Automata

Thomas Tuerk (tuerk@in.tum.de)

February 6, 2012

Contents

1 Labeled transition systems

theory *LTS*
imports *Main*
begin

This theory defines labeled transition systems (LTS).

1.1 Auxiliary Definitions

lemma *lists-product* :
$w \in lists\ (\Sigma 1 \times \Sigma 2) \longleftrightarrow (map\ fst\ w) \in lists\ \Sigma 1 \wedge (map\ snd\ w) \in lists\ \Sigma 2$
⟨*proof*⟩

lemma *lists-inter* : $w \in lists\ (\Sigma 1 \cap \Sigma 2) = (w \in lists\ \Sigma 1 \wedge w \in lists\ \Sigma 2)$
⟨*proof*⟩

lemma *lists-rev* [*simp*] :
$(rev\ w) \in lists\ \Sigma \longleftrightarrow w \in lists\ \Sigma$
⟨*proof*⟩

lemma *rev-image-lists* [*simp*] :
$rev\ `\ (lists\ A) = lists\ A$
⟨*proof*⟩

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 $w = w_1 \dots w_n$ over Σ then a word $p = p_0 \dots p_n$ 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 $(w_1, r_1) \dots (w_n, r_n)$ 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 $(\sigma q' \# xs) = ((q, fst\ \sigma q', snd\ \sigma q') \in \Delta \wedge LTS\text{-}is\text{-}path\ \Delta\ (snd\ \sigma q')\ xs)$

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* :
  $\pi \neq [] \Longrightarrow$
   $(LTS\text{-}is\text{-}path\ \Delta\ q\ (\pi\ @\ \pi') = (LTS\text{-}is\text{-}path\ \Delta\ q\ \pi \wedge LTS\text{-}is\text{-}path\ \Delta\ (snd\ (last\ \pi))\ \pi'))$
⟨*proof*⟩

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 $(\sigma \# w)$ $q' =$
     $(\exists q''.\ (q, \sigma, q'') \in \Delta \wedge LTS\text{-}is\text{-}reachable\ \Delta\ q''\ w\ q')$

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 $\pi \equiv map\ snd\ \pi$

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

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

lemma [*simp*] : *LTS-path* $[] = []$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path-L* $[] = []$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path* $(\sigma q \# \pi) = (snd\ \sigma q \# LTS\text{-}path\ \pi)$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path-L* $(\sigma q \# \pi) = (fst\ \sigma q \# LTS\text{-}path\text{-}L\ \pi)$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path* $\pi = [] \longleftrightarrow \pi = []$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path-L* $\pi = [] \longleftrightarrow \pi = []$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path-last-S* q $[] = q$ ⟨*proof*⟩
lemma [*simp*] : *LTS-path-last-S* q $(x \# xs) = LTS\text{-}path\text{-}last\text{-}S\ (snd\ x)\ xs$
   ⟨*proof*⟩

lemma *path-last-S-in* :
  LTS-path-last-S q $\pi \in (insert\ q\ (set\ (LTS\text{-}path\ \pi)))$
⟨*proof*⟩

lemma *LTS-is-reachable-alt-def* :
  LTS-is-reachable Δ q w $q' = (\exists \pi.\ LTS\text{-}is\text{-}path\ \Delta\ q\ \pi \wedge LTS\text{-}path\text{-}last\text{-}S\ q\ \pi = q' \wedge LTS\text{-}path\text{-}L\ \pi = w)$
  (**is** - = $(\exists \pi.\ ?pathPred\ q\ w\ \pi)$)
⟨*proof*⟩

lemma *LTS-is-reachable-concat* :
   LTS-is-reachable Δ q $(w\ @\ w')$ $q' =$
    $(\exists q''.\ LTS\text{-}is\text{-}reachable\ \Delta\ q\ w\ q'' \wedge LTS\text{-}is\text{-}reachable\ \Delta\ q''\ w'\ q')$
⟨*proof*⟩

lemma *LTS-is-reachable-snoc* [*simp*] :
LTS-is-reachable $\Delta\ q\ (w\ @\ [\sigma])\ q' =$
$(\exists\, q''.\ \textit{LTS-is-reachable}\ \Delta\ q\ w\ q'' \wedge (q'', \sigma, q') \in \Delta)$
⟨*proof*⟩

Unreachability is often interesting as well.

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

lemma *LTS-is-unreachable-not-refl* [*simp*] :
$\neg(\textit{LTS-is-unreachable}\ \Delta\ \Sigma\ q\ q)$
⟨*proof*⟩

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*: $w \in \textit{lists}\ \Sigma$
shows *LTS-is-unreachable* $\Delta\ \Sigma\ q''\ q'$
⟨*proof*⟩

lemma *LTS-is-unreachable-reachable-end* :
$[\![\textit{LTS-is-unreachable}\ \Delta\ \Sigma\ q\ q';\ \textit{LTS-is-reachable}\ \Delta\ q''\ w\ q';\ w \in \textit{lists}\ \Sigma]\!] \Longrightarrow$
LTS-is-unreachable $\Delta\ \Sigma\ q\ q''$
⟨*proof*⟩

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 $\Delta = (\forall\, q\ \sigma\ q1'\ q2'.\ ((q, \sigma, q1') \in \Delta \wedge (q, \sigma, q2') \in \Delta) \longrightarrow (q1' = q2'))$

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

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

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

lemma *DLTS-to-LTS-alt-def* [*simp*]:
$(q, \sigma, q') \in \textit{DLTS-to-LTS}\ \delta \longleftrightarrow (\delta(q, \sigma) = \textit{Some}\ q')$
⟨*proof*⟩

lemma *DLTS-to-LTS---LTS-is-weak-deterministic* [*simp*] :
LTS-is-weak-deterministic (*DLTS-to-LTS* δ)
⟨*proof*⟩

lemma *DLTS-to-LTS---LTS-is-deterministic* :
$[\![\bigwedge q\ \sigma.\ [\![q \in \mathcal{Q};\ \sigma \in \Sigma]\!] \Longrightarrow \neg\ (\delta\ (q, \sigma) = \textit{None})]\!] \Longrightarrow$
LTS-is-deterministic $\mathcal{Q}\ \Sigma$ (*DLTS-to-LTS* δ)
⟨*proof*⟩

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

lemma *LTS-to-DLTS-in* :
$(q, \sigma, q') \in \Delta \Longrightarrow (\exists\, q''.\ \textit{LTS-to-DLTS}\ \Delta\ (q,\sigma) = \textit{Some}\ q'' \wedge (q, \sigma, q'') \in \Delta)$
⟨*proof*⟩

lemma *LTS-to-DLTS-is-some* :
LTS-to-DLTS Δ (q,σ) = *Some* $q' \Longrightarrow (q,\ \sigma,\ q') \in \Delta$
⟨*proof*⟩

lemma *LTS-to-DLTS-is-none* :
(*LTS-to-DLTS* Δ (q,σ) = *None*) $\longleftrightarrow$ $(\forall\, q'.\ (q,\ \sigma,\ q') \notin \Delta)$
⟨*proof*⟩

lemma *LTS-to-DLTS-is-some-det* :
LTS-is-weak-deterministic $\Delta \Longrightarrow$
((*LTS-to-DLTS* Δ $(q,\ \sigma)$ = *Some* q') $\longleftrightarrow$ $(q,\ \sigma,\ q') \in \Delta$)
⟨*proof*⟩

lemma *DLTS-to-LTS-inv* [*simp*] :
LTS-to-DLTS (*DLTS-to-LTS* δ) = δ
⟨*proof*⟩

lemma *LTS-to-DLTS-inv* :
LTS-is-weak-deterministic $\Delta \Longrightarrow$
DLTS-to-LTS (*LTS-to-DLTS* Δ) = Δ
⟨*proof*⟩

lemma *LTS-is-reachable---LTS-is-weak-deterministic* :
assumes *det-*Δ: *LTS-is-weak-deterministic* Δ
shows $[\![$*LTS-is-reachable* Δ q w q'; *LTS-is-reachable* Δ q w $q''$$]\!] \Longrightarrow (q' = q'')$
⟨*proof*⟩

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-path-of δ q [] = *Some* []
| *DLTS-path-of* δ q $(\sigma$ # $w)$ =
Option.bind $(\delta\ (q,\ \sigma))$ $(\lambda q'.$ *Option.map* $(\lambda ql.\ q'$ # $ql)$ $($*DLTS-path-of* $\delta\ q'\ w))$

lemma *LTS-is-path---DLTS-path-of* :
shows *LTS-is-path* (*DLTS-to-LTS* δ) q wr =
(*DLTS-path-of* δ q (*LTS-path-L* wr) = *Some* (*LTS-path* wr))
⟨*proof*⟩

fun *DLTS-reach* :: $('q,\ 'a)$ *DLTS* $\Rightarrow$ $'q \Rightarrow$ $'a$ *list* $\Rightarrow$ $'q$ *option* **where**
DLTS-reach δ q [] = *Some* q
| *DLTS-reach* δ q $(\sigma$ # $w)$ =
Option.bind $(\delta\ (q,\ \sigma))$ $(\lambda q'.$ *DLTS-reach* $\delta\ q'\ w)$

lemma *DLTS-reach-Concat* [*simp*] :
DLTS-reach δ q $(w$ @ $w')$ = *Option.bind* (*DLTS-reach* δ q w) $(\lambda q'.$ *DLTS-reach* $\delta\ q'\ w')$
⟨*proof*⟩

lemma *DLTS-reach-alt-def* :
DLTS-reach δ q w = *Option.map* $(\lambda wl.$ *last* $(q$ # $wl))$ (*DLTS-path-of* δ q w)
⟨*proof*⟩

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*⟩

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')
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma *rtrancl-LTS-forget-labels* :
rtrancl (*LTS-forget-labels* Δ) =
$\{(q, q'). (\exists w.$ *LTS-is-reachable* Δ q w $q')\}$
⟨*proof*⟩

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* $\Delta\}$
⟨*proof*⟩

1.5 Restricting and extending the transition relation

lemma *LTS-is-path-mono* :
⟦$\Delta \subseteq \Delta'$; *LTS-is-path* Δ q π⟧ $\Longrightarrow$ *LTS-is-path* Δ' q π
⟨*proof*⟩

lemma *LTS-is-reachable-mono* :
⟦$\Delta \subseteq \Delta'$; *LTS-is-reachable* Δ q w q'⟧ $\Longrightarrow$ *LTS-is-reachable* Δ' q w q'
⟨*proof*⟩

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* $\Delta 1$ $\Delta 2$ = $((p,\sigma,p') \in \Delta 1 \wedge (q,\sigma,q') \in \Delta 2)$
⟨*proof*⟩

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))) $\in \Delta 2$)
⟨*proof*⟩

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* π $\longleftrightarrow$
LTS-is-path $\Delta 1$ (*fst pq*) (*LTS-product-path1* π) $\wedge$ *LTS-is-path* $\Delta 2$ (*snd pq*) (*LTS-product-path2* π)
⟨*proof*⟩

lemma *LTS-product-path1-L* [*simp*]: *LTS-path-L* (*LTS-product-path1* π) = *LTS-path-L* π
⟨*proof*⟩

lemma *LTS-product-path2-L* [*simp*]: *LTS-path-L* (*LTS-product-path2* π) = *LTS-path-L* π
⟨*proof*⟩

lemma *LTS-product-path1-S*: *LTS-path* (*LTS-product-path1* π) = *map fst* (*LTS-path* π)
⟨*proof*⟩

lemma *LTS-product-path2-S*: *LTS-path* (*LTS-product-path2* π) = *map snd* (*LTS-path* π)
⟨*proof*⟩

lemma *LTS-product-path1-last-S*: *LTS-path-last-S* (*fst pq*) (*LTS-product-path1* π) = *fst* (*LTS-path-last-S pq* π)
⟨*proof*⟩

lemma *LTS-product-path2-last-S*: *LTS-path-last-S* (*snd pq*) (*LTS-product-path2* π) = *snd* (*LTS-path-last-S pq* π)
⟨*proof*⟩

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'*))
⟨*proof*⟩

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* $\Delta 1$ $\Delta 2$)
⟨*proof*⟩

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* $\Delta 1$ $\Delta 2$)
⟨*proof*⟩

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* $\longleftrightarrow$ *n* = *m*
⟨*proof*⟩

lemma *not-finite-NFA-states-UNIV* : ~(*finite* (*UNIV*::*'a set*))
⟨*proof*⟩

end

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

instance ⟨*proof*⟩
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* ∈ *lists* (Σ $\mathcal{A}$)) ∧ (∃ *q*∈($\mathcal{I}$ $\mathcal{A}$). ∃ *q*′∈($\mathcal{F}$ $\mathcal{A}$). *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* ∈ *lists* (Σ $\mathcal{A}$)) ∧ (∃ *q*′∈ ($\mathcal{F}$ $\mathcal{A}$). *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*}, $\mathcal{F}$ = $\mathcal{F}$ $\mathcal{A}$ |)
⟨*proof*⟩

definition $\mathcal{L}$-*left* **where**
$\mathcal{L}$-*left* $\mathcal{A}$ *q* = {*w*. (*w* ∈ *lists* (Σ $\mathcal{A}$)) ∧ (∃ *i* ∈ ($\mathcal{I}$ $\mathcal{A}$). *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*} |)
⟨*proof*⟩

lemma *NFA-accept-alt-def* : *NFA-accept* $\mathcal{A}$ *w* ⟷ *w* ∈ $\mathcal{L}$ $\mathcal{A}$ ⟨*proof*⟩

lemma $\mathcal{L}$-*alt-def* :
$\mathcal{L}$ $\mathcal{A}$ = $\bigcup$ (($\mathcal{L}$-*in-state* $\mathcal{A}$) ‘ ($\mathcal{I}$ $\mathcal{A}$))
⟨*proof*⟩

lemma *in-*$\mathcal{L}$*-in-state-Nil* [*simp*] : [] ∈ $\mathcal{L}$-*in-state* $\mathcal{A}$ *q* ⟷ (*q* ∈ $\mathcal{F}$ $\mathcal{A}$) ⟨*proof*⟩
lemma *in-*$\mathcal{L}$*-in-state-Cons* [*simp*] : (*σ* # *w*) ∈ $\mathcal{L}$-*in-state* $\mathcal{A}$ *q* ⟷
(∃ *q*′. *σ* ∈ Σ $\mathcal{A}$ ∧ (*q*, *σ*, *q*′) ∈ Δ $\mathcal{A}$ ∧ *w* ∈ $\mathcal{L}$-*in-state* $\mathcal{A}$ *q*′) ⟨*proof*⟩

definition *remove-prefix* :: ′*a list* ⇒ (′*a list*) *set* ⇒ (′*a list*) *set* **where**
remove-prefix pre L ≡ *drop* (*length pre*) ‘ (*L* ∩ {*pre* @ *w* |*w*. *True*})

lemma *remove-prefix-alt-def* [*simp*] :
w ∈ *remove-prefix pre L* ⟷ (*pre* @ *w* ∈ *L*)
⟨*proof*⟩

lemma *remove-prefix-Nil* [*simp*] :
remove-prefix [] *L* = *L* ⟨*proof*⟩

lemma *remove-prefix-Combine* [*simp*] :
remove-prefix p2 (*remove-prefix p1 L*) =
remove-prefix (*p1* @ *p2*) *L*
⟨*proof*⟩

lemma $\mathcal{L}$*-in-state-remove-prefix* :
pre ∈ *lists* (Σ $\mathcal{A}$) ⟹
(*remove-prefix pre* ($\mathcal{L}$*-in-state* $\mathcal{A}$ *q*) =
⋃ {$\mathcal{L}$*-in-state* $\mathcal{A}$ *q'* | *q'*. *LTS-is-reachable* (Δ $\mathcal{A}$) *q pre q'*})
⟨*proof*⟩

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* ∈ $\mathcal{F}$ $\mathcal{A}$ ⟷ *q'* ∈ $\mathcal{F}$ $\mathcal{A}$)
⟨*proof*⟩

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*: ⋀*q* σ *q'*. (*q*,σ,*q'*) ∈ Δ $\mathcal{A}$ ⟹ (*q* ∈ $\mathcal{Q}$ $\mathcal{A}$) ∧ (σ ∈ Σ $\mathcal{A}$) ∧ (*q'* ∈ $\mathcal{Q}$ $\mathcal{A}$)
and $\mathcal{I}$*-consistent*: $\mathcal{I}$ $\mathcal{A}$ ⊆ $\mathcal{Q}$ $\mathcal{A}$
and $\mathcal{F}$*-consistent*: $\mathcal{F}$ $\mathcal{A}$ ⊆ $\mathcal{Q}$ $\mathcal{A}$
and *finite-*$\mathcal{Q}$: *finite* ($\mathcal{Q}$ $\mathcal{A}$)
and *finite-*Σ: *finite* (Σ $\mathcal{A}$)

lemma *NFA-intro* [*intro*!] :
⟦⋀*q* σ *q'*. (*q*,σ,*q'*) ∈ Δ $\mathcal{A}$ ⟹ (*q* ∈ $\mathcal{Q}$ $\mathcal{A}$) ∧ (σ ∈ Σ $\mathcal{A}$) ∧ (*q'* ∈ $\mathcal{Q}$ $\mathcal{A}$);
$\mathcal{I}$ $\mathcal{A}$ ⊆ $\mathcal{Q}$ $\mathcal{A}$; $\mathcal{F}$ $\mathcal{A}$ ⊆ $\mathcal{Q}$ $\mathcal{A}$; *finite* ($\mathcal{Q}$ $\mathcal{A}$); *finite* (Σ $\mathcal{A}$)⟧ ⟹ *NFA* $\mathcal{A}$
⟨*proof*⟩

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

lemma *dummy-NFA---is-NFA* :
NFA (*dummy-NFA q a*)
⟨*proof*⟩

lemma (**in** *NFA*) *wf-NFA* : *NFA* $\mathcal{A}$
⟨*proof*⟩

lemma (**in** *NFA*) *finite-*$\mathcal{I}$:
finite ($\mathcal{I}$ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *finite-*$\mathcal{F}$:
finite ($\mathcal{F}$ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) Δ*-subset* :
Δ $\mathcal{A}$ ⊆ $\mathcal{Q}$ $\mathcal{A}$ × Σ $\mathcal{A}$ × $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma (**in** *NFA*) *finite-*Δ :
finite (Δ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-*Δ*-cons---is-path* :
shows ⟦*q* ∈ $\mathcal{Q}$ $\mathcal{A}$; *LTS-is-path* (Δ $\mathcal{A}$) *q* π⟧ ⟹
π ∈ *lists* (Σ $\mathcal{A}$ × $\mathcal{Q}$ $\mathcal{A}$)

