) *set* $\Rightarrow$
($'a \times 'q$ *set*) *set* $\Rightarrow$ ($'a \times 'q$ *set*) *set* $\Rightarrow$ *bool*
where
Hopcroft-update-splitters-pred-aux-upper Q splitted P L L' $\longleftrightarrow$
($\forall$ *a p*. (*a*, *p*) $\in L'$ −−>
((*a*, *p*) $\in L \land$ ($\forall$ *pa pb*. (*p*, *pa*, *pb*) $\notin$ *splitted*)) $\lor$
($\exists$ *p' pa pb*. (*p'*, *pa*, *pb*) $\in$ *splitted* $\land$ $a \in Q$ $\land$ (*p* = *pa* $\lor$ *p* = *pb*)))

definition *Hopcroft-update-splitters-pred-aux-lower-not-splitted* ::
$'a$ *set* $\Rightarrow$ ($'q$ *set* $\times$ $'q$ *set* $\times$ $'q$ *set*) *set* $\Rightarrow$ ($'q$ *set*) *set* $\Rightarrow$

$('a \times 'q\ set)\ set \Rightarrow ('a \times 'q\ set)\ set \Rightarrow bool$
where
Hopcroft-update-splitters-pred-aux-lower-not-splitted Q *splitted* $P\ L\ L' \longleftrightarrow$
$(\forall\ a\ p.\ ((a,\ p) \in L \wedge a \in Q \wedge p \in P \wedge$
$(\forall\ pa\ pb.\ (p,\ pa,\ pb) \notin splitted))$ --->
$(a,\ p) \in L')$

definition *Hopcroft-update-splitters-pred-aux-lower-splitted-in-L* ::
$'a\ set \Rightarrow ('q\ set \times 'q\ set \times 'q\ set)\ set \Rightarrow ('q\ set)\ set \Rightarrow$
$('a \times 'q\ set)\ set \Rightarrow ('a \times 'q\ set)\ set \Rightarrow bool$
where
Hopcroft-update-splitters-pred-aux-lower-splitted-in-L Q *splitted* $P\ L\ L' \longleftrightarrow$
$(\forall\ a\ p\ pa\ pb.$
$((a,\ p) \in L \wedge a \in Q \wedge (p,\ pa,\ pb) \in splitted)$ --->
$((a,\ pa) \in L') \wedge (a,\ pb) \in L')$

definition *Hopcroft-update-splitters-pred-aux-lower-splitted* ::
$'a\ set \Rightarrow ('q\ set \times 'q\ set \times 'q\ set)\ set \Rightarrow ('q\ set)\ set \Rightarrow$
$('a \times 'q\ set)\ set \Rightarrow ('a \times 'q\ set)\ set \Rightarrow bool$
where
Hopcroft-update-splitters-pred-aux-lower-splitted Q *splitted* $P\ L\ L' \longleftrightarrow$
$(\forall\ a\ p\ pa\ pb.$
$(a \in Q \wedge (p,\ pa,\ pb) \in splitted)$ --->
$((a,\ pa) \in L') \vee (a,\ pb) \in L')$

definition *Hopcroft-update-splitters-pred-aux-lower* ::
$'a\ set \Rightarrow ('q\ set \times 'q\ set \times 'q\ set)\ set \Rightarrow ('q\ set)\ set \Rightarrow$
$('a \times 'q\ set)\ set \Rightarrow ('a \times 'q\ set)\ set \Rightarrow bool$
where
Hopcroft-update-splitters-pred-aux-lower Q *splitted* $P\ L\ L' \longleftrightarrow$
Hopcroft-update-splitters-pred-aux-lower-not-splitted Q *splitted* $P\ L\ L' \wedge$
Hopcroft-update-splitters-pred-aux-lower-splitted-in-L Q *splitted* $P\ L\ L' \wedge$
Hopcroft-update-splitters-pred-aux-lower-splitted Q *splitted* $P\ L\ L'$

lemmas *Hopcroft-update-splitters-pred-aux-lower-full-def* =
Hopcroft-update-splitters-pred-aux-lower-def
[*unfolded Hopcroft-update-splitters-pred-aux-lower-not-splitted-def*
Hopcroft-update-splitters-pred-aux-lower-splitted-def
Hopcroft-update-splitters-pred-aux-lower-splitted-in-L-def]

definition *Hopcroft-update-splitters-pred-aux* **where**
Hopcroft-update-splitters-pred-aux Q *splitted* $P\ L\ L' \longleftrightarrow$
Hopcroft-update-splitters-pred-aux-lower Q *splitted* $P\ L\ L' \wedge$
Hopcroft-update-splitters-pred-aux-upper Q *splitted* $P\ L\ L'$

lemmas *Hopcroft-update-splitters-pred-aux-full-def* =
Hopcroft-update-splitters-pred-aux-def
[*unfolded Hopcroft-update-splitters-pred-aux-lower-full-def*
Hopcroft-update-splitters-pred-aux-upper-def,
simplified]

lemma *Hopcroft-update-splitters-pred-aux-I* [*intro*!]:
⟦*Hopcroft-update-splitters-pred-aux-lower-not-splitted* Q *splitted* $P\ L\ L'$;
Hopcroft-update-splitters-pred-aux-lower-splitted Q *splitted* $P\ L\ L'$;
Hopcroft-update-splitters-pred-aux-lower-splitted-in-L Q *splitted* $P\ L\ L'$;
Hopcroft-update-splitters-pred-aux-upper Q *splitted* $P\ L\ L'$⟧ $\Longrightarrow$
Hopcroft-update-splitters-pred-aux Q *splitted* $P\ L\ L'$
unfolding *Hopcroft-update-splitters-pred-aux-def*
Hopcroft-update-splitters-pred-aux-lower-def
by *simp*

lemma *Hopcroft-update-splitters-pred-aux-upper---is-upper* :
$[\![$*Hopcroft-update-splitters-pred-aux-upper Q splitted P L* L'; $L'' \subseteq L'$$]\!] \Longrightarrow$
Hopcroft-update-splitters-pred-aux-upper Q splitted P L L''
unfolding *Hopcroft-update-splitters-pred-aux-upper-def*
by *auto*

lemma *Hopcroft-update-splitters-pred-aux-lower---is-lower* :
$[\![$*Hopcroft-update-splitters-pred-aux-lower Q splitted P L* L'; $L' \subseteq L''$$]\!] \Longrightarrow$
Hopcroft-update-splitters-pred-aux-lower Q splitted P L L''
unfolding *Hopcroft-update-splitters-pred-aux-lower-full-def*
by *blast*

lemma *Hopcroft-update-splitters-pred-aux-Emp-Id* :
Hopcroft-update-splitters-pred-aux Q {} *P L L*
unfolding *Hopcroft-update-splitters-pred-aux-def*
by (*simp add*: *Hopcroft-update-splitters-pred-aux-upper-def*
Hopcroft-update-splitters-pred-aux-lower-full-def)

definition *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred* **where**
Hopcroft-update-splitters-pred-aux-splitted-equiv-pred splitted1 splitted2 $\longleftrightarrow$
($\forall$ *p2 p2a p2b*. (*p2*, *p2a*, *p2b*) $\in$ *splitted1* $\longrightarrow$
(*p2*, *p2a*, *p2b*) $\in$ *splitted2* $\vee$ (*p2*, *p2b*, *p2a*) $\in$ *splitted2*) $\wedge$
($\forall$ *p2 p2a p2b*. (*p2*, *p2a*, *p2b*) $\in$ *splitted2* $\longrightarrow$
(*p2*, *p2a*, *p2b*) $\in$ *splitted1* $\vee$ (*p2*, *p2b*, *p2a*) $\in$ *splitted1*)

lemma *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-nin* :
Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2 $\Longrightarrow$
($\forall$ *pa pb*. (*p*, *pa*, *pb*) $\notin$ *sp1*) $\longleftrightarrow$ ($\forall$ *pa pb*. (*p*, *pa*, *pb*) $\notin$ *sp2*)
unfolding *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-def*
by *metis*

lemma *Hopcroft-update-splitters-pred-aux-splitted-upper-swap-rule* :
assumes *equiv-pred*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2*
and *pre*: *Hopcroft-update-splitters-pred-aux-upper Q sp1 P L* L'
shows *Hopcroft-update-splitters-pred-aux-upper Q sp2 P L* L'
using *pre*
unfolding *Hopcroft-update-splitters-pred-aux-upper-def*
apply (*simp add*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-nin* [*OF equiv-pred*])
apply (*insert equiv-pred* [*unfolded Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-def*])
apply *metis*
done

lemma *Hopcroft-update-splitters-pred-aux-splitted-lower-not-splitted-swap-rule* :
assumes *equiv-pred*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2*
and *pre*: *Hopcroft-update-splitters-pred-aux-lower-not-splitted Q sp1 P L* L'
shows *Hopcroft-update-splitters-pred-aux-lower-not-splitted Q sp2 P L* L'
using *pre*
unfolding *Hopcroft-update-splitters-pred-aux-lower-not-splitted-def*
by (*simp add*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-nin* [*OF equiv-pred*])

lemma *Hopcroft-update-splitters-pred-aux-splitted-lower-splitted-swap-rule* :
assumes *equiv-pred*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2*
and *pre*: *Hopcroft-update-splitters-pred-aux-lower-splitted Q sp1 P L* L'
shows *Hopcroft-update-splitters-pred-aux-lower-splitted Q sp2 P L* L'
using *pre equiv-pred* [*unfolded Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-def*]
unfolding *Hopcroft-update-splitters-pred-aux-lower-splitted-def*
by *metis*

lemma *Hopcroft-update-splitters-pred-aux-splitted-lower-splitted-in-L-swap-rule* :
assumes *equiv-pred*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2*
and *pre*: *Hopcroft-update-splitters-pred-aux-lower-splitted-in-L Q sp1 P L* L'

shows *Hopcroft-update-splitters-pred-aux-lower-splitted-in-L Q sp2 P L L′*
using *pre equiv-pred* [*unfolded Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-def*]
unfolding *Hopcroft-update-splitters-pred-aux-lower-splitted-in-L-def*
by *metis*

lemma *Hopcroft-update-splitters-pred-aux-splitted-swap-rule* :
assumes *equiv-pred*: *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2*
  and *pre*: *Hopcroft-update-splitters-pred-aux Q sp1 P L L′*
shows *Hopcroft-update-splitters-pred-aux Q sp2 P L L′*
using *Hopcroft-update-splitters-pred-aux-splitted-upper-swap-rule* [*OF equiv-pred*, *of Q P L L′*]
  Hopcroft-update-splitters-pred-aux-splitted-lower-splitted-swap-rule
   [*OF equiv-pred*, *of Q P L L′*]
  Hopcroft-update-splitters-pred-aux-splitted-lower-splitted-in-L-swap-rule
   [*OF equiv-pred*, *of Q P L L′*]
  Hopcroft-update-splitters-pred-aux-splitted-lower-not-splitted-swap-rule
   [*OF equiv-pred*, *of Q P L L′*]
  pre
unfolding *Hopcroft-update-splitters-pred-aux-def*
   Hopcroft-update-splitters-pred-aux-lower-def
by *simp*

definition *Hopcroft-update-splitters-pred* ::
 $('q, 'a, 'x)$ *NFA-rec-scheme* $\Rightarrow$ $'q$ *set* $\Rightarrow$ $'a \Rightarrow ('q$ *set*$)$ *set* $\Rightarrow$
   $('a \times 'q$ *set*$)$ *set* $\Rightarrow$ $('a \times 'q$ *set*$)$ *set* $\Rightarrow$ *bool*
where
 Hopcroft-update-splitters-pred $\mathcal{A}$ *pp aa P L L′* $\longleftrightarrow$
 Hopcroft-update-splitters-pred-aux $(\Sigma\ \mathcal{A})$ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*) *P*
  $(L - \{(aa, pp)\})$ L'

lemma *Hopcroft-update-splitters-pred-exists*:
 $\exists L'$. *Hopcroft-update-splitters-pred* $\mathcal{A}$ *pp aa P L L′*
proof −
 let *?L′* =
  $\{(a, p) \mid a\ p.\ (a, p) \in L - \{(aa, pp)\} \wedge$
  $(\forall pa\ pb.\ (p, pa, pb) \notin$ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)$)\} \cup$
  $\{(a, pa) \mid a\ p\ pa\ pb.\ a \in \Sigma\ \mathcal{A} \wedge (p, pa, pb) \in$ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)$\} \cup$
  $\{(a, pb) \mid a\ p\ pa\ pb.\ a \in \Sigma\ \mathcal{A} \wedge (p, pa, pb) \in$ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)$\}$

 have *Hopcroft-update-splitters-pred* $\mathcal{A}$ *pp aa P L ?L′*
  unfolding *Hopcroft-update-splitters-pred-def*
     Hopcroft-update-splitters-pred-aux-full-def
  by *auto*
 thus *?thesis* **by** *blast*
qed

lemma *Hopcroft-update-splitters-pred-in-E2* [*consumes 2*, *case-names no-split split*]:
assumes *L′-OK*: *Hopcroft-update-splitters-pred* $\mathcal{A}$ *pp aa P L L′*
  and *in-L′*: $(a, p) \in L'$
obtains (*no-split*) $(a, p) \in L - \{(aa, pp)\}$ $\bigwedge pa\ pb.\ (p, pa, pb) \notin$ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)
  | (*split*) *p′ pa pb* **where** $(p', pa, pb) \in$ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)
      $a \in \Sigma\ \mathcal{A}$ $p = pa \vee p = pb$
proof −
 from *L′-OK* **have** *Hopcroft-update-splitters-pred-aux-upper* $(\Sigma\ \mathcal{A})$
     (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*) *P* $(L - \{(aa, pp)\})$ L'
  unfolding *Hopcroft-update-splitters-pred-def*
     Hopcroft-update-splitters-pred-aux-def
  by *fast*
 with *in-L′ split no-split*
 show *?thesis* **unfolding** *Hopcroft-update-splitters-pred-aux-upper-def*
 by *blast*

qed

lemma *Hopcroft-update-splitters-pred---nin-splitted-in-L* :
assumes L'*-OK*: *Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L* L'
  and *p-in*: $p \in P$
  and *aa-in*: $aa \in \Sigma\ \mathcal{A}$
  and *nin-splitted*: $\bigwedge pa\ pb.\ (p,\ pa,\ pb) \notin$ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*
  and *in-L*: $(aa,\ p) \in L - \{(a,\ p2)\}$
shows $(aa,\ p) \in L'$
proof –
  from L'*-OK* **have** *Hopcroft-update-splitters-pred-aux-lower-not-splitted* $(\Sigma\ \mathcal{A})$ (*Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*) *P* $(L - \{(a,\ p2)\})\ L'$
    unfolding *Hopcroft-update-splitters-pred-def*
      Hopcroft-update-splitters-pred-aux-def
      Hopcroft-update-splitters-pred-aux-lower-def
    by *simp*

  with *in-L aa-in p-in nin-splitted*
  show *?thesis*
  unfolding *Hopcroft-update-splitters-pred-aux-lower-not-splitted-def*
  by *blast*
qed

lemma *Hopcroft-update-splitters-pred---in-splitted-in-L* :
  ⟦*Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L* L'; $aa \in \Sigma\ \mathcal{A}$;
  $(p,\ pa,\ pb) \in$ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*; $(aa,\ p) \in L - \{(a,\ p2)\}$⟧ $\Longrightarrow$
  $(aa,\ pa) \in L' \wedge (aa,\ pb) \in L'$
unfolding *Hopcroft-update-splitters-pred-def*
  Hopcroft-update-splitters-pred-aux-def
  Hopcroft-update-splitters-pred-aux-lower-def
  Hopcroft-update-splitters-pred-aux-lower-splitted-in-L-def
by *metis*

lemma *Hopcroft-update-splitters-pred---in-splitted* :
  ⟦*Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L* L'; $aa \in \Sigma\ \mathcal{A}$;
  $(p,\ pa,\ pb) \in$ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*⟧ $\Longrightarrow$
  $(aa,\ pa) \in L' \vee (aa,\ pb) \in L'$
unfolding *Hopcroft-update-splitters-pred-def*
  Hopcroft-update-splitters-pred-aux-def
  Hopcroft-update-splitters-pred-aux-lower-def
  Hopcroft-update-splitters-pred-aux-lower-splitted-def
by *metis*

lemma *Hopcroft-update-splitters-pred---label-in* :
⟦*Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L* L';
  $\bigwedge ap.\ ap \in L \Longrightarrow$ *fst ap* $\in \Sigma\ \mathcal{A}$;
  $(a',\ p) \in L'$⟧ $\Longrightarrow a' \in (\Sigma\ \mathcal{A})$
by (*rule Hopcroft-update-splitters-pred-in-E2*) *auto*

lemma *Hopcroft-update-splitters-pred---splitted-emp* :
assumes L'*-OK*: *Hopcroft-update-splitters-pred* $\mathcal{A}$ *p a P L* L'
  and *L-OK*: $\bigwedge ap.\ ap \in L \Longrightarrow$ *fst ap* $\in \Sigma\ \mathcal{A} \wedge$ *snd ap* $\in P$
  and *splitted-emp*: *Hopcroft-splitted* $\mathcal{A}$ *p a* {} *P* = {}
shows $L' = L - \{(a,\ p)\}$
proof (*intro set-eqI iffI*)
  fix *aap2*
  assume *aap2-in*: $aap2 \in L - \{(a,\ p)\}$
  obtain *aa p2* **where** *aap2-eq* [*simp*]: $aap2 = (aa,\ p2)$ **by** (*rule PairE*)

  from *L-OK*[*of aap2*] *aap2-in* **have** *aap2-wf*: $aa \in \Sigma\ \mathcal{A}\ p2 \in P$ **by** *simp-all*

  from *aap2-in splitted-emp*

Hopcroft-update-splitters-pred---nin-splitted-in-L
[*OF* L'-*OK* *aap2-wf* (*2*) *aap2-wf* (*1*)]
show $aap2 \in L'$
by *simp*
next
fix *aap2*
assume *aap2-in*: $aap2 \in L'$
obtain *aa p2* **where** *aap2-eq* [*simp*]: $aap2 = (aa, p2)$ **by** (*rule PairE*)

show $aap2 \in L - \{(a, p)\}$
apply (*rule Hopcroft-update-splitters-pred-in-E2* [*OF* L'-*OK* *aap2-in*[*unfolded aap2-eq*]])
apply (*simp-all add*: *splitted-emp*)
done
qed

4.4.3 The abstract Algorithm

definition *Hopcroft-abstract-invar* **where**
Hopcroft-abstract-invar $\mathcal{A} = (\lambda(P, L)$.
is-weak-language-equiv-partition $\mathcal{A}$ P $\wedge$
$(\forall ap \in L.\ fst\ ap \in \Sigma\ \mathcal{A} \wedge snd\ ap \in P) \wedge$
$(\forall p1\ a\ p2.\ (a \in \Sigma\ \mathcal{A} \wedge (\exists p1' \in P.\ p1 \subseteq p1') \wedge p2 \in P \wedge$
split-language-equiv-partition-pred $\mathcal{A}$ *p1 a p2*) $\longrightarrow$
$(\exists p2'.\ (a, p2') \in L \wedge$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2'*)))

lemma *Hopcroft-abstract-invarI* [*intro*!] :
⟦*is-weak-language-equiv-partition* $\mathcal{A}$ P;
$(\bigwedge a\ p.\ (a,p) \in L \Longrightarrow a \in \Sigma\ \mathcal{A} \wedge p \in P)$;
$(\bigwedge p1\ a\ p2.$ ⟦*is-weak-language-equiv-partition* $\mathcal{A}$ P;
$\bigwedge a\ p.\ (a,p) \in L \Longrightarrow a \in \Sigma\ \mathcal{A} \wedge p \in P$;
$a \in \Sigma\ \mathcal{A}$; $p2 \in P$; $\exists p1' \in P.\ p1 \subseteq p1'$;
split-language-equiv-partition-pred $\mathcal{A}$ *p1 a p2*⟧ $\Longrightarrow$
$(\exists p2'.\ (a, p2') \in L \wedge$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2'*))⟧ $\Longrightarrow$
Hopcroft-abstract-invar $\mathcal{A}$ (P, L)
unfolding *Hopcroft-abstract-invar-def*
by *auto metis*

definition *Hopcroft-abstract-init* $\mathcal{A} \equiv$ (*Hopcroft-accepting-partition* $\mathcal{A}$, $(\Sigma\ \mathcal{A} \times \{\mathcal{F}\ \mathcal{A}\})$)

lemma (**in** *DFA*) *Hopcroft-abstract-invar-init* :
assumes $\mathcal{F}$-*OK*: $\mathcal{F}\ \mathcal{A} \neq \{\}$
shows *Hopcroft-abstract-invar* $\mathcal{A}$ (*Hopcroft-abstract-init* $\mathcal{A}$)
unfolding *Hopcroft-abstract-init-def*
proof
show *is-weak-language-equiv-partition* $\mathcal{A}$ (*Hopcroft-accepting-partition* $\mathcal{A}$)
by (*rule is-weak-language-equiv-partition-init*)
next
fix *a p*
assume $(a,p) \in \Sigma\ \mathcal{A} \times \{\mathcal{F}\ \mathcal{A}\}$
thus $a \in \Sigma\ \mathcal{A} \wedge p \in$ *Hopcroft-accepting-partition* $\mathcal{A}$
unfolding *Hopcroft-accepting-partition-alt-def* [*OF wf-NFA*]
using $\mathcal{F}$-*OK*
by *simp*
next
fix *p1 a p2*
let *?P* = *Hopcroft-accepting-partition* $\mathcal{A}$
assume *p2-in*: $p2 \in$ *?P*
and *split-pred*: *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2*
and *a-in*: $a \in \Sigma\ \mathcal{A}$
and *p1-sub-in*: $\exists p1' \in$ *Hopcroft-accepting-partition* $\mathcal{A}$. $p1 \subseteq p1'$

have *p1-sub*: $p1 \subseteq \mathcal{Q}\ \mathcal{A}$

proof –
from *p1-sub-in*
obtain *p1′* **where** *p1′-in*: $p1' \in$ *Hopcroft-accepting-partition* $\mathcal{A}$
and *p1-sub*: $p1 \subseteq p1'$
by *blast*
from *p1′-in* **have** $p1' \subseteq \mathcal{Q}\ \mathcal{A}$
unfolding *Hopcroft-accepting-partition-alt-def* [*OF wf-NFA*]
using $\mathcal{F}$*-consistent*
by *blast*
with *p1-sub* **show** $p1 \subseteq \mathcal{Q}\ \mathcal{A}$ **by** *simp*
qed
have *no-split-*$\mathcal{Q}$: ¬ (*split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a* ($\mathcal{Q}\ \mathcal{A}$))
unfolding *split-language-equiv-partition-pred-def*
split-language-equiv-partition-alt-def [*OF p1-sub a-in*]
by (*simp add*: δ*-in-*Δ*-iff*) (*metis* δ*-wf*)

have *pred-eq*:
split-language-equiv-partition-pred $\mathcal{A}$ *p1 a* ($\mathcal{F}\ \mathcal{A}$) =
split-language-equiv-partition-pred $\mathcal{A}$ *p1 a* ($\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}$)
apply (*rule split-language-equiv-partition-pred-split-neg*
[*OF - - no-split-*$\mathcal{Q}$])
apply (*insert* $\mathcal{F}$*-consistent*)
apply *auto*
done

with *split-pred p2-in a-in*
have $(a, \mathcal{F}\ \mathcal{A}) \in \Sigma\ \mathcal{A} \times \{\mathcal{F}\ \mathcal{A}\} \wedge$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a* ($\mathcal{F}\ \mathcal{A}$)
unfolding *Hopcroft-accepting-partition-alt-def* [*OF wf-NFA*]
by *auto*
thus $\exists\, p2'.\ (a,\ p2') \in \Sigma\ \mathcal{A} \times \{\mathcal{F}\ \mathcal{A}\} \wedge$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2′*
by *blast*
qed

definition *Hopcroft-abstract-b* **where**
Hopcroft-abstract-b PL = (*snd PL* ≠ {})

definition *Hopcroft-abstract-f* **where**
Hopcroft-abstract-f $\mathcal{A}$ =
(λ(*P*, *L*). *do* {
ASSERT (*Hopcroft-abstract-invar* $\mathcal{A}$ (*P*, *L*));
ASSERT (*L* ≠ {});
(*a*,*p*) ← *SPEC* (λ(*a*,*p*). (*a*,*p*) ∈ *L*);
(*P′*, *L′*) ← *SPEC* (λ(*P′*, *L′*). *Hopcroft-update-splitters-pred* $\mathcal{A}$ *p a P L L′* ∧
(*P′* = *Hopcroft-split* $\mathcal{A}$ *p a* {} *P*));
RETURN (*P′*, *L′*)
})

definition *Hopcroft-abstract* **where**
Hopcroft-abstract $\mathcal{A}$ =
(*if* ($\mathcal{Q}\ \mathcal{A}$ = {}) *then RETURN* {} *else* (
if ($\mathcal{F}\ \mathcal{A}$ = {}) *then RETURN* {$\mathcal{Q}\ \mathcal{A}$} *else* (
do {
(*P*, -) ← *WHILEIT* (*Hopcroft-abstract-invar* $\mathcal{A}$) *Hopcroft-abstract-b*
(*Hopcroft-abstract-f* $\mathcal{A}$) (*Hopcroft-abstract-init* $\mathcal{A}$);
RETURN P
})))

lemma (**in** *DFA*) *Hopcroft-abstract-invar-OK*:
WHILE-invar-OK Hopcroft-abstract-b (*Hopcroft-abstract-f* $\mathcal{A}$) (*Hopcroft-abstract-invar* $\mathcal{A}$)
unfolding *Hopcroft-abstract-f-def Hopcroft-abstract-b-def-raw*
apply (*rule WHILE-invar-OK-I*)

apply (*simp add*: *prod-case-beta*)
apply (*intro refine-vcg*)
apply (*simp add*: *image-iff*)
apply (*clarify*)
apply (*simp*)
proof −
  fix P L a $p2$ L'
  assume *invar-PL*: *Hopcroft-abstract-invar* $\mathcal{A}$ (P, L)
  assume *p2a-in*: $(a, p2) \in L$
  assume L'*-OK*: *Hopcroft-update-splitters-pred* $\mathcal{A}$ $p2$ a P L L'

  from *invar-PL p2a-in*
  have *a-in*: $a \in \Sigma\ \mathcal{A}$ **and** *p2-in-P*: $p2 \in P$
    unfolding *Hopcroft-abstract-invar-def*
    by (*simp-all add*: *Ball-def*)

  from *invar-PL* **have**
    is-part: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) P
    unfolding *Hopcroft-abstract-invar-def is-weak-language-equiv-partition-def*
    by *simp*

  show *Hopcroft-abstract-invar* $\mathcal{A}$ (*Hopcroft-split* $\mathcal{A}$ $p2$ a $\{\}$ P, L')
  proof (*rule Hopcroft-abstract-invarI*)
    from *Hopcroft-split-correct-simple* [*of* P $p2$ a] *a-in p2-in-P invar-PL*
    show
      is-weak-language-equiv-partition $\mathcal{A}$ (*Hopcroft-split* $\mathcal{A}$ $p2$ a $\{\}$ P)
      unfolding *Hopcroft-abstract-invar-def*
      by *simp*
  next
    fix a' p
    assume a'*p-in*: $(a', p) \in L'$

    have a'*-in*: $a' \in \Sigma\ \mathcal{A}$
      using *Hopcroft-update-splitters-pred---label-in* [*OF* L'*-OK* - a'*p-in*]
            invar-PL
      unfolding *Hopcroft-abstract-invar-def*
      by (*auto simp add*: *Ball-def*)

    note *p-in-split* = *Hopcroft-split-in2* [*OF is-part a-in*, *of* p $p2$]

    from L'*-OK* a'*p-in* **have** *p-in*: $p \in$ *Hopcroft-split* $\mathcal{A}$ $p2$ a $\{\}$ P
    proof (*cases rule*: *Hopcroft-update-splitters-pred-in-E2*)
      case *no-split*
      from *no-split*(*1*) *invar-PL*
      have $p \in P$
        unfolding *Hopcroft-abstract-invar-def*
        by *auto*
      with *no-split*(*2*)
      show *?thesis*
        unfolding *p-in-split* **by** *simp*
    next
      case (*split p0 pa pb*)
      thus *?thesis*
        unfolding *p-in-split*
        by *auto*
    qed

