Converting Linear Temporal Logic to Deterministic (Generalized) Rabin Automata

Salomon Sickert

May 26, 2024

Abstract

Recently a new method directly translating linear temporal logic (LTL) formulas to deterministic (generalized) Rabin automata was described in [1].

Compared to the existing approaches of constructing a non-deterministic Buechi-automaton in the first step and then applying a determinization procedure (e.g. some variant of Safra's construction) in a second step, this new approach preservers a relation between the formula and the states of the resulting automaton. While the old approach produced a monolithic structure, the new method is compositional. Furthermore it was shown in some cases the resulting automata were much smaller than the automata generated by existing approaches. In order to guarantee the correctness of the construction this entry contains a complete formalisation and verification of the translation. Furthermore from this basis executable code is generated.

Contents

1 Auxiliary Facts

theory *Preliminaries2*
imports *Main HOL−Library.Infinite-Set*
begin

1.1 Finite and Infinite Sets

lemma *finite-product*:
assumes *fst*: *finite* (*fst* ‘ *A*)
and *snd*: *finite* (*snd* ‘ *A*)
shows *finite A*
⟨*proof*⟩

1.2 Cofinite Filters

lemma *almost-all-commutative*:
finite $S \Longrightarrow (\forall x \in S.\ \forall_\infty i.\ P\ x\ (i::nat)) = (\forall_\infty i.\ \forall x \in S.\ P\ x\ i)$
⟨*proof*⟩

lemma *almost-all-commutative'*:
finite $S \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow \forall_\infty i.\ P\ x\ (i::nat)) \Longrightarrow (\forall_\infty i.\ \forall x \in S.\ P\ x\ i)$
⟨*proof*⟩

fun *index*
where
index $P = (if\ \forall_\infty i.\ P\ i\ then\ Some\ (LEAST\ i.\ \forall j \geq i.\ P\ j)\ else\ None)$

lemma *index-properties*:
fixes $i :: nat$
shows $index\ P = Some\ i \Longrightarrow 0 < i \Longrightarrow \neg\ P\ (i - 1)$
and $index\ P = Some\ i \Longrightarrow j \geq i \Longrightarrow P\ j$
⟨*proof*⟩

end

2 Auxiliary Map Facts

theory *Map2*
imports *Main*
begin

lemma *map-of-tabulate*:
$map\text{-}of\ (map\ (\lambda x.\ (x,\ f\ x))\ xs)\ x \neq None \longleftrightarrow x \in set\ xs$
⟨*proof*⟩

lemma *map-of-tabulate-simp*:
$map\text{-}of\ (map\ (\lambda x.\ (x,\ f\ x))\ xs)\ x = (if\ x \in set\ xs\ then\ Some\ (f\ x)\ else\ None)$
⟨*proof*⟩

lemma *dom-map-update*:
$dom\ (m\ (k \mapsto v)) = dom\ m \cup \{k\}$
⟨*proof*⟩

lemma *map-equal*:
$dom\ m = dom\ m' \Longrightarrow (\bigwedge x.\ x \in dom\ m \Longrightarrow m\ x = m'\ x) \Longrightarrow m = m'$
⟨*proof*⟩

lemma *map-reduce*:
assumes $dom\ m = \{a\} \cup B$
shows $\exists m'.\ dom\ m' = B \wedge (\forall x \in B.\ m\ x = m'\ x)$
⟨*proof*⟩

end

3 Auxiliary Mapping Facts

theory *Mapping2*
imports *Main Map2 HOL−Library.Mapping*
begin

lemma *lookup-delete*:
$Mapping.lookup\ (Mapping.delete\ k\ m)\ k = None$
⟨*proof*⟩

lemma *lookup-tabulate*:
$Mapping.lookup\ (Mapping.tabulate\ xs\ f)\ x = (if\ x \in set\ xs\ then\ Some\ (f\ x)\ else\ None)$
⟨*proof*⟩

lemma *lookup-tabulate-Some*:
$x \in set\ xs \Longrightarrow the\ (Mapping.lookup\ (Mapping.tabulate\ xs\ f)\ x) = f\ x$
⟨*proof*⟩

lemma *finite-keys-tabulate*:
$finite\ (Mapping.keys\ (Mapping.tabulate\ xs\ f))$
⟨*proof*⟩

lemma *keys-empty-iff-map-empty*:
$Mapping.keys\ m = \{\} \longleftrightarrow m = Mapping.empty$
⟨*proof*⟩

lemma *mapping-equal*:
$Mapping.keys\ m = Mapping.keys\ m' \Longrightarrow (\bigwedge x.\ x \in Mapping.keys\ m \Longrightarrow Mapping.lookup\ m\ x = Mapping.lookup\ m'\ x) \Longrightarrow m = m'$
⟨*proof*⟩