⟨*proof*⟩

lemma (**in** *NFA*) *NFA-Δ-cons---LTS-is-reachable* :
⟦*LTS-is-reachable* (Δ $\mathcal{A}$) q w q'⟧ ⟹ ($q \in \mathcal{Q}\ \mathcal{A} \longrightarrow q' \in \mathcal{Q}\ \mathcal{A}$) ∧ $w \in$ *lists* (Σ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *LTS-is-reachable---labels* :
LTS-is-reachable (Δ $\mathcal{A}$) q w q' ⟹ $w \in$ *lists* (Σ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-accept-wf-def* :
NFA-accept $\mathcal{A}$ w = (∃ q∈(*I* $\mathcal{A}$). ∃ q'∈(*F* $\mathcal{A}$). *LTS-is-reachable* (Δ $\mathcal{A}$) q w q')
⟨*proof*⟩

lemma (**in** *NFA*) *LTS-is-reachable-from-initial-alt-def* :
accessible (*LTS-forget-labels* (Δ $\mathcal{A}$)) (*I* $\mathcal{A}$) =
{q'. (∃ q∈ (*I* $\mathcal{A}$). ∃ w. *LTS-is-reachable* (Δ $\mathcal{A}$) q w q')}
⟨*proof*⟩

lemma (**in** *NFA*) *LTS-is-reachable-from-initial-subset* :
accessible (*LTS-forget-labels* (Δ $\mathcal{A}$)) (*I* $\mathcal{A}$) ⊆ $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma (**in** *NFA*) *LTS-is-reachable-from-initial-finite* :
finite (*accessible* (*LTS-forget-labels* (Δ $\mathcal{A}$)) (*I* $\mathcal{A}$))
⟨*proof*⟩

2.2 Constructing from a list representation

fun *construct-NFA-aux* **where**
construct-NFA-aux $\mathcal{A}$ (*q1*, *l*, *q2*) =
(| $\mathcal{Q}$=*insert q1* (*insert q2* ($\mathcal{Q}$ $\mathcal{A}$)),
Σ=Σ $\mathcal{A}$ ∪ *set l*,
Δ=Δ $\mathcal{A}$ ∪ { (*q1*,*a*,*q2*) | *a*. *a* ∈ *set l*},
I = *I* $\mathcal{A}$, *F* = *F* $\mathcal{A}$|)

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

lemma *NFA-construct-alt-def* :
NFA-construct (*Q*, *A*, *D*, *I*, *F*) =
(| $\mathcal{Q}$=*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*⟩

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*⟩

lemma *NFA-construct-exists* :
fixes $\mathcal{A}$ :: ('*q*, '*a*) *NFA-rec*
assumes *wf-A*: *NFA* $\mathcal{A}$
shows ∃ *Q A D I F*. $\mathcal{A}$ = *NFA-construct* (*Q*, *A*, *D*, *I*, *F*)

⟨*proof*⟩

2.3 Removing states

definition *NFA-remove-states* :: ('*q*, '*a*, '*x*) *NFA-rec-scheme* ⇒ '*q set* ⇒ ('*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*) ∈ Δ $\mathcal{A}$ ∧ *s1* ∉ *S* ∧ *s2* ∉ *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* ⟨*proof*⟩
lemma [*simp*] : $\mathcal{Q}$ (*NFA-remove-states* $\mathcal{A}$ *S*) = $\mathcal{Q}$ $\mathcal{A}$ − *S* ⟨*proof*⟩
lemma [*simp*] : $\mathcal{F}$ (*NFA-remove-states* $\mathcal{A}$ *S*) = $\mathcal{F}$ $\mathcal{A}$ − *S* ⟨*proof*⟩
lemma [*simp*] : Σ (*NFA-remove-states* $\mathcal{A}$ *S*) = Σ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : *x* ∈ Δ (*NFA-remove-states* $\mathcal{A}$ *S*) ⟷
x ∈ Δ $\mathcal{A}$ ∧ *fst x* ∉ *S* ∧ *snd* (*snd x*) ∉ *S* ⟨*proof*⟩

lemma *NFA-remove-states-Δ-subset* : Δ (*NFA-remove-states* $\mathcal{A}$ *S*) ⊆ Δ $\mathcal{A}$ ⟨*proof*⟩
lemma *NFA-remove-states---is-well-formed* : *NFA* $\mathcal{A}$ ⟹ *NFA* (*NFA-remove-states* $\mathcal{A}$ *S*) ⟨*proof*⟩

lemma *NFA-remove-states-empty* [*simp*] : *NFA-remove-states* $\mathcal{A}$ {} = $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-remove-states-NFA-remove-states* [*simp*] : *NFA-remove-states* (*NFA-remove-states* $\mathcal{A}$ *S1*) *S2* =
NFA-remove-states $\mathcal{A}$ (*S1* ∪ *S2*)
⟨*proof*⟩

lemma *NFA-remove-states-$\mathcal{L}$-subset* : $\mathcal{L}$ (*NFA-remove-states* $\mathcal{A}$ *S*) ⊆ $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

lemma *LTS-is-reachable-NFA-remove-states* :
assumes *Q-unrech*: ⋀*q'*. *q'*∈ *Q* ⟹ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) *q q'*
and *w-in-Σ*: *w* ∈ *lists* (Σ $\mathcal{A}$)
shows *LTS-is-reachable* (Δ $\mathcal{A}$) *q w q'* ⟷ *LTS-is-reachable* (Δ (*NFA-remove-states* $\mathcal{A}$ *Q*)) *q w q'*
⟨*proof*⟩

lemma *NFA-remove-states-$\mathcal{L}$-in-state-iff* :
assumes *Q-unrech*: ⋀*q'*. *q'*∈*Q* ⟹ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) *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*⟩

lemma *NFA-remove-states-$\mathcal{L}$-iff* :
assumes *unreach-asm*: ⋀*q q'*. ⟦*q* ∈ $\mathcal{I}$ $\mathcal{A}$; *q'* ∈ *Q*⟧ ⟹ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) *q q'*
shows $\mathcal{L}$ (*NFA-remove-states* $\mathcal{A}$ *Q*) = $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-remove-states-accept-iff* :
assumes *unreach-asm*: ⋀*q q'*. ⟦*q* ∈ $\mathcal{I}$ $\mathcal{A}$; *q'*∈ *Q*⟧ ⟹ *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) *q q'*
shows *NFA-accept* (*NFA-remove-states* $\mathcal{A}$ *Q*) *w* = *NFA-accept* $\mathcal{A}$ *w*
⟨*proof*⟩

definition *NFA-unreachable-states* **where**
NFA-unreachable-states $\mathcal{A}$ = {*q'*. ∀ *q*∈$\mathcal{I}$ $\mathcal{A}$. *LTS-is-unreachable* (Δ $\mathcal{A}$) (Σ $\mathcal{A}$) *q q'*}

lemma *NFA-unreachable-states-alt-def* :
NFA-unreachable-states $\mathcal{A}$ = {*q*. $\mathcal{L}$*-left* $\mathcal{A}$ *q* = {}}
⟨*proof*⟩

lemma *NFA-unreachable-states-extend* :
⟦*x* ∉ *NFA-unreachable-states* $\mathcal{A}$; (*x*, σ, *q'*) ∈ Δ $\mathcal{A}$; σ ∈ Σ $\mathcal{A}$⟧ ⟹ *q'* ∉ *NFA-unreachable-states* $\mathcal{A}$
⟨*proof*⟩

definition *NFA-is-initially-connected* **where**

NFA-is-initially-connected $\mathcal{A} \longleftrightarrow \mathcal{Q}\ \mathcal{A} \cap$ *NFA-unreachable-states* $\mathcal{A} = \{\}$

lemma *NFA-is-initially-connected-alt-def* :
NFA-is-initially-connected $\mathcal{A} \longleftrightarrow (\forall\, q \in \mathcal{Q}\ \mathcal{A}.$
$\exists\, i{\in}\mathcal{I}\ \mathcal{A}.\ \exists\, w{\in}$*lists* $(\Sigma\ \mathcal{A}).$ *LTS-is-reachable* $(\Delta\ \mathcal{A})\ i\ w\ q)$
⟨*proof*⟩

lemma *dummy-NFA---NFA-is-initially-connected* :
NFA-is-initially-connected (*dummy-NFA* $q\ a$)
⟨*proof*⟩

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

lemma *NFA-remove-unreachable-states-*$\mathcal{L}$ [*simp*] :
$\mathcal{L}$ (*NFA-remove-unreachable-states* $\mathcal{A}$) = $\mathcal{L}\ \mathcal{A}$
⟨*proof*⟩

lemma *NFA-remove-unreachable-states-*$\mathcal{L}$*-in-state* [*simp*] :
assumes *q-in-Q*: $q \in \mathcal{Q}$ (*NFA-remove-unreachable-states* $\mathcal{A}$)
shows $\mathcal{L}$*-in-state* (*NFA-remove-unreachable-states* $\mathcal{A}$) $q = \mathcal{L}$*-in-state* $\mathcal{A}\ q$
⟨*proof*⟩

lemma *NFA-remove-unreachable-states-accept-iff* [*simp*] :
NFA-accept (*NFA-remove-unreachable-states* $\mathcal{A}$) w = *NFA-accept* $\mathcal{A}\ w$
⟨*proof*⟩

lemma *NFA-remove-unreachable-states---is-well-formed* [*simp*] :
NFA $\mathcal{A} \Longrightarrow$ *NFA* (*NFA-remove-unreachable-states* $\mathcal{A}$)
⟨*proof*⟩

lemma *NFA-unreachable-states-*$\mathcal{I}$:
$q \in \mathcal{I}\ \mathcal{A} \Longrightarrow q \notin$ *NFA-unreachable-states* $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-remove-unreachable-states-*$\mathcal{I}$ [*simp*] :
$\mathcal{I}$ (*NFA-remove-unreachable-states* $\mathcal{A}$) = $\mathcal{I}\ \mathcal{A}$
⟨*proof*⟩

lemma [*simp*] : Σ (*NFA-remove-unreachable-states* $\mathcal{A}$) = $\Sigma\ \mathcal{A}$
⟨*proof*⟩

lemma *NFA-unreachable-states-NFA-remove-unreachable-states* :
NFA-unreachable-states (*NFA-remove-unreachable-states* $\mathcal{A}$) =
NFA-unreachable-states $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-remove-unreachable-states-NFA-remove-unreachable-states* [*simp*] :
NFA-remove-unreachable-states (*NFA-remove-unreachable-states* $\mathcal{A}$) = *NFA-remove-unreachable-states* $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-remove-unreachable-states---NFA-is-initially-connected* :
NFA-is-initially-connected (*NFA-remove-unreachable-states* $\mathcal{A}$)
⟨*proof*⟩

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 $\mathcal{A}\ f \equiv$
(| $\mathcal{Q}$=f ‘ ($\mathcal{Q}\ \mathcal{A}$), Σ=$\Sigma\ \mathcal{A}$, Δ = { ($f\ s1,\ a,\ f\ s2$) | $s1\ a\ s2.\ (s1,a,s2) \in \Delta\ \mathcal{A}$},

$\mathcal{I}$=f ‘ ($\mathcal{I}$ $\mathcal{A}$), $\mathcal{F}$ = f ‘ ($\mathcal{F}$ $\mathcal{A}$) ⟧

lemma [*simp*] : $\mathcal{I}$ (*NFA-rename-states* $\mathcal{A}$ f) = f ‘ $\mathcal{I}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{Q}$ (*NFA-rename-states* $\mathcal{A}$ f) = f ‘ $\mathcal{Q}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{F}$ (*NFA-rename-states* $\mathcal{A}$ f) = f ‘ $\mathcal{F}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Σ (*NFA-rename-states* $\mathcal{A}$ S) = Σ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : (fq, σ, fq') ∈ Δ (*NFA-rename-states* $\mathcal{A}$ f) ⟷
(∃ q q'. (q, σ, q') ∈ Δ $\mathcal{A}$ ∧ (fq = f q) ∧ (fq' = f q'))
⟨*proof*⟩

lemma *NFA-rename-states---is-well-formed* :
NFA $\mathcal{A}$ ⟹ *NFA* (*NFA-rename-states* $\mathcal{A}$ f)
⟨*proof*⟩

lemma *NFA-rename-states-id* [*simp*] : *NFA-rename-states* $\mathcal{A}$ *id* = $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-rename-states-NFA-rename-states* [*simp*] :
NFA-rename-states (*NFA-rename-states* $\mathcal{A}$ *f1*) *f2* =
NFA-rename-states $\mathcal{A}$ (*f2* ∘ *f1*)
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-rename-states-agree-on-Q* :
assumes *f12-agree*: ⋀q. q ∈ $\mathcal{Q}$ $\mathcal{A}$ ⟹ *f1* q = *f2* q
shows *NFA-rename-states* $\mathcal{A}$ *f1* = *NFA-rename-states* $\mathcal{A}$ *f2*
⟨*proof*⟩

lemma *LTS-is-reachable---NFA-rename-statesE* :
LTS-is-reachable (Δ $\mathcal{A}$) q w q' ⟹
LTS-is-reachable (Δ (*NFA-rename-states* $\mathcal{A}$ f)) (f q) w (f q')
⟨*proof*⟩

lemma $\mathcal{L}$*-in-state-rename-subset1* :
$\mathcal{L}$*-in-state* $\mathcal{A}$ q ⊆ $\mathcal{L}$*-in-state* (*NFA-rename-states* $\mathcal{A}$ f) (f q)
⟨*proof*⟩

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

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

lemma *NFA-is-strong-equivalence-rename-funE* :
⟦*NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f;
q ∈ $\mathcal{Q}$ $\mathcal{A}$; q'∈ $\mathcal{Q}$ $\mathcal{A}$⟧ ⟹
f q = f q' ⟷ ($\mathcal{L}$*-in-state* $\mathcal{A}$ q = $\mathcal{L}$*-in-state* $\mathcal{A}$ q')
⟨*proof*⟩

lemma *NFA-is-strong-equivalence-rename-fun---weaken* :
NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ f ⟹
NFA-is-equivalence-rename-fun $\mathcal{A}$ f
⟨*proof*⟩

lemma *NFA-is-strong-equivalence-rename-fun-exists* :
∃f::('a ⇒ 'a). (*NFA-is-strong-equivalence-rename-fun* $\mathcal{A}$ f ∧ (∀ q ∈ $\mathcal{Q}$ $\mathcal{A}$. f q ∈ $\mathcal{Q}$ $\mathcal{A}$))
⟨*proof*⟩

lemma *NFA-is-strong-equivalence-rename-fun---isomorph* :
fixes *f1* :: 'a ⇒ '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*⟩

lemma (**in** *NFA*) $\mathcal{L}$*-in-state-rename-subset2* :
assumes *equiv-f* : *NFA-is-equivalence-rename-fun* $\mathcal{A}$ *f*
assumes *q-in-Q* : $q \in \mathcal{Q}\ \mathcal{A}$
shows $\mathcal{L}$*-in-state* (*NFA-rename-states* $\mathcal{A}$ *f*) (*f q*) $\subseteq$ $\mathcal{L}$*-in-state* $\mathcal{A}$ *q*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *NFA-isomorphic-sym-impl* :

assumes *wf-NFA1* : *NFA* $\mathcal{A}1$
  and *equiv-*$\mathcal{A}1$*-*$\mathcal{A}2$: *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic* $\mathcal{A}2$ $\mathcal{A}1$
⟨*proof*⟩

lemma *NFA-isomorphic-sym* :
assumes *wf-NFA1*: *NFA* $\mathcal{A}1$
  and *wf-NFA2*: *NFA* $\mathcal{A}2$
shows *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$ $\longleftrightarrow$ *NFA-isomorphic* $\mathcal{A}2$ $\mathcal{A}1$
⟨*proof*⟩

lemma *NFA-isomorphic---implies-well-formed* :
assumes *wf-NFA1*: *NFA* $\mathcal{A}1$
  and *equiv-*$\mathcal{A}1$*-*$\mathcal{A}2$: *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA* $\mathcal{A}2$
⟨*proof*⟩

lemma *NFA-isomorphic-trans* :
assumes *equiv-*$\mathcal{A}1$*-*$\mathcal{A}2$: *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$
  and *equiv-*$\mathcal{A}2$*-*$\mathcal{A}3$: *NFA-isomorphic* $\mathcal{A}2$ $\mathcal{A}3$
shows *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}3$
⟨*proof*⟩

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

definition *NFA-isomorphic-wf* **where**
  NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2$ ≡ *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$ ∧ *NFA* $\mathcal{A}1$

lemma *NFA-isomorphic-wf-*$\mathcal{L}$:
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows $\mathcal{L}$ $\mathcal{A}1$ = $\mathcal{L}$ $\mathcal{A}2$
⟨*proof*⟩

lemma *NFA-isomorphic-wf-*Σ :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows Σ $\mathcal{A}1$ = Σ $\mathcal{A}2$
⟨*proof*⟩

lemma *NFA-isomorphic-wf-accept* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-accept* $\mathcal{A}1$ *w* = *NFA-accept* $\mathcal{A}2$ *w*
⟨*proof*⟩

lemma *NFA-isomorphic-wf-alt-def* :
  NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2$ $\longleftrightarrow$
   NFA-isomorphic $\mathcal{A}1$ $\mathcal{A}2$ ∧ *NFA* $\mathcal{A}1$ ∧ *NFA* $\mathcal{A}2$
⟨*proof*⟩

lemma *NFA-isomorphic-wf-sym* :
  NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2$ = *NFA-isomorphic-wf* $\mathcal{A}2$ $\mathcal{A}1$
⟨*proof*⟩

lemma *NFA-isomorphic-wf-trans* :
  ⟦*NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$; *NFA-isomorphic-wf* $\mathcal{A}2$ $\mathcal{A}3$⟧ ⟹
   NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}3$
⟨*proof*⟩

lemma *NFA-isomorphic-wf-refl* :
  NFA $\mathcal{A}1$ ⟹ *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}1$
⟨*proof*⟩

lemma *NFA-isomorphic-wf-intro* :
  ⟦*NFA* $\mathcal{A}1$; *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$⟧ ⟹ *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$