    from a'*-in p-in*
    show $a' \in \Sigma\ \mathcal{A} \wedge p \in$ *Hopcroft-split* $\mathcal{A}$ $p2$ a $\{\}$ P
      by *simp*
  next
    fix $p1$ aa $p2'$

```
let ?P′ = Hopcroft-split 𝒜 p2 a {} P
assume p1-subset-P′: ∃ p1′ ∈ ?P′. p1 ⊆ p1′
  and p2′-in-P′: p2′ ∈ ?P′
  and aa-in: aa ∈ Σ 𝒜
  and split-pred: split-language-equiv-partition-pred 𝒜 p1 aa p2′
  and weak-part: is-weak-language-equiv-partition 𝒜 ?P′
  and L′-invar: ⋀a p. (a, p) ∈ L′ ⟹ a ∈ Σ 𝒜 ∧ p ∈ ?P′

let ?in-L′-ex = ∃ p2′′. (aa, p2′′) ∈ L′ ∧ split-language-equiv-partition-pred 𝒜 p1 aa p2′′
have in-L-impl: ⋀p2′′. ⟦(aa, p2′′) ∈ L; split-language-equiv-partition-pred 𝒜 p1 aa p2′′⟧
          ⟹ ?in-L′-ex
proof −
 fix p2′′
 assume aa-p2′′-in-L: (aa, p2′′) ∈ L and
      split-pred-p2′′: split-language-equiv-partition-pred 𝒜 p1 aa p2′′

 from aa-p2′′-in-L invar-PL
 have p2′′-in-P: p2′′ ∈ P
   unfolding Hopcroft-abstract-invar-def by auto

 thus ?in-L′-ex
 proof (cases (aa, p2′′) ∈ L′)
   case True with split-pred-p2′′
   show ?thesis by blast
 next
   case False note aa-p2′′-nin-L′ = this
   have aa-p2′′-neq: (aa, p2′′) ≠ (a, p2)
   proof −
     from p1-subset-P′ obtain p1′ where
       p1′-in-P′: p1′ ∈ ?P′ and
       p1-sub: p1 ⊆ p1′ by blast

     from split-language-equiv-partition-pred---superset-p1 [OF split-pred-p2′′ p1-sub]
        p1′-in-P′
        split-language-equiv-partition-pred---split-not-eq
     show ?thesis by blast
   qed
   with aa-p2′′-nin-L′ aa-p2′′-in-L
   have (∃ pa pb. (p2′′, pa, pb) ∈ Hopcroft-splitted 𝒜 p2 a {} P)
      using Hopcroft-update-splitters-pred---nin-splitted-in-L [OF L′-OK p2′′-in-P aa-in]
      by blast
   then obtain p2a′′ p2b′′ where p2′′-in-split:
      (p2′′, p2a′′, p2b′′) ∈ Hopcroft-splitted 𝒜 p2 a {} P by blast

   have p2ab′′-in-L′: (aa, p2a′′) ∈ L′ ∧ (aa, p2b′′) ∈ L′
     using aa-p2′′-neq aa-p2′′-in-L
     by (rule-tac Hopcroft-update-splitters-pred---in-splitted-in-L
               [OF L′-OK aa-in p2′′-in-split])
        simp

   have split-language-equiv-partition-pred 𝒜 p1 aa p2a′′ ∨
        split-language-equiv-partition-pred 𝒜 p1 aa p2b′′
   proof (rule split-language-equiv-partition-pred-split [OF - split-pred-p2′′])
     from p2′′-in-split
       have p2ab′′-eq: split-language-equiv-partition 𝒜 p2′′ a p2 = (p2a′′, p2b′′)
       unfolding Hopcroft-splitted-def
       by simp-all

     from is-partition-in-subset[OF is-part p2′′-in-P]
     have p′′-subset: p2′′ ⊆ 𝒬 𝒜 .

     from split-language-equiv-partition-union [OF p2ab′′-eq]
```

show $p2a'' \cup p2b'' = p2''$ **by** *simp*
qed
with *p2ab''-in-L'* **show** *?thesis* **by** *blast*
qed
qed

have *in-P-impl* :
$\bigwedge p2''$. $[\![p2'' \in P;$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2''*$]\!] \Longrightarrow$
?in-L'-ex
proof −
fix *p2''*
assume *p2''-in-P* : $p2'' \in P$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2''*

have *p1-subset-exists*: $\exists p1' \in P.\ p1 \subseteq p1'$
proof −
from *p1-subset-P'* **obtain** *p1'* **where**
p1'-in-P': $p1' \in ?P'$ **and**
p1-sub: $p1 \subseteq p1'$ **by** *blast*

from *p1'-in-P'* **obtain** *p1'' p1a' p1b'*
where *p1ab'-eq*: *split-language-equiv-partition* $\mathcal{A}$ *p1'' a p2* = (*p1a'*, *p1b'*)
and *p1'-eq*: $p1' = p1a' \vee p1' = p1b'$
and *p1''-in-P*: $p1'' \in P$
by (*auto simp add*: *Hopcroft-split-in*)
from *is-partition-in-subset* [*OF is-part p1''-in-P*]
split-language-equiv-partition-union [*OF p1ab'-eq*]
have *p1''-eq* : $p1'' = p1a' \cup p1b'$ **by** *fast*
hence *p1-subset*: $p1 \subseteq p1''$
using *p1'-eq p1-sub* **by** *blast*

from *p1-subset p1''-in-P*
show *?thesis* **by** *blast*
qed

with *aa-in invar-PL p1-subset-exists p2''-in-P*
obtain *p2'''* **where**
$(aa,\ p2''') \in L$ **and**
split-language-equiv-partition-pred $\mathcal{A}$ *p1 aa p2'''*
unfolding *Hopcroft-abstract-invar-def*
by *blast*
with *in-L-impl* **show** *?in-L'-ex*
by *simp*
qed

show *?in-L'-ex*
using *is-part a-in p2'-in-P'*
proof (*cases rule*: *Hopcroft-split-in2-E*)
case *in-P*
from *in-P(1) in-P-impl split-pred* **show** *?thesis*
by *blast*
next
case (*in-splitted p2-0 p2-0a p2-0b*)
note *p2-0-in-splitted* = *in-splitted(1)*
note *p2'-eq* = *in-splitted(2)*
def *p2'-inv* ≡ *if p2' = p2-0a then p2-0b else p2-0a*

from *p2-0-in-splitted* **have** *p2-0-in-P*: $p2\text{-}0 \in P$
unfolding *Hopcroft-splitted-def* **by** *simp*

show *?thesis*
proof (*cases* $(aa,\ p2') \in L' \vee$ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2-0*)
case *True* **with** *p2-0-in-P in-P-impl*

show *?thesis*
using *split-pred* **by** *blast*
next
case *False*
hence *aa-p2'-nin-L'* : (*aa*, *p2'*) $\notin$ *L'*
and *no-split-p2-0*: $\neg$(*split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2-0*)
by *simp-all*

from *Hopcroft-update-splitters-pred---in-splitted*
[*OF L'-OK aa-in p2-0-in-splitted*]
p2'-eq aa-p2'-nin-L'
have *aa-p2'-inv-in-L'*: (*aa*, *p2'-inv*) $\in$ *L'*
unfolding *p2'-inv-def* **by** *auto*

from *is-partition-in-subset* [*OF is-part p2-0-in-P*] **have**
p2-0-subset: *p2-0* $\subseteq$ $\mathcal{Q}$ $\mathcal{A}$ **.**

from *p2-0-in-splitted* **have** *split*: *split-language-equiv-partition* $\mathcal{A}$ *p2-0 a p2* =
(*p2-0a*, *p2-0b*)
unfolding *Hopcroft-splitted-def*
by *simp*
note *p2-0ab-union* = *split-language-equiv-partition-union* [*OF split*]
note *p2-0ab-disj* = *split-language-equiv-partition-disjoint* [*OF split*]

from *split-language-equiv-partition-pred-split-neg* [*OF p2-0ab-union*[*symmetric*]
p2-0ab-disj no-split-p2-0]
have *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2-0a* =
split-language-equiv-partition-pred $\mathcal{A}$ *p1 aa p2-0b* **.**
with *split-pred p2'-eq*
have *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2-0a* $\wedge$
split-language-equiv-partition-pred $\mathcal{A}$ *p1 aa p2-0b* **by** *blast*
with *p2'-inv-def* **have**
split-language-equiv-partition-pred $\mathcal{A}$ *p1 aa p2'-inv* **by** *metis*
with *aa-p2'-inv-in-L'* **show** *?thesis* **by** *blast*
qed
qed
qed
qed

lemma (**in** *NFA*) *Hopcroft-abstract-invar---implies-finite-L*:
assumes *invar*: *Hopcroft-abstract-invar* $\mathcal{A}$ *PL*
shows *finite* (*snd PL*)
proof −
obtain *P L* **where** *PL-eq*[*simp*]: *PL* = (*P*, *L*) **by** (*rule PairE*)

from *invar*
have *L-sub*: *L* $\subseteq$ Σ $\mathcal{A}$ $\times$ *P* **and** *part-P*: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) *P*
unfolding *Hopcroft-abstract-invar-def is-weak-language-equiv-partition-def*
by *auto*

from *is-partition-finite* [*OF finite-*$\mathcal{Q}$ *part-P*] *finite-*Σ
have *finite* ((Σ $\mathcal{A}$) $\times$ *P*) **by** *auto*
with *L-sub* **show** *finite* (*snd PL*) **by** (*simp*, *rule finite-subset*)
qed

definition *Hopcroft-abstract-variant* **where**
Hopcroft-abstract-variant $\mathcal{A}$ = (*measure* (λP. *card* ($\mathcal{Q}$ $\mathcal{A}$) − *card P*)) <**lex**> (*measure card*)

lemma (**in** *DFA*) *Hopcroft-abstract-variant-OK*:
WHILE-variant-OK Hopcroft-abstract-b (*Hopcroft-abstract-f* $\mathcal{A}$) (*Hopcroft-abstract-invar* $\mathcal{A}$)
(*Hopcroft-abstract-variant* $\mathcal{A}$)
proof (*rule WHILE-variant-OK---extend-invariant*)

show *WHILE-invar-OK Hopcroft-abstract-b* (*Hopcroft-abstract-f* $\mathcal{A}$) (*Hopcroft-abstract-invar* $\mathcal{A}$)
  by (*fact Hopcroft-abstract-invar-OK*)
show *wf* (*Hopcroft-abstract-variant* $\mathcal{A}$)
unfolding *Hopcroft-abstract-variant-def*
by (*intro wf-lex-prod wf-measure*)
next
fix *PL*
assume *invar* : *Hopcroft-abstract-invar* $\mathcal{A}$ *PL*
  and *cond*: *Hopcroft-abstract-b PL*
obtain *P L* **where** *PL-eq*[*simp*]: *PL* = (*P*, *L*) **by** (*rule PairE*)

from *invar cond*
show *Hopcroft-abstract-f* $\mathcal{A}$ *PL* $\leq$ *SPEC* ($\lambda PL'$. (PL', *PL*) $\in$ *Hopcroft-abstract-variant* $\mathcal{A}$)
unfolding *Hopcroft-abstract-f-def Hopcroft-abstract-b-def PL-eq*
apply (*simp only*: *prod-case-beta*)
apply (*intro refine-vcg*)
apply (*simp-all*)
apply (*clarify*)
proof −
  fix *a p* L'
  assume *ap-in*: (*a*, *p*) $\in$ *L*
    and L'*-OK*: *Hopcroft-update-splitters-pred* $\mathcal{A}$ *p a P L* L'
  let *?P'* = *Hopcroft-split* $\mathcal{A}$ *p a* {} *P*

  from *ap-in invar*
  have *a-in*: *a* $\in$ Σ $\mathcal{A}$ **and**
      p-in: *p* $\in$ *P* **and**
      equiv-part-P: *is-weak-language-equiv-partition* $\mathcal{A}$ *P*
    unfolding *Hopcroft-abstract-invar-def*
    by *auto*
  from *equiv-part-P* **have** *part-P*: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) *P*
    unfolding *is-weak-language-equiv-partition-def* **by** *fast*

  show ((*?P'*, L'), (*P*, *L*)) $\in$ *Hopcroft-abstract-variant* $\mathcal{A}$
  proof (*cases ?P'* = *P*)
    case *True* **note** *split-eq* = *this*

    from *Hopcroft-split-eq* [*OF split-eq a-in part-P*]
    have *Hopcroft-splitted* $\mathcal{A}$ *p a* {} *P* = {} **.**
    hence L'*-eq*: L' = *L* − {(*a*, *p*)}
      using *invar*
      using *Hopcroft-update-splitters-pred---splitted-emp* [*OF* L'*-OK*]
      unfolding *Hopcroft-abstract-invar-def PL-eq*
      by *simp*

    have *fin-L*: *finite L*
      using *Hopcroft-abstract-invar---implies-finite-L* [*OF invar*] **by** *simp*
    show *?thesis*
      unfolding *Hopcroft-abstract-variant-def*
      apply (*simp add*: L'*-eq split-eq*)
      apply (*rule card-Diff1-less* [*OF fin-L ap-in*])
    done
  next
    case *False* **note** *split-neq* = *this*

    from *Hopcroft-split-correct-simple* [*OF equiv-part-P p-in a-in*] *split-neq*
    have *card-leq*: *card P* < *card* (*Hopcroft-split* $\mathcal{A}$ *p a* {} *P*) **and**
        equiv-part-P': *is-weak-language-equiv-partition* $\mathcal{A}$ (*Hopcroft-split* $\mathcal{A}$ *p a* {} *P*)
      by *simp-all*

    from *equiv-part-P'* **have** *part-P'*: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) (*Hopcroft-split* $\mathcal{A}$ *p a* {} *P*)
      unfolding *is-weak-language-equiv-partition-def* **by** *fast*

```
    from is-partition-card-P [OF finite-Q part-P]
         is-partition-card-P [OF finite-Q part-P']
         card-leq
    have card (Q A) − card (Hopcroft-split A p a {} P) < card (Q A) − card P
      by simp
    thus ?thesis
      unfolding Hopcroft-abstract-variant-def
      by (simp add: split-neq)
  qed
 qed
qed

lemma (in DFA) Hopcroft-abstract-variant-exists:
  WHILE-variant-exists Hopcroft-abstract-b (Hopcroft-abstract-f A) (Hopcroft-abstract-invar A)
unfolding WHILE-variant-exists-def
using Hopcroft-abstract-variant-OK by blast

lemma (in DFA) Hopcroft-abstract-correct :
  Hopcroft-abstract A ≤ SPEC (λP. P = Myhill-Nerode-partition A)
proof (cases Q A = {} ∨ F A = {})
  case True thus ?thesis
    unfolding Hopcroft-abstract-def
    using Myhill-Nerode-partition---Q-emp [of A]
          Myhill-Nerode-partition---F-emp [of A]
    by simp blast
next
  case False thus ?thesis
    unfolding Hopcroft-abstract-def
  apply (rule-tac if-rule, simp)
  apply (rule-tac if-rule, simp)
  apply (rule-tac bind-rule)
  apply (simp add: prod-case-beta)
  apply (rule WHILEIT-rule-manual)
  apply (simp add: Hopcroft-abstract-invar-init)
  apply (simp add: Hopcroft-abstract-variant-exists)
  proof −
    fix PL
    assume invar: Hopcroft-abstract-invar A PL
       and not-cond: ¬ (Hopcroft-abstract-b PL)
    obtain P L where PL-eq[simp]: PL = (P, L) by (rule PairE)

    from not-cond have L-eq[simp]: L = {}
      unfolding Hopcroft-abstract-b-def by simp

    from invar have P = Myhill-Nerode-partition A
      unfolding Hopcroft-abstract-invar-def PL-eq L-eq
      apply (rule-tac split-language-equiv-partition-final)
      apply (simp-all add: Ball-def)
      apply (metis subset-refl)
    done

    thus fst PL = Myhill-Nerode-partition A
      unfolding PL-eq by simp
  qed
qed
```

4.5 Implementing step

Above the next state of the loop was aquired using a specification. Now, let's refine this specification with an inner loop.

definition *Hopcroft-set-step-invar* **where**
Hopcroft-set-step-invar $\mathcal{A}$ *p a P L P′ σ* =
(*Hopcroft-update-splitters-pred-aux* ($\Sigma\ \mathcal{A}$) (*Hopcroft-splitted* $\mathcal{A}$ *p a* {} (*P* − *P′*)) *P*
(*L* − {(*a*, *p*)}) (*snd σ*) ∧ *fst σ* = *Hopcroft-split* $\mathcal{A}$ *p a* {} (*P* − *P′*) ∪ *P′*)

definition *Hopcroft-set-step* **where**
Hopcroft-set-step $\mathcal{A}$ *p a P L* =
(*do* { *PS* ← *SPEC* (λ*P′*. (*P′* ⊆ *P* ∧
(∀ *p′* ∈ *P*. *split-language-equiv-partition-pred* $\mathcal{A}$ *p′ a p* ⟶ *p′* ∈ *P′*)));
(*P′*, *L′*) ← *FOREACHi* (*Hopcroft-set-step-invar* $\mathcal{A}$ *p a P L*) *PS*
(λ*p′* (*P′*, *L′*). *do* {
let (*pt′*, *pf′*) = *split-language-equiv-partition* $\mathcal{A}$ *p′ a p*;
if (*pt′* = {} ∨ *pf′* = {}) *then* (*RETURN* (*P′*, *L′*)) *else*
do {
(*pmin*, *pmax*) ← *SPEC* (λ*pmm*. (*pmm* = (*pt′*, *pf′*)) ∨ (*pmm* = (*pf′*, *pt′*)));
let P′ = (*P′* − {*p′*}) ∪ {*pt′*, *pf′*};
let L′ = ({(*a*, *p″*). (*a*, *p″*) ∈ *L′* ∧ *p″* ≠ *p′*} ∪
{(*a*, *pmin*) | *a*. *a* ∈ Σ $\mathcal{A}$} ∪ {(*a*, *pmax*) |*a*. (*a*, *p′*) ∈ *L′*});
RETURN (*P′*, *L′*)
}
}) (*P*, *L* − {(*a*, *p*)});
RETURN (*P′*, *L′*)
})

lemma *Hopcroft-set-step-correct-aux1* :
assumes *P′-subset*: *P′* ⊆ *P*
and *P-part*: *is-partition* ($\mathcal{Q}\ \mathcal{A}$) *P*
and *P′-prop*: ⋀*p′*. ⟦*p′* ∈ *P*; *split-language-equiv-partition-pred* $\mathcal{A}$ *p′ a p*⟧ ⟹ *p′* ∈ *P′*
shows *Hopcroft-set-step-invar* $\mathcal{A}$ *p a P L P′* (*P*, *L* − {(*a*, *p*)})
proof −
have *splitted-eq-Emp*: *Hopcroft-splitted* $\mathcal{A}$ *p a* {} (*P* − *P′*) = {}
unfolding *Hopcroft-splitted-def split-language-equiv-partition-pred-def*
apply (*rule set-eqI*)
apply (*simp*)
apply (*clarify*)
proof −
fix *p′ pt′ pf′*
assume *p′* ∈ *P p′* ∉ *P′ pt′* ≠ {} *pf′* ≠ {} **and**
[*symmetric*]:(*pt′*, *pf′*) = *split-language-equiv-partition* $\mathcal{A}$ *p′ a p*
with *P′-prop* [*unfolded split-language-equiv-partition-pred-def*, *of p′*]
show *False* **by** *simp*
qed

{ **fix** *p′ pt′ pf′*
assume *p′* ∈ *P p′* ∉ *P′* **and** *split-eq*: *split-language-equiv-partition* $\mathcal{A}$ *p′ a p* = (*pt′*, *pf′*)
with *P′-prop*[*of p′*] **have** *ptf′-eq*: *pt′* = {} ∨ *pf′* = {}
unfolding *split-language-equiv-partition-pred-def* **by** *simp blast*

from *split-language-equiv-partition-union* [*OF split-eq*] **have** *p′-eq*: *p′* = *pt′* ∪ *pf′* **.**

{ **fix** *e*
assume *e* ∈ *pt′*
with *ptf′-eq* **have** *pf′* = {} **by** *auto*
with *p′-eq* **have** *pt′* = *p′* **by** *simp*
with ‹*p′* ∈ *P*› ‹*p′* ∉ *P′*› **have** *pt′* ∈ *P pt′* ∉ *P′* **by** *simp-all*
} **note** *prop1* = *this*

{ **fix** *e*
assume *e* ∈ *pf′*
with *ptf′-eq* **have** *pt′* = {} **by** *auto*
with *p′-eq* **have** *pf′* = *p′* **by** *simp*
with ‹*p′* ∈ *P*› ‹*p′* ∉ *P′*› **have** *pf′* ∈ *P pf′* ∉ *P′* **by** *simp-all*

} **note** *prop2* = *this*
note *prop1 prop2*
} **note** *split-eq-P-aux* = *this*

from *P-part* **have** *split-eq-P-aux2*: {} ∉ *P* **unfolding** *is-partition-def* **by** *simp*

have *split-eq-P*: *Hopcroft-split* 𝒜 *p a* {} (*P* − *P*′) = *P* − *P*′
apply (*rule set-eqI*)
apply (*auto simp add*: *Hopcroft-split-in Bex-def*)
apply (*simp add*: *split-eq-P-aux*(*1*))
apply (*simp add*: *split-eq-P-aux*(*2*))
apply (*simp add*: *split-eq-P-aux*(*3*))
apply (*simp add*: *split-eq-P-aux*(*4*))
apply (*simp add*: *split-eq-P-aux2*)
proof −
fix *p*′
assume *p*′ ∈ *P p*′ ∉ *P*′
obtain *pt*′ *pf*′ **where** *split-eq*: *split-language-equiv-partition* 𝒜 *p*′ *a p* = (*pt*′, *pf*′)
by (*rule PairE*)

note *p*′*-eq* = *split-language-equiv-partition-union* [*OF split-eq*]

from *split-eq P*′*-prop*[*OF* ‹*p*′ ∈ *P*›] ‹*p*′ ∉ *P*′› **have** *ptf*′*-eq*: *pt*′ = {} ∨ *pf*′ = {}
unfolding *split-language-equiv-partition-pred-def* **by** *simp blast*
show ∃ *xa*. *xa* ∈ *P* ∧
xa ∉ *P*′ ∧
(*case split-language-equiv-partition* 𝒜 *xa a p of*
(*p1a*, *p1b*) ⇒ *p*′ = *p1a* ∨ *p*′ = *p1b*)
apply (*rule exI* [**where** *x* = *p*′])
apply (*simp add*: ‹*p*′ ∈ *P*› ‹*p*′ ∉ *P*′› *split-eq*)
apply (*insert ptf*′*-eq*)
apply (*auto simp add*: *p*′*-eq*)
done
qed

show *?thesis*
unfolding *Hopcroft-set-step-invar-def*
using *P*′*-subset*
by (*auto simp add*: *Hopcroft-update-splitters-pred-aux-Emp-Id splitted-eq-Emp split-eq-P*)
qed

lemma *Hopcroft-set-step-correct-aux2* :
assumes *P-part*: *is-partition* (𝒬 𝒜) *P*
and *invar*: *Hopcroft-set-step-invar* 𝒜 *p a P L P*′ (*P*′′, *L*′′)
and *p*′*-in*: *p*′ ∈ *P*′
and *P*′*-subset*: *P*′ ⊆ *P*
and *p*′*-split*: *split-language-equiv-partition* 𝒜 *p*′ *a p* = (*pt*′, *pf*′)
and *ptf-eq*: *pt*′ = {} ∨ *pf*′ = {}
shows *Hopcroft-set-step-invar* 𝒜 *p a P L* (*P*′ − {*p*′}) (*P*′′, *L*′′)
proof −
from *p*′*-in P*′*-subset*
have *P-diff-eq*: *P* − (*P*′ − {*p*′}) = *insert p*′ (*P* − *P*′) **by** *auto*

from *p*′*-in P*′*-subset P-part* **have** *p*′*-neq-emp*: *p*′ ≠ {}
unfolding *is-partition-def* **by** *auto*

from *ptf-eq p*′*-neq-emp*
have *split-aux-eq*: *Hopcroft-split-aux* 𝒜 *p a* {} *p*′ = {*p*′}
unfolding *Hopcroft-split-aux-def p*′*-split*
apply (*simp add*: *split-language-equiv-partition-union*[*OF p*′*-split*])
apply (*cases pt*′ = {})
apply (*simp-all*)

done

have *splitted-aux-eq*: *Hopcroft-splitted-aux* $\mathcal{A}$ *p a* {} p' = {}
unfolding *Hopcroft-splitted-aux-def*
by (*simp add*: p'*-split ptf-eq*)

from *invar* **show** *?thesis*
unfolding *Hopcroft-set-step-invar-def*
apply (*simp add*: *P-diff-eq splitted-aux-eq split-aux-eq*)
apply (*simp add*: *Hopcroft-split-def*)
apply (*auto simp add*: p'*-in*)
done
qed

lemma *Hopcroft-set-step-correct-aux3* :
assumes *L-OK*: $\bigwedge a\ p.\ (a,\ p) \in L \Longrightarrow a \in \Sigma\ \mathcal{A}$
and *invar*: *Hopcroft-set-step-invar* $\mathcal{A}$ *p a P L* P' $(P'',\ L'')$
and *P-part*: *is-partition* $(\mathcal{Q}\ \mathcal{A})$ P
and p'*-in-*P': $p' \in P'$
and P'*-subset*: $P' \subseteq P$
and p'*-split*: *split-language-equiv-partition* $\mathcal{A}$ p' *a p* = $(pt',\ pf')$
and pt'*-neq*: $pt' \neq \{\}$
and pf'*-neq*: $pf' \neq \{\}$
and *pmm*: $((pmin,\ pmax) = (pt',\ pf')) \vee ((pmin,\ pmax) = (pf',\ pt'))$
shows *Hopcroft-set-step-invar* $\mathcal{A}$ *p a P L* $(P' - \{p'\})$
$(insert\ pt'\ (insert\ pf'\ (P'' - \{p'\})),$
$\{(a,\ p'').\ (a,\ p'') \in L'' \wedge p'' \neq p'\} \cup \{(a,\ pmin)\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup$
$\{(a,\ pmax)\ |a.\ (a,\ p') \in L''\})$
proof −
from p'*-in-*P' P'*-subset* **have** $p' \in P$ **by** *auto*
hence *P-diff-eq*: $P - (P' - \{p'\}) = insert\ p'\ (P - P')$ **by** *auto*

from pt'*-neq* pf'*-neq*
have *split-aux-eq*: *Hopcroft-split-aux* $\mathcal{A}$ *p a* {} $p' = \{pt',\ pf'\}$
unfolding *Hopcroft-split-aux-def* p'*-split*
by *simp*

from *split-language-equiv-partition-union*[*OF* p'*-split*]
split-language-equiv-partition-disjoint [*OF* p'*-split*]
pt'*-neq* pf'*-neq*
have *ptf*$'$*-neq-*p': $pt' \neq p'$ $pf' \neq p'$ **by** *auto*

have *splitted-aux-eq*: *Hopcroft-splitted-aux* $\mathcal{A}$ *p a* {} $p' = \{(p',\ pt',\ pf')\}$
unfolding *Hopcroft-splitted-aux-def*
by (*simp add*: p'*-split* pt'*-neq* pf'*-neq*)

{ fix p''
assume p'*-in-split*: $p' \in$ *split-language-equiv-partition-set* $\mathcal{A}$ *a p* p''
and p''*-in*: $p'' \in P$

from p'*-in-split* **have** $p' \subseteq p''$
unfolding *split-language-equiv-partition-set-def*
apply (*simp split*: *prod.splits*)
apply (*metis split-language-equiv-partition-subset*)
done

from *is-partition-distinct-subset*[*OF P-part* p''*-in* ‹$p' \subseteq p''$›] ‹$p' \in P$›
have p''*-eq*: $p'' = p'$ **by** *simp*

with p'*-in-split* *ptf*$'$*-neq-*p' **have** *False*
unfolding *split-language-equiv-partition-set-def*
by (*simp add*: p'*-split*)

}
hence $p' \notin$ *Hopcroft-split* $\mathcal{A}\ p\ a\ \{\}\ (P - P')$
  unfolding *Hopcroft-split-def*
  by (*auto simp add*: *Ball-def*)
hence *split-diff-eq*: *Hopcroft-split* $\mathcal{A}\ p\ a\ \{\}\ (P - P') - \{p'\} =$ *Hopcroft-split* $\mathcal{A}\ p\ a\ \{\}\ (P - P')$
  by *auto*

have *prop1*: *insert* pt' (*insert* pf' (*Hopcroft-split* $\mathcal{A}\ p\ a\ \{\}\ (P - P') \cup P' - \{p'\})) =$
      Hopcroft-split $\mathcal{A}\ p\ a\ \{\}\ (P - (P' - \{p'\})) \cup (P' - \{p'\})$
  apply (*simp add*: *P-diff-eq split-aux-eq Un-Diff split-diff-eq*)
  apply (*simp add*: *Hopcroft-split-def*)
done

{
  def $L' \equiv L - \{(a, p)\}$
  def $L''' \equiv (\{(a, p'').\ (a, p'') \in L'' \land p'' \neq p'\} \cup \{(a, pmin)\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup$
   $\{(a, pmax)\ |a.\ (a, p') \in L''\})$
  def *splitted* $\equiv$ *Hopcroft-splitted* $\mathcal{A}\ p\ a\ \{\}\ (P - P')$
  def $splitted' \equiv$ *Hopcroft-splitted* $\mathcal{A}\ p\ a\ \{\}\ (P - (P' - \{p'\}))$
  assume *pre*[*folded splitted-def* L'*-def*]: *Hopcroft-update-splitters-pred-aux* $(\Sigma\ \mathcal{A})$
    (*Hopcroft-splitted* $\mathcal{A}\ p\ a\ \{\}\ (P - P'))\ P\ L'\ L''$

  have $splitted'$*-eq*[*simp*]: $splitted' =$ *insert* (p', pt', pf') *splitted*
    unfolding $splitted'$*-def splitted-def*
    apply (*simp add*: *P-diff-eq splitted-aux-eq*)
    apply (*simp add*: *Hopcroft-splitted-def*)
  done

  { fix $p''\ pt''\ pf''$
    assume *in-splitted*: $(p'', pt'', pf'') \in$ *splitted*