fun *mapping-generator* :: $('a \Rightarrow 'b\ list) \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ mapping\ set$
where
$mapping\text{-}generator\ V\ [] = \{Mapping.empty\}$

| *mapping-generator V (k#ks)* = {*Mapping.update k v m* | *v m.* *v* ∈ *set* (*V k*) ∧ *m* ∈ *mapping-generator V ks*}

fun *mapping-generator-list* :: (′*a* ⇒ ′*b list*) ⇒ ′*a list* ⇒ (′*a*, ′*b*) *mapping list*
where
mapping-generator-list V [] = [*Mapping.empty*]
| *mapping-generator-list V* (*k*#*ks*) = *concat* (*map* (λ*m*. *map* (λ*v*. *Mapping.update k v m*) (*V k*)) (*mapping-generator-list V ks*))

lemma *mapping-generator-code* [*code*]:
mapping-generator V K = *set* (*mapping-generator-list V K*)
⟨*proof*⟩

lemma *mapping-generator-set-eq*:
mapping-generator V K = {*m*. *Mapping.keys m* = *set K* ∧ (∀ *k* ∈ (*set K*). *the* (*Mapping.lookup m k*) ∈ *set* (*V k*))}
⟨*proof*⟩

end

4 Deterministic Transition Systems

theory *DTS*
imports *Main HOL−Library.Omega-Words-Fun Auxiliary/Mapping2 KBPs.DFS*
begin

— DTS are realised by functions

type-synonym (′*a*, ′*b*) *DTS* = ′*a* ⇒ ′*b* ⇒ ′*a*
type-synonym (′*a*, ′*b*) *transition* = (′*a* × ′*b* × ′*a*)

4.1 Infinite Runs

fun *run* :: (′*q*,′*a*) *DTS* ⇒ ′*q* ⇒ ′*a word* ⇒ ′*q word*
where
run δ *q*$_0$ *w* 0 = *q*$_0$
| *run* δ *q*$_0$ *w* (*Suc i*) = δ (*run* δ *q*$_0$ *w i*) (*w i*)

fun *run*$_t$ :: (′*q*,′*a*) *DTS* ⇒ ′*q* ⇒ ′*a word* ⇒ (′*q* ∗ ′*a* ∗ ′*q*) *word*
where
run$_t$ δ *q*$_0$ *w i* = (*run* δ *q*$_0$ *w i*, *w i*, *run* δ *q*$_0$ *w* (*Suc i*))

lemma *run-foldl*:
run Δ *q*$_0$ *w i* = *foldl* Δ *q*$_0$ (*map w* [0..<*i*])

⟨*proof*⟩

lemma *run$_t$-foldl*:
run$_t$ Δ q_0 w i = (*foldl* Δ q_0 (*map* w [*0..<i*]), w i, *foldl* Δ q_0 (*map* w [*0..<Suc i*]))
⟨*proof*⟩

4.2 Reachable States and Transitions

definition *reach* :: *'a set* ⇒ (*'b*, *'a*) *DTS* ⇒ *'b* ⇒ *'b set*
where
reach Σ δ q_0 = {*run* δ q_0 w n | w n. *range* w ⊆ Σ}

definition *reach$_t$* :: *'a set* ⇒ (*'b*, *'a*) *DTS* ⇒ *'b* ⇒ (*'b*, *'a*) *transition set*
where
reach$_t$ Σ δ q_0 = {*run$_t$* δ q_0 w n | w n. *range* w ⊆ Σ}

lemma *reach-foldl-def*:
assumes Σ ≠ {}
shows *reach* Σ δ q_0 = {*foldl* δ q_0 w | w. *set* w ⊆ Σ}
⟨*proof*⟩

lemma *reach$_t$-foldl-def*:
reach$_t$ Σ δ q_0 = {(*foldl* δ q_0 w, ν, *foldl* δ q_0 (w@[ν])) | w ν. *set* w ⊆ Σ ∧ ν ∈ Σ} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *reach-card-0*:
assumes Σ ≠ {}
shows *infinite* (*reach* Σ δ q_0) ⟷ *card* (*reach* Σ δ q_0) = *0*
⟨*proof*⟩

lemma *reach$_t$-card-0*:
assumes Σ ≠ {}
shows *infinite* (*reach$_t$* Σ δ q_0) ⟷ *card* (*reach$_t$* Σ δ q_0) = *0*
⟨*proof*⟩