⟨*proof*⟩

lemma *NFA-isomorphic-wf-D* :
NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2 \Longrightarrow$ *NFA* $\mathcal{A}1$
NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2 \Longrightarrow$ *NFA* $\mathcal{A}2$
NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2 \Longrightarrow$ *NFA-isomorphic* $\mathcal{A}1$ $\mathcal{A}2$
NFA-isomorphic-wf $\mathcal{A}1$ $\mathcal{A}2 \Longrightarrow$ *NFA-isomorphic* $\mathcal{A}2$ $\mathcal{A}1$
⟨*proof*⟩

lemma *NFA-isomorphic---NFA-rename-states* :
inj-on f ($\mathcal{Q}$ $\mathcal{A}$) $\Longrightarrow$ *NFA-isomorphic* $\mathcal{A}$ (*NFA-rename-states* $\mathcal{A}$ *f*)
⟨*proof*⟩

lemma *NFA-isomorphic-wf---NFA-rename-states* :
⟦*inj-on f* ($\mathcal{Q}$ $\mathcal{A}$); *NFA* $\mathcal{A}$⟧ $\Longrightarrow$ *NFA-isomorphic-wf* $\mathcal{A}$ (*NFA-rename-states* $\mathcal{A}$ *f*)
⟨*proof*⟩

lemma *NFA-isomorphic-wf---rename-states-cong* :
fixes $\mathcal{A}1$ :: ('*q1*, '*a*) *NFA-rec*
fixes $\mathcal{A}2$ :: ('*q2*, '*a*) *NFA-rec*
assumes *inj-f1* : *inj-on f1* ($\mathcal{Q}$ $\mathcal{A}1$) **and**
inj-f2 : *inj-on f2* ($\mathcal{Q}$ $\mathcal{A}2$) **and**
equiv: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*NFA-rename-states* $\mathcal{A}1$ *f1*) (*NFA-rename-states* $\mathcal{A}2$ *f2*)
⟨*proof*⟩

lemma *NFA-isomorphic-wf---NFA-remove-unreachable-states* :
assumes *equiv*: *NFA-isomorphic-wf* $\mathcal{A}1$ $\mathcal{A}2$
shows *NFA-isomorphic-wf* (*NFA-remove-unreachable-states* $\mathcal{A}1$) (*NFA-remove-unreachable-states* $\mathcal{A}2$)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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* $\mathcal{A}2$
⟨*proof*⟩

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* ⇒ '*a set* ⇒ ('*q*, '*a*) *NFA-rec* **where**
NFA-initial-automaton I A ≡ (| $\mathcal{Q}$={}, Σ=*A*, Δ = {}, $\mathcal{I}$=*I*, $\mathcal{F}$ = {} |)

definition *NFA-insert-state* :: ('*q* ⇒ *bool*) ⇒ ('*q*, '*a*) *LTS* ⇒ '*q* ⇒ ('*q*, '*a*) *NFA-rec* ⇒ ('*q*, '*a*) *NFA-rec* **where**
NFA-insert-state FP D q $\mathcal{A}$ ≡
(| $\mathcal{Q}$=*insert q* ($\mathcal{Q}$ $\mathcal{A}$), Σ=Σ $\mathcal{A}$, Δ = Δ $\mathcal{A}$ ∪ {*qsq* . *qsq* ∈ *D* ∧ *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*)
⟨*proof*⟩

lemma *fold-NFA-insert-state* :
finite Q ⟹ *Finite-Set.fold* (*NFA-insert-state FP D*) $\mathcal{A}$ Q =
⦇ $\mathcal{Q}$=Q ∪ ($\mathcal{Q}$ $\mathcal{A}$), Σ=Σ $\mathcal{A}$, Δ = Δ $\mathcal{A}$ ∪ {qsq. qsq ∈ D ∧ *fst* qsq ∈ Q},
$\mathcal{I}$=$\mathcal{I}$ $\mathcal{A}$, $\mathcal{F}$ = ($\mathcal{F}$ $\mathcal{A}$) ∪ {q. q ∈ Q ∧ FP q} ⦈
⟨*proof*⟩

lemma *NFA-construct-reachable-simp* :
finite (*accessible* (*LTS-forget-labels D*) *I*) ⟹
NFA-construct-reachable I A FP D =
⦇$\mathcal{Q}$ = *accessible* (*LTS-forget-labels D*) *I*, Σ = *A*, Δ = {qsq. qsq ∈ D ∧
fst qsq ∈ *accessible* (*LTS-forget-labels D*) *I*}, $\mathcal{I}$ = *I*,
$\mathcal{F}$ = {q ∈ *accessible* (*LTS-forget-labels D*) *I*. FP q}⦈
⟨*proof*⟩

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*: ⋀q. q ∈ $\mathcal{Q}$ $\mathcal{A}$ ⟹ FP q ⟷ q ∈ $\mathcal{F}$ $\mathcal{A}$
and *D-OK*: D = Δ $\mathcal{A}$
shows
(*NFA-remove-unreachable-states* $\mathcal{A}$) = (*NFA-construct-reachable I A FP D*)
(**is** *?ls* = *?rs*)
⟨*proof*⟩

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' ⟷
DD ⊆ *dom rm'* ∧
(∀ q r'. *rm* q = *Some* r' ⟶ *rm'* q = *Some* r') ∧
inj-on rm' (*S* ∩ *dom rm'*)

lemma *NFA-construct-reachable-map-OK-I* [*intro*!] :
⟦*DD* ⊆ *dom rm'*; ⋀q r'. *rm* q = *Some* r' ⟹ *rm'* q = *Some* r'; *inj-on rm'* (*S* ∩ *dom rm'*)⟧ ⟹
NFA-construct-reachable-map-OK S rm DD rm'
⟨*proof*⟩

lemma *NFA-construct-reachable-map-OK-insert-DD* :
NFA-construct-reachable-map-OK S rm (*insert q DD*) *rm'* ⟷
q ∈ *dom rm'* ∧ *NFA-construct-reachable-map-OK S rm DD rm'*
⟨*proof*⟩

lemma *NFA-construct-reachable-map-OK-trans* :
⟦*NFA-construct-reachable-map-OK S rm DD rm'*;
NFA-construct-reachable-map-OK S rm' DD' rm'';
DD'' ⊆ *DD* ∪ *DD'*⟧ ⟹
NFA-construct-reachable-map-OK S rm DD'' rm''
⟨*proof*⟩

definition *NFA-construct-reachable-abstract-impl-invar* **where**

NFA-construct-reachable-abstract-impl-invar I A FP D ≡ (λ((*rm*, $\mathcal{A}$), *wl*).
(∃ *s*. *NFA-construct-reachable-map-OK* (*accessible* (*LTS-forget-labels D*) (*set I*)) *empty*
(*s* ∪ *set I* ∪ *set wl* ∪ {*q'*. ∃ *a q*. *q*∈*s* ∧ (*q*,*a*,*q'*)∈ *D*}) *rm* ∧
(*accessible* (*LTS-forget-labels D*) (*set I*) =
accessible-restrict (*LTS-forget-labels D*) *s* (*set wl*)) ∧
($\mathcal{A}$ = *NFA-rename-states*
(⦇$\mathcal{Q}$ = *s*, Σ = *A*, Δ = {*qsq*. *qsq* ∈ *D* ∧ *fst qsq* ∈ *s*}, $\mathcal{I}$ = *set I*,
$\mathcal{F}$ = {*q* ∈ *s*. *FP q*}⦈) (*the* ∘ *rm*))))

definition *NFA-construct-reachable-abstract-impl-weak-invar* **where**
NFA-construct-reachable-abstract-impl-weak-invar I A FP D ≡ (λ(*rm*, $\mathcal{A}$).
(∃ *s*. *NFA-construct-reachable-map-OK* (*accessible* (*LTS-forget-labels D*) (*set I*)) *empty*
(*s* ∪ *set I* ∪ {*q'*. ∃ *a q*. *q*∈*s* ∧ (*q*,*a*,*q'*)∈ *D*}) *rm* ∧
s ⊆ *accessible* (*LTS-forget-labels D*) (*set I*) ∧
($\mathcal{A}$ = *NFA-rename-states*
(⦇$\mathcal{Q}$ = *s*, Σ = *A*, Δ = {*qsq*. *qsq* ∈ *D* ∧ *fst qsq* ∈ *s*}, $\mathcal{I}$ = *set I*,
$\mathcal{F}$ = {*q* ∈ *s*. *FP q*}⦈) (*the* ∘ *rm*))))

lemma *NFA-construct-reachable-abstract-impl-invar-weaken* :
assumes *invar*: *NFA-construct-reachable-abstract-impl-invar I A FP D* ((*rm*, $\mathcal{A}$), *wl*)
shows *NFA-construct-reachable-abstract-impl-weak-invar I A FP D* (*rm*, $\mathcal{A}$)
⟨*proof*⟩

fun *NFA-construct-reachable-abstract-impl-foreach-invar* **where**
NFA-construct-reachable-abstract-impl-foreach-invar S D rm D0 q it (*rm'*, *D'*, *N*) =
(*let D''* = {(*a*, *q'*) . (*q*, *a*, *q'*) ∈ *D*} − *it*; *qS* = *snd* ‘ *D''* *in*
(*NFA-construct-reachable-map-OK S rm qS rm'* ∧
set N = *qS* ∧
D' = *D0* ∪ {(*the* (*rm'* *q*), *a*, *the* (*rm'* *q'*)) | *a q'*. (*a*, *q'*) ∈ *D''*}))
declare *NFA-construct-reachable-abstract-impl-foreach-invar.simps* [*simp del*]

definition *NFA-construct-reachable-abstract-impl-step* **where**
NFA-construct-reachable-abstract-impl-step S D rm D0 q =
FOREACHi
(*NFA-construct-reachable-abstract-impl-foreach-invar S D rm D0 q*)
{(*a*, *q'*) . (*q*, *a*, *q'*) ∈ *D*}
(λ(*a*, *q'*) (*rm*, *D'*, *N*). *do* {
(*rm'*, *r'*) ← *SPEC* (λ(*rm'*, *r'*). *NFA-construct-reachable-map-OK S rm* {*q'*} *rm'* ∧ *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'*) ∈ *D*}
and *inj-rm*: *inj-on rm* (*S* ∩ *dom rm*)
and *q-in-dom*: *q* ∈ *dom rm*
shows *NFA-construct-reachable-abstract-impl-step S D rm D0 q* ≤
SPEC (*NFA-construct-reachable-abstract-impl-foreach-invar S D rm D0 q* {})
⟨*proof*⟩

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*, $\mathcal{A}$), -) ← *WORKLISTIT* (*NFA-construct-reachable-abstract-impl-invar I A FP D*)
(λ-. *True*)
(λ(*rm*, $\mathcal{A}$) *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*, $\mathcal{A}$));
if (*the* (*rm q*) ∈ $\mathcal{Q}$ $\mathcal{A}$) *then*
(*RETURN* ((*rm*, $\mathcal{A}$), []))

```
      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))
⟨proof⟩


definition NFA-construct-reachable-abstract2-impl where
  NFA-construct-reachable-abstract2-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) ← NFA-construct-reachable-abstract-impl-step
              (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-abstract2-impl-correct :
fixes D :: ('q × 'a × 'q) set and I
defines S ≡ accessible (LTS-forget-labels D) (set I)
assumes fin-S: finite S
   and fin-D: ⋀q. finite {(a, q'). (q, a, q') ∈ D}
shows NFA-construct-reachable-abstract2-impl I A FP D ≤ ⇓Id ((NFA-construct-reachable-abstract-impl I A FP
D)::('q2, 'a) NFA-rec nres)
⟨proof⟩
```

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

```
definition NFA-rename-labels :: ('q, 'a, 'x) NFA-rec-scheme ⇒ ('a ⇒ 'b) ⇒
                        ('q, 'b) NFA-rec where
NFA-rename-labels A f ≡ (| Q = Q A,
                    Σ = f ' (Σ A),
                    Δ = { (p, f σ, q) | p σ q. (p,σ,q) ∈ Δ A},
                    I = I A,
                    F = F A |)
```

lemma [*simp*] : $\mathcal{Q}$ (*NFA-rename-labels* $\mathcal{A}$ f) = $\mathcal{Q}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Σ (*NFA-rename-labels* $\mathcal{A}$ f) = f ‘ (Σ $\mathcal{A}$) ⟨*proof*⟩
lemma [*simp*] : (p, $f\sigma$, q) ∈ Δ (*NFA-rename-labels* $\mathcal{A}$ f) ⟷
($\exists$ σ. (p, σ, q) ∈ Δ $\mathcal{A}$ ∧ ($f\sigma$ = f σ))
⟨*proof*⟩
lemma [*simp*] : $\mathcal{I}$ (*NFA-rename-labels* $\mathcal{A}$ f) = $\mathcal{I}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{F}$ (*NFA-rename-labels* $\mathcal{A}$ f) = $\mathcal{F}$ $\mathcal{A}$ ⟨*proof*⟩

lemma (**in** *NFA*) *NFA-rename-labels---is-well-formed* :
NFA (*NFA-rename-labels* $\mathcal{A}$ f)
⟨*proof*⟩

lemma *lists---NFA-rename-labels* :
w ∈ *lists* (Σ $\mathcal{A}$) ⟹ (*map* f w) ∈ *lists* (Σ (*NFA-rename-labels* $\mathcal{A}$ f))
⟨*proof*⟩

lemma *NFA-rename-labels-id* [*simp*] : *NFA-rename-labels* $\mathcal{A}$ *id* = $\mathcal{A}$
⟨*proof*⟩

lemma *LTS-is-reachable---NFA-rename-labels* :
LTS-is-reachable (Δ $\mathcal{A}$) q w q' ⟹
LTS-is-reachable (Δ (*NFA-rename-labels* $\mathcal{A}$ f)) q (*map* f w) q'
⟨*proof*⟩

lemma *LTS-is-reachable---NFA-rename-labelsE* :
LTS-is-reachable (Δ (*NFA-rename-labels* $\mathcal{A}$ f)) q w q' ⟹
$\exists$ w'. w = (*map* f w') ∧ *LTS-is-reachable* (Δ $\mathcal{A}$) q w' q'
⟨*proof*⟩

lemma *NFA-accept---NFA-rename-labels* :
assumes *NFA-accept* $\mathcal{A}$ w
shows *NFA-accept* (*NFA-rename-labels* $\mathcal{A}$ f) (*map* f w)
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-accept---NFA-rename-labelsE* :
NFA-accept (*NFA-rename-labels* $\mathcal{A}$ f) w ⟹ $\exists$ w'. w = (*map* f w') ∧ *NFA-accept* $\mathcal{A}$ w'
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-accept---NFA-rename-labels-iff* :
NFA-accept (*NFA-rename-labels* $\mathcal{A}$ f) w ⟷ ($\exists$ w'. w = (*map* f w') ∧ *NFA-accept* $\mathcal{A}$ w')
⟨*proof*⟩

lemma (**in** *NFA*) $\mathcal{L}$*-NFA-rename-labels* [*simp*] :
$\mathcal{L}$ (*NFA-rename-labels* $\mathcal{A}$ f) = (*map* f) ‘ $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-isomorphic-wf---NFA-rename-labels-cong* :
assumes *equiv*: *NFA-isomorphic-wf* *A1* *A2*
shows *NFA-isomorphic-wf* (*NFA-rename-labels* *A1* *fl*) (*NFA-rename-labels* *A2* *fl*)
⟨*proof*⟩

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* ⇒ ('*q2*::{*NFA-states*}, '*a*) *NFA-rec* **where**
NFA-normalise-states $\mathcal{A}$ = (*NFA-rename-states* $\mathcal{A}$ (*SOME* (f::'q ⇒ '*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 ∃*f*::('*a* ⇒ '*b*). *inj-on f A*
⟨*proof*⟩

lemma *NFA-isomorphic-wf-normalise-states* :
fixes 𝒜:: ('*q*, '*a*, -) *NFA-rec-scheme*
assumes *wf-A*: *NFA* 𝒜
shows *NFA-isomorphic-wf* 𝒜 ((*NFA-normalise-states* 𝒜)::('*q2*::{*NFA-states*},'*a*) *NFA-rec*)
⟨*proof*⟩

lemma *NFA-isomorphic-wf---NFA-normalise-states* :
NFA-isomorphic-wf 𝒜*1* 𝒜*2* ⟹ *NFA-isomorphic-wf* 𝒜*1* (*NFA-normalise-states* 𝒜*2*)
⟨*proof*⟩

lemma *NFA-isomorphic-wf---NFA-normalise-states-cong* :
fixes 𝒜*1*::('*q1*, '*a*) *NFA-rec*
  and 𝒜*2*::('*q2*, '*a*) *NFA-rec*
shows *NFA-isomorphic-wf* 𝒜*1* 𝒜*2* ⟹
 NFA-isomorphic-wf (*NFA-normalise-states* 𝒜*1*) (*NFA-normalise-states* 𝒜*2*)
⟨*proof*⟩

lemma *NFA-normalise-states-*Σ [*simp*] :
Σ (*NFA-normalise-states* 𝒜) = Σ 𝒜
⟨*proof*⟩

lemma *NFA-normalise-states-*ℒ [*simp*] :
NFA 𝒜 ⟹ ℒ (*NFA-normalise-states* 𝒜) = ℒ 𝒜
⟨*proof*⟩

lemma *NFA-normalise-states-accept* [*simp*] :
NFA 𝒜 ⟹ *NFA-accept* (*NFA-normalise-states* 𝒜) *w* = *NFA-accept* 𝒜 *w*
⟨*proof*⟩

lemma *NFA-is-initially-connected---normalise-states* :
assumes *connected*: *NFA-is-initially-connected* 𝒜
shows *NFA-is-initially-connected* (*NFA-normalise-states* 𝒜)
⟨*proof*⟩

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* ⇒ *bool* ⇒ *bool*) ⇒ ('*q1*, '*a*, '*x1*) *NFA-rec-scheme* ⇒
  ('*q2*, '*a*, '*x2*) *NFA-rec-scheme* ⇒ ('*q1* × '*q2*, '*a*) *NFA-rec* **where**
bool-comb-NFA bc 𝒜*1* 𝒜*2* == (|
  𝒬=𝒬 𝒜*1* × 𝒬 𝒜*2*,
  Σ=Σ 𝒜*1* ∩ Σ 𝒜*2*,
  Δ = *LTS-product* (Δ 𝒜*1*) (Δ 𝒜*2*),
  ℐ = ℐ 𝒜*1* × ℐ 𝒜*2*,
  ℱ = {*q*. *q* ∈ 𝒬 𝒜*1* × 𝒬 𝒜*2* ∧ *bc* (*fst q* ∈ ℱ 𝒜*1*) (*snd q* ∈ ℱ 𝒜*2*)}|)

lemma [*simp*] : ℐ (*bool-comb-NFA bc* 𝒜*1* 𝒜*2*) = ℐ 𝒜*1* × ℐ 𝒜*2* ⟨*proof*⟩
lemma [*simp*] : 𝒬 (*bool-comb-NFA bc* 𝒜*1* 𝒜*2*) = 𝒬 𝒜*1* × 𝒬 𝒜*2* ⟨*proof*⟩
lemma [*simp*] : ℱ (*bool-comb-NFA bc* 𝒜*1* 𝒜*2*) = {*q*. *q* ∈ 𝒬 𝒜*1* × 𝒬 𝒜*2* ∧ *bc* (*fst q* ∈ ℱ 𝒜*1*) (*snd q* ∈ ℱ 𝒜*2*)}
⟨*proof*⟩
lemma [*simp*] : Σ (*bool-comb-NFA bc* 𝒜*1* 𝒜*2*) = Σ 𝒜*1* ∩ Σ 𝒜*2* ⟨*proof*⟩
lemma [*simp*] : Δ (*bool-comb-NFA bc* 𝒜*1* 𝒜*2*) = *LTS-product* (Δ 𝒜*1*) (Δ 𝒜*2*) ⟨*proof*⟩