    { assume $p'' \in P\ p'' \notin P'$
        and $pt'' \neq \{\}\ pf'' \neq \{\}$
        and *split-*p''*-eq*: *split-language-equiv-partition* $\mathcal{A}\ p''\ a\ p = (pt'', pf'')$

      from *split-language-equiv-partition-subset* [*OF split-*p''*-eq*]
      have $pt'' \subseteq p''\ pf'' \subseteq p''$ **by** *simp-all*

      from *split-language-equiv-partition-union* [*OF split-*p''*-eq*]
        split-language-equiv-partition-disjoint [*OF split-*p''*-eq*]
        $\langle pt'' \neq \{\}\rangle\ \langle pf'' \neq \{\}\rangle$
      have $pt'' \neq p''\ pf'' \neq p''$ **by** *auto*

      with *is-partition-distinct-subset*[*OF P-part* $\langle p'' \in P\rangle\ \langle pt'' \subseteq p''\rangle$]
        is-partition-distinct-subset[*OF P-part* $\langle p'' \in P\rangle\ \langle pf'' \subseteq p''\rangle$]
        $\langle p'' \notin P'\rangle\ \langle p' \in P'\rangle\ \langle p' \in P\rangle\ \langle p'' \in P\rangle$
      have $(p'' \neq p') \land (pt'' \neq p') \land (pf'' \neq p')$ **by** *auto*
    }
    with *in-splitted*
    have $p'' \neq p' \land pt'' \neq p' \land pf'' \neq p'$
      unfolding *splitted-def Hopcroft-splitted-def*
      by *simp*
  } note *in-splitted* = *this*

  have *Hopcroft-update-splitters-pred-aux* $(\Sigma\ \mathcal{A})$
    (*Hopcroft-splitted* $\mathcal{A}\ p\ a\ \{\}\ (P - (P' - \{p'\})))\ P\ (L - \{(a, p)\})$
    $(\{(a, p'').\ (a, p'') \in L'' \land p'' \neq p'\} \cup \{(a, pmin)\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup$
    $\{(a, pmax)\ |a.\ (a, p') \in L''\})$
    unfolding $splitted'$*-def*[*symmetric*] L'*-def*[*symmetric*] L'''*-def*[*symmetric*]
  proof
    from *invar* **have**
      Hopcroft-update-splitters-pred-aux-lower-not-splitted $(\Sigma\ \mathcal{A})$ *splitted* $P\ L'\ L''$

unfolding *Hopcroft-set-step-invar-def Hopcroft-update-splitters-pred-aux-def Hopcroft-update-splitters-pred-aux-lower-def* L'*-def*[*symmetric*] *splitted-def*[*symmetric*]
by *simp*
thus *Hopcroft-update-splitters-pred-aux-lower-not-splitted* (Σ $\mathcal{A}$) *splitted'* P L' L'''
unfolding *Hopcroft-update-splitters-pred-aux-lower-not-splitted-def* L'''*-def*
by *auto*
next
from *invar* **have**
Hopcroft-update-splitters-pred-aux-lower-splitted (Σ $\mathcal{A}$) *splitted* P L' L''
unfolding *Hopcroft-set-step-invar-def Hopcroft-update-splitters-pred-aux-def Hopcroft-update-splitters-pred-aux-lower-def* L'*-def*[*symmetric*] *splitted-def*[*symmetric*]
by *simp*
thus *Hopcroft-update-splitters-pred-aux-lower-splitted* (Σ $\mathcal{A}$) *splitted'* P L' L'''
unfolding *Hopcroft-update-splitters-pred-aux-lower-splitted-def*
using *pmm*
apply (*simp add*: *conj-disj-distribL all-conj-distrib*)
apply (*simp add*: L'''*-def*)
apply (*rule conjI*)
apply *metis*
apply (*metis in-splitted*)
done
next
from *invar* **have** *pre*:
Hopcroft-update-splitters-pred-aux-upper (Σ $\mathcal{A}$) *splitted* P L' L''
unfolding *Hopcroft-set-step-invar-def Hopcroft-update-splitters-pred-aux-def* L'*-def*[*symmetric*] *splitted-def*[*symmetric*]
by *simp*
show *Hopcroft-update-splitters-pred-aux-upper* (Σ $\mathcal{A}$) *splitted'* P L' L'''
unfolding *Hopcroft-update-splitters-pred-aux-upper-def* L'''*-def*
apply (*simp add*: *conj-disj-distribL all-conj-distrib in-splitted*)
apply (*intro conjI allI impI*)
apply (*insert pre*[*unfolded Hopcroft-update-splitters-pred-aux-upper-def*]) []
apply *metis*

apply (*insert pmm*)[]
apply (*simp*)
apply (*metis*)

apply (*subgoal-tac* $a \in \Sigma\ \mathcal{A}$)
apply (*insert pmm*)[]
apply (*simp*)
apply (*metis*)

apply (*insert pre*[*unfolded Hopcroft-update-splitters-pred-aux-upper-def*]) []
apply (*simp add*: L'*-def*)
apply (*metis L-OK*)
done
next
from *invar* **have** *pre1*:
Hopcroft-update-splitters-pred-aux-lower-not-splitted (Σ $\mathcal{A}$) *splitted* P L' L''
unfolding *Hopcroft-set-step-invar-def Hopcroft-update-splitters-pred-aux-def Hopcroft-update-splitters-pred-aux-lower-def* L'*-def*[*symmetric*] *splitted-def*[*symmetric*]
by *simp*

from *invar* **have** *pre2*:
Hopcroft-update-splitters-pred-aux-lower-splitted-in-L (Σ $\mathcal{A}$) *splitted* P L' L''
unfolding *Hopcroft-set-step-invar-def Hopcroft-update-splitters-pred-aux-def Hopcroft-update-splitters-pred-aux-lower-def* L'*-def*[*symmetric*] *splitted-def*[*symmetric*]

by *simp*
show *Hopcroft-update-splitters-pred-aux-lower-splitted-in-L* (Σ 𝒜) *splitted′ P L′ L′′′*
unfolding *Hopcroft-update-splitters-pred-aux-lower-splitted-in-L-def*
apply (*simp add*: *conj-disj-distribL all-conj-distrib*)
apply (*simp add*: *L′′′-def*)
apply (*rule conjI*)

apply (*intro allI impI*)
apply (*subgoal-tac* (*a*, *p′*) ∈ *L′′*)
apply (*insert pmm*)[]
apply (*simp add*: *ptf′-neq-p′*)
apply *metis*

apply (*insert pre1*[*unfolded Hopcroft-update-splitters-pred-aux-lower-not-splitted-def*])[]
apply (*metis in-splitted* ‹*p′* ∈ *P*›)

apply (*insert pmm pre2*[*unfolded Hopcroft-update-splitters-pred-aux-lower-splitted-in-L-def*])[]
apply (*metis in-splitted*)
done
qed
} **note** *prop2* = *this*

from *invar* **show** *?thesis*
unfolding *Hopcroft-set-step-invar-def*
by (*simp add*: *prop1 prop2*)
qed

lemma *Hopcroft-set-step-correct-aux4* :
assumes *invar*: *Hopcroft-set-step-invar* 𝒜 *p a P L* {} (*P′′*, *L′′*)
shows *Hopcroft-update-splitters-pred-aux* (Σ 𝒜) (*Hopcroft-splitted* 𝒜 *p a* {} *P*) *P*
(*L* − {(*a*, *p*)}) *L′′* ∧
P′′ = *Hopcroft-split* 𝒜 *p a* {} *P*
using *invar*
unfolding *Hopcroft-set-step-invar-def*
by *simp*

lemma (**in** *DFA*) *Hopcroft-set-step-correct* :
assumes *part-P*: *is-partition* (𝒬 𝒜) *P*
and *L-OK*: $\bigwedge$*a p*. (*a*, *p*) ∈ *L* ⟹ *a* ∈ Σ 𝒜
shows *Hopcroft-set-step* 𝒜 *p a P L* ≤
SPEC (λ(*P′*, *L′*). *Hopcroft-update-splitters-pred* 𝒜 *p a P L L′* ∧ *P′* = *Hopcroft-split* 𝒜 *p a* {} *P*)
unfolding *Hopcroft-set-step-def Hopcroft-update-splitters-pred-def*
apply (*intro refine-vcg*)
apply (*simp-all*)
apply (*metis finite-subset* [*OF* - *is-partition-finite* [*OF finite-𝒬 part-P*]])

apply (*rule Hopcroft-set-step-correct-aux1* [*OF* - *part-P*]) **apply** (*simp*) **apply** *blast*

apply (*rule Hopcroft-set-step-correct-aux2* [*OF part-P*]) **apply** *fast+*

apply (*rule Hopcroft-set-step-correct-aux3* [*OF* - - *part-P*]) **apply** (*simp add*: *L-OK*) **apply** *fast+*

apply (*erule Hopcroft-set-step-correct-aux4*)
done

definition *Hopcroft-set-f* **where**
Hopcroft-set-f 𝒜 =
(λ(*P*, *L*). *do* {
ASSERT (*Hopcroft-abstract-invar* 𝒜 (*P*, *L*));
ASSERT (*L* ≠ {});
(*a*,*p*) ← *SPEC* (λ(*a*,*p*). (*a*,*p*) ∈ *L*);
(*P′*, *L′*) ← *Hopcroft-set-step* 𝒜 *p a P L*;

$RETURN\ (P', L')$
})

4.6 Precomputing Predecessors

In the next refinement step the predecessors of the currently chosen set p with respect to label a is precomputed.

definition *Hopcroft-precompute-step* **where**
$Hopcroft\text{-}precompute\text{-}step\ \mathcal{A}\ p\ a\ pre\ P\ L =$
$(do\ \{\ let\ PS = \{p\ .\ p \in P \wedge (p \cap pre \neq \{\})\};$
$(P', L') \leftarrow FOREACHi\ (Hopcroft\text{-}set\text{-}step\text{-}invar\ \mathcal{A}\ p\ a\ P\ L)\ PS$
$(\lambda p'\ (P', L').\ do\ \{$
$let\ (pt', pct', pf', pcf') =$
$(\{q\ .\ q \in p' \wedge q \in pre\},\ card\ \{q\ .\ q \in p' \wedge q \in pre\},$
$\{q\ .\ q \in p' \wedge q \notin pre\},\ card\ \{q\ .\ q \in p' \wedge q \notin pre\});$
$if\ (pcf' = 0)\ then\ (RETURN\ (P', L'))\ else$
$do\ \{$
$let\ (pmin, pmax) = (if\ (pcf' < pct')\ then\ (pf', pt')\ else\ (pt', pf'));$
$let\ P' = (P' - \{p'\}) \cup \{pt', pf'\};$
$let\ L' = (\{(a, p'').\ (a, p'') \in L' \wedge p'' \neq p'\} \cup$
$\{(a, pmin)\ |\ a.\ a \in \Sigma\ \mathcal{A}\} \cup \{(a, pmax)\ |a.\ (a, p') \in L'\});$
$RETURN\ (P', L')$
}
$\})\ (P, L - \{(a, p)\});$
$RETURN\ (P', L')$
})

lemma *Hopcroft-precompute-step-correct* :
assumes *pre-OK*: $pre = \{q.\ \exists\, q'.\ q' \in p \wedge (q, a, q') \in \Delta\ \mathcal{A}\}$
and *P-fin*: $\bigwedge p.\ p \in P \Longrightarrow finite\ p$
shows $Hopcroft\text{-}precompute\text{-}step\ \mathcal{A}\ p\ a\ pre\ P\ L \leq \Downarrow Id\ (Hopcroft\text{-}set\text{-}step\ \mathcal{A}\ p\ a\ P\ L)$
unfolding *Hopcroft-precompute-step-def Hopcroft-set-step-def*
apply (*rule bind2let-refine*
[**where** $R' = build\text{-}rel\ id\ (\lambda P'.\ \forall\, p{\in}P'.\ finite\ p \wedge p \cap pre \neq \{\})$])
apply (*simp add*: *pw-le-iff refine-pw-simps del*: *br-def*)
apply (*simp add*: *split-language-equiv-partition-pred-def split-language-equiv-partition-def*
split-set-def pre-OK)
apply (*insert P-fin*)[]
apply (*auto*)[]

apply (*subgoal-tac* $\forall\, p'.\ split\text{-}language\text{-}equiv\text{-}partition\ \mathcal{A}\ p'\ a\ p =$
$(\{q.\ q \in p' \wedge q \in pre\}, \{q.\ q \in p' \wedge q \notin pre\})$)
apply (*refine-rcg*)
apply (*rule inj-on-id*)
apply (*simp*)
apply (*rule IdI*)
apply (*simp-all*)

apply (*auto*)[]
apply (*simp-all add*: *split-language-equiv-partition-def split-set-def pre-OK Bex-def*)
done

4.7 Data Refinement

Till now, the algorithm has been refined in several steps. However, the datastructures remained unchanged. Let's now use efficient datastructure. Currently the state consists of a partition of the states and a set of pairs of labels and state sets. In the following the partition will be implemented using maps.

The partition P will be represented by a triple (im, sm, n). This triple consists of a finite map nm mapping indices (natural numbers) to sets, a map sm mapping states to the index of the set it is in, and finally a natural number n that determines the number of used indices.

type-synonym $'q\ partition\text{-}map = (nat \Rightarrow ('q\ set)\ option) * ('q \Rightarrow nat\ option) * nat$

fun *partition-map-α* :: *'q partition-map* ⇒ (*'q set*) *set* **where**
partition-map-α (*im*, *sm*, *n*) = {*p* | *i p*. *i* < *n* ∧ *im i* = *Some p*}

fun *partition-map-invar* :: *'q partition-map* ⇒ *bool* **where**
partition-map-invar (*im*, *sm*, *n*) ⟷
dom im = {*i* . *i* < *n*} ∧
(∀ *i*. *im i* ≠ *Some* {}) ∧
(∀ *q i*. (*sm q* = *Some i*) ⟷ (∃ *p*. *im i* = *Some p* ∧ *q* ∈ *p*))

definition *partition-index-map* :: *'q partition-map* ⇒ (*nat* ⇒ *'q set*) **where**
partition-index-map P i = *the* ((*fst P*) *i*)

definition *partition-state-map* :: *'q partition-map* ⇒ (*'q* ⇒ *nat*) **where**
partition-state-map P q = *the* (*fst* (*snd P*) *q*)

definition *partition-free-index* :: *'q partition-map* ⇒ *nat* **where**
partition-free-index P = *snd* (*snd P*)
lemma *partition-free-index-simp*[*simp*] :
partition-free-index (*im*, *sm*, *n*) = *n* **unfolding** *partition-free-index-def* **by** *simp*

definition *partition-map-empty* :: *'q partition-map* **where**
partition-map-empty = (*empty*, *empty*, *0*)

lemma *partition-map-empty-correct* :
partition-map-α (*partition-map-empty*) = {}
partition-map-invar (*partition-map-empty*)
unfolding *partition-map-empty-def*
by *simp-all*

definition *partition-map-sing* :: *'q set* ⇒ (*'q partition-map*) **where**
partition-map-sing Q =
(*empty* (*0* ↦ *Q*) , (λ*q*. *if* (*q* ∈ *Q*) *then Some 0 else None*), *1*)

lemma *partition-map-sing-correct* :
partition-map-α (*partition-map-sing Q*) = {*Q*}
partition-map-invar (*partition-map-sing Q*) ⟷ *Q* ≠ {}
unfolding *partition-map-sing-def*
by *simp-all*

lemma *partition-index-map-inj-on* :
assumes *invar*: *partition-map-invar P*
and *p-OK*: ⋀*i*. *i* ∈ *p* ⟹ *i* < *partition-free-index P*
shows *inj-on* (*partition-index-map P*) *p*
unfolding *inj-on-def*
proof (*intro ballI impI*)
fix *i1 i2*
assume *i1* ∈ *p*
assume *i2* ∈ *p*
assume *map-i12-eq*: *partition-index-map P i1* = *partition-index-map P i2*

obtain *im sm n* **where** *P-eq*[*simp*]: *P* = (*im*, *sm*, *n*) **by** (*metis PairE*)

from *p-OK*[*OF* ‹*i1* ∈ *p*›] **have** *i1* < *n* **by** *simp*
with *invar* **have** *i1* ∈ *dom im* **by** *simp*
then obtain *s1* **where** *im-i1-eq*: *im i1* = *Some s1*
by (*simp add*: *dom-def*, *blast*)

from *p-OK*[*OF* ‹*i2* ∈ *p*›] **have** *i2* < *n* **by** *simp*
with *invar* **have** *i2* ∈ *dom im* **by** *simp*
then obtain *s2* **where** *im-i2-eq*: *im i2* = *Some s2*
by (*simp add*: *dom-def*, *blast*)

from *map-i12-eq* ‹*im i1* = *Some s1*› ‹*im i2* = *Some s2*›
have *s2-eq*[*simp*]: *s2* = *s1* **unfolding** *partition-index-map-def* **by** *simp*

from *invar* **have** *im i1* $\neq$ *Some* {} **by** *simp*
with *im-i1-eq* **have** *s1* $\neq$ {} **by** *simp*
then obtain *q* **where** *q* $\in$ *s1* **by** *auto*

from *invar im-i1-eq* ‹*q* $\in$ *s1*› **have** *sm q* = *Some i1* **by** *simp*
moreover
from *invar im-i2-eq* ‹*q* $\in$ *s1*› **have** *sm q* = *Some i2* **by** *simp*
finally show *i1* = *i2* **by** *simp*
qed

lemma *partition-map-is-partition* :
assumes *invar*: *partition-map-invar P*
shows *is-partition* ($\bigcup$ (*partition-map-α P*)) (*partition-map-α P*)
proof
fix *p*
assume *p* $\in$ *partition-map-α P*
thus *p* $\subseteq$ $\bigcup$*partition-map-α P*
by *auto*
next
fix *p*
assume *p-in*: *p* $\in$ *partition-map-α P*

obtain *im sm n* **where** *P-eq*[*simp*]: *P* = (*im*, *sm*, *n*) **by** (*metis PairE*)
from *p-in* **obtain** *i* **where** *p-eq*: *im i* = *Some p* **and** *i* < *n* **by** *auto*

from *invar* **have** *im i* $\neq$ *Some* {} **by** *simp*
with ‹*im i* = *Some p*› **show** *p* $\neq$ {} **by** *simp*
next
fix *q p1 p2*
assume *p1-in*: *p1* $\in$ *partition-map-α P*
and *p2-in*: *p2* $\in$ *partition-map-α P*
and *q-in-p12*: *q* $\in$ *p1 q* $\in$ *p2*

obtain *im sm n* **where** *P-eq*[*simp*]: *P* = (*im*, *sm*, *n*) **by** (*metis PairE*)

from *p1-in* **obtain** *i1* **where** *p1-eq*: *im i1* = *Some p1* **and** *i1* < *n* **by** *auto*
from *p2-in* **obtain** *i2* **where** *p2-eq*: *im i2* = *Some p2* **and** *i2* < *n* **by** *auto*

from *invar q-in-p12* **have** *sm q* = *Some i1 sm q* = *Some i2*
by (*simp-all add*: *p1-eq p2-eq*)
hence *i2* = *i1* **by** *simp*
with *p1-eq p2-eq* **show** *p1* = *p2*
by *simp*
qed *simp*

lemma *partition-index-map-disj* :
assumes *invar*: *partition-map-invar P*
and *i-j-OK*: *i* < *partition-free-index P j* < *partition-free-index P*
shows (*partition-index-map P i* $\cap$ *partition-index-map P j* = {}) $\longleftrightarrow$ (*i* $\neq$ *j*)
proof −
from *partition-index-map-inj-on* [*OF invar*, *of* {*i*, *j*}] *i-j-OK*
have *step1*: (*i* = *j*) = (*partition-index-map P i* = *partition-index-map P j*)
by (*auto simp add*: *image-iff*)

note *is-part* = *partition-map-is-partition* [*OF invar*]

obtain *im sm n* **where** *P-eq*[*simp*]: *P* = (*im*, *sm*, *n*) **by** (*metis PairE*)

from *invar i-j-OK*
have $i \in dom\ im$ **and** $j \in dom\ im$ **by** (*simp-all*)
then obtain *pi pj* **where** *im-i-eq*: $im\ i = Some\ pi$ **and** *im-j-eq*: $im\ j = Some\ pj$ **by** *blast*

from *i-j-OK*
have *ij-in*: $partition\text{-}index\text{-}map\ P\ i \in partition\text{-}map\text{-}\alpha\ P \wedge$
$partition\text{-}index\text{-}map\ P\ j \in partition\text{-}map\text{-}\alpha\ P$
apply (*simp add*: *partition-index-map-def im-i-eq im-j-eq*)
apply (*metis im-i-eq im-j-eq*)
done

from *is-part ij-in*
show *?thesis*
unfolding *step1 is-partition-def*
by *auto*
qed

lemma *partition-map-is-partition-eq* :
assumes *invar*: *partition-map-invar P*
and *part*: *is-partition PP* (*partition-map-α P*)
shows $PP = (\bigcup\ (partition\text{-}map\text{-}\alpha\ P))$
proof −
note *partition-map-is-partition*[*OF invar*]
with *part* **show** *?thesis*
unfolding *is-partition-def* **by** *simp*
qed

lemma *partition-state-map-le* :
assumes *invar*: *partition-map-invar P*
and *q-in*: $q \in (\bigcup\ (partition\text{-}map\text{-}\alpha\ P))$
shows $partition\text{-}state\text{-}map\ P\ q < partition\text{-}free\text{-}index\ P$
proof −
obtain *im sm n* **where** *P-eq*[*simp*]: $P = (im, sm, n)$ **by** (*metis PairE*)
from *q-in* **obtain** *i s* **where** $im\ i = Some\ s \ \ q \in s\ i < n$ **by** *auto*
from *invar* ‹$q \in s$› ‹$im\ i = Some\ s$› **have** *sm-q-eq*: $sm\ q = Some\ i$ **by** *simp*
from *sm-q-eq* ‹$i < n$› **show** *?thesis* **by** (*simp add*: *partition-state-map-def*)
qed

fun *partition-map-split* :: $'q\ partition\text{-}map \Rightarrow nat \Rightarrow 'q\ set => 'q\ set \Rightarrow 'q\ partition\text{-}map$ **where**
$partition\text{-}map\text{-}split\ (im, sm, n)\ i\ pmin\ pmax =$
$(im\ (i \mapsto pmax)\ (n \mapsto pmin),$
$\lambda q.\ if\ (q \in pmin)\ then\ Some\ n\ else\ sm\ q,$
$Suc\ n)$

definition *Hopcroft-map-state-α* **where**
$Hopcroft\text{-}map\text{-}state\text{-}\alpha\ \sigma =$
$(partition\text{-}map\text{-}\alpha\ (fst\ \sigma), (apsnd\ (partition\text{-}index\text{-}map\ (fst\ \sigma)))\ `\ (snd\ \sigma))$

definition *Hopcroft-map-state-invar* **where**
$Hopcroft\text{-}map\text{-}state\text{-}invar\ \sigma =$
$(partition\text{-}map\text{-}invar\ (fst\ \sigma) \wedge$
$(\forall\ ap \in (snd\ \sigma).\ snd\ ap < partition\text{-}free\text{-}index\ (fst\ \sigma)))$

definition *Hopcroft-map-state-rel* **where**
$Hopcroft\text{-}map\text{-}state\text{-}rel = build\text{-}rel\ Hopcroft\text{-}map\text{-}state\text{-}\alpha\ Hopcroft\text{-}map\text{-}state\text{-}invar$

lemma *Hopcroft-map-state-rel-sv*[*refine*] :
single-valued Hopcroft-map-state-rel
unfolding *Hopcroft-map-state-rel-def*

by (*rule br-single-valued*)

definition *Hopcroft-map-step-invar* **where**
$Hopcroft\text{-}map\text{-}step\text{-}invar\ \mathcal{A}\ p\ a\ P\ L\ P'\ \sigma \longleftrightarrow$
$\ Hopcroft\text{-}set\text{-}step\text{-}invar\ \mathcal{A}\ (partition\text{-}index\text{-}map\ P\ p)\ a\ (partition\text{-}map\text{-}\alpha\ P)$
$\ ((apsnd\ (partition\text{-}index\text{-}map\ P))\ `\ L)\ ((partition\text{-}index\text{-}map\ P)\ `\ P')$
$\ \ (Hopcroft\text{-}map\text{-}state\text{-}\alpha\ \sigma) \wedge Hopcroft\text{-}map\text{-}state\text{-}invar\ \sigma$

definition *Hopcroft-map-step* **where**
$Hopcroft\text{-}map\text{-}step\ \mathcal{A}\ p\ a\ pre\ P\ L =$
$(do\ \{\ ASSERT\ (pre \subseteq dom\ (fst\ (snd\ P)));$
$\quad let\ PS = (partition\text{-}state\text{-}map\ P)\ `\ pre;$
$\quad (P', L') \leftarrow FOREACHi\ (Hopcroft\text{-}map\text{-}step\text{-}invar\ \mathcal{A}\ p\ a\ P\ L)\ PS$
$\quad\ (\lambda(p'::nat)\ (P'::'q\ partition\text{-}map,\ L').\ do\ \{$
$\quad\quad ASSERT\ (p' \in dom\ (fst\ P'));$
$\quad\quad let\ (pt', pct', pf', pcf') =$
$\quad\quad (\{q\ .\ q \in partition\text{-}index\text{-}map\ P'\ p' \wedge q \in pre\},\ card\ \{q\ .\ q \in partition\text{-}index\text{-}map\ P'\ p' \wedge q \in pre\},$
$\quad\quad\ \{q\ .\ q \in partition\text{-}index\text{-}map\ P'\ p' \wedge q \notin pre\},\ card\ \{q\ .\ q \in partition\text{-}index\text{-}map\ P'\ p' \wedge q \notin pre\});$
$\quad\quad if\ (pcf' = 0)\ then\ (RETURN\ (P', L'))\ else$
$\quad\quad do\ \{$
$\quad\quad\ let\ (pmin, pmax) = (if\ (pcf' < pct')\ then\ (pf', pt')\ else\ (pt', pf'));$
$\quad\quad\ let\ L' = \{(a,\ partition\text{-}free\text{-}index\ P')\ |\ a.\ a \in \Sigma\ \mathcal{A}\} \cup L';$
$\quad\quad\ let\ P' = partition\text{-}map\text{-}split\ P'\ p'\ pmin\ pmax;$
$\quad\quad\ RETURN\ (P', L')$
$\quad\quad \}$
$\quad \})\ (P,\ L - \{(a, p)\});$
$\quad RETURN\ (P', L')$
$\ \})$

lemma *Hopcroft-map-step-correct* :
fixes $P :: 'q\ partition\text{-}map$
 and $L :: ('a \times nat)\ set$
 and $\mathcal{A} :: ('q, 'a, 'X)\ NFA\text{-}rec\text{-}scheme$
assumes *PL-OK*: $((P,L), (P', L')) \in Hopcroft\text{-}map\text{-}state\text{-}rel$
 and *p-le*: $p < partition\text{-}free\text{-}index\ P$
 and *p′-eq*: $p' = partition\text{-}index\text{-}map\ P\ p$
 and *part-P′*: $is\text{-}partition\ (\mathcal{Q}\ \mathcal{A})\ P'$
 and *pre-subset*: $pre \subseteq \mathcal{Q}\ \mathcal{A}$
shows $Hopcroft\text{-}map\text{-}step\ \mathcal{A}\ p\ a\ pre\ P\ L \le \Downarrow Hopcroft\text{-}map\text{-}state\text{-}rel\ (Hopcroft\text{-}precompute\text{-}step\ \mathcal{A}\ p'\ a\ pre\ P'\ L')$
unfolding *Hopcroft-precompute-step-def Hopcroft-map-step-def*
apply (*rule refine*)
defer
apply (*rule refine*)
apply (*rule bind-refine* [**where** *R′=Hopcroft-map-state-rel*])
prefer *2*
apply (*simp add*: *Hopcroft-map-state-rel-def pw-le-iff refine-pw-simps*)
apply (*rule FOREACHi-refine*)
proof −
 from *PL-OK* **have** *invar-P*: *partition-map-invar P* **and** *P′-eq*: $P' = partition\text{-}map\text{-}\alpha\ P$
  and *L-OK*: $\bigwedge a\ p.\ (a, p) \in L \Longrightarrow p < partition\text{-}free\text{-}index\ P$
  and *L′-eq*: $L' = apsnd\ (partition\text{-}index\text{-}map\ P)\ `\ L$
 unfolding *Hopcroft-map-state-rel-def Hopcroft-map-state-invar-def-raw*
  Hopcroft-map-state-α-def-raw
 by *auto*

 have *Q-eq*: $\bigcup partition\text{-}map\text{-}\alpha\ P = \mathcal{Q}\ \mathcal{A}$
 using *partition-map-is-partition-eq*[*OF invar-P*, *folded P′-eq*, *OF part-P′*] *P′-eq*
 by *simp*

 obtain *im sm n* **where** *P-eq*[*simp*]: $P = (im, sm, n)$ **by** (*metis PairE*)

 from *partition-state-map-le*[*OF invar-P*, *unfolded Q-eq*]

have *P-state-map-leq*: $\bigwedge q.\ q \in \mathcal{Q}\ \mathcal{A} \Longrightarrow$ *partition-state-map* (*im*, *sm*, *n*) $q < n$ **by** *simp*

have *map-pre-subset*: *partition-state-map* P ‘ *pre* $\subseteq \{i.\ i < n\}$
  apply (*auto*)
  apply (*rule-tac P-state-map-leq*)
  apply (*insert pre-subset*)
  apply *auto*
done

have *im-inj-on*: $\bigwedge S.\ S \subseteq \{i.\ i < n\} \Longrightarrow$ *inj-on* (*partition-index-map* P) S
  apply (*rule partition-index-map-inj-on*[*OF invar-P*])
  apply (*auto*)
done

show *inj-on* (*partition-index-map* P) (*partition-state-map* P ‘ *pre*)
  apply (*rule im-inj-on*)
  apply (*rule map-pre-subset*)
done