4.2.1 Relation to runs

lemma *run-subseteq-reach*:
assumes *range* w ⊆ Σ
shows *range* (*run* δ q_0 w) ⊆ *reach* Σ δ q_0
and *range* (*run$_t$* δ q_0 w) ⊆ *reach$_t$* Σ δ q_0
⟨*proof*⟩

lemma *limit-subseteq-reach*:
assumes *range* $w \subseteq \Sigma$
shows *limit* (*run* δ q_0 w) $\subseteq$ *reach* Σ δ q_0
and *limit* (run_t δ q_0 w) $\subseteq$ $reach_t$ Σ δ q_0
⟨*proof*⟩

lemma run_t-*finite*:
assumes *finite* (*reach* Σ δ q_0)
assumes *finite* Σ
assumes *range* $w \subseteq \Sigma$
defines $r \equiv run_t$ δ q_0 w
shows *finite* (*range* r)
⟨*proof*⟩

4.2.2 Compute reach Using DFS

definition $Q_L :: 'a$ *list* $\Rightarrow$ $('b, 'a)$ *DTS* $\Rightarrow$ $'b \Rightarrow 'b$ *set*
where
Q_L Σ δ q_0 = (*if* $\Sigma \neq []$ *then gen-dfs* (λq. *map* (δ q) Σ) *Set.insert* ($\in$) {} $[q_0]$ *else* {})

definition *list-dfs* :: (($'a$, $'b$) *transition* $\Rightarrow$ ($'a$, $'b$) *transition list*) $\Rightarrow$ ($'a$, $'b$) *transition list* => ($'a$, $'b$) *transition list* => ($'a$, $'b$) *transition list*
where
list-dfs succ S start $\equiv$ *gen-dfs succ List.insert* (λx *xs*. $x \in$ *set xs*) *S start*

definition $\delta_L :: 'a$ *list* $\Rightarrow$ $('b, 'a)$ *DTS* $\Rightarrow$ $'b \Rightarrow ('b, 'a)$ *transition set*
where
δ_L Σ δ q_0 = *set* (
let
start = *map* ($\lambda\nu$. (q_0, ν, δ q_0 ν)) Σ;
succ = λ(-, -, q). (*map* ($\lambda\nu$. (q, ν, δ q ν)) Σ)
in
(*list-dfs succ* [] *start*))

lemma Q_L-*reach*:
assumes *finite* (*reach* (*set* Σ) δ q_0)
shows Q_L Σ δ q_0 = *reach* (*set* Σ) δ q_0
⟨*proof*⟩

lemma δ_L-*reach*:
assumes *finite* ($reach_t$ (*set* Σ) δ q_0)
shows δ_L Σ δ q_0 = $reach_t$ (*set* Σ) δ q_0

⟨*proof*⟩

lemma *reach-reach$_t$-fst*:
reach Σ δ q_0 = *fst* ‘ *reach$_t$* Σ δ q_0
⟨*proof*⟩

lemma *finite-reach*:
finite (*reach$_t$* Σ δ q_0) ⟹ *finite* (*reach* Σ δ q_0)
⟨*proof*⟩

lemma *finite-reach$_t$*:
assumes *finite* (*reach* Σ δ q_0)
assumes *finite* Σ
shows *finite* (*reach$_t$* Σ δ q_0)
⟨*proof*⟩

lemma Q_L*-eq-*δ_L:
assumes *finite* (*reach$_t$* (*set* Σ) δ q_0)
shows Q_L Σ δ q_0 = *fst* ‘ (δ_L Σ δ q_0)
⟨*proof*⟩

4.3 Product of DTS

fun *simple-product* :: ('*a*, '*c*) *DTS* ⇒ ('*b*, '*c*) *DTS* ⇒ ('*a* × '*b*, '*c*) *DTS* (- × -)
where
$\delta_1 \times \delta_2$ = ($\lambda(q_1, q_2)\ \nu$. ($\delta_1\ q_1\ \nu$, $\delta_2\ q_2\ \nu$))

lemma *simple-product-run*:
fixes $\delta_1\ \delta_2\ w\ q_1\ q_2$
defines $\varrho \equiv$ *run* ($\delta_1 \times \delta_2$) (q_1, q_2) w
defines $\varrho_1 \equiv$ *run* $\delta_1\ q_1\ w$
defines $\varrho_2 \equiv$ *run* $\delta_2\ q_2\ w$
shows $\varrho\ i$ = ($\varrho_1\ i$, $\varrho_2\ i$)
⟨*proof*⟩

theorem *finite-reach-simple-product*:
assumes *finite* (*reach* Σ $\delta_1\ q_1$)
assumes *finite* (*reach* Σ $\delta_2\ q_2$)
shows *finite* (*reach* Σ ($\delta_1 \times \delta_2$) (q_1, q_2))
⟨*proof*⟩