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))
⟨*proof*⟩

lemma $\mathcal{L}$*-product-NFA* :
⟦*NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$⟧ $\Longrightarrow$ $\mathcal{L}$ (*product-NFA* $\mathcal{A}1$ $\mathcal{A}2$) = $\mathcal{L}$ $\mathcal{A}1$ $\cap$ $\mathcal{L}$ $\mathcal{A}2$
⟨*proof*⟩

lemma *bool-comb-NFA---is-well-formed* :
⟦ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$⟧ $\Longrightarrow$ *NFA* (*bool-comb-NFA* *bc* $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

lemma *product-NFA---is-well-formed* :
⟦ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$ ⟧ $\Longrightarrow$ *NFA* (*product-NFA* $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

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* $op\wedge$ $\mathcal{A}1$ $\mathcal{A}2$
⟨*proof*⟩

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 $\mathcal{A}2$
⟨*proof*⟩

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)
⟨*proof*⟩

lemma $\mathcal{L}$*-efficient-product-NFA* :
⟦ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$⟧ $\Longrightarrow$ $\mathcal{L}$ (*efficient-product-NFA* $\mathcal{A}1$ $\mathcal{A}2$) = $\mathcal{L}$ $\mathcal{A}1$ $\cap$ $\mathcal{L}$ $\mathcal{A}2$
⟨*proof*⟩

lemma *efficient-bool-comb-NFA---is-well-formed* :
⟦ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$ ⟧ $\Longrightarrow$ *NFA* (*efficient-bool-comb-NFA* *bc* $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

lemma *efficient-product-NFA---is-well-formed* :
⟦ *NFA* $\mathcal{A}1$; *NFA* $\mathcal{A}2$ ⟧ $\Longrightarrow$ *NFA* (*efficient-product-NFA* $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

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* ($\mathcal{I}$ $\mathcal{A}1$ × $\mathcal{I}$ $\mathcal{A}2$) (Σ $\mathcal{A}1$ $\cap$ Σ $\mathcal{A}2$)
($\lambda q.$ q $\in$ $\mathcal{Q}$ $\mathcal{A}1$ × $\mathcal{Q}$ $\mathcal{A}2$ $\wedge$ *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*⟩

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* ($\mathcal{I}$ *A1* × $\mathcal{I}$ *A2*) (Σ *A1* ∩ Σ *A2*)
(λ*q*. *q* ∈ $\mathcal{Q}$ *A1* × $\mathcal{Q}$ *A2* ∧ *q* ∈ ($\mathcal{F}$ *A1* × $\mathcal{F}$ *A2*))
(*LTS-product* (Δ *A1*) (Δ *A2*)))
⟨*proof*⟩

lemma *NFA-isomorphic-bool-comb-NFA* :
assumes *equiv-1*: *NFA-isomorphic-wf A1 A1'*
and *equiv-2*: *NFA-isomorphic-wf A2 A2'*
shows *NFA-isomorphic-wf* (*bool-comb-NFA bc A1 A2*) (*bool-comb-NFA bc A1' A2'*)
⟨*proof*⟩

lemma *NFA-isomorphic-efficient-bool-comb-NFA* :
assumes *equiv-1*: *NFA-isomorphic-wf A1 A1'*
and *equiv-2*: *NFA-isomorphic-wf A2 A2'*
shows *NFA-isomorphic-wf* (*efficient-bool-comb-NFA bc A1 A2*) (*efficient-bool-comb-NFA bc A1' A2'*)
⟨*proof*⟩

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 A1 A2 = *NFA-normalise-states* (*efficient-bool-comb-NFA bc A1 A2*)

lemma *NFA-bool-comb---isomorphic-wf* :
NFA A1 ⟹ *NFA A2* ⟹
NFA-isomorphic-wf (*efficient-bool-comb-NFA bc A1 A2*)
(*NFA-bool-comb bc A1 A2*)
⟨*proof*⟩

lemma *NFA-isomorphic-efficient-NFA-bool-comb* :
assumes *equiv-1*: *NFA-isomorphic-wf A1 A1'*
and *equiv-2*: *NFA-isomorphic-wf A2 A2'*
shows *NFA-isomorphic-wf* (*NFA-bool-comb bc A1 A2*) (*NFA-bool-comb bc A1' A2'*)
⟨*proof*⟩

lemma *NFA-bool-comb-NFA*---Σ [*simp*] :
Σ (*NFA-bool-comb bc A1 A2*) = Σ *A1* ∩ Σ *A2*
⟨*proof*⟩

2.10 Reversal

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

lemma [*simp*] : $\mathcal{Q}$ (*NFA-reverse A*) = $\mathcal{Q}$ *A* ⟨*proof*⟩
lemma [*simp*] : Σ (*NFA-reverse A*) = Σ *A* ⟨*proof*⟩
lemma [*simp*] : $\mathcal{I}$ (*NFA-reverse A*) = $\mathcal{F}$ *A* ⟨*proof*⟩
lemma [*simp*] : $\mathcal{F}$ (*NFA-reverse A*) = $\mathcal{I}$ *A* ⟨*proof*⟩

lemma [*simp*] : *p*σ*q* ∈ Δ (*NFA-reverse A*) ⟷ ((*snd* (*snd p*σ*q*)), (*fst* (*snd p*σ*q*)), (*fst p*σ*q*)) ∈ Δ *A*
⟨*proof*⟩

lemma *NFA-reverse---is-well-formed* :
NFA A ⟹ *NFA* (*NFA-reverse A*)
⟨*proof*⟩

lemma *NFA-reverse-NFA-reverse* :
NFA-reverse (*NFA-reverse A*) = *A*
⟨*proof*⟩

lemma *NFA-reverse---LTS-is-reachable* :
LTS-is-reachable (Δ (*NFA-reverse A*)) *p w q* ⟷ *LTS-is-reachable* (Δ *A*) *q* (*rev w*) *p*
⟨*proof*⟩

lemma *NFA-reverse---accept* :
NFA-accept (*NFA-reverse* $\mathcal{A}$) $w \longleftrightarrow$ *NFA-accept* $\mathcal{A}$ (*rev* w)
⟨*proof*⟩

lemma *NFA-reverse---$\mathcal{L}$* :
$\mathcal{L}$ (*NFA-reverse* $\mathcal{A}$) = {*rev* w | w. $w \in \mathcal{L}$ $\mathcal{A}$}
⟨*proof*⟩

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}
⟨*proof*⟩

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}
⟨*proof*⟩

lemma *unreachable-states-NFA-reverse-def* :
NFA-unreachable-states $\mathcal{A}$ = {q. $\mathcal{L}$-*in-state* (*NFA-reverse* $\mathcal{A}$) q = {}}
⟨*proof*⟩

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*⟩

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. $q \in \mathcal{Q}$ $\mathcal{A}$ ∧ $\mathcal{L}$-*in-state* $\mathcal{A}$ q ∩ $L \neq$ {}} |)

lemma [*simp*] : $\mathcal{Q}$ (*NFA-right-quotient* $\mathcal{A}$ L) = $\mathcal{Q}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Σ (*NFA-right-quotient* $\mathcal{A}$ L) = Σ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{I}$ (*NFA-right-quotient* $\mathcal{A}$ L) = $\mathcal{I}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Δ (*NFA-right-quotient* $\mathcal{A}$ L) = Δ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{F}$ (*NFA-right-quotient* $\mathcal{A}$ L) = {q. $q \in \mathcal{Q}$ $\mathcal{A}$ ∧ $\mathcal{L}$-*in-state* $\mathcal{A}$ q ∩ $L \neq$ {}} ⟨*proof*⟩

lemma *NFA-right-quotient---is-well-formed* :
NFA $\mathcal{A}$ ⟹ *NFA* (*NFA-right-quotient* $\mathcal{A}$ L)
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-right-quotient---accepts* :
NFA-accept (*NFA-right-quotient* $\mathcal{A}$ L) $w \longleftrightarrow$
($\exists$ *w2* ∈ L. *NFA-accept* $\mathcal{A}$ (w @ *w2*))
⟨*proof*⟩

lemma *NFA-right-quotient---alt-def* :
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' . $\exists$ q w. $q' \in \mathcal{Q}$ $\mathcal{A}$ ∧ $q \in \mathcal{F}$ $\mathcal{A}$ ∧ $w \in$ *rev* ‘ L ∩ *lists* (Σ $\mathcal{A}$) ∧
LTS-is-reachable {(q, σ, p) . (p, σ, q) ∈ Δ $\mathcal{A}$} q w q'} |)
⟨*proof*⟩

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*⟩

lemma *NFA-right-quotient---restrict-L*:
NFA-right-quotient $\mathcal{A}$ (*L* ∩ *lists* (Σ $\mathcal{A}$)) = *NFA-right-quotient* $\mathcal{A}$ *L*
⟨*proof*⟩

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* ∩ Σ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *NFA-right-quotient-lists-alt-def* :
shows *NFA-right-quotient-lists* $\mathcal{A}$ *Q* =
⦇ $\mathcal{Q}$ = $\mathcal{Q}$ $\mathcal{A}$,
Σ = Σ $\mathcal{A}$,
Δ = Δ $\mathcal{A}$,
$\mathcal{I}$ = $\mathcal{I}$ $\mathcal{A}$,
$\mathcal{F}$ = *accessible* {(*q*,*p*) . ∃σ. (*p*, σ, *q*) ∈ Δ $\mathcal{A}$ ∧ σ ∈ *Q*} ($\mathcal{F}$ $\mathcal{A}$) ⦈
⟨*proof*⟩

end

3 Deterministic Finite Automata

theory *DFA*
imports *Main LTS NFA*
begin

3.1 Basic Definitions

definition *NFA-is-deterministic* **where**
NFA-is-deterministic $\mathcal{A}$ ≡
(*LTS-is-deterministic* ($\mathcal{Q}$ $\mathcal{A}$) (Σ $\mathcal{A}$) (Δ $\mathcal{A}$) ∧ (∃ *q0*. $\mathcal{I}$ $\mathcal{A}$ = {*q0*}))

lemma *dummy-NFA---NFA-is-deterministic* :
NFA-is-deterministic (*dummy-NFA q a*)
⟨*proof*⟩

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* $\mathcal{A}$ ⟨*proof*⟩
lemma (**in** *DFA*) *wf-DFA* : *DFA* $\mathcal{A}$ ⟨*proof*⟩

lemma *DFA-alt-def* :
DFA $\mathcal{A}$ ≡ *NFA-is-deterministic* $\mathcal{A}$ ∧ *NFA* $\mathcal{A}$
⟨*proof*⟩

lemma *DFA-implies-NFA*[*simp*] :
DFA $\mathcal{A}$ ⟹ *NFA* $\mathcal{A}$ ⟨*proof*⟩

lemma *dummy-NFA---is-DFA* :

DFA (*dummy-NFA* q a)
⟨*proof*⟩

lemma (**in** *detNFA*) *deterministic* : *LTS-is-deterministic* ($\mathcal{Q}$ $\mathcal{A}$) (Σ $\mathcal{A}$) (Δ $\mathcal{A}$) $\wedge$ ($\exists$ $q0$. $\mathcal{I}$ $\mathcal{A}$ = {$q0$})
⟨*proof*⟩

lemma (**in** *detNFA*) *weak-deterministic* : *LTS-is-weak-deterministic* (Δ $\mathcal{A}$)
⟨*proof*⟩

3.2 The unique initial state

lemma (**in** *detNFA*) *unique-initial* : $\exists!$ x. $x \in \mathcal{I}$ $\mathcal{A}$
⟨*proof*⟩

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*⟩

lemma (**in** *DFA*) i-*is-state* : i $\mathcal{A} \in \mathcal{Q}$ $\mathcal{A}$ ⟨*proof*⟩
lemma (**in** *DFA*) $\mathcal{Q}$-*not-Emp*: $\mathcal{Q}$ $\mathcal{A} \neq$ {} ⟨*proof*⟩

lemma (**in** *detNFA*) $\mathcal{L}$-*in-state*-i : $\mathcal{L}$-*in-state* $\mathcal{A}$ (i $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$ ⟨*proof*⟩

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 $\mathcal{A}$
⟨*proof*⟩

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

lemma (**in** *DFA*) δ-*wf* :
assumes δ $\mathcal{A}$ (q,σ) = *Some* q'
shows $q \in \mathcal{Q}$ $\mathcal{A} \wedge$ ($\sigma \in \Sigma$ $\mathcal{A}$) $\wedge$ ($q' \in \mathcal{Q}$ $\mathcal{A}$)
⟨*proof*⟩

3.4 Lemmas about deterministic automata

lemma (**in** *DFA*) *DFA-LTS-is-reachable-DLTS-reach-simp* :
LTS-is-reachable (Δ $\mathcal{A}$) q w $q' \longleftrightarrow$ (*DLTS-reach* (δ $\mathcal{A}$) q w = *Some* q')
⟨*proof*⟩

lemma (**in** *DFA*) *DFA-*δ*-is-none-iff* :
(δ $\mathcal{A}$ (q, σ) = *None*) $\longleftrightarrow \neg$ ($q \in \mathcal{Q}$ $\mathcal{A} \wedge \sigma \in \Sigma$ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *DFA*) *DFA-*δ*-is-some* :
⟦$q \in \mathcal{Q}$ $\mathcal{A}$; $\sigma \in \Sigma$ $\mathcal{A}$⟧ $\Longrightarrow \neg$ (δ $\mathcal{A}$ (q, σ) = *None*)
⟨*proof*⟩

lemma (**in** *DFA*) *DFA-reach-is-none-iff* :
DLTS-reach (δ $\mathcal{A}$) q w = *None* $\longleftrightarrow \neg$(($w \neq$ [] −−> $q \in \mathcal{Q}$ $\mathcal{A}$) $\wedge$ $w \in$ *lists* (Σ $\mathcal{A}$))
⟨*proof*⟩

lemma (**in** *DFA*) *DFA-DLTS-reach-is-some* :
⟦$q \in \mathcal{Q}$ $\mathcal{A}$; $w \in$ *lists* (Σ $\mathcal{A}$)⟧ $\Longrightarrow \neg$ (*DLTS-reach* (δ $\mathcal{A}$) q w = *None*)
⟨*proof*⟩

lemma (**in** *DFA*) *DFA-DLTS-reach-wf* : ⟦ $q \in \mathcal{Q}\ \mathcal{A}$; *DLTS-reach* ($\delta\ \mathcal{A}$) $q\ w$ = *Some* q'⟧ ⟹ $q' \in \mathcal{Q}\ \mathcal{A} \wedge$ $w \in$ *lists* ($\Sigma\ \mathcal{A}$)
⟨*proof*⟩

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*⟩

lemma (**in** *DFA*) *DFA-accept-alt-def* :
 NFA-accept $\mathcal{A}\ w =$
  (*case* (*DLTS-reach* ($\delta\ \mathcal{A}$) (i $\mathcal{A}$) w) *of None* ⇒ *False* | *Some* q' ⇒ $q' \in \mathcal{F}\ \mathcal{A}$)
⟨*proof*⟩

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, $q1'$) ∈ $\Delta\ \mathcal{A}$
  and *in-D2*: ($q2$, σ, $q2'$) ∈ $\Delta\ \mathcal{A}'$
  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*⟩

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* ($\Delta\ \mathcal{A}$) $q1\ w\ q1'$
  and *DLTS-reach-2*: *LTS-is-reachable* ($\Delta\ \mathcal{A}'$) $q2\ w\ q2'$
  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* $\mathcal{A}'\ q2'$
⟨*proof*⟩

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*⟩

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* $\mathcal{A}\ f$)
⟨*proof*⟩

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*⟩

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* $\mathcal{A}2$
⟨*proof*⟩

lemma *NFA-normalise-states-DFA* [*simp*] :
assumes *wf-A*: *DFA* $\mathcal{A}$
shows *DFA* (*NFA-normalise-states* $\mathcal{A}$)
⟨*proof*⟩

3.5 Determinisation

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

lemma [*simp*] : $\mathcal{I}$ (*determinise-NFA* $\mathcal{A}$) = {$\mathcal{I}\ \mathcal{A}$} ⟨*proof*⟩
lemma [*simp*] : ($q \in \mathcal{Q}$ (*determinise-NFA* $\mathcal{A}$)) ⟷ $q \subseteq \mathcal{Q}\ \mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Σ (*determinise-NFA* $\mathcal{A}$) = $\Sigma\ \mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $q \in \mathcal{F}$ (*determinise-NFA* $\mathcal{A}$) ⟷ ($q \subseteq (\mathcal{Q}\ \mathcal{A})$) ∧ $q \cap \mathcal{F}\ \mathcal{A} \neq \{\}$ ⟨*proof*⟩

lemma [*simp*] : $Q\sigma Q' \in \Delta$ (*determinise-NFA* $\mathcal{A}$) ⟷
((snd (snd $Q\sigma Q'$) = {$q'.\ \exists\, q{\in}(fst\ Q\sigma Q').\ (q, fst\ (snd\ Q\sigma Q'), q') \in \Delta\ \mathcal{A}$}) ∧
(fst $Q\sigma Q' \subseteq \mathcal{Q}\ \mathcal{A}$) ∧ (($fst$ (snd $Q\sigma Q'$)) $\in \Sigma\ \mathcal{A}$))
⟨*proof*⟩

lemma *determinise-NFA-is-detNFA* :
NFA-is-deterministic (*determinise-NFA* $\mathcal{A}$)
⟨*proof*⟩

interpretation *det*: *detNFA determinise-NFA* $\mathcal{A}$ ⟨*proof*⟩

lemma *determinised*-i [*simp*] : i (*determinise-NFA* $\mathcal{A}$) = $\mathcal{I}\ \mathcal{A}$ ⟨*proof*⟩
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* {$q'$ |q q'. $q \in Q \wedge (q, \sigma, q') \in \Delta\ \mathcal{A}$} *else None*)
⟨*proof*⟩

lemma *determinise-NFA---is-well-formed* :
NFA $\mathcal{A}$ ⟹ *NFA* (*determinise-NFA* $\mathcal{A}$)
⟨*proof*⟩

lemma *determinise-NFA---DFA* :
NFA $\mathcal{A}$ ⟹ *DFA* (*determinise-NFA* $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *determinise-NFA---DLTS-reach* :
shows $Q \subseteq \mathcal{Q}\ \mathcal{A}$ ⟹
DLTS-reach (δ (*determinise-NFA* $\mathcal{A}$)) Q w =
(*if* ($w \in lists\ (\Sigma\ \mathcal{A})$) *then Some* {$q'$. $\exists\, q{\in}Q$. *LTS-is-reachable* ($\Delta\ \mathcal{A}$) q w q'} *else None*)
⟨*proof*⟩

lemma (**in** *NFA*) *determinise-NFA---$\mathcal{L}$-in-state* :
assumes *Q-subset*: $Q \subseteq \mathcal{Q}\ \mathcal{A}$
shows $\mathcal{L}$-*in-state* (*determinise-NFA* $\mathcal{A}$) $Q = \bigcup$ {$\mathcal{L}$-*in-state* $\mathcal{A}$ q | $q.\ q \in Q$}
(**is** *?s1* = *?s2*)
⟨*proof*⟩

lemma (**in** *NFA*) *determinise-NFA-$\mathcal{L}$* [*simp*] :
$\mathcal{L}$ (*determinise-NFA* $\mathcal{A}$) = $\mathcal{L}\ \mathcal{A}$
⟨*proof*⟩

lemma (**in** *NFA*) *determinise-NFA-accept* [*simp*] :
NFA-accept (*determinise-NFA* $\mathcal{A}$) w = *NFA-accept* $\mathcal{A}$ w
⟨*proof*⟩