show $((P,\ L - \{(a,\ p)\}),\ (P',\ L' - \{(a,\ p')\})) \in$ *Hopcroft-map-state-rel*
proof −
  have (*apsnd* (*partition-index-map* (*im*, *sm*, *n*))) ‘ $(L - \{(a,\ p)\})$ =
        (((*apsnd* (*partition-index-map* (*im*, *sm*, *n*))) ‘ L) −
         ((*apsnd* (*partition-index-map* (*im*, *sm*, *n*))) ‘ $\{(a,\ p)\}$))
    apply (*rule inj-on-image-set-diff* [**where** C = *UNIV* $\times \{i.\ i < n\}$])
    apply (*simp add*: *apsnd-def*)
    apply (*rule map-pair-inj-on*)
    apply (*rule inj-on-id*)
    apply (*rule im-inj-on*[*unfolded P-eq*], *simp*)
    apply (*insert p-le L-OK*)
    apply *auto*
  done

  hence *apsnd* (*partition-index-map* (*im*, *sm*, *n*)) ‘ $L - \{(a,\ p')\}$ =
         apsnd (*partition-index-map* (*im*, *sm*, *n*)) ‘ $(L - \{(a,\ p)\})$ **by** (*simp add*: p'*-eq*)

  with *PL-OK* **show** *?thesis*
    by (*simp add*: *Hopcroft-map-state-rel-def Hopcroft-map-state-*α*-def-raw*
                Hopcroft-map-state-invar-def)
qed

show $\{p \in P'.\ p \cap pre \neq \{\}\}$ = *partition-index-map* P ‘ *partition-state-map* P ‘ *pre*
  (**is** *?ls* = *?rs*)
proof (*intro set-eqI iffI*)
  fix p
  assume $p \in$ *?ls*
  hence $p \in P'$ **and** $p \cap pre \neq \{\}$ **by** *simp-all*
  from ‹$p \cap pre \neq \{\}$› **obtain** q **where** $q \in p$ **and** $q \in pre$ **by** *auto*

  from ‹$q \in p$› ‹$p \in P'$› P'*-eq* **obtain** i **where** $i < n$ **and** *im-i-eq*: *im* i = *Some* p
    by *simp blast*

  from *invar-P* **have** *sm-q-eq*: *sm* q = *Some* i
    by (*simp add*: *im-i-eq* ‹$q \in p$›)

  show $p \in$ *?rs*
    apply (*simp del*: *ex-simps add*: *image-iff Bex-def partition-index-map-def*
              partition-state-map-def ex-simps[*symmetric*])
    apply (*rule exI* [**where** $x = q$])
    apply (*simp add*: *im-i-eq sm-q-eq* ‹$q \in pre$›)
  done
next

fix *p*
assume $p \in$ *?rs*
then obtain *q* **where** *q-in-pre*: $q \in$ *pre* **and** *p-eq*: $p =$ *the* (*im* (*the* (*sm q*)))
apply (*simp del*: *ex-simps add*: *image-iff Bex-def partition-index-map-def partition-state-map-def ex-simps*[*symmetric*])
apply *blast*
done

from *q-in-pre* **have** $q \in \mathcal{Q}\ \mathcal{A}$ **using** *pre-subset* **by** *auto*
with $\mathcal{Q}$*-eq*[*symmetric*] **obtain** *i s* **where** $i<n$ **and** *im-i-eq*: *im i* = *Some s* **and** $q \in s$ **by** *auto*
with *invar-P* **have** *sm-q-eq*: *sm q* = *Some i* **by** *simp*

from *p-eq* **have** *s-eq*[*simp*]: $s = p$
by (*simp add*: *im-i-eq sm-q-eq*)

show $p \in$ *?ls*
apply (*simp add*: P'*-eq*)
apply (*rule conjI*)
apply (*rule exI* [**where** $x = i$])
apply (*simp add*: ‹$i < n$› *im-i-eq*)
apply (*simp add*: *set-eq-iff*)
apply (*rule exI* [**where** $x = q$])
apply (*simp add*: ‹$q \in s$›[*simplified*] *q-in-pre*)
done
qed

let $?\Phi'' = (\lambda PS\ PL''$ - -.
(*partition-free-index P* $\leq$ *partition-free-index* (*fst* PL'')) $\wedge$
($\forall\, i \in PS$. *fst* (*fst* PL'') i = *fst P i*))
show $?\Phi''$ (*partition-state-map P* ‘ *pre*) (P, $L - \{(a, p)\}$)
(*partition-index-map P* ‘ *partition-state-map P* ‘ *pre*) (P', $L' - \{(a, p')\}$)
by *simp*

show *pre-subset*: *pre* $\subseteq$ *dom* (*fst* (*snd P*))
using *invar-P pre-subset*
apply (*simp add*: *dom-def subset-iff* $\mathcal{Q}$*-eq*[*symmetric*])
apply *metis*
done

{ **fix** *it* σ it' σ'
assume *Hopcroft-set-step-invar* $\mathcal{A}$ p' *a* P' L' it' σ'
(σ, σ') $\in$ *Hopcroft-map-state-rel*
it' = *partition-index-map P* ‘ *it*
thus *Hopcroft-map-step-invar* $\mathcal{A}$ *p a P L it* σ
unfolding *Hopcroft-map-state-rel-def Hopcroft-map-step-invar-def*
by (*simp add*: L'*-eq* p'*-eq* P'*-eq*)
}

show *single-valued Hopcroft-map-state-rel*
unfolding *Hopcroft-map-state-rel-def* **by** (*simp del*: *br-def*)

apply-end *clarify*
apply-end (*simp only*: *fst-conv*)
apply-end (*rule refine*)

{ **fix** *i it* im' sm' n' *sL* x' it' P'' L'' i'
assume $i \in it$
and *it-subset*: *it* $\subseteq$ *partition-state-map P* ‘ *pre*
and *pt-neq-emp-pre*: *partition-index-map P* ‘ *it* $\subseteq \{p \in P'.\ p \cap pre \neq \{\}\}$
and *map-step-invar*: *Hopcroft-map-step-invar* $\mathcal{A}$ *p a P L it* ((im', sm', n'), *sL*)
and *Hopcroft-set-step-invar* $\mathcal{A}$ p' *a* P' L' (*partition-index-map P* ‘ *it*) (P'', L'')
and *in-rel*: (((im', sm', n'), *sL*), P'', L'') $\in$ *Hopcroft-map-state-rel*

and *map-i-i′-eq*: *partition-index-map P i* = *partition-index-map P i′*
and *i′* ∈ *it*
partition-free-index P ≤ *partition-free-index* (*im′*, *sm′*, *n′*)
and *im′-eq*: ∀ *i*∈*it*. *im′ i* = *fst P i*

from *map-pre-subset* ⟨*i* ∈ *it*⟩ *it-subset* **have** *i-le*: *i* < *n*
by (*simp add*: *subset-iff*)

from *map-pre-subset* ⟨*i′* ∈ *it*⟩ *it-subset* **have** *i′-le*: *i′* < *n*
by (*simp add*: *subset-iff*)

from *im-inj-on*[*of* {*i*, *i′*}] *map-i-i′-eq* **have** *i′-eq*[*simp*]: *i′* = *i*
by (*simp add*: *subset-iff i-le i′-le image-iff*)

def *p′′* ≡ *partition-index-map P i*
def *pt′* ≡ {*q* ∈ *p′′*. *q* ∈ *pre*}
def *pf′* ≡ {*q* ∈ *p′′*. *q* ∉ *pre*}

from *im′-eq* ⟨*i*∈*it*⟩ **have** *p′′-intro*: *partition-index-map* (*im′*, *sm′*, *n′*) *i* = *p′′*
unfolding *p′′-def partition-index-map-def* **by** *simp*

from ⟨*i* ∈ *it*⟩ *pt-neq-emp-pre* **have** *pt′-neq-emp*: *pt′* ≠ {}
unfolding *pt′-def p′′-def*
by *auto*

def *pmin* ≡ *if card pf′* < *card pt′ then pf′ else pt′*
def *pmax* ≡ *if card pf′* < *card pt′ then pt′ else pf′*

have *pminmax*: (*if card pf′* < *card pt′ then* (*pf′*, *pt′*) *else* (*pt′*, *pf′*)) = (*pmin*, *pmax*)
unfolding *pmin-def pmax-def* **by** *simp*

from ⟨*partition-free-index P* ≤ *partition-free-index* (*im′*, *sm′*, *n′*)⟩ **have**
n-le: *n* ≤ *n′* **by** *simp*

show *i* ∈ *dom im′*
using *n-le i-le map-step-invar*
unfolding *Hopcroft-map-step-invar-def Hopcroft-map-state-invar-def*
by *simp*

have (*if card pf′* = *0 then RETURN* ((*im′*, *sm′*, *n′*), *sL*)
else let P′ = *partition-map-split* (*im′*, *sm′*, *n′*) *i pmin pmax*;
L′ = {(*a*, *partition-free-index P′* − *1*) |*a*. *a* ∈ Σ 𝒜} ∪ *sL*
in RETURN (*P′*, *L′*))
≤ ⇓ {(σ, σ′).
(σ, σ′) ∈ *Hopcroft-map-state-rel* ∧
partition-free-index P ≤ *partition-free-index* (*fst* σ) ∧
(∀ *i*∈*it* − {*i*}. *fst* (*fst* σ) *i* = *fst P i*)}
(*if card pf′* = *0 then RETURN* (*P′′*, *L′′*)
else let P′ = *P′′* − {*partition-index-map P i′*} ∪ {*pt′*, *pf′*};
L′ = {(*a*, *p′′*). (*a*, *p′′*) ∈ *L′′* ∧ *p′′* ≠ *partition-index-map P i′*} ∪
{(*a*, *pmin*) |*a*. *a* ∈ Σ 𝒜} ∪
{(*a*, *pmax*) |*a*. (*a*, *partition-index-map P i′*) ∈ *L′′*}
in RETURN (*P′*, *L′*))
proof (*cases card pf′* = *0*)
case *True*
with *in-rel n-le im′-eq*
show *?thesis* **by** (*simp add*: *Bex-def Hopcroft-map-state-rel-def pw-le-iff refine-pw-simps*)
next
case *False* **note** *card-pf′-neq* = *this*
hence *pf′-neq-emp*: *pf′* ≠ {} **by** *auto*

with *pt′-neq-emp* **have** *pminmax-neq-emp*: *pmin* ≠ {} *pmax* ≠ {}

unfolding *pmin-def pmax-def* **by** *simp-all*

have p''*-eq-minmax*: $p'' = pmin \cup pmax$
unfolding *pmin-def pmax-def pt'-def pf'-def* **by** *auto*
have *pminmax-disj*: $pmin \cap pmax = \{\}$
unfolding *pmin-def pmax-def pt'-def pf'-def* **by** *auto*

from *map-pre-subset n-le it-subset* **have** *n'-not-in*: $n' \notin it$
apply (*rule-tac notI*)
apply (*subgoal-tac* $n' < n$)
apply *simp*
apply (*simp add*: *subset-iff*)
done

from *invar-P* $\langle i < n \rangle$ **have** $i \in (dom\ im)$ **by** *simp*
hence *im-i-eq*: $im\ i = Some\ p''$
by (*auto simp add*: *dom-def set-eq-iff p''-def partition-index-map-def*)

have *in-rel'*:
$(((im'(i \mapsto pmax,\ n' \mapsto pmin),\ \lambda q.\ if\ q \in pmin\ then\ Some\ n'\ else\ sm'\ q,\ Suc\ n'),$
$\{(a,\ n')\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup sL),\ insert\ pt'\ (insert\ pf'\ (P'' - \{p''\})),$
$\{(a,\ p''').\ (a,\ p''') \in L'' \wedge p''' \neq p''\} \cup \{(a,\ pmin)\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup$
$\{(a,\ pmax)\ |a.\ (a,\ p'') \in L''\}) \in Hopcroft\text{-}map\text{-}state\text{-}rel$
proof −
from *i-le n-le* **have** *i-le'*: $i < n'$ **by** *simp*
hence *i-neq-n'*: $i \neq n'$ **by** *simp*

from *in-rel* **have** *invar-P''*: *partition-map-invar* $(im',\ sm',\ n')$ **and**
P''-eq: $P'' = partition\text{-}map\text{-}\alpha\ (im',\ sm',\ n')$
and *sL-OK*: $\bigwedge a\ i.\ (a,\ i) \in sL \Longrightarrow i < n'$
and *L''-eq*: $L'' = apsnd\ (partition\text{-}index\text{-}map\ (im',\ sm',\ n'))$ ` sL
unfolding *Hopcroft-map-state-rel-def Hopcroft-map-state-invar-def-raw*
Hopcroft-map-state-α-def-raw
by *auto*

from *invar-P'' i-le'* **have** $i \in (dom\ im')$ **by** *simp*
hence *im'-i-eq*: $im'\ i = Some\ p''$
by (*auto simp add*: *dom-def p''-intro*[*symmetric*] *partition-index-map-def*)

have *invar-OK*: *Hopcroft-map-state-invar*
$((im'(i \mapsto pmax,\ n' \mapsto pmin),\ \lambda q.\ if\ q \in pmin\ then\ Some\ n'\ else\ sm'\ q,\ Suc\ n'),$
$\{(a,\ n')\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup sL)$
apply (*simp add*: *Hopcroft-map-state-invar-def i-neq-n' pminmax-neq-emp Ball-def*
im'-eq all-conj-distrib imp-conjR)
apply (*intro conjI allI impI*)
proof −
from *invar-P''* **have** $dom\ im' = \{i.\ i < n'\}$ **by** *simp*
with *i-le n-le* **show** $insert\ n'\ (insert\ i\ (dom\ im')) = \{i.\ i < Suc\ n'\}$
by *auto*
next
fix i'
from *invar-P''* **show** $im'\ i' \neq Some\ \{\}$ **by** *simp*
next
fix $a\ i'$
assume $(a,\ i') \in sL$
from *sL-OK*[*OF this*] **have** $i' < n'$ **.**
thus $i' < Suc\ n'$ **by** *simp*
next
fix q
assume $q \in pmin$
thus $q \notin pmax$ **using** *pminmax-disj* **by** *auto*
next

fix q
from *invar-P''* **have** $n' \notin$ *dom im'* **by** *simp*
hence *im' n'* = *None* **unfolding** *dom-def* **by** *simp*
with *invar-P''* **show** $sm'\ q \neq Some\ n'$
by *simp*
next
fix $q\ i'$
assume $q \notin pmin\ i' \neq i\ i' \neq n'$
with *invar-P''* **show** $(sm'\ q = Some\ i') = (\exists p.\ im'\ i' = Some\ p \wedge q \in p)$
by *simp*
next
fix $q\ i'\ p$
assume $q \in pmin\ i' \neq i$ **and**
im'-i'-eq: $im'\ i' = Some\ p$

from *im'-i'-eq* **have** $i' \in dom\ im'$ **by** (*simp add*: *dom-def*)
with *invar-P''* **have** $i' < n'$ **by** *simp*

from ‹$q \in pmin$› **have** $q \in p''$
unfolding *p''-eq-minmax* **by** *simp*

from *partition-index-map-disj* [*OF invar-P''*, *of i i'*] ‹$q \in p''$› ‹$i' < n'$› *i-le n-le*
show $q \notin p$
by (*simp add*: *partition-index-map-def p''-intro*[*symmetric*] *im'-i'-eq* ‹$i' \neq i$›[*symmetric*] *set-eq-iff*)
next
fix q
assume *q-nin-min*: $q \notin pmin$

from *invar-P''* **have** $(sm'\ q = Some\ i) \longleftrightarrow q \in p''$
by (*simp add*: *partition-index-map-def im'-i-eq*)
with *q-nin-min* **show** $(sm'\ q = Some\ i) \longleftrightarrow q \in pmax$
unfolding *p''-eq-minmax* **by** *simp*
qed

have *alpha-OK*: (*insert pt'* (*insert pf'* $(P'' - \{p''\})$),
$\{(a, p''').\ (a, p''') \in L'' \wedge p''' \neq p''\} \cup \{(a, pmin)\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup$
$\{(a, pmax)\ |a.\ (a, p'') \in L''\}) =$
Hopcroft-map-state-α
$((im'(i \mapsto pmax,\ n' \mapsto pmin),\ \lambda q.\ if\ q \in pmin\ then\ Some\ n'\ else\ sm'\ q,\ Suc\ n'),$
$\{(a, n')\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup sL)$
apply (*simp add*: *Hopcroft-map-state-α-def P''-eq*)
apply (*intro conjI set-eqI*)
apply (*simp-all split*: *prod.splits add*: *image-iff Bex-def i-neq-n'*)
prefer *2*
apply *clarify*
apply *simp*
apply (*rename-tac a pp*)
proof −
fix *pp*
show $(pp = pt' \vee pp = pf' \vee (\exists i{<}n'.\ im'\ i = Some\ pp) \wedge pp \neq p'') =$
$(\exists ia.\ (ia = i \longrightarrow i < Suc\ n' \wedge pmax = pp) \wedge$
$(ia \neq i \longrightarrow (ia = n' \longrightarrow pmin = pp) \wedge (ia \neq n' \longrightarrow ia < Suc\ n' \wedge im'\ ia = Some\ pp)))$
proof (*cases* $pp = pt' \vee pp = pf'$)
case *True* **note** *pp-eq-ptf'* = *this*
show *?thesis*
proof (*cases pp* = *pmax*)
case *True* **with** *pp-eq-ptf' i-le'* **show** *?thesis*
by *auto*
next
case *False*
with *pp-eq-ptf'* **have** *pp* = *pmin*
unfolding *pmax-def pmin-def* **by** *auto*

with *pp-eq-ptf′ i-le′* **show** *?thesis*
by *auto*
qed
next
case *False*
hence *pp-neq*: $pp \neq pt'\ pp \neq pf'\ pp \neq pmin\ pp \neq pmax$
unfolding *pmin-def pmax-def* **by** *simp-all*

{ fix i'
have $((i' < n' \wedge im'\ i' = Some\ pp) \wedge pp \neq p'') =$
$(i' \neq i \wedge i' \neq n' \wedge i' < Suc\ n' \wedge im'\ i' = Some\ pp)$
proof $(cases\ i' < n' \wedge im'\ i' = Some\ pp)$
case *False* **thus** *?thesis* **by** *auto*
next
case *True*
with *partition-index-map-inj-on* [*OF invar-P″*, *of* $\{i', i\}$] *i-le′*
show *?thesis*
apply (*simp add*: *image-iff partition-index-map-def Ball-def*
p″-intro[*symmetric*])
apply *auto*
done
qed
} note *step* = *this*

show *?thesis*
apply (*simp del*: *ex-simps add*: *pp-neq pp-neq*[*symmetric*] *ex-simps*[*symmetric*])
apply (*rule iff-exI*)
apply (*metis step*)
done
qed
next
fix *a pp*
{ fix *a pp*
have $(a, pp) \in L'' \longleftrightarrow$
$(\exists\, i'.\ (a, i') \in sL \wedge pp = the\ (im'\ i'))$
unfolding *L″-eq*
by (*simp add*: *image-iff Bex-def partition-index-map-def*)
moreover
{ fix i'
have $(a, i') \in sL \wedge pp = the\ (im'\ i') \longleftrightarrow$
$(a, i') \in sL \wedge i' < n' \wedge (im'\ i' = Some\ pp)$
proof $(cases\ (a, i') \in sL)$
case *False* **thus** *?thesis* **by** *simp*
next
case *True* **note** *ai′-in-sL* = *this*
from *sL-OK*[*OF ai′-in-sL*] **have** $i' < n'$ **.**
with *invar-P″* **have** $i' \in dom\ im'$ **by** *simp*
with *ai′-in-sL* ‹$i' < n'$› **show** *?thesis* **by** *auto*
qed
}
ultimately have $(a, pp) \in L'' \longleftrightarrow$
$(\exists\, i'.\ (a, i') \in sL \wedge i' < n' \wedge (im'\ i' = Some\ pp))$
by *simp*
} note *in-L″-eq* = *this*

{ fix i'
have $(im'\ i' = Some\ p'') \longleftrightarrow i' = i$
proof
assume $i' = i$
thus $im'\ i' = Some\ p''$
by (*simp add*: *im′-i-eq*)
next

assume im'-i'-eq: $im'\ i' = Some\ p''$
hence $i' \in dom\ im'$ **by** ($simp\ add$: dom-def)
with $invar$-P'' **have** $i' < n'$ **by** $simp$

from $partition$-$index$-map-inj-on [$OF\ invar$-P'', $of\ \{i,\ i'\}$] ‹$i' < n'$› i-le'
show $i' = i$
by ($auto\ simp\ add$: $image$-$iff\ Bex$-$def\ partition$-$index$-map-$def\ im'$-i'-$eq\ im'$-i-eq)
qed
} **note** im'-eq-$p'' = this$

show $((a,\ pp) \in L'' \wedge pp \neq p'' \vee pp = pmin \wedge a \in \Sigma\ \mathcal{A} \vee pp = pmax \wedge (a,\ p'') \in L'') =$
$(\exists\, i'.\ (i' = n' \wedge a \in \Sigma\ \mathcal{A} \vee (a,\ i') \in sL) \wedge$
$pp = partition$-$index$-map
$(im'(i \mapsto pmax,\ n' \mapsto pmin),\ \lambda q.\ if\ q \in pmin\ then\ Some\ n'\ else\ sm'\ q,\ Suc\ n')\ i')$
(**is** $?ls = (\exists\, i'.\ (i' = n' \wedge a \in \Sigma\ \mathcal{A} \vee (a,\ i') \in sL) \wedge pp = ?pm\ i')$)
unfolding in-L''-eq
apply ($rule\ iffI$)
apply ($elim\ disjE\ conjE\ exE$)
prefer 4
apply ($elim\ exE\ conjE\ disjE$)
proof −
fix i'
assume $pp = ?pm\ i'\ i' = n'\ a \in \Sigma\ \mathcal{A}$
hence $pp = pmin \wedge a \in \Sigma\ \mathcal{A}$
by ($simp\ add$: $partition$-$index$-map-def)
thus $(\exists\, i'.\ (a,\ i') \in sL \wedge i' < n' \wedge im'\ i' = Some\ pp) \wedge pp \neq p'' \vee$
$pp = pmin \wedge a \in \Sigma\ \mathcal{A} \vee$
$pp = pmax \wedge (\exists\, i'.\ (a,\ i') \in sL \wedge i' < n' \wedge im'\ i' = Some\ p'')$ **by** $simp$
next
assume $pp = pmin\ a \in \Sigma\ \mathcal{A}$
thus $\exists\, i'.\ (i' = n' \wedge a \in \Sigma\ \mathcal{A} \vee (a,\ i') \in sL) \wedge$
$pp = ?pm\ i'$
apply ($rule$-$tac\ exI$[**where** $x = n'$])
apply ($simp\ add$: $partition$-$index$-map-def)
done
next
fix i'
assume $pp = pmax\ (a,\ i') \in sL\ i' < n'\ im'\ i' = Some\ p''$
moreover **from** ‹$im'\ i' = Some\ p''$› **have** $i' = i$ **by** ($simp\ add$: im'-eq-p'')
ultimately
show $\exists\, i'.\ (i' = n' \wedge a \in \Sigma\ \mathcal{A} \vee (a,\ i') \in sL) \wedge$
$pp = ?pm\ i'$
apply ($rule$-$tac\ exI$[**where** $x = i$])
apply ($simp\ add$: $partition$-$index$-map-def)
done
next
fix i'
assume $pp \neq p''\ (a,\ i') \in sL\ i' < n'\ im'\ i' = Some\ pp$
moreover **from** ‹$im'\ i' = Some\ pp$› ‹$pp \neq p''$›
have $i' \neq i$ **using** im'-eq-p'' [$of\ i'$, $symmetric$]
by $simp$
ultimately
show $\exists\, i'.\ (i' = n' \wedge a \in \Sigma\ \mathcal{A} \vee (a,\ i') \in sL) \wedge$
$pp = ?pm\ i'$
apply ($rule$-$tac\ exI$[**where** $x = i'$])
apply ($simp\ add$: $partition$-$index$-map-def)
done
next
fix i'
assume pp-eq: $pp = ?pm\ i'$ **and** in-sL: $(a,\ i') \in sL$

from sL-OK[$OF\ in$-sL] **have** i'-le: $i' < n'$ **.**

with *invar-P''* **have** $i' \in dom\ im'$ **by** *simp*
then obtain *pp'* **where** *im'-i'-eq*: $im'\ i' = Some\ pp'$ **by** *blast*

show $(\exists i'.\ (a,\ i') \in sL \wedge i' < n' \wedge im'\ i' = Some\ pp) \wedge pp \neq p'' \vee$
$pp = pmin \wedge a \in \Sigma\ \mathcal{A} \vee$
$pp = pmax \wedge (\exists i'.\ (a,\ i') \in sL \wedge i' < n' \wedge im'\ i' = Some\ p'')$
proof $(cases\ i' = i)$
case *True* **note** $i'\text{-}eq = this$
with *pp-eq i-le'* **have** $pp = pmax$
by (*simp add*: *partition-index-map-def*)
with *in-sL i'-eq* **have** $pp = pmax \wedge (\exists i'.\ (a,\ i') \in sL \wedge i' < n' \wedge im'\ i' = Some\ p'')$
apply *simp*
apply (*rule exI* [**where** $x = i$])
apply (*simp add*: *im'-i-eq i-le'*)
done
thus *?thesis* **by** *blast*
next
case *False* **note** $i'\text{-}neq = this$
with *im'-i'-eq pp-eq i'-le* **have** *pp-eq*: $pp = pp'$
by (*simp add*: *partition-index-map-def*)

with *i'-neq im'-eq-p''* [*of i'*] **have** *pp-neq*: $pp \neq p''$
by (*simp add*: *im'-i'-eq*)

from *in-sL i'-le pp-eq* **have** $\exists i'.\ ((a,\ i') \in sL \wedge i' < n' \wedge im'\ i' = Some\ pp)$
apply (*rule-tac exI* [**where** $x = i'$])
apply (*simp add*: *im'-i'-eq*)
done
with $\langle pp \neq p'' \rangle$ **show** *?thesis* **by** *blast*
qed
qed
qed
from *alpha-OK invar-OK* **show** *?thesis* **by** (*simp add*: *Hopcroft-map-state-rel-def*)
qed

from *in-rel' n-le i-le card-pf'-neq n'-not-in im'-eq*
show *?thesis*
by (*simp add*: *partition-index-map-def im-i-eq Ball-def pw-le-iff refine-pw-simps Hopcroft-map-state-rel-def*)
qed
thus $(let\ (pt',\ pct',\ pf',\ pcf') =$
$(\{q \in partition\text{-}index\text{-}map\ (im',\ sm',\ n')\ i.\ q \in pre\},$
$card\ \{q \in partition\text{-}index\text{-}map\ (im',\ sm',\ n')\ i.\ q \in pre\},$
$\{q \in partition\text{-}index\text{-}map\ (im',\ sm',\ n')\ i.\ q \notin pre\},$
$card\ \{q \in partition\text{-}index\text{-}map\ (im',\ sm',\ n')\ i.\ q \notin pre\})$
$in\ if\ pcf' = 0\ then\ RETURN\ ((im',\ sm',\ n'),\ sL)$
$else\ let\ (pmin,\ pmax) = if\ pcf' < pct'\ then\ (pf',\ pt')\ else\ (pt',\ pf');$
$L' = \{(a,\ partition\text{-}free\text{-}index\ (im',\ sm',\ n'))\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup sL;$
$P' = partition\text{-}map\text{-}split\ (im',\ sm',\ n')\ i\ pmin\ pmax$
$in\ RETURN\ (P',\ L'))$
$\leq \Downarrow \{(\sigma,\ \sigma').$
$(\sigma,\ \sigma') \in Hopcroft\text{-}map\text{-}state\text{-}rel\ \wedge$
$partition\text{-}free\text{-}index\ P \leq partition\text{-}free\text{-}index\ (fst\ \sigma)\ \wedge$
$(\forall i{\in}it - \{i\}.\ fst\ (fst\ \sigma)\ i = fst\ P\ i)\}$
$(let\ (pt',\ pct',\ pf',\ pcf') =$
$(\{q \in partition\text{-}index\text{-}map\ P\ i'.\ q \in pre\},$
$card\ \{q \in partition\text{-}index\text{-}map\ P\ i'.\ q \in pre\},$
$\{q \in partition\text{-}index\text{-}map\ P\ i'.\ q \notin pre\},$
$card\ \{q \in partition\text{-}index\text{-}map\ P\ i'.\ q \notin pre\})$
$in\ if\ pcf' = 0\ then\ RETURN\ (P'',\ L'')$
$else\ let\ (pmin,\ pmax) = if\ pcf' < pct'\ then\ (pf',\ pt')\ else\ (pt',\ pf');$
$P' = P'' - \{partition\text{-}index\text{-}map\ P\ i'\} \cup \{pt',\ pf'\};$