4.4 (Generalised) Product of DTS

fun *product* :: $('a \Rightarrow ('b, 'c)\ DTS) \Rightarrow ('a \rightharpoonup 'b, 'c)\ DTS\ (\Delta_\times)$
where
$\Delta_\times\ \delta_m = (\lambda q\ \nu.\ (\lambda x.\ case\ q\ x\ of\ None \Rightarrow None \mid Some\ y \Rightarrow Some\ (\delta_m\ x\ y\ \nu)))$

lemma *product-run-None*:
fixes $\iota_m\ \delta_m\ w$
defines $\varrho \equiv run\ (\Delta_\times\ \delta_m)\ \iota_m\ w$
assumes $\iota_m\ k = None$
shows $\varrho\ i\ k = None$
⟨*proof*⟩

lemma *product-run-Some*:
fixes $\iota_m\ \delta_m\ w\ q_0\ k$
defines $\varrho \equiv run\ (\Delta_\times\ \delta_m)\ \iota_m\ w$
defines $\varrho' \equiv run\ (\delta_m\ k)\ q_0\ w$
assumes $\iota_m\ k = Some\ q_0$
shows $\varrho\ i\ k = Some\ (\varrho'\ i)$
⟨*proof*⟩

theorem *finite-reach-product*:
assumes *finite* $(dom\ \iota_m)$
assumes $\bigwedge x.\ x \in dom\ \iota_m \Longrightarrow finite\ (reach\ \Sigma\ (\delta_m\ x)\ (the\ (\iota_m\ x)))$
shows *finite* $(reach\ \Sigma\ (\Delta_\times\ \delta_m)\ \iota_m)$
⟨*proof*⟩

4.5 Simple Product Construction Helper Functions and Lemmas

fun *embed-transition-fst* :: $('a, 'c)\ transition \Rightarrow ('a \times 'b, 'c)\ transition\ set$
where
$embed\text{-}transition\text{-}fst\ (q, \nu, q') = \{((q, x), \nu, (q', y)) \mid x\ y.\ True\}$

fun *embed-transition-snd* :: $('b, 'c)\ transition \Rightarrow ('a \times 'b, 'c)\ transition\ set$
where
$embed\text{-}transition\text{-}snd\ (q, \nu, q') = \{((x, q), \nu, (y, q')) \mid x\ y.\ True\}$

lemma *embed-transition-snd-unfold*:
$embed\text{-}transition\text{-}snd\ t = \{((x, fst\ t), fst\ (snd\ t), (y, snd\ (snd\ t))) \mid x\ y.\ True\}$
⟨*proof*⟩

fun *project-transition-fst* :: $('a \times 'b, 'c)$ *transition* $\Rightarrow$ $('a, 'c)$ *transition*
where
project-transition-fst $(x, \nu, y) = (fst\ x, \nu, fst\ y)$

fun *project-transition-snd* :: $('a \times 'b, 'c)$ *transition* $\Rightarrow$ $('b, 'c)$ *transition*
where
project-transition-snd $(x, \nu, y) = (snd\ x, \nu, snd\ y)$

lemma
fixes $\delta_1\ \delta_2\ w\ q_1\ q_2$
defines $\varrho \equiv run_t\ (\delta_1 \times \delta_2)\ (q_1, q_2)\ w$
defines $\varrho_1 \equiv run_t\ \delta_1\ q_1\ w$
defines $\varrho_2 \equiv run_t\ \delta_2\ q_2\ w$
shows *product-run-project-fst*: *project-transition-fst* $(\varrho\ i) = \varrho_1\ i$
and *product-run-project-snd*: *project-transition-snd* $(\varrho\ i) = \varrho_2\ i$
and *product-run-embed-fst*: $\varrho\ i \in$ *embed-transition-fst* $(\varrho_1\ i)$
and *product-run-embed-snd*: $\varrho\ i \in$ *embed-transition-snd* $(\varrho_2\ i)$
⟨*proof*⟩

lemma
fixes $\delta_1\ \delta_2\ w\ q_1\ q_2$
defines $\varrho \equiv run_t\ (\delta_1 \times \delta_2)\ (q_1, q_2)\ w$
defines $\varrho_1 \equiv run_t\ \delta_1\ q_1\ w$
defines $\varrho_2 \equiv run_t\ \delta_2\ q_2\ w$
assumes *finite* (*range* ϱ)
shows *product-run-finite-fst*: *finite* (*range* ϱ_1)
and *product-run-finite-snd*: *finite* (*range* ϱ_2)
⟨*proof*⟩

lemma
fixes $\delta_1\ \delta_2\ w\ q_1\ q_2$
defines $\varrho \equiv run_t\ (\delta_1 \times \delta_2)\ (q_1, q_2)\ w$
defines $\varrho_1 \equiv run_t\ \delta_1\ q_1\ w$
assumes *finite* (*range* ϱ)
shows *product-run-project-limit-fst*: *project-transition-fst* ‘ *limit* ϱ = *limit* ϱ_1
and *product-run-embed-limit-fst*: *limit* $\varrho \subseteq \bigcup$ (*embed-transition-fst* ‘ (*limit* ϱ_1))
⟨*proof*⟩