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*⟩

lemma *DFA---NFA-remove-unreachable-states* :
DFA $\mathcal{A} \Longrightarrow$ *DFA* (*NFA-remove-unreachable-states* $\mathcal{A}$)
⟨*proof*⟩

lemma *efficient-determinise-NFA-is-detNFA* :
NFA-is-deterministic (*efficient-determinise-NFA* $\mathcal{A}$)
⟨*proof*⟩

lemma *efficient-determinise-NFA-is-DFA* :
NFA $\mathcal{A} \Longrightarrow$ *DFA* (*efficient-determinise-NFA* $\mathcal{A}$)
⟨*proof*⟩

lemma *efficient-determinise-NFA-*Σ [*simp*] :
Σ (*efficient-determinise-NFA* $\mathcal{A}$) = Σ $\mathcal{A}$
⟨*proof*⟩

lemma (**in** *NFA*) *efficient-determinise-NFA-accept* [*simp*] :
NFA-accept (*efficient-determinise-NFA* $\mathcal{A}$) w = *NFA-accept* $\mathcal{A}$ w
⟨*proof*⟩

lemma (**in** *NFA*) *efficient-determinise-NFA-*$\mathcal{L}$ [*simp*] :
$\mathcal{L}$ (*efficient-determinise-NFA* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

lemma *efficient-determinise-NFA---is-initially-connected* :
NFA-is-initially-connected (*efficient-determinise-NFA* $\mathcal{A}$)
⟨*proof*⟩

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, σ, {q'. $\exists$ $q{\in}Q$. (q, σ, q') $\in \Delta$ $\mathcal{A}$}) | Q σ. $Q \subseteq \mathcal{Q}$ $\mathcal{A} \wedge \sigma \in \Sigma$ $\mathcal{A}$})
⟨*proof*⟩

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*⟩

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* $\mathcal{A}2$)
⟨*proof*⟩

definition *NFA-determinise* :: ('q::{*NFA-states*}, 'a, -) *NFA-rec-scheme* ⇒ ('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* $\mathcal{A}$)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *NFA-determinise-is-DFA* :
assumes *wf-A* : *NFA* $\mathcal{A}$
shows *DFA* (*NFA-determinise* $\mathcal{A}$)
⟨*proof*⟩

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*⟩

lemma *NFA-determinise-*$\mathcal{L}$:
NFA $\mathcal{A} \Longrightarrow \mathcal{L}$ (*NFA-determinise* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

3.6 Right quotient

lemma *NFA-right-quotient---is-well-formed-DFA* :
DFA $\mathcal{A} \Longrightarrow$ *DFA* (*NFA-right-quotient* $\mathcal{A}$ *L*)
⟨*proof*⟩

3.7 Complement

definition *DFA-complement* :: ('*q*, '*a*, '*x*) *NFA-rec-scheme* ⇒ ('*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* $\mathcal{A}$) = $\mathcal{Q}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Σ (*DFA-complement* $\mathcal{A}$) = Σ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : Δ (*DFA-complement* $\mathcal{A}$) = Δ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{I}$ (*DFA-complement* $\mathcal{A}$) = $\mathcal{I}$ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : $\mathcal{F}$ (*DFA-complement* $\mathcal{A}$) = $\mathcal{Q}$ $\mathcal{A}$ − $\mathcal{F}$ $\mathcal{A}$ ⟨*proof*⟩

lemma [*simp*] : δ (*DFA-complement* $\mathcal{A}$) = δ $\mathcal{A}$ ⟨*proof*⟩
lemma [*simp*] : i (*DFA-complement* $\mathcal{A}$) = i $\mathcal{A}$ ⟨*proof*⟩

lemma *DFA-complement---is-well-formed* : *NFA* $\mathcal{A} \Longrightarrow$ *NFA* (*DFA-complement* $\mathcal{A}$)
⟨*proof*⟩

lemma *DFA-complement-of-DFA-is-DFA* :
DFA $\mathcal{A} \Longrightarrow$ *DFA* (*DFA-complement* $\mathcal{A}$)
⟨*proof*⟩

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

lemma *DFA-complement-DFA-complement* [*simp*] :
NFA $\mathcal{A} \Longrightarrow$ *DFA-complement* (*DFA-complement* $\mathcal{A}$) = $\mathcal{A}$
⟨*proof*⟩

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*⟩

lemma *DFA-complement-*$\mathcal{L}$*---lists-*Σ :
$\mathcal{L}$ (*DFA-complement* $\mathcal{A}$) ⊆ *lists* (Σ $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *DFA*) *DFA-complement-*$\mathcal{L}$ [*simp*] :
shows $\mathcal{L}$ (*DFA-complement* $\mathcal{A}$) = *lists* (Σ $\mathcal{A}$) − $\mathcal{L}$ $\mathcal{A}$

⟨*proof*⟩

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*⟩

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*⟩

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 $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

lemma *bool-comb-DFA-NFA-accept* :
assumes *DFA-*$\mathcal{A}1$: *DFA* $\mathcal{A}1$
  and *DFA-*$\mathcal{A}2$: *DFA* $\mathcal{A}2$
shows *NFA-accept* (*bool-comb-NFA* f $\mathcal{A}1$ $\mathcal{A}2$) w =
    ($w \in$ *lists* (Σ $\mathcal{A}1$) $\cap$ *lists* (Σ $\mathcal{A}2$) $\wedge$ f (*NFA-accept* $\mathcal{A}1$ w) (*NFA-accept* $\mathcal{A}2$ w))
⟨*proof*⟩

lemma *efficient-bool-comb-DFA-accept* :
assumes *DFA-*$\mathcal{A}1$: *DFA* $\mathcal{A}1$
  and *DFA-*$\mathcal{A}2$: *DFA* $\mathcal{A}2$
shows *NFA-accept* (*efficient-bool-comb-NFA* f $\mathcal{A}1$ $\mathcal{A}2$) w =
    ($w \in$ *lists* (Σ $\mathcal{A}1$) $\cap$ *lists* (Σ $\mathcal{A}2$) $\wedge$ f (*NFA-accept* $\mathcal{A}1$ w) (*NFA-accept* $\mathcal{A}2$ w))
⟨*proof*⟩

lemma *NFA-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* (*NFA-bool-comb* f $\mathcal{A}1$ $\mathcal{A}2$)
⟨*proof*⟩

lemma *NFA-bool-comb-DFA---NFA-accept* :
assumes *DFA-*$\mathcal{A}1$: *DFA* $\mathcal{A}1$
  and *DFA-*$\mathcal{A}2$: *DFA* $\mathcal{A}2$
shows *NFA-accept* (*NFA-bool-comb* f $\mathcal{A}1$ $\mathcal{A}2$) w =
    ($w \in$ *lists* (Σ $\mathcal{A}1$) $\cap$ *lists* (Σ $\mathcal{A}2$) $\wedge$ f (*NFA-accept* $\mathcal{A}1$ w) (*NFA-accept* $\mathcal{A}2$ w))
⟨*proof*⟩

3.9 Minimisation

definition *DFA-is-smaller-equiv* **where**
  DFA-is-smaller-equiv $\mathcal{A}$ $\mathcal{A}'$ ≡
   DFA $\mathcal{A}'$ $\wedge$ (Σ $\mathcal{A}$ = Σ $\mathcal{A}'$) $\wedge$ ($\mathcal{L}$ $\mathcal{A}$ = $\mathcal{L}$ $\mathcal{A}'$) $\wedge$ (*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*) $\wedge$
    ($\forall \mathcal{A}'$::('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* $\mathcal{A}$
⟨*proof*⟩

lemma $\mathcal{L}$*-truncate* [*simp*] :
$\mathcal{L}$ (*NFA-rec.truncate* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-truncate* [*simp*] :
NFA (*NFA-rec.truncate* $\mathcal{A}$) = *NFA* $\mathcal{A}$
⟨*proof*⟩

lemma *NFA-is-deterministic-truncate* [*simp*] :
NFA-is-deterministic (*NFA-rec.truncate* $\mathcal{A}$) = *NFA-is-deterministic* $\mathcal{A}$
⟨*proof*⟩

lemma *detNFA-truncate* [*simp*] :
detNFA (*NFA-rec.truncate* $\mathcal{A}$) = *detNFA* $\mathcal{A}$
⟨*proof*⟩

lemma *DFA-truncate* [*simp*] :
DFA (*NFA-rec.truncate* $\mathcal{A}$) = *DFA* $\mathcal{A}$
⟨*proof*⟩

lemma *DFA-is-smaller-equiv-truncate* [*simp*] : *DFA-is-smaller-equiv* $\mathcal{A}$ (*NFA-rec.truncate* $\mathcal{A}'$) ⟷ *DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$
⟨*proof*⟩

lemma *DFA-is-minimal-gen-def* :
fixes $\mathcal{A}$:: ('*q*, '*a*, '*X*) *NFA-rec-scheme*
shows *DFA-is-minimal* $\mathcal{A}$ ⟷ *DFA* $\mathcal{A}$ ∧ (∀ $\mathcal{A}'$::('*q*, '*a*, '*X2*) *NFA-rec-scheme*. ¬(*DFA-is-smaller-equiv* $\mathcal{A}$ $\mathcal{A}'$))
⟨*proof*⟩

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* $\mathcal{A}$
⟨*proof*⟩

lemma (**in** *minDFA*) *wf-minDFA* :
minDFA $\mathcal{A}$ ⟨*proof*⟩

lemma (**in** *minDFA*) *DFA-is-minimal---DLTS-reachable* :
$q \in \mathcal{Q}$ $\mathcal{A}$ ⟹ $q \notin$ *NFA-unreachable-states* $\mathcal{A}$
⟨*proof*⟩

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

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

theorem (**in** *DFA*) *DFA-is-minimal-alt-def* :
DFA-is-minimal $\mathcal{A} \longleftrightarrow$
(*NFA-is-initially-connected* $\mathcal{A} \wedge$
inj-on ($\mathcal{L}$*-in-state* $\mathcal{A}$) ($\mathcal{Q}\ \mathcal{A}$))
(**is** - = (*?P1* $\wedge$ *?P2*))
⟨*proof*⟩

lemma *DFA-is-minimal-alt-DFA-def* :
DFA-is-minimal $\mathcal{A} \longleftrightarrow$
(*DFA* $\mathcal{A} \wedge$ *NFA-is-initially-connected* $\mathcal{A} \wedge$
inj-on ($\mathcal{L}$*-in-state* $\mathcal{A}$) ($\mathcal{Q}\ \mathcal{A}$))
(**is** - = (*?P1* $\wedge$ *?P2*))
⟨*proof*⟩

lemma *NFA-isomorphic---DFA-is-minimal* :
fixes $\mathcal{A}1$ :: (*'q1*, *'a*, *'X*) *NFA-rec-scheme*
fixes $\mathcal{A}2$ :: (*'q2*, *'a*) *NFA-rec*
assumes *eq-A12*: *NFA-isomorphic* $\mathcal{A}1\ \mathcal{A}2$
and *min-A1*: *DFA-is-minimal* $\mathcal{A}1$
shows *DFA-is-minimal* $\mathcal{A}2$
⟨*proof*⟩

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*: $\Sigma\ \mathcal{A}1 = \Sigma\ \mathcal{A}2$
and *min-A1*: *DFA-is-minimal* $\mathcal{A}1$
and *min-A2*: *DFA-is-minimal* $\mathcal{A}2$
shows $\exists f.$ *inj-on* f ($\mathcal{Q}\ \mathcal{A}1$) $\wedge$
($\forall\, q{\in}\mathcal{Q}\ \mathcal{A}1.\ (f\ q) \in \mathcal{Q}\ \mathcal{A}2 \wedge \mathcal{L}$*-in-state* $\mathcal{A}2\ (f\ q) = \mathcal{L}$*-in-state* $\mathcal{A}1\ q$)
⟨*proof*⟩

lemma *DFA-is-minimal---equiv-states-bijection-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*: $\Sigma\ \mathcal{A}1 = \Sigma\ \mathcal{A}2$
and *min-A1*: *DFA-is-minimal* $\mathcal{A}1$
and *min-A2*: *DFA-is-minimal* $\mathcal{A}2$
shows $\exists f.$ *bij-betw* f ($\mathcal{Q}\ \mathcal{A}1$) ($\mathcal{Q}\ \mathcal{A}2$) $\wedge$
($\forall\, q{\in}\mathcal{Q}\ \mathcal{A}1.\ \mathcal{L}$*-in-state* $\mathcal{A}2\ (f\ q) = \mathcal{L}$*-in-state* $\mathcal{A}1\ q$)
⟨*proof*⟩

lemma *DFA-is-minimal---isomorph-wf* :
fixes $\mathcal{A}1$ :: (*'q1*, *'a*, *'X*) *NFA-rec-scheme*
fixes $\mathcal{A}2$ :: (*'q2*, *'a*) *NFA-rec*
assumes *min-A1*: *DFA-is-minimal* $\mathcal{A}1$

and *min-A2*: *DFA-is-minimal* $\mathcal{A}2$
and *L-eq*: $\mathcal{L}\ \mathcal{A}1 = \mathcal{L}\ \mathcal{A}2$
and Σ*-eq*: $\Sigma\ \mathcal{A}1 = \Sigma\ \mathcal{A}2$
shows *NFA-isomorphic-wf* $\mathcal{A}1\ \mathcal{A}2$
⟨*proof*⟩

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* $\mathcal{A} \equiv$ *efficient-determinise-NFA* (*NFA-reverse* $\mathcal{A}$)

definition *Brzozowski*
where *Brzozowski* $\mathcal{A} \equiv$ *Brzozowski-halfway* (*Brzozowski-halfway* $\mathcal{A}$)

lemma *Brzozowski-halfway---well-formed* :
NFA $\mathcal{A} \Longrightarrow$ *NFA* (*Brzozowski-halfway* $\mathcal{A}$)
⟨*proof*⟩

lemma *Brzozowski---well-formed* :
NFA $\mathcal{A} \Longrightarrow$ *NFA* (*Brzozowski* $\mathcal{A}$)
⟨*proof*⟩

lemma *Brzozowski-*Σ [*simp*] :
Σ (*Brzozowski* $\mathcal{A}$) = $\Sigma\ \mathcal{A}$
⟨*proof*⟩

lemma *Brzozowski-halfway-yields-DFA* :
assumes *NFA* $\mathcal{A}$
shows *DFA* (*Brzozowski-halfway* $\mathcal{A}$)
⟨*proof*⟩

lemma *Brzozowski-yields-DFA* :
assumes *NFA* $\mathcal{A}$
shows *DFA* (*Brzozowski* $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *Brzozowski-halfway---*$\mathcal{L}$:
shows $\mathcal{L}$ (*Brzozowski-halfway* $\mathcal{A}$) = {*rev* w |w. $w \in \mathcal{L}\ \mathcal{A}$}
⟨*proof*⟩

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

theorem (**in** *NFA*) *Brzozowski---*$\mathcal{L}$:
$\mathcal{L}$ (*Brzozowski* $\mathcal{A}$) = $\mathcal{L}\ \mathcal{A}$
⟨*proof*⟩

theorem (**in** *NFA*) *Brzozowski---NFA-accept* :
NFA-accept (*Brzozowski* $\mathcal{A}$) w = *NFA-accept* $\mathcal{A}$ w
⟨*proof*⟩

lemma (**in** *DFA*) *Brzozowski-halfway---minimal* :
assumes *connected*: *NFA-is-initially-connected* $\mathcal{A}$
shows *DFA-is-minimal* (*Brzozowski-halfway* $\mathcal{A}$)
⟨*proof*⟩

theorem (**in** *NFA*) *Brzozowski---minimal* :
DFA-is-minimal (*Brzozowski* $\mathcal{A}$)
⟨*proof*⟩

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*⟩

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*⟩

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*:
$\forall \mathcal{A}$. *NFA* $\mathcal{A} \longrightarrow$
  $\mathcal{L}$(*NFA-minimise* $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$ ∧
  Σ(*NFA-minimise* $\mathcal{A}$) = Σ $\mathcal{A}$ ∧
  DFA-is-minimal (*NFA-minimise* $\mathcal{A}$)
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

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* $\mathcal{A}2$
⟨*proof*⟩

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*⟩

lemma *Brzozowski---minimise* :
  NFA $\mathcal{A}$ ⟹ *NFA-isomorphic-wf* (*Brzozowski* $\mathcal{A}$) (*NFA-minimise* $\mathcal{A}$)
⟨*proof*⟩

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 ?A-min*)
⟨*proof*⟩

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*⟩

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* ⇒ ('*q*, '*a*) *NFA-rec* **where**
 Hopcroft-minimise $\mathcal{A}$ ≡ *NFA-rename-states* $\mathcal{A}$ (*SOME f*.
  NFA-is-strong-equivalence-rename-fun $\mathcal{A}$ *f* ∧ (∀ *q* ∈ $\mathcal{Q}$ $\mathcal{A}$. *f q* ∈ $\mathcal{Q}$ $\mathcal{A}$))

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*⟩

lemma *Hopcroft-minimise-*$\mathcal{Q}$:
fixes $\mathcal{A}$ :: ('*q*, '*a*, '*x*) *NFA-rec-scheme*
shows $\mathcal{Q}$ (*Hopcroft-minimise* $\mathcal{A}$) ⊆ $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

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*⟩

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* ⇒ ('*a set*) *set* ⇒ *bool* **where**
 is-partition Q P ⟷
  (⋃ *P* = *Q*) ∧ ({} ∉ *P*) ∧ (∀ *p1 p2*. *p1* ∈ *P* ∧ *p2* ∈ *P* ∧ *p1* ≠ *p2* ⟶ *p1* ∩ *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* $Q\ P$
⟨*proof*⟩

lemma *is-partition-Nil-Q* :
is-partition $\{\}\ P \longleftrightarrow P = \{\}$
⟨*proof*⟩

lemma *is-partition-Nil* [*simp*] :
is-partition $Q\ \{\} \longleftrightarrow Q = \{\}$
⟨*proof*⟩