$L' = \{(a,\ p'').\ (a,\ p'') \in L'' \wedge p'' \neq \textit{partition-index-map}\ P\ i'\} \cup$
$\{(a,\ pmin)\ |a.\ a \in \Sigma\ \mathcal{A}\} \cup$
$\{(a,\ pmax)\ |a.\ (a,\ \textit{partition-index-map}\ P\ i') \in L''\}$
in RETURN $(P',\ L')$)
apply (*simp split del*: *if-splits*
del: *P-eq*
add: *p''-def*[*symmetric*] *pt'-def*[*symmetric*] *pf'-def*[*symmetric*] *p''-intro*)
apply (*unfold pminmax*)
apply (*cases card pf' = 0*)
apply (*simp-all*)
done
}
qed

definition *Hopcroft-map-f* **where**
Hopcroft-map-f $\mathcal{A}$ =
(λ(*P*, *L*). *do* {
ASSERT (*Hopcroft-abstract-invar* $\mathcal{A}$ (*Hopcroft-map-state-α* (*P*, *L*)));
ASSERT ($L \neq \{\}$);
(*a*,*i*) ← *SPEC* (λ(*a*,*i*). (*a*,*i*) ∈ *L*);
ASSERT (*i* ∈ *dom* (*fst P*));
let pre = $\{q.\ \exists\, q'.\ q' \in$ *partition-index-map P i* $\wedge\ (q,\ a,\ q') \in \Delta\ \mathcal{A}\}$;
$(P',\ L')$ ← *Hopcroft-map-step* $\mathcal{A}$ *i a pre P L*;
RETURN $(P',\ L')$
})

lemma (**in** *DFA*) *Hopcroft-map-f-correct* :
assumes *PL-OK*: (*PL*, *PL'*) ∈ *Hopcroft-map-state-rel*
shows *Hopcroft-map-f* $\mathcal{A}$ *PL* ≤ ⇓*Hopcroft-map-state-rel* (*Hopcroft-abstract-f* $\mathcal{A}$ *PL'*)
proof −
obtain *P L* **where** *PL-eq*[*simp*]: *PL* = (*P*, *L*) **by** (*rule PairE*)
obtain *P' L'* **where** *PL'-eq*[*simp*]: *PL'* = (*P'*, *L'*) **by** (*rule PairE*)

from *PL-OK* **have** *PL-α*: *Hopcroft-map-state-α* (*P*, *L*) = (*P'*, *L'*)
and *invar*: *Hopcroft-map-state-invar* (*P*, *L*)
by (*simp-all add*: *Hopcroft-map-state-rel-def*)

from *PL-α* **have** *P'-eq*: *P'* = *partition-map-α P*
and *L'-eq*: *L'* = *apsnd* (*partition-index-map P*) ‘ *L*
by (*simp-all add*: *Hopcroft-map-state-α-def*)

show *?thesis*
unfolding *Hopcroft-map-f-def Hopcroft-abstract-f-def*
apply *simp*
apply (*refine-rcg*)
apply (*simp-all only*: *PL-α*)
apply (*simp add*: *L'-eq*)
prefer *3*
apply *clarify* **apply** *simp*
proof −
show (*RES L*) ≤ ⇓ (*build-rel* (*apsnd* (*partition-index-map P*)) ($\lambda x.\ x \in L$)) (*RES L'*)
apply (*simp del*: *br-def add*: *pw-le-iff refine-pw-simps*)
apply (*simp add*: *L'-eq subset-iff image-iff Bex-def*)
apply *blast*
done
next
fix *a i a' p'*
assume *abstr-invar*: *Hopcroft-abstract-invar* $\mathcal{A}$ (*P'*, *L'*)
and *in-rel*: ((*a*, *i*), (*a'*, *p'*)) ∈ *build-rel* (*apsnd* (*partition-index-map P*)) ($\lambda x.\ x \in L$)

from *in-rel* **have** *a'-eq*[*simp*]: *a'* = *a*
and *p'-eq*: *p'* = *partition-index-map P i*

```
          and ai-in-L: (a, i) ∈ L
     by simp-all

   obtain im sm n where P-eq: P = (im, sm, n) by (metis PairE)

   show i ∈ dom (fst P)
     using ai-in-L invar
     unfolding Hopcroft-map-state-invar-def
     by (simp add: P-eq Ball-def)

   from abstr-invar have is-part-P': is-partition (Q 𝒜) P' and
                         L'-OK: ⋀a p. (a, p) ∈ L' ⟹ a ∈ Σ 𝒜
     unfolding Hopcroft-abstract-invar-def is-weak-language-equiv-partition-def by auto
   def pre ≡ {q. ∃q'. q' ∈ p' ∧ (q, a, q') ∈ Δ 𝒜}

   have Hopcroft-map-step 𝒜 i a {q. ∃q'. q' ∈ partition-index-map P i ∧ (q, a, q') ∈ Δ 𝒜} P L
    ≤ ⇓ Hopcroft-map-state-rel
       (Hopcroft-precompute-step 𝒜 p' a' {q. ∃q'. q' ∈ p' ∧ (q, a', q') ∈ Δ 𝒜} P' L')
     apply (unfold pre-def[symmetric] p'-eq[symmetric] a'-eq)
     apply (rule-tac Hopcroft-map-step-correct)
     apply (insert PL-OK, simp)[]
     apply (insert ai-in-L invar, simp add: Hopcroft-map-state-invar-def Ball-def) []
     apply (simp add: p'-eq)
     apply (simp add: is-part-P')
     apply (insert Δ-consistent, auto simp add: pre-def)
   done
   also note Hopcroft-precompute-step-correct
   also note Hopcroft-set-step-correct
   finally show Hopcroft-map-step 𝒜 i a
     {q. ∃q'. q' ∈ partition-index-map P i ∧ (q, a, q') ∈ Δ 𝒜} P L
    ≤ ⇓ Hopcroft-map-state-rel
       (SPEC (λ(P'', L'').
              Hopcroft-update-splitters-pred 𝒜 p' a' P' L' L'' ∧
              P'' = Hopcroft-split 𝒜 p' a' {} P'))
     by (simp add: is-part-P' L'-OK is-partition-memb-finite[OF finite-Q is-part-P'])
 qed
qed


definition partition-map-init where
 partition-map-init 𝒜 =
  (if (Q 𝒜 − ℱ 𝒜 = {}) then
    partition-map-sing (ℱ 𝒜)
   else
    (empty (0 ↦ (ℱ 𝒜)) (1 ↦ (Q 𝒜 − ℱ 𝒜)),
     (λq. if (q ∈ ℱ 𝒜) then Some 0 else
          if (q ∈ Q 𝒜) then Some 1 else None),
     2))

lemma partition-map-init-correct :
assumes wf-A: NFA 𝒜
    and f-neq-emp: ℱ 𝒜 ≠ {}
shows partition-map-α (partition-map-init 𝒜) = Hopcroft-accepting-partition 𝒜 ∧
      partition-map-invar (partition-map-init 𝒜)
proof (cases Q 𝒜 ⊆ ℱ 𝒜)
 case True
   with f-neq-emp show ?thesis
   unfolding partition-map-init-def Hopcroft-accepting-partition-alt-def[OF wf-A]
   by (auto simp add: partition-map-sing-correct)
next
 case False
 with f-neq-emp show ?thesis
```

unfolding *partition-map-init-def Hopcroft-accepting-partition-alt-def* [*OF wf-A*]
by *auto*
qed

lemma *partition-map-init---index-0*:
partition-index-map (*partition-map-init* $\mathcal{A}$) *0* = $\mathcal{F}$ $\mathcal{A}$
unfolding *partition-map-init-def partition-index-map-def partition-map-sing-def*
by *auto*

lemma *partition-map-init---index-le*:
0 < *partition-free-index* (*partition-map-init* $\mathcal{A}$)
unfolding *partition-map-init-def partition-index-map-def partition-map-sing-def*
by *auto*

definition *Hopcroft-map-init* **where**
Hopcroft-map-init $\mathcal{A}$ = (*partition-map-init* $\mathcal{A}$, {(*a*, *0*) | *a*. *a* ∈ Σ $\mathcal{A}$})

definition *Hopcroft-map* **where**
Hopcroft-map $\mathcal{A}$ =
(*if* ($\mathcal{Q}$ $\mathcal{A}$ = {}) *then RETURN* (*partition-map-empty*) *else* (
if ($\mathcal{F}$ $\mathcal{A}$ = {}) *then RETURN* (*partition-map-sing* ($\mathcal{Q}$ $\mathcal{A}$)) *else* (
do {
(*P*, -) ← *WHILET* (λ*PL*. (*snd PL* ≠ {}))
(*Hopcroft-map-f* $\mathcal{A}$) (*Hopcroft-map-init* $\mathcal{A}$);
RETURN P
})))

lemma (**in** *DFA*) *Hopcroft-map-correct* :
Hopcroft-map $\mathcal{A}$ ≤ ⇓(*build-rel partition-map-α partition-map-invar*) (*Hopcroft-abstract* $\mathcal{A}$)
unfolding *Hopcroft-map-def Hopcroft-abstract-def*
apply *refine-rcg*
apply (*simp-all add*: *partition-map-empty-correct partition-map-sing-correct*)
proof −
assume $\mathcal{F}$ $\mathcal{A}$ ≠ {}
thus (*Hopcroft-map-init* $\mathcal{A}$, *Hopcroft-abstract-init* $\mathcal{A}$) ∈ *Hopcroft-map-state-rel*
unfolding *Hopcroft-map-state-rel-def Hopcroft-abstract-init-def Hopcroft-map-init-def*
apply (*simp add*: *Hopcroft-map-state-α-def Hopcroft-map-state-invar-def*
partition-map-init-correct[*OF wf-NFA*] *partition-map-init---index-le*)
apply (*rule set-eqI*)
apply (*simp add*: *image-iff partition-map-init---index-0*)
apply (*auto*)
done
next
fix *PL* **and** *PL′* :: (*′q set set*) × (*′a* × (*′q set*)) *set*
assume (*PL*, *PL′*) ∈ *Hopcroft-map-state-rel*
thus (*snd PL* ≠ {}) = *Hopcroft-abstract-b PL′*
unfolding *Hopcroft-abstract-b-def Hopcroft-map-state-rel-def*
by (*cases PL*, *simp add*: *Hopcroft-map-state-α-def*)
next
fix *PL* **and** *PL′* :: (*′q set set*) × (*′a* × (*′q set*)) *set*
assume (*PL*, *PL′*) ∈ *Hopcroft-map-state-rel*
thus *Hopcroft-map-f* $\mathcal{A}$ *PL* ≤ ⇓ *Hopcroft-map-state-rel* (*Hopcroft-abstract-f* $\mathcal{A}$ *PL′*)
by (*rule Hopcroft-map-f-correct*)
next
fix *P L P′* **and** *L′* :: (*′a* × (*′q set*)) *set*
assume ((*P*, *L*), (*P′*, *L′*)) ∈ *Hopcroft-map-state-rel*
thus *P′* = *partition-map-α P* ∧ *partition-map-invar P*
by (*simp add*: *Hopcroft-map-state-rel-def Hopcroft-map-state-α-def*
Hopcroft-map-state-invar-def)
qed

lemma (**in** *DFA*) *Hopcroft-map-correct-full* :

Hopcroft-map $\mathcal{A}$ ≤ *SPEC* (λ*P*.
  (*partition-map-α P* = *Myhill-Nerode-partition* $\mathcal{A}$) ∧
  partition-map-invar P)
proof −
 note *Hopcroft-map-correct*
 also note *Hopcroft-abstract-correct*
 finally show *?thesis* **by** (*simp add*: *pw-le-iff refine-pw-simps*)
qed

Using this encoding of the partition, it's even easier to construct a rename function

lemma *partition-map-to-rename-fun-OK* :
assumes *invar*: *partition-map-invar P*
  and *part-map-OK*: *partition-map-α P* = *Myhill-Nerode-partition* $\mathcal{A}$
shows *dom* (*fst* (*snd P*)) = $\mathcal{Q}$ $\mathcal{A}$
   NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ (λ*q*. *states-enumerate* (*the* ((*fst* (*snd P*)) *q*)))
proof −
 obtain *im sm n* **where** *P-eq*[*simp*]: *P* = (*im*, *sm*, *n*) **by** (*metis PairE*)

 from *invar* **have** *dom-i*: ⋀*i Q*. *im i* = *Some Q* ⟹ *i* < *n* **by** (*auto simp add*: *dom-def*)
 from *invar* **have** *sm-eq*: ⋀*q i*. (*sm q* = *Some i*) = (∃ *p*. *im i* = *Some p* ∧ *q* ∈ *p*) **by** *simp*

 from *part-map-OK dom-i* **have**
  is-part: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) (*partition-map-α P*) **and**
  strong: ⋀*Q i*. *im i* = *Some Q* ⟹ *is-strong-language-equiv-set* $\mathcal{A}$ *Q*
  apply (*simp-all add*: *is-strong-language-equiv-partition-fun-alt-def dom-def*)
  apply *metis*
 done

 from *is-part* **have** *Q-eq*: $\mathcal{Q}$ $\mathcal{A}$ = ⋃{*Q*. ∃ *i*<*n*. *im i* = *Some Q*}
  unfolding *is-partition-def* **by** *simp*

 from *invar*
 show *dom* (*fst* (*snd P*)) = $\mathcal{Q}$ $\mathcal{A}$
  by (*auto simp add*: *dom-def Q-eq*)

 show *NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ (λ*q*. *states-enumerate* (*the* ((*fst* (*snd P*)) *q*)))
  unfolding *NFA-is-strong-equivalence-rename-fun-def*
 proof (*intro ballI*)
  fix *q q'*
  assume *q* ∈ $\mathcal{Q}$ $\mathcal{A}$ *q'* ∈ $\mathcal{Q}$ $\mathcal{A}$

  with *Q-eq* **obtain** *i i' Q Q'* **where**
   iQ-props: *i* < *n* *i'* < *n* *im i* = *Some Q* *im i'* = *Some Q'* *q* ∈ *Q* *q'* ∈ *Q'*
   by *auto*
  with *invar* **have** *sm-eq*: *sm q* = *Some i* *sm q'* = *Some i'* **by** *auto*

  from *invar iQ-props sm-eq*
  have *im-ii'*: *im i* = *im i'* ⟹ *i* = *i'*
   apply *simp*
   apply (*subgoal-tac sm q'* = *Some i*)
   apply (*simp add*: *sm-eq*)
   apply *auto*
  done

  from *iQ-props* **have** *Q* ∈ *partition-map-α P* *Q'* ∈ *partition-map-α P* **by** *auto*
  with *part-map-OK* **have**
    Q ∈ *Myhill-Nerode-partition* $\mathcal{A}$
    Q' ∈ *Myhill-Nerode-partition* $\mathcal{A}$
   by *simp-all*
  then obtain *q1 q1'* **where**
    Q = {*q'* ∈ $\mathcal{Q}$ $\mathcal{A}$. $\mathcal{L}$*-in-state* $\mathcal{A}$ *q'* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1*} *q1* ∈ $\mathcal{Q}$ $\mathcal{A}$
    Q' = {*q'* ∈ $\mathcal{Q}$ $\mathcal{A}$. $\mathcal{L}$*-in-state* $\mathcal{A}$ *q'* = $\mathcal{L}$*-in-state* $\mathcal{A}$ *q1'*} *q1'* ∈ $\mathcal{Q}$ $\mathcal{A}$

by (*auto simp add*: *Myhill-Nerode-partition-alt-def*)
with *iQ-props im-ii′* **have** $(i = i') = (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q')$
apply (*simp add*: *Myhill-Nerode-partition-alt-def*)
apply (*rule iffI*)
apply (*auto*)
done
thus $((states\text{-}enumerate\ (the\ (fst\ (snd\ P)\ q)) = states\text{-}enumerate\ (the\ (fst\ (snd\ P)\ q')))) = (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q')$
by (*simp add*: *sm-eq states-enumerate-eq*)
qed
qed

4.8 Code Generation

locale *Hopcroft-impl-locale* =
s : *StdSet s-ops* +
sp : *StdSet sp-ops* +
n : *StdSet n-ops* +
sm : *StdMap sm-ops* +
im : *StdMap im-ops* +
sm-it : *set-iteratei set-op-α sp-ops set-op-invar sp-ops sm-it* +
ns-it : *set-iteratei set-op-α n-ops set-op-invar n-ops ns-it* +
split-it : *set-iteratei set-op-α sp-ops set-op-invar sp-ops split-it* +
s-image: *set-image set-op-α s-ops set-op-invar s-ops set-op-α n-ops set-op-invar n-ops s-image*
for *s-ops* :: (*′q*, *′q-set*, -) *set-ops-scheme*
and *sp-ops* :: (*′q*, *′q-set-p*, -) *set-ops-scheme*
and *n-ops* :: (*nat*, *′n-set*, -) *set-ops-scheme*
and *sm-ops* :: (*′q*, *nat*, *′sm*, -) *map-ops-scheme*
and *im-ops* :: (*nat*, *′q-set-p*, *′im*, -) *map-ops-scheme*
and *sm-it* :: (*′q-set-p*, *′q*, *′sm*) *iteratori*
and *split-it* :: (*′q-set-p*, *′q*, (*′q-set-p* × *nat* × *′q-set-p* × *nat*)) *iteratori*
and *ns-it*:: (*′n-set*, *nat*, (*′im* × *′sm* × *nat*) × (*′a* × *nat*) *list*) *iteratori*
and *s-image* +
fixes *q-set-convert* :: *′q-set* ⇒ *′q-set-p*
assumes *q-set-convert-correct* :
s.invar Q ⟹ *sp.invar* (*q-set-convert Q*)
s.invar Q ⟹ *sp.α* (*q-set-convert Q*) = *s.α Q*
begin

definition *im-ext-invar* **where**
$im\text{-}ext\text{-}invar\ im \equiv (im.invar\ im) \wedge (\forall\, i\ Q.\ im.\alpha\ im\ i = Some\ Q \longrightarrow sp.invar\ Q)$

definition *im-ext-α* **where**
$im\text{-}ext\text{-}\alpha\ im \equiv (\lambda i.\ Option.map\ sp.\alpha\ (im.\alpha\ im\ i))$

definition *partition-impl-invar* **where**
$partition\text{-}impl\text{-}invar\ P = (im\text{-}ext\text{-}invar\ (fst\ P) \wedge (sm.invar\ (fst\ (snd\ P))))$

definition *partition-impl-α* **where**
$partition\text{-}impl\text{-}\alpha\ P = (im\text{-}ext\text{-}\alpha\ (fst\ P),\ sm.\alpha\ (fst\ (snd\ P)),\ snd\ (snd\ P))$

definition *partition-impl-rel* **where**
partition-impl-rel = *build-rel partition-impl-α partition-impl-invar*

lemma *partition-impl-rel-sv*[*refine*] :
single-valued partition-impl-rel
unfolding *partition-impl-rel-def*
by (*rule br-single-valued*)

definition *partition-impl-empty* **where**
partition-impl-empty = (*im.empty*, *sm.empty*, *0*)

lemma *partition-impl-empty-correct* :
(*partition-impl-empty*, *partition-map-empty*) ∈ *partition-impl-rel*
unfolding *partition-impl-empty-def partition-map-empty-def partition-impl-rel-def*
by (*simp add*: *partition-impl-invar-def partition-impl-α-def sm.correct im.correct*
im-ext-invar-def im-ext-α-def)

definition *sm-update* **where**
sm-update Q n sm = *sm-it* (λ-. *True*) (λ*q sm*. *sm.update q n sm*) *Q sm*

lemma *sm-update-correct* :
assumes *invar-sm*: *sm.invar sm*
and *invar-Q*: *sp.invar Q*
shows *sm.invar* (*sm-update Q n sm*) ∧
sm.α (*sm-update Q n sm*) = (λ*q*. *if* (*q* ∈ *sp.α Q*) *then Some n else sm.α sm q*)
unfolding *sm-update-def*
apply (*rule-tac sm-it.iteratei-rule-insert-P* [**where** *I* = λ*Q σ*. *sm.invar σ* ∧ *sm.α σ* = (λ*q*. *if q* ∈ *Q then Some n else sm.α sm q*)])
apply (*auto simp add*: *sm.correct invar-sm invar-Q*)
done

definition *partition-impl-sing* :: ′*q-set* ⇒ (′*im* × ′*sm* × *nat*) **where**
partition-impl-sing Q ≡
(*let QP* = *q-set-convert Q in*
(*im.sng 0 QP*, *sm-update QP 0 sm.empty*, *1*))

lemma *partition-impl-sing-correct* :
assumes *Q-in-rel*: (*Q*, *QQ*) ∈ *build-rel s.α s.invar*
shows (*partition-impl-sing Q*, *partition-map-sing QQ*) ∈ *partition-impl-rel*
using *Q-in-rel*
by (*auto simp add*: *partition-impl-rel-def partition-map-sing-def partition-impl-sing-def*
partition-impl-α-def partition-impl-invar-def
im-ext-α-def im.correct im-ext-invar-def Let-def
sm-update-correct sm.correct q-set-convert-correct)

definition *partition-impl-init* **where**
partition-impl-init F Q =
(*let QF* = *s.diff Q F in*
(*if* (*s.isEmpty QF*) *then partition-impl-sing F else*
(*let FP* = *q-set-convert F in*
let QFP = *q-set-convert QF in*
(*im.update 0 FP* (*im.sng 1 QFP*), *sm-update FP 0* (*sm-update QFP 1 sm.empty*), *2*))))

lemma *partition-impl-init-correct* :
assumes *Q-in-rel*: (*Q*, $\mathcal{Q}$ $\mathcal{A}$) ∈ *build-rel s.α s.invar*
and *F-in-rel*: (*F*, $\mathcal{F}$ $\mathcal{A}$) ∈ *build-rel s.α s.invar*
shows (*partition-impl-init F Q*, *partition-map-init* $\mathcal{A}$) ∈ *partition-impl-rel*
proof (*cases* $\mathcal{Q}$ $\mathcal{A}$ − $\mathcal{F}$ $\mathcal{A}$ = {})
case *True*
with *partition-impl-sing-correct* [*OF F-in-rel*] *Q-in-rel F-in-rel*
show *?thesis* **by** (*simp add*: *partition-impl-init-def Let-def s.correct partition-map-init-def*)
next
case *False*
with *Q-in-rel F-in-rel*
show *?thesis*
by (*auto simp add*: *partition-impl-init-def Let-def s.correct partition-map-init-def*
im.correct sm-update-correct partition-impl-rel-def partition-impl-α-def
sm.correct im-ext-α-def partition-impl-invar-def im-ext-invar-def
q-set-convert-correct)
qed

definition *partition-impl-split* **where**
partition-impl-split P i pmin pmax =

$(im.update\ (snd\ (snd\ P))\ pmin\ (im.update\ i\ pmax\ (fst\ P)),$
$sm\text{-}update\ pmin\ (snd\ (snd\ P))\ (fst\ (snd\ P)),\ Suc\ (snd\ (snd\ P)))$

lemma *partition-impl-split-correct* :
assumes *pmin-in-rel*: $(pmin, pmin') \in build\text{-}rel\ sp.\alpha\ sp.invar$
and *pmax-in-rel*: $(pmax, pmax') \in build\text{-}rel\ sp.\alpha\ sp.invar$
and *P-in-rel*: $(P, P') \in partition\text{-}impl\text{-}rel$
shows $(partition\text{-}impl\text{-}split\ P\ i\ pmin\ pmax,$
$partition\text{-}map\text{-}split\ P'\ i\ pmin'\ pmax') \in partition\text{-}impl\text{-}rel$
using *assms*
apply (*cases P*, *rename-tac im sm n*)
apply (*cases P'*, *rename-tac im' sm' n'*)
apply (*simp add*: *partition-impl-rel-def*
partition-impl-invar-def sm-update-correct partition-impl-α-def
im-ext-invar-def im.correct im-ext-α-def partition-impl-split-def)
apply (*simp add*: *fun-eq-iff*)
done

definition *Hopcroft-impl-split-count-set* :: $('q \Rightarrow bool) \Rightarrow 'q\text{-}set\text{-}p \Rightarrow ('q\text{-}set\text{-}p \times nat \times 'q\text{-}set\text{-}p \times nat)$
where
$Hopcroft\text{-}impl\text{-}split\text{-}count\text{-}set\ P\ S =$
$split\text{-}it\ (\lambda\text{-}.\ True)\ (\lambda q\ (pt', pct', pf', pcf').$
$(if\ (P\ q)\ then\ (sp.ins\text{-}dj\ q\ pt', Suc\ pct', pf', pcf')\ else$
$(pt', pct', sp.ins\text{-}dj\ q\ pf', Suc\ pcf')))$
$S\ (sp.empty, 0, sp.empty, 0)$

lemma *Hopcroft-impl-split-count-set-correct* :
assumes *invar-S*: *sp.invar S*
shows $let\ (pt, pct, pf, pcf) = Hopcroft\text{-}impl\text{-}split\text{-}count\text{-}set\ P\ S\ in$
$(sp.invar\ pt \land sp.invar\ pf \land sp.\alpha\ pt = \{q.\ q \in sp.\alpha\ S \land P\ q\} \land$
$(pct = card\ (sp.\alpha\ pt)) \land sp.\alpha\ pf = \{q.\ q \in sp.\alpha\ S \land \neg(P\ q)\} \land$
$(pcf = card\ (sp.\alpha\ pf)))$
proof −
let $?I = \lambda SS\ (pt, pct, pf, pcf).$
$(sp.invar\ pt \land sp.invar\ pf \land (sp.\alpha\ pt = \{q.\ q \in SS \land P\ q\}) \land$
$(pct = card\ (sp.\alpha\ pt)) \land (sp.\alpha\ pf = \{q.\ q \in SS \land \neg(P\ q)\}) \land$
$(pcf = card\ (sp.\alpha\ pf)))$

have $?I\ (sp.\alpha\ S)\ (Hopcroft\text{-}impl\text{-}split\text{-}count\text{-}set\ P\ S)$
unfolding *Hopcroft-impl-split-count-set-def*
apply (*rule split-it.iteratei-rule-insert-P* [**where** $I = ?I$])
apply (*simp-all add*: *invar-S sp.correct split-def*)
apply (*auto*)
apply (*rule-tac card-insert-disjoint*[*symmetric*])
apply (*metis sp.finite*)
apply *simp*
apply (*rule-tac card-insert-disjoint*[*symmetric*])
apply (*metis sp.finite*)
apply *simp*
done
thus *?thesis* **by** *simp*
qed

lemma *Hopcroft-impl-split-count-set-correctD* :
assumes $Hopcroft\text{-}impl\text{-}split\text{-}count\text{-}set\ P\ S = (pt, pct, pf, pcf)$
and *sp.invar S*
shows $(sp.invar\ pt \land sp.invar\ pf \land \{q.\ q \in sp.\alpha\ S \land P\ q\} = sp.\alpha\ pt \land$
$(card\ (sp.\alpha\ pt) = pct) \land \{q.\ q \in sp.\alpha\ S \land \neg(P\ q)\} = sp.\alpha\ pf \land$
$(card\ (sp.\alpha\ pf) = pcf))$
apply (*insert assms Hopcroft-impl-split-count-set-correct* [*of S P*])

apply (*simp add*: *split-def*)
done

definition *Hopcroft-impl-step* **where**
Hopcroft-impl-step pre (*AL*::*'a list*) *P L* =
 do {
  let PS = *s-image* (λq. (*the* (*sm.lookup q* (*fst* (*snd P*))))) *pre*;
  ASSERT (*n.invar PS*);
  (P', L') ← *FOREACH* (*n.α PS*)
   (λ(i'::*nat*) (P', L'). *do* {
    let (pt', pct', pf', pcf') = *Hopcroft-impl-split-count-set* (λq. *s.memb q pre*)
      (*the* (*im.lookup* i' (*fst* P')));
    if ($pcf' = 0$) *then* (*RETURN* (P', L')) *else*
    do {
     let (*pmin*, *pmax*) = (*if* ($pcf' < pct'$) *then* (pf', pt') *else* (pt', pf'));
     let L' = *map* (λa. (*a*, *snd* (*snd* P'))) *AL* @ L';
     let P' = *partition-impl-split* P' i' *pmin pmax*;
     RETURN (P', L')
    }
   }) (*P*, *L*);
  RETURN (P', L')
 }

lemma *Hopcroft-impl-step-correct* :
assumes *PL-OK*: ((*P*,*L*), (P',$L' - \{(a, p)\}$)) ∈ (*rprod partition-impl-rel* (*build-rel set distinct*))
 and *AL-OK*: *distinct AL set AL* = Σ $\mathcal{A}$
 and *pre-OK*: (*pre*, *pre'*) ∈ *build-rel s.α s.invar*
shows *Hopcroft-impl-step pre AL P L* ≤ ⇓(*rprod partition-impl-rel* (*build-rel set distinct*))
  (*Hopcroft-map-step* $\mathcal{A}$ *p a pre'* P' L')
unfolding *Hopcroft-impl-step-def Hopcroft-map-step-def*
using [[*goals-limit* = *20*]]
apply (*refine-rcg*)
— preprocess goals
defer
 apply (*rule inj-on-id*)
 apply *simp*
— goal solved
defer
 apply (*rule PL-OK*)
— goal solved
 apply (*intro rprod-sv partition-impl-rel-sv br-single-valued*)
— goal solved
 apply (*clarify*, *simp*)+
 apply (*rename-tac it im' sm' n' im sm n LL pt ptc pf pfc i Q*)
defer
 apply (*intro rprod-sv partition-impl-rel-sv br-single-valued*)
— goal solved
 apply *simp*
— goal solved
 apply (*intro rprod-sv partition-impl-rel-sv br-single-valued*)
— goal solved
 apply (*clarify*, *simp*)+
 apply (*rename-tac it im' sm' n' im sm n LL pt ptc pf pfc p-min' p-max' p-min p-max i Q*)
 apply (*intro conjI*)
defer
defer
defer
defer
 apply (*rule partition-impl-rel-sv*)
— goal solved
 apply (*simp*)