lemma
fixes $\delta_1\ \delta_2\ w\ q_1\ q_2$
defines $\varrho \equiv run_t\ (\delta_1 \times \delta_2)\ (q_1, q_2)\ w$
defines $\varrho_2 \equiv run_t\ \delta_2\ q_2\ w$

assumes *finite* (*range* ϱ)
shows *product-run-project-limit-snd*: *project-transition-snd* ‘ *limit* ϱ = *limit* ϱ_2
and *product-run-embed-limit-snd*: *limit* $\varrho \subseteq \bigcup$ (*embed-transition-snd* ‘ (*limit* ϱ_2))
⟨*proof*⟩

lemma
fixes δ_1 δ_2 w q_1 q_2
defines $\varrho \equiv run_t$ ($\delta_1 \times \delta_2$) (q_1, q_2) w
defines $\varrho_1 \equiv run_t$ δ_1 q_1 w
defines $\varrho_2 \equiv run_t$ δ_2 q_2 w
assumes *finite* (*range* ϱ)
shows *product-run-embed-limit-finiteness-fst*: *limit* $\varrho \cap (\bigcup$ (*embed-transition-fst* ‘ S)) = {} $\longleftrightarrow$ *limit* $\varrho_1 \cap S$ = {} (**is** *?thesis1*)
and *product-run-embed-limit-finiteness-snd*: *limit* $\varrho \cap (\bigcup$ (*embed-transition-snd* ‘ S')) = {} $\longleftrightarrow$ *limit* $\varrho_2 \cap S'$ = {} (**is** *?thesis2*)
⟨*proof*⟩

4.6 Product Construction Helper Functions and Lemmas

fun *embed-transition* :: $'a \Rightarrow$ ($'b$, $'c$) *transition* $\Rightarrow$ ($'a \rightharpoonup 'b$, $'c$) *transition set* ($\uparrow$-)
where
$\uparrow_x$ (q, ν, q') = {(m, ν, m') | m m'. m x = *Some* q $\wedge$ m' x = *Some* q'}

fun *project-transition* :: $'a \Rightarrow$ ($'a \rightharpoonup 'b$, $'c$) *transition* $\Rightarrow$ ($'b$, $'c$) *transition* ($\downarrow$-)
where
$\downarrow_x$ (m, ν, m') = (*the* (m x), ν, *the* (m' x))

fun *embed-pair* :: $'a \Rightarrow$ (($'b$, $'c$) *transition set* $\times$ ($'b$, $'c$) *transition set*) $\Rightarrow$ (($'a \rightharpoonup 'b$, $'c$) *transition set* $\times$ ($'a \rightharpoonup 'b$, $'c$) *transition set*) ($\uparrow$-)
where
$\uparrow_x$ (S, S') = ($\bigcup(\uparrow_x$ ‘ S), $\bigcup(\uparrow_x$ ‘ S'))

fun *project-pair* :: $'a \Rightarrow$ (($'a \rightharpoonup 'b$, $'c$) *transition set* $\times$ ($'a \rightharpoonup 'b$, $'c$) *transition set*) $\Rightarrow$ (($'b$, $'c$) *transition set* $\times$ ($'b$, $'c$) *transition set*) ($\downarrow$-)
where
$\downarrow_x$ (S, S') = ($\downarrow_x$ ‘ S, $\downarrow_x$ ‘ S')

lemma *embed-transition-unfold*:
embed-transition x t = {(m, *fst* (*snd* t), m') | m m'. m x = *Some* (*fst* t) $\wedge$ m' x = *Some* (*snd* (*snd* t))}

⟨*proof*⟩

lemma
fixes ι_m δ_m w q_0
fixes $x :: 'a$
defines $\varrho \equiv run_t\ (\Delta_\times\ \delta_m)\ \iota_m\ w$
defines $\varrho' \equiv run_t\ (\delta_m\ x)\ q_0\ w$
assumes $\iota_m\ x = Some\ q_0$
shows *product-run-project*: $\downarrow_x\ (\varrho\ i) = \varrho'\ i$
and *product-run-embed*: $\varrho\ i \in \uparrow_x\ (\varrho'\ i)$
⟨*proof*⟩

lemma
fixes ι_m δ_m w q_0 x
defines $\varrho \equiv run_t\ (\Delta_\times\ \delta_m)\ \iota_m\ w$
defines $\varrho' \equiv run_t\ (\delta_m\ x)\ q_0\ w$
assumes $\iota_m\ x = Some\ q_0$
assumes *finite* (*range* ϱ)
shows *product-run-project-limit*: $\downarrow_x\ `\ limit\ \varrho = limit\ \varrho'$
and *product-run-embed-limit*: $limit\ \varrho \subseteq \bigcup\ (\uparrow_x\ `\ (limit\ \varrho'))$
⟨*proof*⟩

lemma *product-run-embed-limit-finiteness*:
fixes ι_m δ_m w q_0 k
defines $\varrho \equiv run_t\ (\Delta_\times\ \delta_m)\ \iota_m\ w$
defines $\varrho' \equiv run_t\ (\delta_m\ k)\ q_0\ w$
assumes $\iota_m\ k = Some\ q_0$
assumes *finite* (*range* ϱ)
shows $limit\ \varrho \cap (\bigcup\ (\uparrow_k\ `\ S)) = \{\} \longleftrightarrow limit\ \varrho' \cap S = \{\}$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