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*⟩

lemma *is-partition-in-subset* :
assumes *is-part*: *is-partition* $Q\ P$
and *p-in*: $p \in P$
shows $p \subseteq Q$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *is-partition-distinct-subset* :
assumes *is-part*: *is-partition* $Q\ P$
and *p-in*: $p \in P$
and p'*-sub*: $p' \subseteq p$
shows $p' \notin P - \{p\}$
⟨*proof*⟩

lemma *is-partition-finite* :
assumes *fin-Q*: *finite* Q
and *is-part*: *is-partition* $Q\ P$
shows *finite* P
⟨*proof*⟩

lemma *is-partition-card* :
assumes *fin-Q*: *finite* Q
and *is-part*: *is-partition* $Q\ P$
shows *card* $Q =$ *setsum card* P
⟨*proof*⟩

lemma *is-partition-card-P* :
assumes *fin-Q*: *finite* Q
and *is-part*: *is-partition* $Q\ P$
shows *card* $P \leq$ *card* Q
⟨*proof*⟩

lemma *is-partition-find* :
assumes *is-part*: *is-partition* $Q\ P$

and *q-in-Q*: $q \in Q$
shows $\exists!p.\ p \in P \wedge q \in p$
⟨*proof*⟩

lemma *is-partition-refine* :
assumes *fin-Q*: *finite Q*
and *p-nin*: $p \notin P$
and *is-part*: *is-partition Q* (*insert p P*)
and *p1-p2-neq-emp*: $p1 \neq \{\}$ $p2 \neq \{\}$
and *p1-p2-disj*: $p1 \cap p2 = \{\}$
and *p1-p2-union*: $p = p1 \cup p2$
shows *is-partition Q* (*insert p1* (*insert p2 P*)) ∧
card (*insert p1* (*insert p2 P*)) = *Suc* (*card* (*insert p P*))
⟨*proof*⟩

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*⟩

definition *relation-of-partition* **where**
relation-of-partition P = {(*q1*, *q2*). ∃ *Q* ∈ *P*. *q1* ∈ *Q* ∧ *q2* ∈ *Q*}

lemma *relation-of-partition-is-equiv* :
assumes *part-P*: *is-partition Q P*
shows *equiv Q* (*relation-of-partition P*)
⟨*proof*⟩

definition *partition-less-eq* **where**
partition-less-eq P1 P2 ⟷ *relation-of-partition P1* ⊆ *relation-of-partition P2*

lemma *relation-of-partition-inverse* :
assumes *part-P*: *is-partition Q P*
shows *Q* // (*relation-of-partition P*) = *P*
(**is** *?ls* = *?rs*)
⟨*proof*⟩

lemma *quotient-inverse* :
assumes *eq-r*: *equiv Q r*
shows (*relation-of-partition* (*Q* // *r*)) = *r*
(**is** *?ls* = *?rs*)
⟨*proof*⟩

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 $\mathcal{A}$ *p* ≡
(*p* ⊆ $\mathcal{Q}$ $\mathcal{A}$) ∧
((*p* ⊆ $\mathcal{F}$ $\mathcal{A}$) ∨ (*p* ∩ $\mathcal{F}$ $\mathcal{A}$ = {})) ∧
(∀ *q1* ∈ $\mathcal{Q}$ $\mathcal{A}$. ∀ *q2* ∈ $\mathcal{Q}$ $\mathcal{A}$.
$\mathcal{L}$-*in-state* $\mathcal{A}$ *q1* = $\mathcal{L}$-*in-state* $\mathcal{A}$ *q2* ∧
q1 ∈ *p* ⟶ *q2* ∈ *p*)

lemma *is-weak-language-equiv-setI* [*intro*!] :
⟦*p* ⊆ $\mathcal{Q}$ $\mathcal{A}$;

$(p \subseteq \mathcal{F}\ \mathcal{A}) \vee (p \cap \mathcal{F}\ \mathcal{A} = \{\});$
$\bigwedge q1\ q2.\ [\![q1 \in \mathcal{Q}\ \mathcal{A};\ q2 \in \mathcal{Q}\ \mathcal{A};\ q1 \in p;$
$\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2]\!] \Longrightarrow$
$q2 \in p]\!] \Longrightarrow$
is-weak-language-equiv-set $\mathcal{A}\ p$
⟨*proof*⟩

lemma *is-weak-language-equiv-setD* :
[[*is-weak-language-equiv-set* $\mathcal{A}\ p$;
$q1 \in \mathcal{Q}\ \mathcal{A};\ q2 \in \mathcal{Q}\ \mathcal{A};\ \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2$]] ⟹
$(q1 \in p) \longleftrightarrow (q2 \in p)$
⟨*proof*⟩

definition *is-weak-language-equiv-partition* **where**
is-weak-language-equiv-partition $\mathcal{A}\ P \longleftrightarrow$
is-partition $(\mathcal{Q}\ \mathcal{A})\ P \wedge$
$(\forall p \in P.$ *is-weak-language-equiv-set* $\mathcal{A}\ p)$

lemma *is-weak-language-equiv-partitionI* [*intro*!] :
[[*is-partition* $(\mathcal{Q}\ \mathcal{A})\ P$;
$\bigwedge p.\ p \in P \Longrightarrow$ *is-weak-language-equiv-set* $\mathcal{A}\ p$]] ⟹
is-weak-language-equiv-partition $\mathcal{A}\ P$
⟨*proof*⟩

lemma *is-weak-language-equiv-partitionD1* :
[[*is-weak-language-equiv-partition* $\mathcal{A}\ P$;
$p \in P$]] ⟹ $p \subseteq \mathcal{F}\ \mathcal{A} \vee (p \cap \mathcal{F}\ \mathcal{A} = \{\})$
⟨*proof*⟩

lemma *is-weak-language-equiv-partitionD2* :
[[*is-weak-language-equiv-partition* $\mathcal{A}\ P$;
$q1 \in \mathcal{Q}\ \mathcal{A};\ q2 \in \mathcal{Q}\ \mathcal{A};\ p \in P;$
$\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2;$
$q1 \in p$]] ⟹ $q2 \in p$
⟨*proof*⟩

lemma *is-weak-language-equiv-partitionD2-iff* :
[[*is-weak-language-equiv-partition* $\mathcal{A}\ P$;
$q1 \in \mathcal{Q}\ \mathcal{A};\ q2 \in \mathcal{Q}\ \mathcal{A};\ p \in P;$
$\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2$]] ⟹
$q1 \in p \longleftrightarrow q2 \in p$
⟨*proof*⟩

lemma *is-weak-language-equiv-partitionD3* :
is-weak-language-equiv-partition $\mathcal{A}\ P \Longrightarrow$
is-partition $(\mathcal{Q}\ \mathcal{A})\ P$
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

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* $\mathcal{A} = \{\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}, \mathcal{F}\ \mathcal{A}\} \cap \{s.\ s \neq \{\}\}$
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

definition *Myhill-Nerode-partition* **where**
Myhill-Nerode-partition $\mathcal{A} \equiv (\mathcal{Q}\ \mathcal{A})\ //\ (Myhill\text{-}Nerode\text{-}relation\ \mathcal{A})$

lemma *Myhill-Nerode-partition-alt-def* :
Myhill-Nerode-partition $\mathcal{A} =$
$\{\{q'.\ q' \in \mathcal{Q}\ \mathcal{A} \wedge \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q' = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q\} \mid q.\ q \in \mathcal{Q}\ \mathcal{A}\}$
⟨*proof*⟩

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*⟩

lemma *Myhill-Nerode-partition---$\mathcal{Q}$-emp* :
$\mathcal{Q}\ \mathcal{A} = \{\} \Longrightarrow$ *Myhill-Nerode-partition* $\mathcal{A} = \{\}$
⟨*proof*⟩

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*⟩

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 $\mathcal{A}\ p \equiv$
$(p \subseteq \mathcal{Q}\ \mathcal{A}) \wedge$
$((p \subseteq \mathcal{F}\ \mathcal{A}) \vee (p \cap \mathcal{F}\ \mathcal{A} = \{\})) \wedge$
$(\forall\, q1 \in \mathcal{Q}\ \mathcal{A}.\ \forall\, q2 \in \mathcal{Q}\ \mathcal{A}.$
$q1 \in p \wedge q2 \in p \longrightarrow \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2)$

lemma *is-weak2-language-equiv-set-alt-def* :
is-weak2-language-equiv-set $\mathcal{A}\ p =$
$((p \subseteq \mathcal{Q}\ \mathcal{A}) \wedge$
$(\forall\, q1 \in \mathcal{Q}\ \mathcal{A}.\ \forall\, q2 \in \mathcal{Q}\ \mathcal{A}.$
$q1 \in p \wedge q2 \in p \longrightarrow \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2))$
⟨*proof*⟩

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*⟩

definition *is-strong-language-equiv-set* **where**
is-strong-language-equiv-set $\mathcal{A}$ $p \equiv$
is-weak-language-equiv-set $\mathcal{A}$ $p \wedge$
is-weak2-language-equiv-set $\mathcal{A}$ p

lemma *is-strong-language-equiv-set-alt-def* :
is-strong-language-equiv-set $\mathcal{A}$ $p =$
$((p \subseteq \mathcal{Q}\ \mathcal{A}) \wedge$
$(\forall\, q1 \in p.\ \forall\, q2 \in \mathcal{Q}\ \mathcal{A}.$
$((q2 \in p)) \longleftrightarrow (\mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q1 = \mathcal{L}\text{-}in\text{-}state\ \mathcal{A}\ q2)))$
⟨*proof*⟩

lemma *is-strong-language-equiv-partition-fun-alt-def* :
($P =$ *Myhill-Nerode-partition* $\mathcal{A}$) $\longleftrightarrow$
(*is-partition* ($\mathcal{Q}$ $\mathcal{A}$) $P \wedge$ ($\forall\, p{\in}P.$ *is-strong-language-equiv-set* $\mathcal{A}$ p))
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

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 $\{\} = (\{\}, \{\})$
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *split-set-union-distint*:
split-set P $S = (S1, S2) \Longrightarrow$
$(S = S1 \cup S2) \wedge (S1 \cap S2 = \{\})$
⟨*proof*⟩

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 deterministic, 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* =
  $(\{q\ .\ \exists\, q'.\ q \in p1 \wedge (\delta\ \mathcal{A})\ (q,\ a) = Some\ q' \wedge q' \in p2\},$
  $\{q\ .\ \exists\, q'.\ q \in p1 \wedge (\delta\ \mathcal{A})\ (q,\ a) = Some\ q' \wedge q' \notin p2\})$
⟨*proof*⟩

lemma *split-language-equiv-partition-disjoint* :
  ⟦*split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)⟧ ⟹
  $p1a \cap p1b = \{\}$
⟨*proof*⟩

lemma *split-language-equiv-partition-union* :
  split-language-equiv-partition $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*) ⟹
  $p1 = p1a \cup p1b$
⟨*proof*⟩

lemma *split-language-equiv-partition-subset* :
assumes *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2* = (*p1a*, *p1b*)
shows $p1a \subseteq p1 \wedge p1b \subseteq p1$
⟨*proof*⟩

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*: $a \in \Sigma\ \mathcal{A}$
  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*⟩

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*⟩

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*: $\bigwedge p.\ p \in P \Longrightarrow (p \subseteq \mathcal{F}\ \mathcal{A}) \vee (p \cap \mathcal{F}\ \mathcal{A} = \{\})$
  and *no-split*: $\bigwedge p1\ a\ p2.$ ⟦$p1 \in P$; $a \in \Sigma\ \mathcal{A}$; $p2 \in P$⟧ ⟹
    ¬(*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*⟩

lemma (**in** *DFA*) *split-language-equiv-partition-final* :
assumes *is-part*: *is-weak-language-equiv-partition* $\mathcal{A}$ *P*
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*)
shows *P* = *Myhill-Nerode-partition* $\mathcal{A}$
⟨*proof*⟩

4.3 Naive implementation

definition *Hopcroft-naive* **where**
Hopcroft-naive $\mathcal{A}$ =
WHILEIT (*is-weak-language-equiv-partition* $\mathcal{A}$)

(λ*P*. ∃ *p1 a p2*. (*p1* ∈ *P* ∧ *a* ∈ Σ $\mathcal{A}$ ∧ *p2* ∈ *P* ∧ *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2*))

(λ*P*. *do* {
(*p1*, *a*, *p2*) ← *SPEC* (λ(*p1*,*a*,*p2*). *p1* ∈ *P* ∧ *a* ∈ Σ $\mathcal{A}$ ∧ *p2* ∈ *P* ∧
split-language-equiv-partition-pred $\mathcal{A}$ *p1 a p2*);
let (*p1a*, *p1b*) = *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2*;
RETURN ((*P* − {*p1*}) ∪ {*p1a*, *p1b*})}) (*Hopcroft-accepting-partition* $\mathcal{A}$)

lemma (**in** *DFA*) *Hopcroft-naive-correct* :
Hopcroft-naive $\mathcal{A}$ ≤ *SPEC* (λ*P*. *P* = *Myhill-Nerode-partition* $\mathcal{A}$)
⟨*proof*⟩

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* $\mathcal{A}$ *p1 a p2*
shows *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2a* ∨
split-language-equiv-partition-pred $\mathcal{A}$ *p1 a p2b*
⟨*proof*⟩

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* $\mathcal{A}$ *p1 a p2*)
shows (*split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2a*) ⟷
(*split-language-equiv-partition-pred* $\mathcal{A}$ *p1 a p2b*)
⟨*proof*⟩

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* ⊆ *p1′*
shows *split-language-equiv-partition-pred* $\mathcal{A}$ *p1′ a p2*
⟨*proof*⟩

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'*} ∩ {*p*. *p* ≠ {}})

definition *Hopcroft-split* **where**
Hopcroft-split $\mathcal{A}$ *p a res P* =
(*res* ∪ ⋃((*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*
⟨*proof*⟩

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* $\mathcal{A}$ *p2 a res p1* =
(*let* (*p1a*, *p1b*) = *split-language-equiv-partition* $\mathcal{A}$ *p1 a p2 in*
(*if* (*p1a* = {} ∨ *p1b* = {}) *then* (*insert p1 res*) *else*
(*insert p1a* (*insert p1b res*))))
⟨*proof*⟩

lemma *Hopcroft-split-Insert* [*simp*] :
Hopcroft-split $\mathcal{A}$ *p2 a res* (*insert p1 P*) =
Hopcroft-split $\mathcal{A}$ *p2 a* (*Hopcroft-split-aux* $\mathcal{A}$ *p2 a res p1*) *P*
⟨*proof*⟩

lemma (**in** *DFA*) *Hopcroft-split-correct* :
assumes *is-part*: *is-weak-language-equiv-partition* $\mathcal{A}$ (*res* ∪ *P*)
and *p2-equiv-set*: *is-weak-language-equiv-set* $\mathcal{A}$ *p2*
and *a-in*: *a* ∈ Σ $\mathcal{A}$
and *res-P-disjoint*: *res* ∩ *P* = {}
shows *is-weak-language-equiv-partition* $\mathcal{A}$
(*Hopcroft-split* $\mathcal{A}$ *p2 a res P*) (**is** *?P1 res P*)
(*Hopcroft-split* $\mathcal{A}$ *p2 a res P*) ≠ (*res* ∪ *P*) ⟶
card (*res* ∪ *P*) < *card* (*Hopcroft-split* $\mathcal{A}$ *p2 a res P*) (**is** *?P2 res P*)
⟨*proof*⟩

lemma (**in** *DFA*) *Hopcroft-split-correct-simple* :
assumes *is-part*: *is-weak-language-equiv-partition* $\mathcal{A}$ *P*
and *p2-in*: *p2* ∈ *P*
and *a-in*: *a* ∈ 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*) ≠ *P* ⟹
card P < *card* (*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
⟨*proof*⟩

lemma (**in** *DFA*) *split-language-equiv-partition-pred---split-not-eq*:
assumes *p1-in-split*: *p1* ∈ (*Hopcroft-split* $\mathcal{A}$ *p2 a* {} *P*)
and *split-pred*: *split-language-equiv-partition-pred* $\mathcal{A}$ *p1 aa p2'*
shows (*aa*, *p2'*) ≠ (*a*, *p2*)
⟨*proof*⟩

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*
$(p = p1a \vee p = p1b)))$
⟨*proof*⟩

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* $\{\} = res$
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*
$p = p1a \vee p = p1b$
⟨*proof*⟩

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*⟩

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
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}upper\ Q\ splitted\ P\ L\ L' \longleftrightarrow$
$(\forall\, a\ p.\ (a,\ p) \in L' \longrightarrow$
$((a,\ p) \in L \wedge (\forall\, pa\ pb.\ (p,\ pa,\ pb) \notin splitted)) \vee$
$(\exists\, p'\ pa\ pb.\ (p',\ pa,\ pb) \in splitted \wedge a \in Q \wedge (p = pa \vee p = pb)))$

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
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\text{-}not\text{-}splitted\ Q\ splitted\ P\ L\ L' \longleftrightarrow$
$(\forall\, a\ p.\ ((a,\ p) \in L \wedge a \in Q \wedge p \in P \wedge$
$(\forall\, pa\ pb.\ (p,\ pa,\ pb) \notin splitted)) \longrightarrow$
$(a,\ p) \in L')$

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
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\text{-}splitted\text{-}in\text{-}L\ Q\ splitted\ P\ L\ L' \longleftrightarrow$
$(\forall\, a\ p\ pa\ pb.$
$((a,\ p) \in L \wedge a \in Q \wedge (p,\ pa,\ pb) \in splitted) \longrightarrow$
$((a,\ pa) \in L') \wedge (a,\ pb) \in L')$

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
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\text{-}splitted\ Q\ splitted\ P\ L\ L' \longleftrightarrow$
$(\forall\, a\ p\ pa\ pb.$
$(a \in Q \wedge (p,\ pa,\ pb) \in splitted) \longrightarrow$
$((a,\ pa) \in L') \vee (a,\ pb) \in L')$

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
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\ Q\ splitted\ P\ L\ L' \longleftrightarrow$
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\text{-}not\text{-}splitted\ Q\ splitted\ P\ L\ L' \wedge$
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\text{-}splitted\text{-}in\text{-}L\ Q\ splitted\ P\ L\ L' \wedge$
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\text{-}splitted\ Q\ splitted\ P\ L\ L'$

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**
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\ Q\ splitted\ P\ L\ L' \longleftrightarrow$
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}lower\ Q\ splitted\ P\ L\ L' \wedge$
$Hopcroft\text{-}update\text{-}splitters\text{-}pred\text{-}aux\text{-}upper\ Q\ splitted\ P\ L\ L'$