proof −
  from *pre-OK*
  show *n.invar* (*s-image* (λ*q*. *the* (*sm.lookup q* (*fst* (*snd P*)))) *pre*)
    by (*simp add*: *s-image.image-correct*)
next
  from *PL-OK pre-OK*
  show *partition-state-map P′* ‘ *pre′* =
        n.α (*s-image* (λ*q*. *the* (*map-op-lookup sm-ops q* (*fst* (*snd P*)))) *pre*)
    by (*simp add*: *partition-impl-rel-def sm.correct partition-impl-invar-def im-ext-invar-def partition-state-map-def-raw partition-impl-*α*-def s-image.image-correct*)
next
  fix *it im′ sm′ n′ im sm n LL pt ptc pf pfc i Q*
  assume *step-invar*: *Hopcroft-map-step-invar* $\mathcal{A}$ *p a P′ L′ it* ((*im′*, *sm′*, *n′*), *set LL*) **and**
        split-set: *Hopcroft-impl-split-count-set* (λ*q*. *s.memb q pre*) (*the* (*im.lookup i im*)) =
          (*pt*, *ptc*, *pf*, *pfc*) **and**
        i-in-it: *i* $\in$ *it* **and**
        im′-i[*simp*]: *im′ i* = *Some Q* **and**
        partition-rel: ((*im*, *sm*, *n*), *im′*, *sm′*, *n′*) $\in$ *partition-impl-rel*

  from *partition-rel im′-i* **obtain** *Q′* **where**
      invar-im: *im.invar im* **and**
      invar-Q′: *sp.invar Q′* **and**
      Q′-eq[*simp*]: *sp*.α *Q′* = *Q* **and**
      im-i[*simp*]: *im*.α *im i* = *Some Q′* **and**
      invar-sm: *sm.invar sm* **and**
      sm-eq[*simp*]: *sm*.α *sm* = *sm′* **and**
      n′-eq[*simp*]: *n′* = *n* **and**
      im′-eq[*simp*]: *im′* = (λ*i*. *Option.map sp*.α (*map-op-*α *im-ops im i*)) **and**
      invar-ext-im: *im-ext-invar im*
    unfolding *partition-impl-rel-def*
    by (*auto simp add*: *partition-impl-invar-def im-ext-invar-def partition-impl-*α*-def im-ext-*α*-def*)

  from *invar-im invar-Q′ Hopcroft-impl-split-count-set-correctD* [*OF split-set*] *pre-OK*
  show (*pfc* = *0*) = (*card* {*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) *i*. *q* $\notin$ *pre′*} = *0*)
    by (*simp add*: *s.correct im.correct partition-index-map-def*)

  show {(*a*, *n′*) |*a*. *a* $\in$ Σ $\mathcal{A}$} $\cup$ *set LL* = (λ*a*. (*a*, *n*)) ‘ *set AL* $\cup$ *set LL* **using** *AL-OK* **by** *auto*

  show *distinct* (*map* (λ*a*. (*a*, *n*)) *AL*) **using** *AL-OK*(*1*) **by** (*induct AL*, *auto*)

  from *step-invar* **have** *Hopcroft-map-state-invar* ((*im′*, *sm′*, *n*), *set LL*)
    by (*simp add*: *Hopcroft-map-step-invar-def*)
  hence $\bigwedge$*a i*. (*a*, *i*) $\in$ *set LL* $\Longrightarrow$ *i* < *n* **by** (*simp add*: *Hopcroft-map-state-invar-def Ball-def*)
  thus (λ*a*. (*a*, *n*)) ‘ *set AL* $\cap$ *set LL* = {} **by** *auto*

  fix *p-min p-max p-min′ p-max′*
  assume *p-minmax′*: (*if card* {*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) *i*. *q* $\notin$ *pre′*}
          < *card* {*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) *i*. *q* $\in$ *pre′*}
       then ({*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) (*id i*). *q* $\notin$ *pre′*},
             {*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) (*id i*). *q* $\in$ *pre′*})
       else ({*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) (*id i*). *q* $\in$ *pre′*},
             {*q* $\in$ *partition-index-map* (*im′*, *sm′*, *n′*) (*id i*). *q* $\notin$ *pre′*})) =
      (*p-min′*, *p-max′*)
  assume *p-minmax*: (*if pfc* < *ptc then* (*pf*, *pt*) *else* (*pt*, *pf*)) = (*p-min*, *p-max*)

  have *minmax-OK*: (*p-min*, *p-min′*) $\in$ *br sp*.α *sp.invar* $\wedge$ (*p-max*, *p-max′*) $\in$ *br sp*.α *sp.invar*
    using *p-minmax′ p-minmax Hopcroft-impl-split-count-set-correctD* [*OF split-set*] *invar-im invar-Q′ pre-OK*
    by (*cases pfc* < *ptc*, *simp-all add*: *im.correct s.correct partition-index-map-def*)

```
  from minmax-OK invar-sm invar-ext-im
  show (partition-impl-split (im, sm, n) i p-min p-max, im′(i ↦ p-max′, n′ ↦ p-min′),
        λq. if q ∈ p-min′ then Some n′ else sm′ q, Suc n′) ∈ partition-impl-rel
    unfolding partition-impl-rel-def
    apply (simp add: partition-impl-α-def im-ext-α-def invar-im im.correct sm-update-correct
                     partition-impl-invar-def im-ext-invar-def partition-impl-split-def)
    apply auto
  done
qed

definition Hopcroft-impl-f where
Hopcroft-impl-f pre-fun AL = (λ(P, L).
 do {
    let (a,i) = hd L;
    let pre = pre-fun a (the (im.lookup i (fst P)));
    (P′, L′) ← Hopcroft-impl-step pre AL P (tl L);
    RETURN (P′, L′)
 })


lemma Hopcroft-impl-f-correct :
assumes PL-OK: ((P,L), (P′,L′)) ∈ (rprod partition-impl-rel (build-rel set distinct))
    and AL-OK: distinct AL set AL = Σ 𝒜
    and pre-fun-OK: ⋀a S. ⟦a ∈ Σ 𝒜; sp.invar S⟧ ⟹
                    (s.invar (pre-fun a S) ∧ (s.α (pre-fun a S) =
                       {q . ∃ q′. (q, a, q′) ∈ Δ 𝒜 ∧ q′ ∈ sp.α S}))
shows (Hopcroft-impl-f pre-fun AL (P,L))  ≤ ⇓(rprod partition-impl-rel (build-rel set distinct))
       (Hopcroft-map-f 𝒜 (P′, L′))
unfolding Hopcroft-impl-f-def Hopcroft-map-f-def
using [[goals-limit = 4]]
apply (refine-rcg)
— preprocess goals
  apply (insert PL-OK)[]
  apply (simp)
— goal solved
  apply (insert PL-OK)[]
  apply (cases L, simp)
  apply (clarify, simp)+
  apply (rename-tac im′ sm′ n′ im sm n a i L′ Q)
  apply (rule Hopcroft-impl-step-correct)
  — process subgoals
    apply (simp)
    apply (simp add: AL-OK)
    apply (simp add: AL-OK)
defer
  apply (simp add: partition-impl-rel-sv)
— goal solved
  apply (simp)
proof −
  fix im′ sm′ n′ im sm n a i L′ Q

  assume abstract-invar: Hopcroft-abstract-invar 𝒜 (Hopcroft-map-state-α ((im′, sm′, n′), insert (a, i) (set L′)))
     and im′-i: im′ i = Some Q
     and partition-rel: ((im, sm, n), im′, sm′, n′) ∈ partition-impl-rel
     and (a, i) ∉ set L′ distinct L′

  from abstract-invar have a-in: a ∈ Σ 𝒜
    by (simp add: Hopcroft-abstract-invar-def Hopcroft-map-state-α-def)

  from partition-rel im′-i obtain Q′ where
     invar-im: im.invar im and
```

invar-Q': *sp.invar Q'* **and**
Q'-eq[simp]: $sp.\alpha$ $Q' = Q$ **and**
im-i[simp]: $im.\alpha$ *im i* = *Some Q'* **and**
invar-sm: *sm.invar sm* **and**
sm-eq[simp]: $sm.\alpha$ *sm* = *sm'* **and**
n'-eq[simp]: $n' = n$ **and**
im'-eq[simp]: *im'* = (λi. *Option.map* (*set-op-α sp-ops*) (*map-op-α im-ops im i*)) **and**
invar-ext-im: *im-ext-invar im*
unfolding *partition-impl-rel-def*
by (*auto simp add*: *partition-impl-invar-def im-ext-invar-def partition-impl-α-def im-ext-α-def*)

from *pre-fun-OK[OF a-in, of (the (map-op-lookup im-ops i im))]*
show (*pre-fun a* (*the* (*map-op-lookup im-ops i im*)),
{q. $\exists q'$. $q' \in$ *partition-index-map* (*im'*, *sm'*, *n'*) $i \wedge (q, a, q') \in \Delta\ \mathcal{A}$})
$\in$ *br* (*set-op-α s-ops*) (*set-op-invar s-ops*)
by (*auto simp add*: *im.correct invar-im invar-Q' partition-index-map-def*)
qed

definition *Hopcroft-impl* **where**
Hopcroft-impl Q F AL pre-fun =
(*if* (*s.isEmpty Q*) *then RETURN* (*partition-impl-empty*) *else* (
if (*s.isEmpty F*) *then RETURN* (*partition-impl-sing Q*) *else* (
do {
(P, -) $\leftarrow$ *WHILET* (λPL. (*snd PL* $\neq$ []))
(*Hopcroft-impl-f pre-fun AL*)
(*partition-impl-init F Q*, *map* (λa. (a, *0*)) *AL*);
RETURN P
})))

lemma *Hopcroft-impl-correct* :
assumes *Q-in-rel*: (Q, $\mathcal{Q}\ \mathcal{A}$) $\in$ *build-rel s.α s.invar*
and *F-in-rel*: (F, $\mathcal{F}\ \mathcal{A}$) $\in$ *build-rel s.α s.invar*
and *AL-OK*: *distinct AL set AL* = $\Sigma\ \mathcal{A}$
and *pre-fun-OK*: $\bigwedge a\ S$. $[\![a \in \Sigma\ \mathcal{A};$ *sp.invar S*$]\!] \Longrightarrow$
(*s.invar* (*pre-fun a S*) $\wedge$ (*s.α* (*pre-fun a S*) =
{q . $\exists q'$. $(q, a, q') \in \Delta\ \mathcal{A} \wedge q' \in$ *sp.α S*}))
shows *Hopcroft-impl Q F AL pre-fun* $\leq \Downarrow$*partition-impl-rel* (*Hopcroft-map* $\mathcal{A}$)
unfolding *Hopcroft-impl-def Hopcroft-map-def*
apply (*refine-rcg*)
— preprocess goals
apply (*insert Q-in-rel*)[]
apply (*simp add*: *s.correct*)
— goal solved
apply (*rule partition-impl-empty-correct*)
— goal solved
apply (*insert F-in-rel*)[]
apply (*simp add*: *s.correct*)
— goal solved
apply (*rule partition-impl-sing-correct[OF Q-in-rel]*)
— goal solved
apply (*subgoal-tac* ((*partition-impl-init F Q*, *map* (λa. (a, *0*)) *AL*), *Hopcroft-map-init* $\mathcal{A}$) $\in$
(*rprod partition-impl-rel* (*build-rel set distinct*)))
apply *assumption*
apply (*simp add*: *Hopcroft-map-init-def AL-OK*)
apply (*intro conjI*)
apply (*rule partition-impl-init-correct* [*OF Q-in-rel F-in-rel*])
apply (*auto*)[]
apply (*simp add*: *distinct-map AL-OK inj-on-def*)
— goal solved
apply (*intro rprod-sv partition-impl-rel-sv br-single-valued*)
— goal solved

apply *auto*[]
— goal solved
apply (*clarify*)
apply (*rule Hopcroft-impl-f-correct* [*OF - AL-OK*])
apply (*simp*)
apply (*simp add*: *pre-fun-OK*)
— goal solved
apply *simp*
done

lemma *set-iteratori-ns-OK* :
assumes *invar-ns*: *n.invar ns*
shows *set-iteratori* ($\lambda c\ f$. *ns-it c f ns*) (*n*.α *ns*)
unfolding *set-iteratori-def*
apply (*intro allI impI*)
apply (*rule ns-it.iteratei-rule*)
apply (*simp-all add*: *invar-ns*)
apply *auto*
done

schematic-lemma *Hopcroft-code-correct-aux* :
shows *RETURN ?code* $\leq$ *Hopcroft-impl Q F AL pre-fun*
unfolding *Hopcroft-impl-def partition-impl-empty-def*
partition-impl-sing-def partition-impl-init-def sm-update-def
Hopcroft-impl-f-def Hopcroft-impl-step-def
Hopcroft-impl-split-count-set-def
partition-impl-split-def
apply (*rule refine-transfer*)+
apply (*rule set-iteratori-ns-OK*)
apply *simp*
apply (*rule refine-transfer*)+
done

definition (**in** −) *Hopcroft-code* **where**
Hopcroft-code
(*s-ops* :: (*'q*, *'q-set*, -) *set-ops-scheme*)
(*sp-ops* :: (*'q*, *'q-set-p*, -) *set-ops-scheme*)
q-set-convert
(*im-ops* :: (*nat*, *'q-set-p*, *'im*, -) *map-ops-scheme*)
(*sm-ops* :: (*'q*, *nat*, *'sm*, -) *map-ops-scheme*)
(*split-it* :: (*'q-set-p*, *'q*, (*'q-set-p* × *nat* × *'q-set-p* × *nat*)) *iteratori*)
(*ns-it* :: (*'n-set*, *nat*, (*'im* × *'sm* × *nat*) × (*'a* × *nat*) *list*) *iteratori*)
s-image
(*sm-it* :: (*'q-set-p*, *'q*, *'sm*) *iteratori*)
Q F AL pre-fun =
(*if set-op-isEmpty s-ops Q then* (*map-op-empty im-ops*, *map-op-empty sm-ops*, *0*)
else if set-op-isEmpty s-ops F then let QP = *q-set-convert Q in* (*map-op-sng im-ops 0 QP*, *sm-it* (λ-. *True*) (λ*q*. *map-op-update sm-ops q 0*) *QP* (*map-op-empty sm-ops*), *1*)
else let (*a*, *b*) = *while* (λ*PL*. *snd PL* $\neq$ [])
(λ(*a*, *b*). *let* (*aa*, *ba*) = *hd b*; *xb* = *pre-fun aa* (*the* (*map-op-lookup im-ops ba* (*fst a*)));
(*ab*, *bb*) = *let xc* = *s-image* (λ*q*. *the* (*map-op-lookup sm-ops q* (*fst* (*snd a*)))) *xb*;
(*ab*, *bb*) = *ns-it* (λ-. *True*)
(λ*x s*. *let* (*ab*, *bb*) = *s*;
(*ac*, *bc*) = *split-it* (λ-. *True*)
(λ*q* (*pt'*, *pct'*, *pf'*, *pcf'*).
if set-op-memb s-ops q xb then (*set-op-ins-dj sp-ops q pt'*, *Suc pct'*, *pf'*, *pcf'*) *else* (*pt'*, *pct'*, *set-op-ins-dj sp-ops q pf'*, *Suc pcf'*))
(*the* (*map-op-lookup im-ops x* (*fst ab*)))
(*set-op-empty sp-ops*, *0*, *set-op-empty sp-ops*, *0*)
in case bc of
(*ad*, *ae*, *be*) ⇒

if be = 0 then (*ab*, *bb*)
else let (*af*, *bf*) = *if be* < *ad then* (*ae*, *ac*) *else* (*ac*, *ae*); *xg* = *map* (λ*a*. (*a*, *snd* (*snd ab*))) *AL* @ *bb*;
xh = (*map-op-update im-ops* (*snd* (*snd ab*)) *af* (*map-op-update im-ops x bf* (*fst ab*)),
sm-it (λ-. *True*) (λ*q*. *map-op-update sm-ops q* (*snd* (*snd ab*))) *af* (*fst* (*snd ab*)), *Suc* (*snd* (*snd ab*)))
in (*xh*, *xg*))
xc (*a*, *tl b*)
in (*ab*, *bb*)
in (*ab*, *bb*))
(*let QF* = *set-op-diff s-ops Q F*
in if set-op-isEmpty s-ops QF then let QP = *q-set-convert F in* (*map-op-sng im-ops 0 QP*, *sm-it* (λ-. *True*) (λ*q*. *map-op-update sm-ops q 0*) *QP* (*map-op-empty sm-ops*), *1*)
else let FP = *q-set-convert F*; *QFP* = *q-set-convert QF*
in (*map-op-update im-ops 0 FP* (*map-op-sng im-ops 1 QFP*),
sm-it (λ-. *True*) (λ*q*. *map-op-update sm-ops q 0*) *FP* (*sm-it* (λ-. *True*) (λ*q*. *map-op-update sm-ops q 1*) *QFP* (*map-op-empty sm-ops*)), *2*),
map (λ*a*. (*a*, *0*)) *AL*)
in a)

lemma *Hopcroft-code-correct* :
shows *RETURN* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*) ≤
Hopcroft-impl Q F AL pre-fun
using *Hopcroft-code-correct-aux*
by (*simp add*: *Hopcroft-code-def*)

lemma *Hopcroft-code-correct-full* :
fixes $\mathcal{A}$ :: (*'q*, *'a*, -) *NFA-rec-scheme*
assumes *wf-A*: *DFA* $\mathcal{A}$
and *Q-in-rel*: (*Q*, $\mathcal{Q}$ $\mathcal{A}$) ∈ *build-rel s.α s.invar*
and *F-in-rel*: (*F*, $\mathcal{F}$ $\mathcal{A}$) ∈ *build-rel s.α s.invar*
and *AL-OK*: *distinct AL set AL* = Σ $\mathcal{A}$
and *pre-fun-OK*: ⋀*a S*. ⟦*a* ∈ Σ $\mathcal{A}$; *sp.invar S*⟧ ⟹
(*s.invar* (*pre-fun a S*) ∧ (*s.α* (*pre-fun a S*) =
{*q* . ∃ *q'*. (*q*, *a*, *q'*) ∈ Δ $\mathcal{A}$ ∧ *q'* ∈ *sp.α S*}))
shows *partition-impl-invar* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)
partition-map-α (*partition-impl-α* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)) = *Myhill-Nerode-partition* $\mathcal{A}$
partition-map-invar (*partition-impl-α* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*))
proof −
note *Hopcroft-code-correct*
also have *Hopcroft-impl Q F AL pre-fun* ≤ ⇓ *partition-impl-rel* (*Hopcroft-map* $\mathcal{A}$)
using *Hopcroft-impl-correct*[*OF Q-in-rel F-in-rel AL-OK pre-fun-OK*] **by** *simp*
also note *DFA.Hopcroft-map-correct-full* [*OF wf-A*]
finally have *RETURN* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*) ≤ ⇓ *partition-impl-rel*
(*SPEC* (λ*P*. *partition-map-α P* = *Myhill-Nerode-partition* $\mathcal{A}$ ∧ *partition-map-invar P*))
by *simp*

thus *partition-impl-invar* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)
partition-map-α (*partition-impl-α* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)) =
Myhill-Nerode-partition $\mathcal{A}$
partition-map-invar (*partition-impl-α* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*))
unfolding *partition-impl-rel-def*
by (*simp-all add*: *pw-le-iff refine-pw-simps*)
qed

definition (**in** −) *Hopcroft-code-rename-map* **where**
Hopcroft-code-rename-map
s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun =
(*fst* (*snd* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)))

lemmas *Hopcroft-code-rename-map-alt-def* = *Hopcroft-code-rename-map-def*[*unfolded Hopcroft-code-def*]

lemma *Hopcroft-code-correct-rename-fun* :
fixes $\mathcal{A}$ :: ('*q*, '*a*, -) *NFA-rec-scheme*
assumes *wf-A*: *DFA* $\mathcal{A}$
and *Q-in-rel*: $(Q, \mathcal{Q}\ \mathcal{A}) \in$ *build-rel s.α s.invar*
and *F-in-rel*: $(F, \mathcal{F}\ \mathcal{A}) \in$ *build-rel s.α s.invar*
and *AL-OK*: *distinct AL set AL* = $\Sigma\ \mathcal{A}$
and *pre-fun-OK*: $\bigwedge a\ S.\ [\![a \in \Sigma\ \mathcal{A};\ sp.invar\ S]\!] \Longrightarrow$
(*s.invar* (*pre-fun a S*) ∧ (*s.α* (*pre-fun a S*) =
$\{q\ .\ \exists\, q'.\ (q,\ a,\ q') \in \Delta\ \mathcal{A} \wedge q' \in sp.\alpha\ S\}$))
shows *sm.invar* (*Hopcroft-code-rename-map s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)
dom (*sm.α* (*Hopcroft-code-rename-map s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)) = $\mathcal{Q}\ \mathcal{A}$
NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ (λ*q*.
states-enumerate (*the* (*sm.lookup q* (*Hopcroft-code-rename-map s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*))))
proof −
from *Hopcroft-code-correct-full*[*of* $\mathcal{A}$ *Q F AL pre-fun*] *assms*
have *invar-impl*: *partition-impl-invar* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)
and *strong-fun*: *partition-map-α* (*partition-impl-α* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)) =
Myhill-Nerode-partition $\mathcal{A}$
and *invar*: *partition-map-invar* (*partition-impl-α* (*Hopcroft-code s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*))
by *simp-all*

from *invar-impl partition-map-to-rename-fun-OK*[*OF invar strong-fun*]
show *sm.invar* (*Hopcroft-code-rename-map s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)
dom (*sm.α* (*Hopcroft-code-rename-map s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*)) = $\mathcal{Q}\ \mathcal{A}$
NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ (λ*q*.
states-enumerate (*the* (*sm.lookup q* (*Hopcroft-code-rename-map s-ops sp-ops q-set-convert im-ops sm-ops split-it ns-it s-image sm-it Q F AL pre-fun*))))
by (*simp-all add*: *Hopcroft-code-rename-map-def dom-def o-def*
partition-impl-α-def partition-impl-invar-def sm.correct)
qed

end

end

5 Presburger Adaptation

theory *Presburger-Adapt*
imports *Main NFA DFA*
implementation/NFASpec
~~/src/HOL/Library/afp/Presburger−Automata/DFS
~~/src/HOL/Library/afp/Presburger−Automata/Presburger-Automata

begin

The translation of Presburger arithmetic to finite automata defined in the AFP Library *Presburger-Automata* consists of building finite automata for Diophantine equations and inequations as well as standard automata constructions. These automata constructions are however defined on a datastructure which is well suited for the specific automata used. Here, let's try to replace these specialised finite automata with general ones.

5.1 DFAs for Diophantine Equations and Inequations

5.1.1 Definition

```
datatype pres-NFA-state =
  pres-NFA-state-error
| pres-NFA-state-int int


fun pres-DFA-eq-ineq-trans-fun where
  pres-DFA-eq-ineq-trans-fun ineq ks pres-NFA-state-error - = pres-NFA-state-error
| pres-DFA-eq-ineq-trans-fun ineq ks (pres-NFA-state-int j) bs =
    (if (ineq ∨ (eval-dioph ks (map nat-of-bool bs)) mod 2 = j mod 2)
        then pres-NFA-state-int ((j − (eval-dioph ks (map nat-of-bool bs))) div 2)
        else pres-NFA-state-error)

fun pres-DFA-is-node where
  pres-DFA-is-node ks l (pres-NFA-state-error) = True
| pres-DFA-is-node ks l (pres-NFA-state-int m) =
   dioph-is-node ks l m

lemma finite-pres-DFA-is-node-set [simp] :
  finite {q. pres-DFA-is-node ks l q}
proof −
  have set-eq: {q. pres-DFA-is-node ks l q} =
   insert pres-NFA-state-error
     (pres-NFA-state-int ‘ {m. dioph-is-node ks l m})
   (is ?s1 = ?s2)
  proof (intro set-eqI)
    fix q
    show (q ∈ ?s1) = (q ∈ ?s2)
    proof (cases q)
      case pres-NFA-state-error thus ?thesis by simp
    next
      case (pres-NFA-state-int m)
      thus ?thesis by auto
    qed
  qed

  show ?thesis
  apply (simp add: set-eq)
  apply (rule finite-imageI)
  apply (insert Presburger-Automata.dioph-dfs.graph-finite)
  apply (simp add: Collect-def)
  done
qed


definition pres-DFA-eq-ineq ::
  bool ⇒ nat ⇒ int list ⇒ int ⇒ (pres-NFA-state, bool list) NFA-rec where
  pres-DFA-eq-ineq ineq n ks l =
    (| Q = {q. pres-DFA-is-node ks l q},
      Σ = {bs . length bs = n},
      Δ = {(q, bs, pres-DFA-eq-ineq-trans-fun ineq ks q bs) | q bs.
            pres-DFA-is-node ks l q ∧ length bs = n},
```

$\mathcal{I} = \{$*pres-NFA-state-int l*$\}$,
$\mathcal{F} = \{$*pres-NFA-state-int m* $|m.$
dioph-is-node ks l m $\wedge$ *0* $\leq$ *m* $\wedge$ (*ineq* $\vee$ *m* = *0*)$\}$ $\|)$

5.1.2 Properties

lemma *pres-DFA-is-node---pres-DFA-eq-ineq-trans-fun* :
assumes *q-OK*: *pres-DFA-is-node ks l q*
and *bs-OK*: *length bs* = *n*
shows *pres-DFA-is-node ks l* (*pres-DFA-eq-ineq-trans-fun i ks q bs*)
proof (*cases pres-DFA-eq-ineq-trans-fun i ks q bs*)
case *pres-NFA-state-error*
thus *?thesis*
unfolding *pres-DFA-eq-ineq-def* **by** *simp*
next
case (*pres-NFA-state-int m′*)
note *q′-eq* = *this*

then obtain *m* **where**
q-eq : *q* = *pres-NFA-state-int m*
and *m′-eq* : *m′* = (*m* − *eval-dioph ks* (*map nat-of-bool bs*)) *div 2*
and *cond*: *i* $\vee$ *eval-dioph ks* (*map nat-of-bool bs*) *mod 2* = *m mod 2*
apply (*cases q*, *simp-all*)
apply (*case-tac i* $\vee$ *eval-dioph ks* (*map nat-of-bool bs*) *mod 2* = *int mod 2*)
apply (*simp-all*)
done

from *q-OK q-eq* **have** *is-node-m*: *dioph-is-node ks l m*
unfolding *pres-DFA-eq-ineq-def*
by *simp*

have *m′* $\in$ *set* (*dioph-ineq-succs n ks m*)
using *bs-OK*
unfolding *dioph-ineq-succs-def List.map-filter-def pres-DFA-eq-ineq-def*
apply (*simp add*: *o-def image-iff m′-eq Bex-def*)
apply (*rule-tac exI* [**where** *x* = *map nat-of-bool bs*])
apply (*simp add*: *nat-of-bool-mk-nat-vecs*)
done
with *dioph-ineq-dfs.succs-is-node* [*OF is-node-m*]
show *?thesis*
unfolding *pres-DFA-eq-ineq-def q′-eq*
by (*simp add*: *list-all-iff*)
qed

lemma *pres-DFA-eq-ineq---is-well-formed* :
DFA (*pres-DFA-eq-ineq i n ks l*)
unfolding *DFA-alt-def NFA-def NFA-is-deterministic-def*
proof (*intro conjI allI impI*)
show $\mathcal{I}$ (*pres-DFA-eq-ineq i n ks l*) $\subseteq$ $\mathcal{Q}$ (*pres-DFA-eq-ineq i n ks l*)
unfolding *pres-DFA-eq-ineq-def*
by (*simp add*: *dioph-is-node-def*)
next
show $\mathcal{F}$ (*pres-DFA-eq-ineq i n ks l*) $\subseteq$ $\mathcal{Q}$ (*pres-DFA-eq-ineq i n ks l*)
unfolding *pres-DFA-eq-ineq-def*
by (*auto simp add*: *subset-iff*)
next
show *finite* (Σ (*pres-DFA-eq-ineq i n ks l*))
unfolding *pres-DFA-eq-ineq-def*
by (*simp*, *rule Presburger-Automata.finite-list*)
next
show *finite* ($\mathcal{Q}$ (*pres-DFA-eq-ineq i n ks l*))
unfolding *pres-DFA-eq-ineq-def*

by *simp*
next
fix q bs q'
assume *in-D*: $(q,\ bs,\ q') \in \Delta$ (*pres-DFA-eq-ineq i n ks l*)

from *in-D* **show** *q-in-Q*: $q \in \mathcal{Q}$ (*pres-DFA-eq-ineq i n ks l*)
unfolding *pres-DFA-eq-ineq-def* **by** *simp*
from *in-D* **show** *bs-in*: $bs \in \Sigma$ (*pres-DFA-eq-ineq i n ks l*)
unfolding *pres-DFA-eq-ineq-def* **by** *simp*

from *in-D*
show $q' \in \mathcal{Q}$ (*pres-DFA-eq-ineq i n ks l*)
using *pres-DFA-is-node---pres-DFA-eq-ineq-trans-fun*
unfolding *pres-DFA-eq-ineq-def*
by *simp*
next
show $\exists$ *q0*. $\mathcal{I}$ (*pres-DFA-eq-ineq i n ks l*) = {*q0*}
unfolding *pres-DFA-eq-ineq-def*
by *simp*
next
show *LTS-is-deterministic* ($\mathcal{Q}$ (*pres-DFA-eq-ineq i n ks l*)) (Σ (*pres-DFA-eq-ineq i n ks l*))
(Δ (*pres-DFA-eq-ineq i n ks l*))
unfolding *pres-DFA-eq-ineq-def LTS-is-deterministic-def LTS-is-weak-deterministic-def*
by *simp*
qed