4.7 Transfer Rules

context includes *lifting-syntax*
begin

lemma *product-parametric* [*transfer-rule*]:
$((A\ ===>\ B\ ===>\ C\ ===>\ B)\ ===>\ (A\ ===>\ rel\text{-}option\ B)\ ===>\ C\ ===>\ A\ ===>\ rel\text{-}option\ B)\ product\ product$
⟨*proof*⟩

lemma *run-parametric* [*transfer-rule*]:
$((A\ ===>\ B\ ===>\ A)\ ===>\ A\ ===>\ ((=)\ ===>\ B)\ ===>\ (=)$

===> A) run run
⟨proof⟩

lemma *reach-parametric* [*transfer-rule*]:
assumes *bi-total B*
assumes *bi-unique B*
shows (*rel-set B* ===> (*A* ===> *B* ===> *A*) ===> *A* ===> *rel-set A*) *reach reach*
⟨proof⟩

end

4.8 Lift to Mapping

lift-definition *product-abs* :: ('*a* ⇒ ('*b*, '*c*) *DTS*) ⇒ (('*a*, '*b*) *mapping*, '*c*) *DTS* (↑Δ×) **is** *product*
parametric *product-parametric* ⟨*proof*⟩

lemma *product-abs-run-None*:
Mapping.lookup ι_m *k* = *None* ⟹ *Mapping.lookup* (*run* (↑Δ× δ_m) ι_m *w i*) *k* = *None*
⟨*proof*⟩

lemma *product-abs-run-Some*:
Mapping.lookup ι_m *k* = *Some* q_0 ⟹ *Mapping.lookup* (*run* (↑Δ× δ_m) ι_m *w i*) *k* = *Some* (*run* (δ_m *k*) q_0 *w i*)
⟨*proof*⟩

theorem *finite-reach-product-abs*:
assumes *finite* (*Mapping.keys* ι_m)
assumes ⋀*x*. *x* ∈ (*Mapping.keys* ι_m) ⟹ *finite* (*reach* Σ (δ_m *x*) (*the* (*Mapping.lookup* ι_m *x*)))
shows *finite* (*reach* Σ (↑Δ× δ_m) ι_m)
⟨*proof*⟩

end

5 Mojmir Automata (Without Final States)

theory *Semi-Mojmir*
imports *Main Auxiliary/Preliminaries2 DTS*
begin

5.1 Definitions

locale *semi-mojmir-def* =
  fixes
    — Alphapet
    Σ :: $'a$ *set*
  fixes
    — Transition Function
    δ :: ($'b$, $'a$) *DTS*
  fixes
    — Initial State
    q_0 :: $'b$
  fixes
    — ω-Word
    w :: $'a$ *word*
begin

definition *sink* :: $'b \Rightarrow$ *bool*
where
  sink $q \equiv (q_0 \neq q) \wedge (\forall \nu \in \Sigma.\ \delta\ q\ \nu = q)$

declare *sink-def* [*code*]

fun *token-run* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ $'b$
where
  token-run $x\ n$ = *run* $\delta\ q_0$ (*suffix* $x\ w$) $(n - x)$

fun *configuration* :: $'b \Rightarrow$ *nat* $\Rightarrow$ *nat set*
where
  configuration $q\ n = \{x.\ x \leq n \wedge$ *token-run* $x\ n = q\}$

fun *oldest-token* :: $'b \Rightarrow$ *nat* $\Rightarrow$ *nat option*
where
  oldest-token $q\ n$ = (*if configuration* $q\ n \neq \{\}$ *then Some* (*Min* (*configuration* $q\ n$)) *else None*)

fun *senior* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
  senior $x\ n$ = *the* (*oldest-token* (*token-run* $x\ n$) n)

fun *older-seniors* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat set*
where
  older-seniors $x\ n = \{s.\ \exists y.\ s =$ *senior* $y\ n \wedge s <$ *senior* $x\ n \wedge \neg$ *sink* (*token-run* $s\ n$)$\}$

fun *rank* :: *nat* ⇒ *nat* ⇒ *nat option*
where
rank x n =
(*if x* ≤ *n* ∧ ¬*sink* (*token-run x n*) *then Some* (*card* (*older-seniors x n*)) *else None*)