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'⟧ ⟹
Hopcroft-update-splitters-pred-aux Q splitted P L L'
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred-aux-upper---is-upper* :
⟦*Hopcroft-update-splitters-pred-aux-upper Q splitted P L L'*; *L''* ⊆ *L'*⟧ ⟹
Hopcroft-update-splitters-pred-aux-upper Q splitted P L L''
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred-aux-lower---is-lower* :
⟦*Hopcroft-update-splitters-pred-aux-lower Q splitted P L L'*; *L'* ⊆ *L''*⟧ ⟹
Hopcroft-update-splitters-pred-aux-lower Q splitted P L L''
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred-aux-Emp-Id* :
Hopcroft-update-splitters-pred-aux Q {} *P L L*
⟨*proof*⟩

definition *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred* **where**
Hopcroft-update-splitters-pred-aux-splitted-equiv-pred splitted1 splitted2 ⟷
(∀ *p2 p2a p2b*. (*p2*, *p2a*, *p2b*) ∈ *splitted1* ⟶
(*p2*, *p2a*, *p2b*) ∈ *splitted2* ∨ (*p2*, *p2b*, *p2a*) ∈ *splitted2*) ∧
(∀ *p2 p2a p2b*. (*p2*, *p2a*, *p2b*) ∈ *splitted2* ⟶
(*p2*, *p2a*, *p2b*) ∈ *splitted1* ∨ (*p2*, *p2b*, *p2a*) ∈ *splitted1*)

lemma *Hopcroft-update-splitters-pred-aux-splitted-equiv-pred-nin* :
Hopcroft-update-splitters-pred-aux-splitted-equiv-pred sp1 sp2 ⟹
(∀ *pa pb*. (*p*, *pa*, *pb*) ∉ *sp1*) ⟷ (∀ *pa pb*. (*p*, *pa*, *pb*) ∉ *sp2*)
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

definition *Hopcroft-update-splitters-pred* ::
 ('*q*, '*a*, '*x*) *NFA-rec-scheme* ⇒ '*q set* ⇒ '*a* ⇒ ('*q set*) *set* ⇒
  ('*a* × '*q set*) *set* ⇒ ('*a* × '*q set*) *set* ⇒ *bool*
where
 Hopcroft-update-splitters-pred $\mathcal{A}$ *pp aa P L L'* ⟷
  Hopcroft-update-splitters-pred-aux (Σ $\mathcal{A}$) (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*) *P*
   (*L* − {(*aa*, *pp*)}) *L'*

lemma *Hopcroft-update-splitters-pred-exists*:
 ∃ *L'*. *Hopcroft-update-splitters-pred* $\mathcal{A}$ *pp aa P L L'*
⟨*proof*⟩

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*) ∈ *L'*
obtains (*no-split*) (*a*, *p*) ∈ *L* − {(*aa*, *pp*)} ⋀*pa pb*. (*p*, *pa*, *pb*) ∉ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)
  | (*split*) *p' pa pb* **where** (*p'*, *pa*, *pb*) ∈ (*Hopcroft-splitted* $\mathcal{A}$ *pp aa* {} *P*)
      a ∈ Σ $\mathcal{A}$ *p* = *pa* ∨ *p* = *pb*
⟨*proof*⟩

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* ∈ *P*
  and *aa-in*: *aa* ∈ Σ $\mathcal{A}$
  and *nin-splitted*: ⋀*pa pb*. (*p*, *pa*, *pb*) ∉ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*
  and *in-L*: (*aa*, *p*) ∈ *L* − {(*a*, *p2*)}
shows (*aa*, *p*) ∈ *L'*
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred---in-splitted-in-L* :
 ⟦*Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L L'*; *aa* ∈ Σ $\mathcal{A}$;
  (*p*, *pa*, *pb*) ∈ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*; (*aa*, *p*) ∈ *L* − {(*a*, *p2*)}⟧ ⟹
  (*aa*, *pa*) ∈ *L'* ∧ (*aa*, *pb*) ∈ *L'*
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred---in-splitted* :
 ⟦*Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L L'*; *aa* ∈ Σ $\mathcal{A}$;
  (*p*, *pa*, *pb*) ∈ *Hopcroft-splitted* $\mathcal{A}$ *p2 a* {} *P*⟧ ⟹
  (*aa*, *pa*) ∈ *L'* ∨ (*aa*, *pb*) ∈ *L'*
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred---label-in* :
⟦*Hopcroft-update-splitters-pred* $\mathcal{A}$ *p2 a P L L'*;
 ⋀*ap*. *ap* ∈ *L* ⟹ *fst ap* ∈ Σ $\mathcal{A}$;
 (*a'*, *p*) ∈ *L'*⟧ ⟹ *a'* ∈ (Σ $\mathcal{A}$)
⟨*proof*⟩

lemma *Hopcroft-update-splitters-pred---splitted-emp* :
assumes *L'-OK*: *Hopcroft-update-splitters-pred* $\mathcal{A}$ *p a P L L'*
  and *L-OK*: ⋀*ap*. *ap* ∈ *L* ⟹ *fst ap* ∈ Σ $\mathcal{A}$ ∧ *snd ap* ∈ *P*
  and *splitted-emp*: *Hopcroft-splitted* $\mathcal{A}$ *p a* {} *P* = {}
shows *L'* = *L* − {(*a*, *p*)}
⟨*proof*⟩

4.4.3 The abstract Algorithm

definition *Hopcroft-abstract-invar* **where**
Hopcroft-abstract-invar $\mathcal{A}$ = (λ(P, L).
is-weak-language-equiv-partition $\mathcal{A}$ P $\wedge$
($\forall$ *ap* $\in$ L. *fst ap* $\in$ Σ $\mathcal{A}$ $\wedge$ *snd ap* $\in$ P) $\wedge$
($\forall$ *p1 a p2*. (a $\in$ Σ $\mathcal{A}$ $\wedge$ ($\exists$ *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$$a$ p. (a,p) $\in$ L $\Longrightarrow$ a $\in$ Σ $\mathcal{A}$ $\wedge$ p $\in$ P);
($\bigwedge$*p1 a p2*. ⟦*is-weak-language-equiv-partition* $\mathcal{A}$ P;
$\bigwedge$$a$ p. (a,p) $\in$ L $\Longrightarrow$ a $\in$ Σ $\mathcal{A}$ $\wedge$ p $\in$ P;
a $\in$ Σ $\mathcal{A}$; *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)
⟨*proof*⟩

definition *Hopcroft-abstract-init* $\mathcal{A}$ ≡ (*Hopcroft-accepting-partition* $\mathcal{A}$, (Σ $\mathcal{A}$ × {$\mathcal{F}$ $\mathcal{A}$}))

lemma (**in** *DFA*) *Hopcroft-abstract-invar-init* :
assumes $\mathcal{F}$-*OK*: $\mathcal{F}$ $\mathcal{A}$ ≠ {}
shows *Hopcroft-abstract-invar* $\mathcal{A}$ (*Hopcroft-abstract-init* $\mathcal{A}$)
⟨*proof*⟩

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) $\in$ L);
(P', L') ← *SPEC* (λ(P', L'). *Hopcroft-update-splitters-pred* $\mathcal{A}$ p a P L L' $\wedge$
(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* $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *NFA*) *Hopcroft-abstract-invar---implies-finite-L*:
assumes *invar*: *Hopcroft-abstract-invar* $\mathcal{A}$ *PL*
shows *finite* (*snd PL*)
⟨*proof*⟩

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*⟩

lemma (**in** *DFA*) *Hopcroft-abstract-variant-exists*:
WHILE-variant-exists Hopcroft-abstract-b (*Hopcroft-abstract-f* $\mathcal{A}$) (*Hopcroft-abstract-invar* $\mathcal{A}$)
⟨*proof*⟩

lemma (**in** *DFA*) *Hopcroft-abstract-correct* :
Hopcroft-abstract $\mathcal{A}$ ≤ *SPEC* (λP. P = *Myhill-Nerode-partition* $\mathcal{A}$)
⟨*proof*⟩

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* (Σ $\mathcal{A}$) (*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* ($\lambda P'$. (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
($\lambda p'$ (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* (λpmm. (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*⟩

lemma *Hopcroft-set-step-correct-aux2* :
assumes P-*part*: *is-partition* ($\mathcal{Q}$ $\mathcal{A}$) P
and *invar*: *Hopcroft-set-step-invar* $\mathcal{A}$ p a P L P' (P'', L'')
and p'-*in*: p' ∈ P'
and P'-*subset*: P' ⊆ P
and p'-*split*: *split-language-equiv-partition* $\mathcal{A}$ p' a p = (pt', pf')
and *ptf-eq*: pt' = {} ∨ pf' = {}
shows *Hopcroft-set-step-invar* $\mathcal{A}$ p a P L (P' − {p'}) (P'', L'')
⟨*proof*⟩

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*⟩

lemma *Hopcroft-set-step-correct-aux4* :
assumes *invar*: *Hopcroft-set-step-invar* $\mathcal{A}\ p\ a\ P\ L\ \{\}\ (P'',\ L'')$
shows *Hopcroft-update-splitters-pred-aux* $(\Sigma\ \mathcal{A})$ (*Hopcroft-splitted* $\mathcal{A}\ p\ a\ \{\}\ P)\ P$
    $(L - \{(a,\ p)\})\ L'' \wedge$
  $P'' = Hopcroft\text{-}split\ \mathcal{A}\ p\ a\ \{\}\ P$
⟨*proof*⟩

lemma (**in** *DFA*) *Hopcroft-set-step-correct* :
assumes *part-P*: *is-partition* $(\mathcal{Q}\ \mathcal{A})\ P$
  and *L-OK*: $\bigwedge a\ p.\ (a,\ p) \in L \Longrightarrow a \in \Sigma\ \mathcal{A}$
shows *Hopcroft-set-step* $\mathcal{A}\ p\ a\ P\ L \leq$
    $SPEC\ (\lambda(P',\ L').\ Hopcroft\text{-}update\text{-}splitters\text{-}pred\ \mathcal{A}\ p\ a\ P\ L\ L' \wedge P' = Hopcroft\text{-}split\ \mathcal{A}\ p\ a\ \{\}\ P)$
⟨*proof*⟩

definition *Hopcroft-set-f* **where**
Hopcroft-set-f $\mathcal{A} =$
 $(\lambda(P,\ L).\ do\ \{$
    $ASSERT\ (Hopcroft\text{-}abstract\text{-}invar\ \mathcal{A}\ (P,\ L))$;
    $ASSERT\ (L \neq \{\})$;
    $(a,p) \leftarrow SPEC\ (\lambda(a,p).\ (a,p) \in L)$;
    $(P',\ L') \leftarrow Hopcroft\text{-}set\text{-}step\ \mathcal{A}\ p\ a\ P\ L$;
    $RETURN\ (P',\ L')$
  $\})$

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-precompute-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-precompute-step* $\mathcal{A}\ p\ a\ pre\ P\ L \leq \Downarrow Id\ (Hopcroft\text{-}set\text{-}step\ \mathcal{A}\ p\ a\ P\ L)$
⟨*proof*⟩

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* :: $'q\ partition\text{-}map \Rightarrow ('q\ set)\ set$ **where**
partition-map-α $(im,\ sm,\ n) = \{p \mid i\ p.\ i < n \wedge im\ i = Some\ p\}$

fun *partition-map-invar* :: $'q\ partition\text{-}map \Rightarrow bool$ **where**
partition-map-invar $(im,\ sm,\ n) \longleftrightarrow$
$dom\ im = \{i\ .\ i < n\} \wedge$
$(\forall\, i.\ im\ i \neq Some\ \{\}) \wedge$
$(\forall\, q\ i.\ (sm\ q = Some\ i) \longleftrightarrow (\exists\, p.\ im\ i = Some\ p \wedge q \in p))$

definition *partition-index-map* :: $'q\ partition\text{-}map \Rightarrow (nat \Rightarrow 'q\ set)$ **where**
partition-index-map $P\ i = the\ ((fst\ P)\ i)$

definition *partition-state-map* :: $'q\ partition\text{-}map \Rightarrow ('q \Rightarrow nat)$ **where**
partition-state-map $P\ q = the\ (fst\ (snd\ P)\ q)$

definition *partition-free-index* :: $'q\ partition\text{-}map \Rightarrow nat$ **where**
partition-free-index $P = snd\ (snd\ P)$
lemma *partition-free-index-simp*[*simp*] :
partition-free-index $(im,\ sm,\ n) = n$ ⟨*proof*⟩

definition *partition-map-empty* :: $'q\ partition\text{-}map$ **where**
partition-map-empty $= (empty,\ empty,\ 0)$

lemma *partition-map-empty-correct* :
partition-map-α $(partition\text{-}map\text{-}empty) = \{\}$
partition-map-invar $(partition\text{-}map\text{-}empty)$
⟨*proof*⟩

definition *partition-map-sing* :: $'q\ set \Rightarrow ('q\ partition\text{-}map)$ **where**
partition-map-sing $Q =$
$(empty\ (0 \mapsto Q)\ ,\ (\lambda q.\ if\ (q \in Q)\ then\ Some\ 0\ else\ None),\ 1)$

lemma *partition-map-sing-correct* :
partition-map-α $(partition\text{-}map\text{-}sing\ Q) = \{Q\}$
partition-map-invar $(partition\text{-}map\text{-}sing\ Q) \longleftrightarrow Q \neq \{\}$
⟨*proof*⟩

lemma *partition-index-map-inj-on* :
assumes *invar*: *partition-map-invar* P
and *p-OK*: $\bigwedge i.\ i \in p \Longrightarrow i < partition\text{-}free\text{-}index\ P$
shows *inj-on* $(partition\text{-}index\text{-}map\ P)\ p$
⟨*proof*⟩

lemma *partition-map-is-partition* :
assumes *invar*: *partition-map-invar P*
shows *is-partition* ($\bigcup$ (*partition-map-α P*)) (*partition-map-α P*)
⟨*proof*⟩

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* ∩ *partition-index-map P j* = {}) ⟷ (*i* ≠ *j*)
⟨*proof*⟩

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-map-α P*))
⟨*proof*⟩

lemma *partition-state-map-le* :
assumes *invar*: *partition-map-invar P*
and *q-in*: *q* ∈ ($\bigcup$ (*partition-map-α P*))
shows *partition-state-map P q* < *partition-free-index P*
⟨*proof*⟩

fun *partition-map-split* :: *'q partition-map* ⇒ *nat* ⇒ *'q set* => *'q set* ⇒ *'q partition-map* **where**
partition-map-split (*im*, *sm*, *n*) *i pmin pmax* =
(*im* (*i* ↦ *pmax*) (*n* ↦ *pmin*),
λ*q*. *if* (*q* ∈ *pmin*) *then Some n else sm q*,
Suc n)

definition *Hopcroft-map-state-α* **where**
Hopcroft-map-state-α σ =
(*partition-map-α* (*fst σ*), (*apsnd* (*partition-index-map* (*fst σ*))) ‘ (*snd σ*))

definition *Hopcroft-map-state-invar* **where**
Hopcroft-map-state-invar σ =
(*partition-map-invar* (*fst σ*) ∧
(∀ *ap* ∈ (*snd σ*). *snd ap* < *partition-free-index* (*fst σ*)))

definition *Hopcroft-map-state-rel* **where**
Hopcroft-map-state-rel = *build-rel Hopcroft-map-state-α Hopcroft-map-state-invar*

lemma *Hopcroft-map-state-rel-sv*[*refine*] :
single-valued Hopcroft-map-state-rel
⟨*proof*⟩

definition *Hopcroft-map-step-invar* **where**
Hopcroft-map-step-invar $\mathcal{A}$ *p a P L P' σ* ⟷
Hopcroft-set-step-invar $\mathcal{A}$ (*partition-index-map P p*) *a* (*partition-map-α P*)
((*apsnd* (*partition-index-map P*)) ‘ *L*) ((*partition-index-map P*) ‘ *P'*)
(*Hopcroft-map-state-α σ*) ∧ *Hopcroft-map-state-invar σ*

definition *Hopcroft-map-step* **where**
Hopcroft-map-step $\mathcal{A}$ *p a pre P L* =
(*do* { *ASSERT* (*pre* ⊆ *dom* (*fst* (*snd P*)));
let PS = (*partition-state-map P*) ‘ *pre*;
(*P'*, *L'*) ← *FOREACHi* (*Hopcroft-map-step-invar* $\mathcal{A}$ *p a P L*) *PS*
(λ(*p'*::*nat*) (*P'*::*'q partition-map*, *L'*). *do* {
ASSERT (*p'* ∈ *dom* (*fst P'*));

```
        let (pt', pct', pf', pcf') =
         ({q . q ∈ partition-index-map P' p' ∧ q ∈ pre}, card {q . q ∈ partition-index-map P' p' ∧ q ∈ pre},
          {q . q ∈ partition-index-map P' p' ∧ q ∉ pre}, card {q . q ∈ partition-index-map P' p' ∧ q ∉ pre});
        if (pcf' = 0) then (RETURN (P', L')) else
        do {
          let (pmin, pmax) = (if (pcf' < pct') then (pf', pt') else (pt', pf'));
          let L' = {(a, partition-free-index P') | a. a ∈ Σ A} ∪ L';
          let P' = partition-map-split P' p' pmin pmax;
          RETURN (P', L')
        }
      }) (P, L − {(a, p)});
    RETURN (P', L')
  })

lemma Hopcroft-map-step-correct :
fixes P :: 'q partition-map
  and L :: ('a × nat) set
  and A :: ('q, 'a, 'X) NFA-rec-scheme
assumes PL-OK: ((P,L), (P', L')) ∈ Hopcroft-map-state-rel
    and p-le: p < partition-free-index P
    and p'-eq: p' = partition-index-map P p
    and part-P': is-partition (Q A) P'
    and pre-subset: pre ⊆ Q A
shows Hopcroft-map-step A p a pre P L ≤ ⇓Hopcroft-map-state-rel (Hopcroft-precompute-step A p' a pre P' L')
⟨proof⟩

definition Hopcroft-map-f where
Hopcroft-map-f A =
 (λ(P, L). do {
      ASSERT (Hopcroft-abstract-invar A (Hopcroft-map-state-α (P, L)));
      ASSERT (L ≠ {});
      (a,i) ← SPEC (λ(a,i). (a,i) ∈ L);
      ASSERT (i ∈ dom (fst P));
      let pre = {q. ∃ q'. q' ∈ partition-index-map P i ∧ (q, a, q') ∈ Δ A};
      (P', L') ← Hopcroft-map-step 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 A PL ≤ ⇓Hopcroft-map-state-rel (Hopcroft-abstract-f A PL')
⟨proof⟩


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⟩

lemma partition-map-init---index-0:
```

partition-index-map (*partition-map-init* $\mathcal{A}$) *0* = $\mathcal{F}$ $\mathcal{A}$
⟨*proof*⟩

lemma *partition-map-init---index-le*:
0 < *partition-free-index* (*partition-map-init* $\mathcal{A}$)
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

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*⟩

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*⟩

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*)