interpretation *pres-DFA-eq-ineq* :
DFA pres-DFA-eq-ineq i n ks l
by (*rule pres-DFA-eq-ineq---is-well-formed*)

lemma *δ-pres-DFA-eq-ineq* :
δ (*pres-DFA-eq-ineq i n ks l*) (q, bs) =
(*if* (*pres-DFA-is-node ks l q* $\wedge$ *length bs* = n) *then*
Some (*pres-DFA-eq-ineq-trans-fun i ks q bs*)
else None)
using *pres-DFA-eq-ineq.δ-in-Δ-iff* [*symmetric*]
pres-DFA-eq-ineq.DFA-δ-is-none-iff
by (*simp add*: *pres-DFA-eq-ineq-def*)

lemma *pres-DFA-eq-ineq-reach-error* :
list-all (*is-alph n*) *bss* $\Longrightarrow$
DLTS-reach (δ (*pres-DFA-eq-ineq i n ks l*)) *pres-NFA-state-error bss* = *Some* (*pres-NFA-state-error*)
proof (*induct bss*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons bs bss*)
from *Cons*(*2*) **have** *len-bs*: *length bs* = n **by** (*simp add*: *is-alph-def*)
from *Cons*(*2*) **have** *bss-OK*: *list-all* (*is-alph n*) *bss* **by** *simp*
note *ind-hyp* = *Cons*(*1*) [*OF bss-OK*]

from *len-bs*
have *step*: δ (*pres-DFA-eq-ineq i n ks l*) (*pres-NFA-state-error*, *bs*) =
Some pres-NFA-state-error
unfolding *δ-pres-DFA-eq-ineq*
by *simp*

from *step ind-hyp*
show *?case* **by** *simp*
qed

lemma *eval-dioph-nats-of-boolss-eq*:
⟦*length bs* = *n*; *list-all* (*is-alph n*) *bss*⟧ ⟹
eval-dioph ks (*nats-of-boolss n* (*bs* # *bss*)) = *j* ⟷
eval-dioph ks (*map nat-of-bool bs*) *mod 2* = *j mod 2* ∧
eval-dioph ks (*nats-of-boolss n bss*) = (*j* − *eval-dioph ks* (*map nat-of-bool bs*)) *div 2*
apply (*subst eval-dioph-div-mod*)
apply (*simp add*: *nats-of-boolss-mod2 nats-of-boolss-div2*)
done

lemma *pres-DFA-eq-ineq-reach-eq* :
assumes *pres-NFA-state-int j* ∈ *𝒬* (*pres-DFA-eq-ineq False n ks l*)
  and *list-all* (*is-alph n*) *bss*
shows *DLTS-reach* (*δ* (*pres-DFA-eq-ineq False n ks l*)) (*pres-NFA-state-int j*) *bss* = *Some* (*pres-NFA-state-int 0*) ⟷
    eval-dioph ks (*nats-of-boolss n bss*) = *j*
using *assms*
proof (*induct bss arbitrary*: *j*)
  case *Nil*
  thus *?case*
  by (*simp add*: *eval-dioph-replicate-0*)
next
  case (*Cons bs bss′ j*)
  from *Cons*(*3*) **have**
    bs-in: *length bs* = *n* **and**
    bss′-in : *list-all* (*is-alph n*) *bss′*
    by (*simp-all add*: *is-alph-def*)
  note *ind-hyp* = *Cons*(*1*) [*OF* - *bss′-in*]
  note *j-in-Q* = *Cons*(*2*)

  have *δ-j-bs-eq*:
  δ (*pres-DFA-eq-ineq False n ks l*) (*pres-NFA-state-int j*, *bs*) = *Some* (*pres-DFA-eq-ineq-trans-fun False ks* (*pres-NFA-state-int j*) *bs*)
    using *j-in-Q bs-in*
    unfolding *δ-pres-DFA-eq-ineq*
    unfolding *pres-DFA-eq-ineq-def*
    by *simp*

  show *?case*
  proof (*cases eval-dioph ks* (*map nat-of-bool bs*) *mod 2* = *j mod 2*)
    case *False*
    thus *?thesis*
      by (*simp add*: *pres-DFA-eq-ineq-reach-error*
        δ-j-bs-eq eval-dioph-nats-of-boolss-eq bs-in bss′-in)
  next
    case *True* **note** *cond-true* = *True*

    have *j′-in-Q*: *pres-DFA-eq-ineq-trans-fun False ks* (*pres-NFA-state-int j*) *bs* ∈ *𝒬* (*pres-DFA-eq-ineq False n ks l*)
    using *bs-in j-in-Q pres-DFA-is-node---pres-DFA-eq-ineq-trans-fun* [**where** *i* = *False*]
    unfolding *pres-DFA-eq-ineq-def*
    by (*simp add*: *cond-true del*: *pres-DFA-eq-ineq-trans-fun.simps*)

    have *j′-eq*: *pres-DFA-eq-ineq-trans-fun False ks* (*pres-NFA-state-int j*) *bs* =
      pres-NFA-state-int ((*j* − *eval-dioph ks* (*map nat-of-bool bs*)) *div 2*)
      by (*simp add*: *cond-true*)

    from *ind-hyp* [*OF j′-in-Q* [*unfolded j′-eq*]]
    show *?thesis*
      by (*simp add*: *pres-DFA-eq-ineq-reach-error*
        δ-j-bs-eq eval-dioph-nats-of-boolss-eq bs-in bss′-in)
  qed

qed

lemma *pres-DFA-eq-correct* :
assumes *bss-OK*: *list-all* (*is-alph n*) *bss*
shows *NFA-accept* (*pres-DFA-eq-ineq False n ks l*) *bss* =
  (*eval-dioph ks* (*nats-of-boolss n bss*) = *l*)
proof −
  have i-*eval*: i (*pres-DFA-eq-ineq False n ks l*) = *pres-NFA-state-int l*
    unfolding *pres-DFA-eq-ineq-def*
    by (*simp add*: i-*def*)

  have $\mathcal{F}$-*eval*: $\mathcal{F}$ (*pres-DFA-eq-ineq False n ks l*) = {*pres-NFA-state-int 0*}
    unfolding *pres-DFA-eq-ineq-def*
    by (*auto simp add*: *dioph-is-node-def*)

  from *pres-DFA-eq-ineq*.i-*is-state* [*of False n ks l*]
  have *l-in-Q*: *pres-NFA-state-int l* ∈ $\mathcal{Q}$ (*pres-DFA-eq-ineq False n ks l*)
    by (*simp add*: i-*eval*)
  note *dioph-OK* = *pres-DFA-eq-ineq-reach-eq* [*OF l-in-Q bss-OK*, *symmetric*]

  have *DLTS-reach* (δ (*pres-DFA-eq-ineq False n ks l*)) (*pres-NFA-state-int l*) *bss* ≠ *None*
    using *bss-OK*
    apply (*simp add*: *pres-DFA-eq-ineq.DFA-reach-is-none-iff*)
    apply (*simp add*: *pres-DFA-eq-ineq-def is-alph-def in-lists-conv-set*
      dioph-is-node-def Ball-set-list-all)
  done
  thus *?thesis*
    by (*auto simp add*: *pres-DFA-eq-ineq.DFA-accept-alt-def dioph-OK* $\mathcal{F}$-*eval* i-*eval*)
qed

lemma *pres-DFA-eq-ineq-reach-ineq* :
assumes *pres-NFA-state-int j* ∈ $\mathcal{Q}$ (*pres-DFA-eq-ineq True n ks l*)
    and *list-all* (*is-alph n*) *bss*
    and *DLTS-reach* (δ (*pres-DFA-eq-ineq True n ks l*)) (*pres-NFA-state-int j*) *bss* = *Some* (*pres-NFA-state-int j'*)
shows *0* ≤ *j'* ⟷ *eval-dioph ks* (*nats-of-boolss n bss*) ≤ *j*
using *assms*
proof (*induct bss arbitrary*: *j j'*)
  case *Nil*
  thus *?case*
  by (*simp add*: *eval-dioph-replicate-0*)
next
  case (*Cons bs bss' j*)
  from *Cons*(*3*) **have**
    bs-in: *length bs* = *n* **and**
    bss'-in : *list-all* (*is-alph n*) *bss'*
    by (*simp-all add*: *is-alph-def*)
  note *ind-hyp* = *Cons*(*1*) [*OF - bss'-in*]
  note *j-in-Q* = *Cons*(*2*)
  note *reach-j-j'* = *Cons*(*4*)

  let *?j'* = (*j* − *eval-dioph ks* (*map nat-of-bool bs*)) *div 2*
  have δ-*j-bs-eq*:
   δ (*pres-DFA-eq-ineq True n ks l*) (*pres-NFA-state-int j*, *bs*) =
    Some (*pres-NFA-state-int ?j'*)
    using *j-in-Q bs-in*
    unfolding δ-*pres-DFA-eq-ineq*
    unfolding *pres-DFA-eq-ineq-def*
    by *simp*

  have *j'-in-Q*: *pres-DFA-eq-ineq-trans-fun True ks* (*pres-NFA-state-int j*) *bs* ∈ $\mathcal{Q}$ (*pres-DFA-eq-ineq True n ks l*)
    using *bs-in j-in-Q pres-DFA-is-node---pres-DFA-eq-ineq-trans-fun* [**where** *i* = *True*]

unfolding *pres-DFA-eq-ineq-def*
by (*simp del*: *pres-DFA-eq-ineq-trans-fun.simps*)

have *j′-eq*: *pres-DFA-eq-ineq-trans-fun True ks* (*pres-NFA-state-int j*) *bs* =
pres-NFA-state-int ?j′
by *simp*

with *ind-hyp* [*OF j′-in-Q* [*unfolded j′-eq*]] *reach-j-j′*
show *?case*
by (*simp add*: *δ-j-bs-eq*
eval-dioph-ineq-div-mod [**where** *xs* = *nats-of-boolss n* (*bs#bss′*)]
nats-of-boolss-div2 bs-in bss′-in nats-of-boolss-mod2)
qed

lemma *pres-DFA-ineq-reach-exists* :
⟦*list-all* (*is-alph n*) *bss*; *dioph-is-node ks l j*⟧ ⟹
∃ *j′*. *DLTS-reach* (*δ* (*pres-DFA-eq-ineq True n ks l*)) (*pres-NFA-state-int j*) *bss* =
Some (*pres-NFA-state-int j′*) ∧ *dioph-is-node ks l j′*
proof (*induct bss arbitrary*: *j*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons bs bss′*)

let *?j′* = (*j* − *eval-dioph ks* (*map nat-of-bool bs*)) *div 2*
have *pres-DFA-eq-ineq-trans-fun True ks* (*pres-NFA-state-int j*) *bs* ∈ *𝒬* (*pres-DFA-eq-ineq False n ks l*)
using *Cons pres-DFA-is-node---pres-DFA-eq-ineq-trans-fun* [**where** *i* = *True*]
unfolding *pres-DFA-eq-ineq-def*
by (*simp del*: *pres-DFA-eq-ineq-trans-fun.simps*)
hence *j′-is-node*: *dioph-is-node ks l ?j′*
unfolding *pres-DFA-eq-ineq-def*
by *simp*

from *Cons j′-is-node*
show *?case*
by (*simp add*: *is-alph-def δ-pres-DFA-eq-ineq*)
qed

lemma *pres-DFA-ineq-correct* :
assumes *bss-OK*: *list-all* (*is-alph n*) *bss*
shows *NFA-accept* (*pres-DFA-eq-ineq True n ks l*) *bss* =
(*eval-dioph ks* (*nats-of-boolss n bss*) ≤ *l*)
proof −
have i-*eval*: i (*pres-DFA-eq-ineq True n ks l*) = *pres-NFA-state-int l*
unfolding *pres-DFA-eq-ineq-def*
by (*simp add*: i-*def*)

have *ℱ-eval*: *ℱ* (*pres-DFA-eq-ineq True n ks l*) = {*pres-NFA-state-int m* |*m*. *dioph-is-node ks l m* ∧ *0* ≤ *m*}
unfolding *pres-DFA-eq-ineq-def*
by *simp*

obtain *j′* **where** *reach-eq*:
DLTS-reach (*δ* (*pres-DFA-eq-ineq True n ks l*)) (*pres-NFA-state-int l*) *bss* =
Some (*pres-NFA-state-int j′*)
and *is-node-j′*: *dioph-is-node ks l j′*
using *pres-DFA-ineq-reach-exists* [*OF bss-OK*, *of ks l l*]
by (*simp add*: *dioph-is-node-def*, *blast*)

have *l-in-Q*: *pres-NFA-state-int l* ∈ *𝒬* (*pres-DFA-eq-ineq True n ks l*)
unfolding *pres-DFA-eq-ineq-def*
by (*simp add*: *dioph-is-node-def*)

from *pres-DFA-eq-ineq-reach-ineq* [*OF l-in-Q bss-OK reach-eq*, *symmetric*]

show *?thesis*
  by (*simp add*: *pres-DFA-eq-ineq.DFA-accept-alt-def* i-*eval* $\mathcal{F}$-*eval*
      reach-eq is-node-j')
qed

5.1.3 Efficiency

5.1.4 Implementation

For using these automata constructions let's replace the new datatype for states with natural numbers and consider only reachable states.

fun *pres-NFA-state-to-nat* **where**
  pres-NFA-state-to-nat pres-NFA-state-error = *0*
| *pres-NFA-state-to-nat* (*pres-NFA-state-int m*) =
  int-encode m + *1*

lemma *pres-NFA-state-nat-eq-0* [*simp*] :
  (*pres-NFA-state-to-nat q* = *0*) $\longleftrightarrow$ *q* = *pres-NFA-state-error*
by (*cases q*, *auto*)

lemma *pres-NFA-state-nat-neq-0* [*simp*] :
  (*pres-NFA-state-to-nat q* = *Suc m*) $\longleftrightarrow$ *q* = *pres-NFA-state-int* (*int-decode m*)
by (*cases q*, *auto*)

lemma *inj-pres-NFA-state-to-nat*:
  inj-on pres-NFA-state-to-nat S
unfolding *inj-on-def Ball-def*
proof (*intro allI impI*)
  fix *q1 q2*
  assume *f-q1*: *pres-NFA-state-to-nat q1* = *pres-NFA-state-to-nat q2*
  thus *q1* = *q2*
    by (*cases pres-NFA-state-to-nat q2*, *simp-all*)
qed

definition *efficient-pres-DFA-eq-ineq* **where**
  efficient-pres-DFA-eq-ineq i n ks l =
   NFA-rename-states (*NFA-remove-unreachable-states* (*pres-DFA-eq-ineq i n ks l*)) *pres-NFA-state-to-nat*

lemma *efficient-pres-DFA-eq-ineq---is-well-formed* :
  DFA (*efficient-pres-DFA-eq-ineq i n ks l*)
unfolding *efficient-pres-DFA-eq-ineq-def*
apply (*intro DFA---inj-rename DFA---NFA-remove-unreachable-states*)
apply (*simp-all add*: *pres-DFA-eq-ineq---is-well-formed inj-pres-NFA-state-to-nat*)
done

lemma *pres-DFA-eq-ineq---isomorphic-wf* :
  NFA-isomorphic-wf (*NFA-remove-unreachable-states* (*pres-DFA-eq-ineq i n ks l*))
      (*efficient-pres-DFA-eq-ineq i n ks l*)
unfolding *efficient-pres-DFA-eq-ineq-def*
by (*intro NFA-isomorphic-wf---NFA-rename-states inj-pres-NFA-state-to-nat*
    NFA-remove-unreachable-states---is-well-formed pres-DFA-eq-ineq.wf-NFA)

lemma *efficient-pres-DFA-eq-ineq---NFA-accept* [*simp*] :
  NFA-accept (*efficient-pres-DFA-eq-ineq i n ks l*) *bss* =
   NFA-accept (*pres-DFA-eq-ineq i n ks l*) *bss*
proof −
  have *equiv-f*: $\bigwedge\mathcal{A}$. *NFA-is-equivalence-rename-fun* $\mathcal{A}$ *pres-NFA-state-to-nat*
    by (*rule NFA-is-equivalence-rename-funI---inj-used*, *fact inj-pres-NFA-state-to-nat*)

  have *wf-1* : *DFA* (*NFA-remove-unreachable-states* (*pres-DFA-eq-ineq i n ks l*))
    by (*intro DFA---NFA-remove-unreachable-states*, *fact pres-DFA-eq-ineq---is-well-formed*)

hence *wf-2*: *NFA* (*NFA-remove-unreachable-states* (*pres-DFA-eq-ineq i n ks l*))
  by (*simp add*: *DFA-alt-def*)

note *NFA.NFA-rename-states---accept* [*OF wf-2 equiv-f*]
thus *?thesis*
  unfolding *efficient-pres-DFA-eq-ineq-def*
  by *simp*
qed

5.2 Existential Quantification

type-synonym *pres-NFA* = (*nat*, *bool list*) *NFA-rec*

definition *pres-DFA-labels-tl* ::
  pres-NFA ⇒ *pres-NFA* **where**
  pres-DFA-labels-tl $\mathcal{A}$ = *NFA-rename-labels* $\mathcal{A}$ *tl*

lemma *pres-DFA-labels-tl---well-formed*:
NFA $\mathcal{A}$ ⟹ *NFA* (*pres-DFA-labels-tl* $\mathcal{A}$)
  by (*simp add*: *pres-DFA-labels-tl-def*
    NFA.NFA-rename-labels---is-well-formed)

lemma *pres-DFA-labels-tl---NFA-accept* :
assumes *wf-A*: *NFA* $\mathcal{A}$
  and Σ-*A*: ⋀*bs*. *bs* ∈ Σ $\mathcal{A}$ ⟹ *bs* ≠ []
shows *NFA-accept* (*pres-DFA-labels-tl* $\mathcal{A}$) *bss* ⟷
  (∃ *bs*. *length bs* = *length bss* ∧ *NFA-accept* $\mathcal{A}$ (*insertll 0 bs bss*))
apply (*simp add*: *pres-DFA-labels-tl-def NFA.NFA-accept---NFA-rename-labels-iff* [*OF wf-A*])
proof
  assume ∃ *bss'*. *bss* = *map tl bss'* ∧ *NFA-accept* $\mathcal{A}$ *bss'*
  then obtain *bss'* **where**
    bss-eq: *bss* = *map tl bss'*
   and *bss'-acc*: *NFA-accept* $\mathcal{A}$ *bss'* **by** *blast*

  have *len-bs* : *length* (*map hd bss'*) = *length bss*
    unfolding *bss-eq* **by** *simp*

  from *bss'-acc* **have** *bss'* ∈ *lists* (Σ $\mathcal{A}$)
    by (*simp add*: *NFA-accept-def*)
  hence *insertll 0* (*map hd bss'*) (*map tl bss'*) = *bss'*
  proof (*induct bss'*)
    case *Nil* **thus** *?case* **by** *simp*
  next
    case (*Cons bs bss*)
    from *Cons*(*1*) **have** *bs* ≠ [] **using** Σ-*A* **by** *simp*
    then obtain *bs-hd bs-tl* **where** *bs-eq*: *bs* = *bs-hd* # *bs-tl*
      by (*cases bs*, *simp*)

    with *Cons*(*3*) **show** *?case*
      by (*simp add*: *insertl-0-eq*)
  qed
  hence *NFA-accept* $\mathcal{A}$ (*insertll 0* (*map hd bss'*) *bss*)
    unfolding *bss-eq*
    using *bss'-acc*
    by *simp*
  with *len-bs* **show** ∃ *bs*. *length bs* = *length bss* ∧ *NFA-accept* $\mathcal{A}$ (*insertll 0 bs bss*) **by** *blast*
next
  assume ∃ *bs*. *length bs* = *length bss* ∧ *NFA-accept* $\mathcal{A}$ (*insertll 0 bs bss*)
  then obtain *bs* **where**
    bss-acc: *NFA-accept* $\mathcal{A}$ (*insertll 0 bs bss*)
   and *len-bs*: *length bs* = *length bss* **by** *blast*

from *len-bs*
have *map tl* (*insertll 0 bs bss*) = *bss*
proof (*induct bss arbitrary*: *bs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons bss-hd bss-tl bs*)
thus *?case*
by (*cases bs*, *simp-all add*: *insertl-0-eq*)
qed
with *bss-acc* **show** $\exists$ *bss*$'$. *bss* = *map tl bss*$'$ $\wedge$ *NFA-accept* $\mathcal{A}$ *bss*$'$
apply (*rule-tac exI* [**where** *x* = *insertll 0 bs bss*])
apply *simp*
done
qed

definition *pres-DFA-exists* ::
nat $\Rightarrow$ *pres-NFA* $\Rightarrow$ *pres-NFA* **where**
pres-DFA-exists n $\mathcal{A}$ = *NFA-right-quotient-lists* (*pres-DFA-labels-tl* $\mathcal{A}$)
{*replicate n False*}

lemma *pres-DFA-exists---well-formed*:
NFA $\mathcal{A}$ $\Longrightarrow$ *NFA* (*pres-DFA-exists n* $\mathcal{A}$)
unfolding *pres-DFA-exists-def*
by (*intro NFA-right-quotient---is-well-formed pres-DFA-labels-tl---well-formed*, *assumption*)

lemma *pres-DFA-exists---NFA-accept* :
assumes *wf-A*: *NFA* $\mathcal{A}$
and Σ-*A*: $\bigwedge$*bs*. *bs* $\in \Sigma\ \mathcal{A} \Longrightarrow$ *bs* $\neq$ []
shows *NFA-accept* (*pres-DFA-exists n* $\mathcal{A}$) *bss* $\longleftrightarrow$
($\exists$ *bs m*. *length bs* = *length bss* + *m* $\wedge$ *NFA-accept* $\mathcal{A}$ (*insertll 0 bs* (*bss* @ *zeros m n*)))
proof −
have *lists-repl*: *lists* {*replicate n False*} = {*zeros m n* | *m*. *True*} (**is** *?s1* = *?s2*)
proof (*intro set-eqI iffI*)
fix *bss*
assume *bss* $\in$ *?s2*
thus *bss* $\in$ *?s1* **by** (*auto simp add*: *zeros-def list-all-iff*)
next
fix *bss*
assume *bss* $\in$ *?s1*
hence $\bigwedge$*bs*. *bs* $\in$ *set bss* $\Longrightarrow$ *bs* = *replicate n False* **by** (*auto simp add*: *list-all-iff*)
hence *bss* = *replicate* (*length bss*) (*replicate n False*)
by (*induct bss*, *auto*)
thus *bss* $\in$ *?s2* **by** (*simp add*: *zeros-def*, *blast*)
qed

note *wf-ex* = *pres-DFA-labels-tl---well-formed* [*OF wf-A*]
note *NFA-accept* = *NFA.NFA-right-quotient---accepts* [*OF wf-ex*, **where** *L* = *lists* {*replicate n False*}]
with *lists-repl* **show** *?thesis*
by (*simp del*: *ex-simps*
add: *pres-DFA-exists-def pres-DFA-labels-tl---NFA-accept* [*OF wf-A* Σ-*A*]
ex-simps[*symmetric*] *zeros-len*)
qed

lemma *nats-of-boolss-append-zeros* :
assumes *bss-in*: *list-all* (*is-alph n*) *bss*
shows *nats-of-boolss n* (*bss* @ *zeros m n*) = *nats-of-boolss n bss*
proof −
have *map-eq*: $\bigwedge$*nl*::*nat list*. *map* (λ(*x*, *y*). *x* + (*2*::*nat*) ^ *length bss* $*$ *y*) (*zip nl* (*replicate* (*length nl*) *0*)) = *nl*
proof −
fix *nl*
show *map* (λ(*x*::*nat*, *y*). *x* + *2* ^ *length bss* $*$ *y*) (*zip nl* (*replicate* (*length nl*) *0*)) = *nl*
proof (*induct nl*)

```
      case Nil thus ?case by simp
    next
      case (Cons x nl)
      thus ?case by (simp)
    qed
  qed

  from map-eq [of nats-of-boolss n bss]
       nats-of-boolss-append [of n bss zeros m n]
  show nats-of-boolss n (bss @ zeros m n) = nats-of-boolss n bss
    by (simp add: bss-in zeros-is-alpha nats-of-boolss-zeros nats-of-boolss-length)
qed


lemma pres-DFA-exists---NFA-accept---nats :
assumes wf-A: NFA 𝒜
    and Σ-A: Σ 𝒜 = {bs. length bs = Suc n}
    and acc-A: ⋀bss. list-all (is-alph (Suc n)) bss ⟹ NFA-accept 𝒜 bss ⟷ P (nats-of-boolss (Suc n) bss)
    and bss-in: list-all (is-alph n) bss
shows NFA-accept (pres-DFA-exists n 𝒜) bss =
       (∃ x. P (x # nats-of-boolss n bss))
proof −
  have Σ-A′: ⋀bs. bs ∈ Σ 𝒜 ⟹ bs ≠ []
  proof −
    fix bs
    assume bs ∈ Σ 𝒜
    with Σ-A have length bs = Suc n by simp
    thus bs ≠ [] by auto
  qed

  have ⋀bs m. length bs = length bss + m ⟹
              NFA-accept 𝒜 (insertll 0 bs (bss @ zeros m n)) =
              P (nats-of-boolss (Suc n) (insertll 0 bs (bss @ zeros m n)))
  proof −
    fix bs :: bool list
    fix m
    assume len-bs: length bs = length bss + m
    with insertll-len2 [of n bss @ (zeros m n) bs 0]
         acc-A
    show NFA-accept 𝒜 (insertll 0 bs (bss @ zeros m n)) =
         P (nats-of-boolss (Suc n) (insertll 0 bs (bss @ zeros m n)))
      by (simp add: bss-in zeros-is-alpha zeros-len)
  qed
  hence step0:
    (∃ bs m. length bs = length bss + m ∧ NFA-accept 𝒜 (insertll 0 bs (bss @ zeros m n))) =
     (∃ bs m. length bs = length bss + m ∧ P (nats-of-boolss (Suc n) (insertll 0 bs (bss @ zeros m n))))
    by auto


  have ⋀bs m. length bs = length bss + m ⟹
       nats-of-boolss (Suc n) (insertll 0 bs (bss @ zeros m n)) =
       nat-of-bools bs # nats-of-boolss n bss
  proof −
   fix bs :: bool list
   fix m
   assume length bs = length bss + m
   hence len-bs: length bs = length (bss @ zeros m n)
     by (simp add: zeros-len)

   have bss′-in: list-all (is-alph n) (bss @ zeros m n)
     by (simp add: bss-in zeros-is-alpha)
```

```
   from nats-of-boolss-insertll [of n bss @ (zeros m n) bs 0, OF bss'-in len-bs]
        nats-of-boolss-append-zeros [of n bss m, OF bss-in]
   show nats-of-boolss (Suc n) (insertll 0 bs (bss @ zeros m n)) =
         nat-of-bools bs # nats-of-boolss n bss
     by (simp add: insertl-0-eq)
  qed
  hence step1:
     (∃ bs m. length bs = length bss + m ∧ P (nats-of-boolss (Suc n) (insertll 0 bs (bss @ zeros m n)))) =
      (∃ bs m. length bs = length bss + m ∧ P (nat-of-bools bs # nats-of-boolss n bss))
    by (metis (no-types))

  have step2: (∃ bs m. length bs = length bss + m ∧ P (nat-of-bools bs # nats-of-boolss n bss)) =
               (∃ x. P (x # nats-of-boolss n bss))
    apply (rule iffI)
    apply (erule exE conjE)+
    apply (erule exI)
    apply (erule exE)
    apply (rule-tac x=bools-of-nat (length bss) x in exI)
    apply (rule-tac x=length (bools-of-nat (length bss) x) − length bss in exI)
    apply (simp add: bools-of-nat-inverse bools-of-nat-length)
  done

  from step0 step1 step2 show ?thesis
    by (simp add: pres-DFA-exists---NFA-accept [OF wf-A Σ-A']
                  nats-of-boolss-insertll)
qed

definition pres-DFA-exists-min where
  pres-DFA-exists-min n 𝒜 = NFA-right-quotient-lists (
     NFA-minimise (pres-DFA-labels-tl 𝒜)) {replicate n False}

find-theorems name: NFA-right-quotient name: well-formed
lemma pres-DFA-exists-min---well-formed :
assumes wf-A: NFA 𝒜
shows DFA (pres-DFA-exists-min n 𝒜)
unfolding pres-DFA-exists-min-def pres-DFA-labels-tl-def
by (metis NFA-right-quotient---is-well-formed-DFA NFA.NFA-rename-labels---is-well-formed
         NFA-minimise-spec(3) assms DFA-is-minimal-gen-def)

lemma pres-DFA-exists-min---well-formed-DFA :
DFA 𝒜 ⟹ DFA (pres-DFA-exists-min n 𝒜)
using pres-DFA-exists-min---well-formed
by (metis DFA-alt-def)

lemma pres-DFA-exists-min---NFA-accept :
assumes wf-A: DFA 𝒜
shows NFA-accept (pres-DFA-exists-min n 𝒜) bss ⟷
       NFA-accept (pres-DFA-exists n 𝒜) bss
proof −
  let ?A1 = pres-DFA-labels-tl 𝒜
  let ?A2 = NFA-minimise ?A1

  from wf-A have NFA-A: NFA 𝒜 by (simp add: DFA-alt-def)
  have NFA-A1: NFA ?A1
    unfolding pres-DFA-labels-tl-def
    by (simp add: NFA.NFA-rename-labels---is-well-formed [OF NFA-A])

  from NFA-minimise-spec(3)[OF NFA-A1]
  have NFA-A2: NFA ?A2 unfolding DFA-is-minimal-def DFA-alt-def by simp

  from NFA-minimise-spec(1)[OF NFA-A1]
  have NFA-accept-A2: ⋀bss. NFA-accept ?A2 bss = NFA-accept ?A1 bss
```

by (*auto simp add*: *$\mathcal{L}$-def*)

show *?thesis*
unfolding *pres-DFA-exists-min-def pres-DFA-exists-def*
by (*simp add*: *NFA.NFA-right-quotient---accepts NFA-A1 NFA-A2 NFA-accept-A2*)
qed

lemma *pres-DFA-exists-min---NFA-accept---nats* :
assumes *wf-A*: *DFA* $\mathcal{A}$
and Σ-*A*: $\Sigma\ \mathcal{A} = \{bs.\ length\ bs = Suc\ n\}$
and *acc-A*: $\bigwedge bss.\ list\text{-}all\ (is\text{-}alph\ (Suc\ n))\ bss \Longrightarrow NFA\text{-}accept\ \mathcal{A}\ bss \longleftrightarrow P\ (nats\text{-}of\text{-}boolss\ (Suc\ n)\ bss)$
and *bss-in*: *list-all* (*is-alph n*) *bss*
shows *NFA-accept* (*pres-DFA-exists-min n* $\mathcal{A}$) *bss* =
($\exists\, x.\ P\ (x\ \#\ nats\text{-}of\text{-}boolss\ n\ bss)$)
proof −
have *wf-A'* : *NFA* $\mathcal{A}$
using *wf-A* **unfolding** *DFA-alt-def* **by** *simp*
show *?thesis* **by** (*metis pres-DFA-exists---NFA-accept---nats* [*OF wf-A'* Σ-*A acc-A bss-in*]
pres-DFA-exists-min---NFA-accept [*OF wf-A*])
qed

lemma Σ-*pres-DFA-exists-min* :
NFA $\mathcal{A} \Longrightarrow \Sigma$ (*pres-DFA-exists-min n* $\mathcal{A}$) = *tl* ‘ $\Sigma\ \mathcal{A}$
by (*simp add*: *pres-DFA-exists-min-def pres-DFA-labels-tl-def NFA-minimise-spec*(*2*)
NFA.NFA-rename-labels---is-well-formed)