fun *senior-states* :: *'b* ⇒ *nat* ⇒ *'b set*
where
senior-states q n =
{*p*. ∃ *x y*. *oldest-token p n* = *Some y* ∧ *oldest-token q n* = *Some x* ∧ *y* < *x* ∧ ¬ *sink p*}

fun *state-rank* :: *'b* ⇒ *nat* ⇒ *nat option*
where
state-rank q n = (*if configuration q n* ≠ {} ∧ ¬*sink q then Some* (*card* (*senior-states q n*)) *else None*)

definition *max-rank* :: *nat*
where
max-rank = *card* (*reach* Σ δ *q*$_0$ − {*q*. *sink q*})

5.1.1 Iterative Computation of State-Ranks

fun *initial* :: *'b* ⇒ *nat option*
where
initial q = (*if q* = *q*$_0$ *then Some 0 else None*)

fun *pre-ranks* :: (*'b* ⇒ *nat option*) ⇒ *'a* ⇒ *'b* ⇒ *nat set*
where
pre-ranks r ν q = {*i* . ∃ *q'*. *r q'* = *Some i* ∧ *q* = δ *q' ν*} ∪ (*if q* = *q*$_0$ *then* {*max-rank*} *else* {})

fun *step* :: (*'b* ⇒ *nat option*) ⇒ *'a* ⇒ (*'b* ⇒ *nat option*)
where
step r ν q = (
if
¬*sink q* ∧ *pre-ranks r ν q* ≠ {}
then
Some (*card* {*q'*. ¬*sink q'* ∧ *pre-ranks r ν q'* ≠ {} ∧ *Min* (*pre-ranks r ν q'*) < *Min* (*pre-ranks r ν q*)})
else
None)

5.1.2 Properties of Tokens

definition *token-squats* :: *nat* ⇒ *bool*
where
token-squats x = (∀ *n*. ¬*sink* (*token-run x n*))

end

locale *semi-mojmir* = *semi-mojmir-def* +
assumes
— The alphabet is finite. Non-emptiness is derived from well-formed w
finite-Σ: *finite* Σ
assumes
— The set of reachable states is finite
finite-reach: *finite* (*reach* Σ δ q_0)
assumes
— w only contains letters from the alphabet
bounded-w: *range w* ⊆ Σ
begin

lemma *nonempty-Σ*: Σ ≠ {}
⟨*proof*⟩

lemma *bounded-w′*: *w i* ∈ Σ
⟨*proof*⟩

lemma *sink-rev-step*:
¬*sink q* ⟹ *q* = δ *q′* ν ⟹ ν ∈ Σ ⟹ ¬*sink q′*
¬*sink q* ⟹ *q* = δ *q′* (*w i*) ⟹ ¬*sink q′*
⟨*proof*⟩

5.2 Token Run

lemma *token-stays-in-sink*:
assumes *sink q*
assumes *token-run x n* = *q*
shows *token-run x* (*n* + *m*) = *q*
⟨*proof*⟩

lemma *token-is-not-in-sink*:
token-run x n ∉ *A* ⟹ *token-run x* (*Suc n*) ∈ *A* ⟹ ¬*sink* (*token-run x n*)
⟨*proof*⟩

lemma *token-run-intial-state*:
token-run x x = q_0
⟨*proof*⟩

lemma *token-run-P*:
assumes ¬ *P* (*token-run x n*)
assumes *P* (*token-run x* (*Suc* (*n* + *m*)))
shows ∃ *m′* ≤ *m*. ¬ *P* (*token-run x* (*n* + *m′*)) ∧ *P* (*token-run x* (*Suc* (*n* + *m′*)))
⟨*proof*⟩

lemma *token-run-merge-Suc*:
assumes *x* ≤ *n*
assumes *y* ≤ *n*
assumes *token-run x n* = *token-run y n*
shows *token-run x* (*Suc n*) = *token-run y* (*Suc n*)
⟨*proof*⟩

lemma *token-run-merge*:
⟦*x* ≤ *n*; *y* ≤ *n*; *token-run x n* = *token-run y n*⟧ ⟹ *token-run x* (*n* + *m*) = *token-run y* (*n* + *m*)
⟨*proof*⟩

lemma *token-run-mergepoint*:
assumes *x* < *y*
assumes *token-run x* (*y* + *n*) = *token-run y* (*y* + *n*)
obtains *m* **where** *x* ≤ (*Suc m*) **and** *y* ≤ (*Suc m*)
and *y* = *Suc m* ∨ *token-run x m* ≠ *token-run y m*
and *token-run x* (*Suc m*) = *token-run y* (*Suc m*)
⟨*proof*⟩