$s.invar\ Q \Longrightarrow sp.\alpha\ \ (q\text{-}set\text{-}convert\ Q) = s.\alpha\ Q$
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**
$partition\text{-}impl\text{-}rel = build\text{-}rel\ partition\text{-}impl\text{-}\alpha\ partition\text{-}impl\text{-}invar$

lemma *partition-impl-rel-sv*[*refine*] :
$single\text{-}valued\ partition\text{-}impl\text{-}rel$
⟨*proof*⟩

definition *partition-impl-empty* **where**
$partition\text{-}impl\text{-}empty = (im.empty,\ sm.empty,\ 0)$

lemma *partition-impl-empty-correct* :
$(partition\text{-}impl\text{-}empty,\ partition\text{-}map\text{-}empty) \in partition\text{-}impl\text{-}rel$
⟨*proof*⟩

definition *sm-update* **where**
$sm\text{-}update\ Q\ n\ sm = sm\text{-}it\ (\lambda\text{-}.\ True)\ (\lambda q\ sm.\ sm.update\ q\ n\ sm)\ Q\ sm$

lemma *sm-update-correct* :
assumes *invar-sm*: $sm.invar\ sm$
and *invar-Q*: $sp.invar\ Q$
shows $sm.invar\ (sm\text{-}update\ Q\ n\ sm) \wedge$
$sm.\alpha\ (sm\text{-}update\ Q\ n\ sm) = (\lambda q.\ if\ (q \in sp.\alpha\ Q)\ then\ Some\ n\ else\ sm.\alpha\ sm\ q)$
⟨*proof*⟩

definition *partition-impl-sing* :: $'q\text{-}set \Rightarrow ('im \times 'sm \times nat)$ **where**
$partition\text{-}impl\text{-}sing\ Q \equiv$
$(let\ QP = q\text{-}set\text{-}convert\ Q\ in$
$(im.sng\ 0\ QP,\ sm\text{-}update\ QP\ 0\ sm.empty,\ 1))$

lemma *partition-impl-sing-correct* :
assumes *Q-in-rel*: $(Q,\ QQ) \in build\text{-}rel\ s.\alpha\ s.invar$
shows $(partition\text{-}impl\text{-}sing\ Q,\ partition\text{-}map\text{-}sing\ QQ) \in partition\text{-}impl\text{-}rel$
⟨*proof*⟩

definition *partition-impl-init* **where**
$partition\text{-}impl\text{-}init\ F\ Q =$
$(let\ QF = s.diff\ Q\ F\ in$
$(if\ (s.isEmpty\ QF)\ then\ partition\text{-}impl\text{-}sing\ F\ else$
$(let\ FP = q\text{-}set\text{-}convert\ F\ in$
$let\ QFP = q\text{-}set\text{-}convert\ QF\ in$
$(im.update\ 0\ FP\ (im.sng\ 1\ QFP),\ sm\text{-}update\ FP\ 0\ (sm\text{-}update\ QFP\ 1\ sm.empty),\ 2))))$

lemma *partition-impl-init-correct* :
assumes *Q-in-rel*: $(Q,\ \mathcal{Q}\ \mathcal{A}) \in build\text{-}rel\ s.\alpha\ s.invar$
and *F-in-rel*: $(F,\ \mathcal{F}\ \mathcal{A}) \in build\text{-}rel\ s.\alpha\ s.invar$
shows $(partition\text{-}impl\text{-}init\ F\ Q,\ partition\text{-}map\text{-}init\ \mathcal{A}) \in partition\text{-}impl\text{-}rel$
⟨*proof*⟩

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-update pmin (*snd* (*snd P*)) (*fst* (*snd P*)), *Suc* (*snd* (*snd P*)))

lemma *partition-impl-split-correct* :
assumes *pmin-in-rel*: (*pmin*, *pmin'*) ∈ *build-rel sp.α sp.invar*
and *pmax-in-rel*: (*pmax*, *pmax'*) ∈ *build-rel sp.α sp.invar*
and *P-in-rel*: (*P*, *P'*) ∈ *partition-impl-rel*
shows (*partition-impl-split P i pmin pmax*,
partition-map-split P' i pmin' pmax') ∈ *partition-impl-rel*
⟨*proof*⟩

definition *Hopcroft-impl-split-count-set* :: ('*q* ⇒ *bool*) ⇒ '*q-set-p* ⇒ ('*q-set-p* × *nat* × '*q-set-p* × *nat*)
where
Hopcroft-impl-split-count-set P S =
split-it (λ-. *True*) (λ*q* (*pt'*, *pct'*, *pf'*, *pcf'*).
(*if* (*P q*) *then* (*sp.ins-dj q pt'*, *Suc pct'*, *pf'*, *pcf'*) *else*
(*pt'*, *pct'*, *sp.ins-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-impl-split-count-set P S in*
(*sp.invar pt* ∧ *sp.invar pf* ∧ *sp.α pt* = {*q*. *q* ∈ *sp.α S* ∧ *P q*} ∧
(*pct* = *card* (*sp.α pt*)) ∧ *sp.α pf* = {*q*. *q* ∈ *sp.α S* ∧ ¬(*P q*)} ∧
(*pcf* = *card* (*sp.α pf*)))
⟨*proof*⟩

lemma *Hopcroft-impl-split-count-set-correctD* :
assumes *Hopcroft-impl-split-count-set P S* = (*pt*, *pct*, *pf*, *pcf*)
and *sp.invar S*
shows (*sp.invar pt* ∧ *sp.invar pf* ∧ {*q*. *q* ∈ *sp.α S* ∧ *P q*} = *sp.α pt* ∧
(*card* (*sp.α pt*) = *pct*) ∧ {*q*. *q* ∈ *sp.α S* ∧ ¬(*P q*)} = *sp.α pf* ∧
(*card* (*sp.α pf*) = *pcf*))
⟨*proof*⟩

definition *Hopcroft-impl-step* **where**
Hopcroft-impl-step pre (*AL*::'*a list*) *P L* =
do {
let PS = *s-image* (λ*q*. (*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* (λ*q*. *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* (λ*a*. (*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*: $((P,L), (P',L' - \{(a,\ p)\})) \in (rprod\ partition\text{-}impl\text{-}rel\ (build\text{-}rel\ set\ distinct))$
  and *AL-OK*: $distinct\ AL \wedge set\ AL = \Sigma\ \mathcal{A}$
  and *pre-OK*: $(pre,\ pre') \in build\text{-}rel\ s.\alpha\ s.invar$
shows $Hopcroft\text{-}impl\text{-}step\ pre\ AL\ P\ L \leq \Downarrow(rprod\ partition\text{-}impl\text{-}rel\ (build\text{-}rel\ set\ distinct))$
  $(Hopcroft\text{-}map\text{-}step\ \mathcal{A}\ p\ a\ pre'\ P'\ L')$
⟨*proof*⟩

definition *Hopcroft-impl-f* **where**
$Hopcroft\text{-}impl\text{-}f\ pre\text{-}fun\ AL = (\lambda(P,\ L).$
 do {
  $let\ (a,i) = hd\ L;$
  $let\ pre = pre\text{-}fun\ a\ (the\ (im.lookup\ i\ (fst\ P)));$
  $(P',\ L') \leftarrow Hopcroft\text{-}impl\text{-}step\ pre\ AL\ P\ (tl\ L);$
  $RETURN\ (P',\ L')$
 })

lemma *Hopcroft-impl-f-correct* :
assumes *PL-OK*: $((P,L), (P',L')) \in (rprod\ partition\text{-}impl\text{-}rel\ (build\text{-}rel\ set\ distinct))$
  and *AL-OK*: $distinct\ AL \wedge set\ AL = \Sigma\ \mathcal{A}$
  and *pre-fun-OK*: $\bigwedge a\ S.\ [\![a \in \Sigma\ \mathcal{A};\ sp.invar\ S]\!] \Longrightarrow$
    $(s.invar\ (pre\text{-}fun\ a\ S) \wedge (s.\alpha\ (pre\text{-}fun\ a\ S) =$
      $\{q\ .\ \exists q'.\ (q,\ a,\ q') \in \Delta\ \mathcal{A} \wedge q' \in sp.\alpha\ S\}))$
shows $(Hopcroft\text{-}impl\text{-}f\ pre\text{-}fun\ AL\ (P,L)) \leq \Downarrow(rprod\ partition\text{-}impl\text{-}rel\ (build\text{-}rel\ set\ distinct))$
  $(Hopcroft\text{-}map\text{-}f\ \mathcal{A}\ (P',\ L'))$
⟨*proof*⟩

definition *Hopcroft-impl* **where**
 $Hopcroft\text{-}impl\ Q\ F\ AL\ pre\text{-}fun =$
  $(if\ (s.isEmpty\ Q)\ then\ RETURN\ (partition\text{-}impl\text{-}empty)\ else\ ($
   $if\ (s.isEmpty\ F)\ then\ RETURN\ (partition\text{-}impl\text{-}sing\ Q)\ else\ ($
    do {
     $(P,\ \text{-}) \leftarrow WHILET\ (\lambda PL.\ (snd\ PL \neq []))$
       $(Hopcroft\text{-}impl\text{-}f\ pre\text{-}fun\ AL)$
       $(partition\text{-}impl\text{-}init\ F\ Q,\ map\ (\lambda a.\ (a,\ 0))\ AL);$
     $RETURN\ P$
    }))))

lemma *Hopcroft-impl-correct* :
assumes *Q-in-rel*: $(Q,\ \mathcal{Q}\ \mathcal{A}) \in build\text{-}rel\ s.\alpha\ s.invar$
  and *F-in-rel*: $(F,\ \mathcal{F}\ \mathcal{A}) \in build\text{-}rel\ s.\alpha\ s.invar$
  and *AL-OK*: $distinct\ AL \wedge set\ AL = \Sigma\ \mathcal{A}$
  and *pre-fun-OK*: $\bigwedge a\ S.\ [\![a \in \Sigma\ \mathcal{A};\ sp.invar\ S]\!] \Longrightarrow$
    $(s.invar\ (pre\text{-}fun\ a\ S) \wedge (s.\alpha\ (pre\text{-}fun\ a\ S) =$
      $\{q\ .\ \exists q'.\ (q,\ a,\ q') \in \Delta\ \mathcal{A} \wedge q' \in sp.\alpha\ S\}))$
shows $Hopcroft\text{-}impl\ Q\ F\ AL\ pre\text{-}fun \leq \Downarrow partition\text{-}impl\text{-}rel\ (Hopcroft\text{-}map\ \mathcal{A})$
⟨*proof*⟩

lemma *set-iteratori-ns-OK* :
assumes *invar-ns*: $n.invar\ ns$
shows $set\text{-}iteratori\ (\lambda c\ f.\ ns\text{-}it\ c\ f\ ns)\ (n.\alpha\ ns)$
⟨*proof*⟩

schematic-lemma *Hopcroft-code-correct-aux* :
shows $RETURN\ ?code \leq Hopcroft\text{-}impl\ Q\ F\ AL\ pre\text{-}fun$
⟨*proof*⟩

definition (**in** −) *Hopcroft-code* **where**
$Hopcroft\text{-}code$
  $(s\text{-}ops :: ('q,\ 'q\text{-}set,\ \text{-})\ set\text{-}ops\text{-}scheme)$

(*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* ≠ []) (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
⟨*proof*⟩

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*⟩

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*, $\mathcal{Q}$ $\mathcal{A}$) ∈ *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 *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*⟩

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*⟩

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},
      I = {*pres-NFA-state-int l*},
      F = {*pres-NFA-state-int m* |*m.*
        dioph-is-node ks l m ∧ 0 ≤ m ∧ (ineq ∨ 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*⟩

lemma *pres-DFA-eq-ineq---is-well-formed* :
  DFA (pres-DFA-eq-ineq i n ks l)
⟨*proof*⟩

interpretation *pres-DFA-eq-ineq* :
  DFA pres-DFA-eq-ineq i n ks l
⟨*proof*⟩

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 ∧ length bs = n) then
    Some (pres-DFA-eq-ineq-trans-fun i ks q bs)
   else None)
⟨*proof*⟩

lemma *pres-DFA-eq-ineq-reach-error* :
list-all (is-alph n) bss ⟹
 DLTS-reach (δ (pres-DFA-eq-ineq i n ks l)) pres-NFA-state-error bss = Some (pres-NFA-state-error)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma *pres-DFA-eq-ineq-reach-ineq* :
assumes *pres-NFA-state-int j ∈ 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*
⟨*proof*⟩

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*⟩

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*⟩

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) ⟷ q = pres-NFA-state-error
⟨*proof*⟩

lemma *pres-NFA-state-nat-neq-0* [*simp*] :
  (pres-NFA-state-to-nat q = Suc m) ⟷ q = pres-NFA-state-int (int-decode m)
⟨*proof*⟩

lemma *inj-pres-NFA-state-to-nat*:
  inj-on pres-NFA-state-to-nat S
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

lemma *pres-DFA-labels-tl---NFA-accept* :
assumes *wf-A*: *NFA* $\mathcal{A}$
and Σ-*A*: $\bigwedge$*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*))
⟨*proof*⟩

definition *pres-DFA-exists* ::
nat ⇒ *pres-NFA* ⇒ *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 $\mathcal{A}$ ⟹ *NFA* (*pres-DFA-exists n* $\mathcal{A}$)
⟨*proof*⟩

lemma *pres-DFA-exists---NFA-accept* :
assumes *wf-A*: *NFA* $\mathcal{A}$
and Σ-*A*: $\bigwedge$*bs*. *bs* ∈ Σ $\mathcal{A}$ ⟹ *bs* ≠ []
shows *NFA-accept* (*pres-DFA-exists n* $\mathcal{A}$) *bss* ⟷
(∃ *bs m*. *length bs* = *length bss* + *m* ∧ *NFA-accept* $\mathcal{A}$ (*insertll 0 bs* (*bss* @ *zeros m n*)))
⟨*proof*⟩

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*⟩

lemma *pres-DFA-exists---NFA-accept---nats* :
assumes *wf-A*: *NFA* $\mathcal{A}$
and Σ-*A*: Σ $\mathcal{A}$ = {*bs*. *length bs* = *Suc n*}

and *acc-A*: $\bigwedge bss.$ *list-all* (*is-alph* (*Suc n*)) *bss* $\Longrightarrow$ *NFA-accept* $\mathcal{A}$ *bss* $\longleftrightarrow$ *P* (*nats-of-boolss* (*Suc n*) *bss*)
and *bss-in*: *list-all* (*is-alph n*) *bss*
shows *NFA-accept* (*pres-DFA-exists n* $\mathcal{A}$) *bss* =
($\exists x.$ *P* (*x* # *nats-of-boolss n bss*))
⟨*proof*⟩

definition *pres-DFA-exists-min* **where**
pres-DFA-exists-min n $\mathcal{A}$ = *NFA-right-quotient-lists* (
NFA-minimise (*pres-DFA-labels-tl* $\mathcal{A}$)) {*replicate n False*}

find-theorems *name*: *NFA-right-quotient name*: *well-formed*
lemma *pres-DFA-exists-min---well-formed* :
assumes *wf-A*: *NFA* $\mathcal{A}$
shows *DFA* (*pres-DFA-exists-min n* $\mathcal{A}$)
⟨*proof*⟩

lemma *pres-DFA-exists-min---well-formed-DFA* :
DFA $\mathcal{A}$ $\Longrightarrow$ *DFA* (*pres-DFA-exists-min n* $\mathcal{A}$)
⟨*proof*⟩

lemma *pres-DFA-exists-min---NFA-accept* :
assumes *wf-A*: *DFA* $\mathcal{A}$
shows *NFA-accept* (*pres-DFA-exists-min n* $\mathcal{A}$) *bss* $\longleftrightarrow$
NFA-accept (*pres-DFA-exists n* $\mathcal{A}$) *bss*
⟨*proof*⟩

lemma *pres-DFA-exists-min---NFA-accept---nats* :
assumes *wf-A*: *DFA* $\mathcal{A}$
and Σ-*A*: Σ $\mathcal{A}$ = {*bs*. *length bs* = *Suc n*}
and *acc-A*: $\bigwedge bss.$ *list-all* (*is-alph* (*Suc n*)) *bss* $\Longrightarrow$ *NFA-accept* $\mathcal{A}$ *bss* $\longleftrightarrow$ *P* (*nats-of-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-of-boolss n bss*))
⟨*proof*⟩

lemma Σ-*pres-DFA-exists-min* :
NFA $\mathcal{A}$ $\Longrightarrow$ Σ (*pres-DFA-exists-min n* $\mathcal{A}$) = *tl* ‘ Σ $\mathcal{A}$
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

lemma Σ-*image-tl* : *tl* ‘ {*bs*::*'a list*. *length bs* = *Suc n*} = {*bs*. *length bs* = *n*}
⟨*proof*⟩

lemma *pres-DFA-forall-min---NFA-accept---nats* :
fixes $\mathcal{A}$ :: *pres-NFA*
assumes *wf-A*: *DFA* $\mathcal{A}$
and Σ-*A*: Σ $\mathcal{A}$ = {*bs*. *length bs* = *Suc n*}
and *acc-A*: $\bigwedge bss.$ *list-all* (*is-alph* (*Suc n*)) *bss* $\Longrightarrow$ *NFA-accept* $\mathcal{A}$ *bss* $\longleftrightarrow$ *P* (*nats-of-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-of-boolss n bss*))
⟨*proof*⟩

lemma Σ-*pres-DFA-forall-min* :
NFA $\mathcal{A} \Longrightarrow \Sigma$ (*pres-DFA-forall-min n* $\mathcal{A}$) = *tl* ‘ Σ $\mathcal{A}$
⟨*proof*⟩

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*⟩

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*
(λ*q*. *case q of pres-NFA-state-error* ⇒ *False*
| *pres-NFA-state-int m* ⇒ (*0* ≤ *m*))

else
(λ*q*. *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*⟩

definition *pres-DFA-exists-min-impl* **where**
pres-DFA-exists-min-impl A AA =
right-quotient-lists (*list-all* (λ*b*. ¬ *b*)) (*minimise-Brzozowski* (*rename-labels-gen AA A tl*))

lemma *im-tl-eq*: *tl* ‘ {*bl*. *length bl* = *Suc n*} = {*bl*. *length bl* = *n*}
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *pres-DFA-exists-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-exists-min-impl A AA*)) (*pres-DFA-exists-min n* $\mathcal{A}$)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* $\mathcal{A}$)
⟨*proof*⟩

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* ⇒ *pf* ⇒ *'cache* ⇒ *'nfa* × *'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∧ 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∨ 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⟶ 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)) ∧
      nfa.invar (fst (nfa-of-pf n p c)) ∧
      NFA-isomorphic-wf (nfa.α (fst (nfa-of-pf n p c)))
                        (DFA-of-pf n p)
⟨proof⟩


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)
⟨proof⟩

lemma eval-pf-impl :
  eval-pf pf [] = accept (pf-to-nfa 0 pf) []
⟨proof⟩


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 (λl. True # l) bl) (s-image (λl. 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}
  ⟨proof⟩

  lemma  presburger-label-set-cache-OK :
    presburger-label-set-cache s.α s.invar c-α c-invar m.empty
    ⟨proof⟩
end

end
```