5.3 Universal Quantification

definition *pres-DFA-forall-min* **where**
pres-DFA-forall-min n $\mathcal{A}$ = *DFA-complement* (*pres-DFA-exists-min n* (*DFA-complement* $\mathcal{A}$))

lemma *pres-DFA-forall-min---well-formed-DFA* :
DFA $\mathcal{A} \Longrightarrow$ *DFA* (*pres-DFA-forall-min n* $\mathcal{A}$)
unfolding *pres-DFA-forall-min-def*
by (*intro DFA-complement-of-DFA-is-DFA pres-DFA-exists-min---well-formed-DFA*,
assumption)

lemma Σ-*image-tl* : *tl* ‘ $\{bs{::}'a\ list.\ length\ bs = Suc\ n\} = \{bs.\ length\ bs = n\}$
proof (*intro set-eqI iffI*)
fix *bs* :: *'a list*
assume $bs \in tl$ ‘ $\{bs.\ length\ bs = Suc\ n\}$
thus $bs \in \{bs.\ length\ bs = n\}$ **by** *auto*
next
fix *bs* :: *'a list*
assume $bs \in \{bs.\ length\ bs = n\}$
thus $bs \in tl$ ‘ $\{bs.\ length\ bs = Suc\ n\}$
apply (*simp add*: *image-iff*)
apply (*rule exI* [**where** $x = e\ \#\ bs$])
apply *simp*
done
qed

lemma *pres-DFA-forall-min---NFA-accept---nats* :
fixes $\mathcal{A}$ :: *pres-NFA*
assumes *wf-A*: *DFA* $\mathcal{A}$
and Σ-*A*: $\Sigma\ \mathcal{A} = \{bs.\ length\ bs = Suc\ n\}$
and *acc-A*: $\bigwedge bss.\ list\text{-}all\ (is\text{-}alph\ (Suc\ n))\ bss \Longrightarrow NFA\text{-}accept\ \mathcal{A}\ bss \longleftrightarrow P\ (nats\text{-}of\text{-}boolss\ (Suc\ n)\ bss)$
and *bss-in*: *list-all* (*is-alph n*) *bss*
shows *NFA-accept* (*pres-DFA-forall-min n* $\mathcal{A}$) *bss* =
($\forall\, x.\ P\ (x\ \#\ nats\text{-}of\text{-}boolss\ n\ bss)$)
proof −

```
  let ?cA = DFA-complement 𝒜
  note wf-ca = DFA-complement-of-DFA-is-DFA [OF wf-A]
  hence wf'-ca: NFA ?cA unfolding DFA-alt-def by simp

  have acc-cA: ⋀bss. list-all (is-alph (Suc n)) bss ⟹ NFA-accept ?cA bss ⟷ ¬(P (nats-of-boolss (Suc n) bss))
    using DFA-complement-word [OF wf-A] Σ-A
    by (simp add: list-all-iff in-lists-conv-set acc-A is-alph-def)

  have acc : NFA-accept (pres-DFA-forall-min n 𝒜) bss ⟷
           ¬ (NFA-accept (pres-DFA-exists-min n ?cA) bss)
    unfolding pres-DFA-forall-min-def
    apply (rule DFA-complement-word [OF pres-DFA-exists-min---well-formed-DFA [OF wf-ca]])
    apply (insert bss-in)
    apply (simp add: Σ-pres-DFA-exists-min[OF wf'-ca] Σ-A Σ-image-tl)
    apply (simp add: list-all-iff is-alph-def in-lists-conv-set)
  done

  with pres-DFA-exists-min---NFA-accept---nats [OF wf-ca - acc-cA bss-in] Σ-A
  show ?thesis by simp
qed

lemma Σ-pres-DFA-forall-min :
  NFA 𝒜 ⟹ Σ (pres-DFA-forall-min n 𝒜) = tl ‘ Σ 𝒜
by (simp add: pres-DFA-forall-min-def Σ-pres-DFA-exists-min DFA-complement---is-well-formed)
```

5.4 Translation

```
fun DFA-of-pf :: nat ⇒ pf ⇒ pres-NFA where
  Eq:     DFA-of-pf n (Eq ks l) = efficient-pres-DFA-eq-ineq False n ks l
| Le:     DFA-of-pf n (Le ks l) = efficient-pres-DFA-eq-ineq True n ks l
| And:    DFA-of-pf n (And p q) = NFA-bool-comb op∧ (DFA-of-pf n p) (DFA-of-pf n q)
| Or:     DFA-of-pf n (Or p q) = NFA-bool-comb op∨ (DFA-of-pf n p) (DFA-of-pf n q)
| Imp:    DFA-of-pf n (Imp p q) = NFA-bool-comb op⟶ (DFA-of-pf n p) (DFA-of-pf n q)
| Exists: DFA-of-pf n (Exist p) = pres-DFA-exists-min n (DFA-of-pf (Suc n) p)
| Forall: DFA-of-pf n (Forall p) = pres-DFA-forall-min n (DFA-of-pf (Suc n) p)
| Neg:    DFA-of-pf n (Neg p) = DFA-complement (DFA-of-pf n p)

lemmas DFA-of-pf-induct =
  DFA-of-pf.induct [case-names Eq Le And Or Imp Exist Forall Neg]

lemma DFA-of-pf---correct:
  DFA (DFA-of-pf n p) ∧
  Σ (DFA-of-pf n p) = {bs. length bs = n} ∧
  (∀ bss. list-all (is-alph n) bss ⟶
    NFA.NFA-accept (DFA-of-pf n p) bss = eval-pf p (nats-of-boolss n bss))
  (is ?P1 n p ∧ ?P2 n p ∧ ?P3 n p)
proof (induct n p rule: DFA-of-pf-induct)
  case (Eq n ks l)
  show ?case
    apply (simp add: efficient-pres-DFA-eq-ineq---is-well-formed pres-DFA-eq-correct)
    apply (simp add: efficient-pres-DFA-eq-ineq-def pres-DFA-eq-ineq-def)
  done
next
  case (Le n ks l)
  show ?case
    apply (simp add: efficient-pres-DFA-eq-ineq---is-well-formed pres-DFA-ineq-correct)
    apply (simp add: efficient-pres-DFA-eq-ineq-def pres-DFA-eq-ineq-def)
  done
next
  case (And n p q)
  thus ?case
    by (simp add: NFA-bool-comb-DFA---NFA-accept NFA-bool-comb-DFA---is-well-formed
```

in-lists-conv-set list-all-iff is-alph-def)
next
case (*Or n p q*)
thus *?case*
by (*simp add*: *NFA-bool-comb-DFA---NFA-accept NFA-bool-comb-DFA---is-well-formed*
in-lists-conv-set list-all-iff is-alph-def)
next
case (*Imp n p q*)
thus *?case*
by (*simp add*: *NFA-bool-comb-DFA---NFA-accept NFA-bool-comb-DFA---is-well-formed*
in-lists-conv-set list-all-iff is-alph-def)
next
case (*Neg n p*)
thus *?case*
by (*simp add*: *DFA-complement-of-DFA-is-DFA DFA-complement-word*
in-lists-conv-set list-all-iff is-alph-def)
next
case (*Exist n p*)
with *pres-DFA-exists-min---NFA-accept---nats* [**where** $\mathcal{A}$ = *DFA-of-pf* (*Suc n*) *p* **and** *n* = *n* **and** *P* = *eval-pf p*]
show *?case*
by (*simp add*: *Σ-pres-DFA-exists-min Σ-image-tl pres-DFA-exists-min---well-formed-DFA*)
next
case (*Forall n p*)
with *pres-DFA-forall-min---NFA-accept---nats* [**where** $\mathcal{A}$ = *DFA-of-pf* (*Suc n*) *p* **and** *n* = *n* **and** *P* = *eval-pf p*]
show *?case*
by (*simp add*: *Σ-pres-DFA-forall-min Σ-image-tl pres-DFA-forall-min---well-formed-DFA*)
qed

5.5 Code Generation

The automata used for presburger arithmetic have label sets that consist of all bitvectors of a certain length. The following locale is used to cache these sets.

locale *presburger-label-set-cache* = *set a-α a-invar*
for *a-α* :: *'al-set* ⇒ (*'a list*) *set* **and** *a-invar* +
fixes *c-α* :: *'cache* ⇒ *nat* ⇒ *'cache* × *'al-set*
fixes *c-invar* :: *'cache* ⇒ *bool*
fixes *init-cache* :: *'cache*
assumes *init-cache-OK* :
c-invar init-cache
assumes *cache-correct* :
c-invar c ⟹ *c-invar* (*fst* (*c-α c n*))
c-invar c ⟹ *a-invar* (*snd* (*c-α c n*))
c-invar c ⟹ *a-α* (*snd* (*c-α c n*)) = {*bs*. *length bs* = *n*}

locale *presburger-locale* =
nfa: *StdNFA nfa-ops* +
presburger-label-set-cache a-α a-invar c-α c-invar c-init +
dfa-construct-no-enc-fun nfa-op-α nfa-ops nfa-op-invar nfa-ops a-α a-invar dfa-construct +
labels-gen: *nfa-rename-labels-gen nfa-op-α nfa-ops nfa-op-invar nfa-ops*
nfa-op-α nfa-ops nfa-op-invar nfa-ops a-α a-invar rename-labels-gen
for *a-α* :: *'bl-set* ⇒ (*bool list*) *set* **and** *a-invar c-init* **and**
c-α :: *'cache* ⇒ *nat* ⇒ (*'cache* × *'bl-set*) **and** *c-invar* **and**
nfa-ops :: (*'q*::{*NFA-states*}, *bool list*, *'nfa*) *nfa-ops* **and**
dfa-construct :: (*pres-NFA-state*,*nat*,*bool list*,*'bl-set*,*'nfa*) *nfa-construct-fun* **and**
rename-labels-gen
begin
definition *pres-DFA-eq-ineq-impl* **where**
pres-DFA-eq-ineq-impl A ineq n ks l =
dfa-construct pres-NFA-state-to-nat (*pres-NFA-state-int l*) *A*
(*if ineq then*
(*λq*. *case q of pres-NFA-state-error* ⇒ *False*

```
                    | pres-NFA-state-int m ⇒ (0 ≤ m))
        else
          (λq. case q of pres-NFA-state-error ⇒ False
                    | pres-NFA-state-int m ⇒ (m = 0)))
        (pres-DFA-eq-ineq-trans-fun ineq ks)

lemma pres-DFA-eq-ineq-impl-correct :
assumes A-OK: a-invar A a-α A = {bs. length bs = n}
shows invar (pres-DFA-eq-ineq-impl A ineq n ks l)
      NFA-isomorphic-wf (α (pres-DFA-eq-ineq-impl A ineq n ks l))
                    (efficient-pres-DFA-eq-ineq ineq n ks l)
proof −
  have invar (pres-DFA-eq-ineq-impl A ineq n ks l) ∧
        NFA-isomorphic-wf (α (pres-DFA-eq-ineq-impl A ineq n ks l))
                    (NFA-remove-unreachable-states (pres-DFA-eq-ineq ineq n ks l))
  unfolding pres-DFA-eq-ineq-impl-def
    apply (rule dfa-construct-no-enc-fun-correct)
    apply (rule pres-DFA-eq-ineq---is-well-formed)
    apply (auto simp add: pres-DFA-eq-ineq-def A-OK inj-pres-NFA-state-to-nat
                  split: pres-NFA-state.split)
  done
  thus invar (pres-DFA-eq-ineq-impl A ineq n ks l)
       NFA-isomorphic-wf (α (pres-DFA-eq-ineq-impl A ineq n ks l))
                    (efficient-pres-DFA-eq-ineq ineq n ks l)
  by (simp-all add: NFA-isomorphic-wf-trans[OF - pres-DFA-eq-ineq---isomorphic-wf])
qed

definition pres-DFA-exists-min-impl where
  pres-DFA-exists-min-impl A AA =
    right-quotient-lists (list-all (λb. ¬ b)) (minimise-Brzozowski (rename-labels-gen AA A tl))

lemma im-tl-eq: tl ‘ {bl. length bl = Suc n} = {bl. length bl = n}
   apply (auto simp add: image-iff length-Suc-conv)[]
   apply (simp add: ex-simps[symmetric] del: ex-simps)
done

lemma pres-DFA-exists-min-impl-correct-invar :
  assumes nfa.invar AA
        a-invar A Σ (nfa.α AA) = {bl. length bl = Suc n}
        a-α A = {bl. length bl = n}
  shows nfa.invar (pres-DFA-exists-min-impl A AA)
  unfolding pres-DFA-exists-min-impl-def pres-DFA-exists-min-def pres-DFA-labels-tl-def
  apply (insert assms)
  apply (intro nfa.correct-isomorphic conjI rename-labels-gen-correct---isomorphic)+
  apply (simp-all add: im-tl-eq)
done

lemma pres-DFA-exists-min-impl-correct-α :
  assumes nfa.invar AA
        NFA-isomorphic-wf (nfa.α AA) 𝒜
        a-invar A Σ (nfa.α AA) = {bl. length bl = Suc n}
        a-α A = {bl. length bl = n}
  shows NFA-isomorphic-wf (nfa.α (pres-DFA-exists-min-impl A AA)) (pres-DFA-exists-min n 𝒜)
  unfolding pres-DFA-exists-min-impl-def pres-DFA-exists-min-def pres-DFA-labels-tl-def
  apply (insert assms)
  apply (intro nfa.correct-isomorphic conjI rename-labels-gen-correct---isomorphic)+
  apply (simp-all add: im-tl-eq)
proof −
   assume iso: NFA-isomorphic-wf (nfa-op-α nfa-ops AA) 𝒜
      and Σ-eq: Σ (nfa-op-α nfa-ops AA) = {bl. length bl = Suc n}
   hence Σ-eq': Σ 𝒜 = {bl. length bl = Suc n}
     by (simp add: NFA-isomorphic-wf-Σ)
```

from *iso* **have** *NFA* $\mathcal{A}$ **unfolding** *NFA-isomorphic-wf-alt-def* **by** *simp*
hence Σ-*min*: Σ (*NFA-minimise* (*NFA-rename-labels* $\mathcal{A}$ *tl*)) = {*bl*. *length bl* = *n*}
by (*simp add*: *NFA-minimise-spec*(*2*) *NFA.NFA-rename-labels---is-well-formed* Σ-*eq*$'$ *im-tl-eq*)

have {*replicate n False*} $\cap$ {*bl*. *length bl* = *n*} =
{*a*. *list-all Not a*} $\cap$ {*bl*. *length bl* = *n*}
apply (*intro set-eqI iffI*)
apply (*induct n*, *simp-all*)
apply (*induct n*, *auto simp add*: *length-Suc-conv*)
done
with Σ-*min*
show {*replicate n False*} $\cap$ Σ (*NFA-minimise* (*NFA-rename-labels* $\mathcal{A}$ *tl*)) =
{*a*. *list-all Not a*} $\cap$ Σ (*NFA-minimise* (*NFA-rename-labels* $\mathcal{A}$ *tl*))
by *metis*
qed

lemmas *pres-DFA-exists-min-impl-correct* =
pres-DFA-exists-min-impl-correct-invar pres-DFA-exists-min-impl-correct-α

definition *pres-DFA-forall-min-impl* **where**
pres-DFA-forall-min-impl A AA =
complement (*pres-DFA-exists-min-impl A* (*complement AA*))

lemma *pres-DFA-forall-min-impl-correct-invar* :
assumes *nfa.invar AA*
a-invar A Σ (*nfa*.α *AA*) = {*bl*. *length bl* = *Suc n*}
a-α *A* = {*bl*. *length bl* = *n*}
shows *nfa.invar* (*pres-DFA-forall-min-impl A AA*)
unfolding *pres-DFA-forall-min-impl-def pres-DFA-forall-min-def pres-DFA-labels-tl-def*
apply (*insert assms*)
apply (*intro nfa.correct-isomorphic conjI*
pres-DFA-exists-min-impl-correct | *assumption*)+
apply (*simp-all add*: *nfa.correct*)
done

lemma *pres-DFA-forall-min-impl-correct-*α :
assumes *nfa.invar AA*
NFA-isomorphic-wf (*nfa*.α *AA*) $\mathcal{A}$
a-invar A Σ (*nfa*.α *AA*) = {*bl*. *length bl* = *Suc n*}
a-α *A* = {*bl*. *length bl* = *n*}
shows *NFA-isomorphic-wf* (*nfa*.α (*pres-DFA-forall-min-impl A AA*)) (*pres-DFA-forall-min n* $\mathcal{A}$)
unfolding *pres-DFA-forall-min-impl-def pres-DFA-forall-min-def pres-DFA-labels-tl-def*
apply (*insert assms*)
apply (*intro nfa.correct-isomorphic conjI*
pres-DFA-exists-min-impl-correct | *assumption*)+
apply (*simp-all add*: *nfa.correct*)
done

lemmas *pres-DFA-forall-min-impl-correct* =
pres-DFA-forall-min-impl-correct-invar pres-DFA-forall-min-impl-correct-α

fun *nfa-of-pf* :: *nat* $\Rightarrow$ *pf* $\Rightarrow$ $'$*cache* $\Rightarrow$ $'$*nfa* $\times$ $'$*cache* **where**
Eq: *nfa-of-pf n* (*Eq ks l*) *c* =
(*let* (*c*$'$, *A*) = *c-*α *c n in*
(*pres-DFA-eq-ineq-impl A False n ks l*, *c*$'$))
| *Le*: *nfa-of-pf n* (*Le ks l*) *c* =
(*let* (*c*$'$, *A*) = *c-*α *c n in*
(*pres-DFA-eq-ineq-impl A True n ks l*, *c*$'$))
| *And*: *nfa-of-pf n* (*And p q*) *c* =
(*let* (*P*, *c*$'$) = *nfa-of-pf n p c in*
let (*Q*, *c*$''$) = *nfa-of-pf n q c*$'$ *in*

(*bool-comb op*$\wedge$ *P Q*, *c''*))
| *Or*: *nfa-of-pf n* (*Or p q*) *c* =
(*let* (*P*, *c'*) = *nfa-of-pf n p c in*
let (*Q*, *c''*) = *nfa-of-pf n q c' in*
(*bool-comb op*$\vee$ *P Q*, *c''*))
| *Imp*: *nfa-of-pf n* (*Imp p q*) *c* =
(*let* (*P*, *c'*) = *nfa-of-pf n p c in*
let (*Q*, *c''*) = *nfa-of-pf n q c' in*
(*bool-comb op*$\longrightarrow$ *P Q*, *c''*))
| *Exists*: *nfa-of-pf n* (*Exist p*) *c* =
(*let* (*c'*, *A*) = *c-*α *c n in*
let (*P*, *c''*) = *nfa-of-pf* (*Suc n*) *p c' in*
(*pres-DFA-exists-min-impl A P*, *c''*))
| *Forall*: *nfa-of-pf n* (*Forall p*) *c* =
(*let* (*c'*, *A*) = *c-*α *c n in*
let (*P*, *c''*) = *nfa-of-pf* (*Suc n*) *p c' in*
(*pres-DFA-forall-min-impl A P*, *c''*))
| *Neg*: *nfa-of-pf n* (*Neg p*) *c* =
(*let* (*P*, *c'*) = *nfa-of-pf n p c in*
(*complement P*, *c'*))

lemmas *nfa-of-pf-induct* =
nfa-of-pf.induct [*case-names Eq Le And Or Imp Exist Forall Neg*]

lemma *nfa-of-pf---correct*:
assumes *c-invar c*
shows *c-invar* (*snd* (*nfa-of-pf n p c*)) $\wedge$
nfa.invar (*fst* (*nfa-of-pf n p c*)) $\wedge$
NFA-isomorphic-wf (*nfa.*α (*fst* (*nfa-of-pf n p c*)))
(*DFA-of-pf n p*)
using *assms*
proof (*induct n p arbitrary*: *c rule*: *DFA-of-pf-induct*)
case (*Eq n ks l c*) **note** *invar-c* = *this*
from *cache-correct* [*OF invar-c*, *of n*]
show *?case*
by (*auto simp add*: *pres-DFA-eq-ineq-impl-correct split*: *prod.split*)
next
case (*Le n ks l c*) **note** *invar-c* = *this*
from *cache-correct* [*OF invar-c*, *of n*]
show *?case*
by (*auto simp add*: *pres-DFA-eq-ineq-impl-correct split*: *prod.split*)
next
case (*And n p q c*)
obtain *P c'* **where** [*simp*]: *nfa-of-pf n p c* = (*P*, *c'*) **by** (*rule PairE*)
obtain *Q c''* **where** [*simp*]: *nfa-of-pf n q c'* = (*Q*, *c''*) **by** (*rule PairE*)

from *And*(*1*)[*of c*] *And*(*2*)[*of c'*] *And*(*3*)
show *?case*
apply *simp*
apply (*intro conjI nfa.correct-isomorphic*)
apply *simp-all*
apply (*metis NFA-isomorphic-wf-*Σ *DFA-of-pf---correct*)+
done
next
case (*Or n p q c*)
obtain *P c'* **where** [*simp*]: *nfa-of-pf n p c* = (*P*, *c'*) **by** (*rule PairE*)
obtain *Q c''* **where** [*simp*]: *nfa-of-pf n q c'* = (*Q*, *c''*) **by** (*rule PairE*)

from *Or*(*1*)[*of c*] *Or*(*2*)[*of c'*] *Or*(*3*)
show *?case*
apply *simp*
apply (*intro conjI nfa.correct-isomorphic*)

```
      apply simp-all
      apply (metis NFA-isomorphic-wf-Σ DFA-of-pf---correct)+
    done
  next
    case (Imp n p q c)
    obtain P c′ where [simp]: nfa-of-pf n p c = (P, c′) by (rule PairE)
    obtain Q c′′ where [simp]: nfa-of-pf n q c′ = (Q, c′′) by (rule PairE)

    from Imp(1)[of c] Imp(2)[of c′] Imp(3)
    show ?case
      apply simp
      apply (intro conjI nfa.correct-isomorphic)
      apply simp-all
      apply (metis NFA-isomorphic-wf-Σ DFA-of-pf---correct)+
    done
  next
    case (Exist n p c)
    obtain c′ A where [simp]: c-α c n = (c′, A) by (rule PairE)
    obtain P c′′ where [simp]: nfa-of-pf (Suc n) p c′ = (P, c′′) by (rule PairE)

    from Exist(1)[of c′] Exist(2) cache-correct [of c n]
    show ?case
      apply simp
      apply (intro conjI pres-DFA-exists-min-impl-correct)
      apply simp-all
      apply (metis NFA-isomorphic-wf-Σ DFA-of-pf---correct)+
    done
  next
    case (Forall n p c)
    obtain c′ A where [simp]: c-α c n = (c′, A) by (rule PairE)
    obtain P c′′ where [simp]: nfa-of-pf (Suc n) p c′ = (P, c′′) by (rule PairE)

    from Forall(1)[of c′] Forall(2) cache-correct [of c n]
    show ?case
      apply simp
      apply (intro conjI pres-DFA-forall-min-impl-correct)
      apply simp-all
      apply (metis NFA-isomorphic-wf-Σ DFA-of-pf---correct)+
    done
  next
    case (Neg n p c)
    obtain P c′ where [simp]: nfa-of-pf n p c = (P, c′) by (rule PairE)

    from Neg(1)[of c] Neg(2)
    show ?case
      apply simp
      apply (intro conjI nfa.correct-isomorphic)
      apply simp-all
    done
  qed


definition pf-to-nfa where
  pf-to-nfa n pf = fst (nfa-of-pf n pf c-init)

lemma pf-to-nfa---correct:
shows nfa.invar (pf-to-nfa n p)
      NFA-isomorphic-wf (nfa.α (pf-to-nfa n p)) (DFA-of-pf n p)
using nfa-of-pf---correct [of c-init n p]
unfolding pf-to-nfa-def
by (simp-all add: init-cache-OK)
```

lemma *eval-pf-impl* :
  eval-pf pf [] = *accept* (*pf-to-nfa 0 pf*) []
proof −
  note *equiv-wf* = *pf-to-nfa---correct* [*of 0 pf*]
  note *NFA-accept-OK* = *accept-correct---isomorphic* [*OF equiv-wf*, *of* []]
  with *DFA-of-pf---correct* [*of 0 pf*] *NFA-accept-OK*
  show *?thesis* **by** *simp*
qed

end

locale *presburger-label-set-cache-by-map-set* =
    s: *StdSet s-ops* + *m*: *StdMap m-ops* +
    set-image set-op-α s-ops set-op-invar s-ops set-op-α s-ops set-op-invar s-ops s-image
    for *s-ops* :: (*bool list*, *'bl*, -) *set-ops-scheme*
    and *m-ops* :: (*nat*, *'bl*, *'m*, -) *map-ops-scheme*
    and *s-image*
begin
  definition *c-invar* **where**
    c-invar m ≡ *m.invar m* ∧
      (∀ *n bl*. *m.α m n* = *Some bl* ⟶ *s.invar bl* ∧ *s.α bl* = {*bv* . *length bv* = *n*})

  primrec *c-α* **where**
     c-α m 0 = (*m*, *s.sng* [])
   | *c-α m* (*Suc n*) =
      (*case* (*m.lookup* (*Suc n*) *m*) *of Some bl* ⇒ (*m*, *bl*) | *None* =>
       (*let* (*m'*, *bl*) = *c-α m n in*
        let bl' = *s.union* (*s-image* (λ*l*. *True* # *l*) *bl*) (*s-image* (λ*l*. *False* # *l*) *bl*) *in*
        let m'' = *m.update* (*Suc n*) *bl' m' in*
        (*m''*, *bl'*)))

  lemma *c-α-correct* :
    assumes *c-invar m*
    shows *c-invar* (*fst* (*c-α m n*)) ∧
           s.invar (*snd* (*c-α m n*)) ∧
           s.α (*snd* (*c-α m n*)) = {*bs*. *length bs* = *n*}
  using *assms*
  proof (*induct n arbitrary*: *m*)
    case *0* **thus** *?case* **by** (*simp add*: *s.correct*)
  next
    case (*Suc n m*)
    note *ind-hyp* = *Suc*(*1*)
    note *invar-m* = *Suc*(*2*)

    show *?case*
    proof (*cases m.lookup* (*Suc n*) *m*)
      case (*Some bl*)
      thus *?thesis*
        apply (*simp add*: *invar-m*)
        apply (*insert invar-m*)
        apply (*simp add*: *c-invar-def m.correct*)
      done
    next
      case *None* **note** *lookup-eq-none*[*simp*] = *this*
      obtain *m' bl* **where** [*simp*]: *c-α m n* = (*m'*, *bl*) **by** (*rule PairE*)

      def *bl'* ≡ *set-op-union s-ops* (*s-image* (*op* # *True*) *bl*) (*s-image* (*op* # *False*) *bl*)

      from *ind-hyp* [*OF invar-m*]
      have *bl'-props*: *s.invar bl' s.α bl'* = {*bs*. *length bs* = *Suc n*} **and** *invar-m'*: *c-invar m'*

```
      apply (simp-all add: s.correct bl′-def image-correct)
      apply (auto simp add: image-iff length-Suc-conv)
    done

    from invar-m′
    show ?thesis
      by (simp add: bl′-def[symmetric] bl′-props c-invar-def m.correct)
  qed
qed

lemma presburger-label-set-cache-OK :
  presburger-label-set-cache s.α s.invar c-α c-invar m.empty
  unfolding presburger-label-set-cache-def
  apply (simp add: c-α-correct)
  apply (simp add: c-invar-def m.correct)
done
end

end
```