5.2.1 Step Lemmas

lemma *token-run-step*:
assumes *x* ≤ *n*
assumes *token-run x n* = *q′*
assumes *q* = *δ q′* (*w n*)
shows *token-run x* (*Suc n*) = *q*
⟨*proof*⟩

lemma *token-run-step′*:
x ≤ *n* ⟹ *token-run x* (*Suc n*) = *δ* (*token-run x n*) (*w n*)
⟨*proof*⟩

5.3 Configuration

5.3.1 Properties

lemma *configuration-distinct*:
$q \neq q' \Longrightarrow \mathit{configuration}\ q\ n \cap \mathit{configuration}\ q'\ n = \{\}$
⟨*proof*⟩

lemma *configuration-finite*:
finite (*configuration* $q\ n$)
⟨*proof*⟩

lemma *configuration-non-empty*:
$x \leq n \Longrightarrow \mathit{configuration}\ (\mathit{token\text{-}run}\ x\ n)\ n \neq \{\}$
⟨*proof*⟩

lemma *configuration-token*:
$x \leq n \Longrightarrow x \in \mathit{configuration}\ (\mathit{token\text{-}run}\ x\ n)\ n$
⟨*proof*⟩

lemmas *configuration-Max-in* = *Max-in*[*OF configuration-finite*]
lemmas *configuration-Min-in* = *Min-in*[*OF configuration-finite*]

5.3.2 Monotonicity

lemma *configuration-monotonic-Suc*:
$x \leq n \Longrightarrow \mathit{configuration}\ (\mathit{token\text{-}run}\ x\ n)\ n \subseteq \mathit{configuration}\ (\mathit{token\text{-}run}\ x\ (\mathit{Suc}\ n))\ (\mathit{Suc}\ n)$
⟨*proof*⟩

5.3.3 Pull-Up and Push-Down

lemma *pull-up-token-run-tokens*:
$[\![x \leq n;\ y \leq n;\ \mathit{token\text{-}run}\ x\ n = \mathit{token\text{-}run}\ y\ n]\!] \Longrightarrow \exists\, q.\ x \in \mathit{configuration}\ q\ n \wedge y \in \mathit{configuration}\ q\ n$
⟨*proof*⟩

lemma *push-down-configuration-token-run*:
$[\![x \in \mathit{configuration}\ q\ n;\ y \in \mathit{configuration}\ q\ n]\!] \Longrightarrow x \leq n \wedge y \leq n \wedge \mathit{token\text{-}run}\ x\ n = \mathit{token\text{-}run}\ y\ n$
⟨*proof*⟩

5.3.4 Step Lemmas

lemma *configuration-step*:
$x \in \mathit{configuration}\ q'\ n \Longrightarrow q = \delta\ q'\ (w\ n) \Longrightarrow x \in \mathit{configuration}\ q\ (\mathit{Suc}\ n)$

⟨*proof*⟩

lemma *configuration-step-non-empty*:
configuration q' $n \neq \{\} \Longrightarrow q = \delta\ q'\ (w\ n) \Longrightarrow$ *configuration* q (*Suc* n) $\neq \{\}$
⟨*proof*⟩

lemma *configuration-rev-step′*:
assumes $x \neq$ *Suc* n
assumes $x \in$ *configuration* q (*Suc* n)
obtains q' **where** $q = \delta\ q'\ (w\ n)$ **and** $x \in$ *configuration* q' n
⟨*proof*⟩

lemma *configuration-rev-step″*:
assumes $x \in$ *configuration* q_0 (*Suc* n)
shows $x =$ *Suc* $n \vee (\exists\, q'.\ q_0 = \delta\ q'\ (w\ n) \wedge x \in$ *configuration* q' $n)$
⟨*proof*⟩

lemma *configuration-step-eq-*q_0:
configuration q_0 (*Suc* n) $= \{$*Suc* $n\} \cup \bigcup \{$*configuration* q' $n \mid q'.\ q_0 = \delta\ q'\ (w\ n)\}$
⟨*proof*⟩

lemma *configuration-rev-step*:
assumes $q \neq q_0$
assumes $x \in$ *configuration* q (*Suc* n)
obtains q' **where** $q = \delta\ q'\ (w\ n)$ **and** $x \in$ *configuration* q' n
⟨*proof*⟩

lemma *configuration-step-eq*:
assumes $q \neq q_0$
shows *configuration* q (*Suc* n) $= \bigcup \{$*configuration* q' $n \mid q'.\ q = \delta\ q'\ (w\ n)\}$
⟨*proof*⟩

lemma *configuration-step-eq-unified*:
shows *configuration* q (*Suc* n) $= \bigcup \{$*configuration* q' $n \mid q'.\ q = \delta\ q'\ (w\ n)\} \cup$ (*if* $q = q_0$ *then* $\{$*Suc* $n\}$ *else* $\{\}$)
⟨*proof*⟩

5.4 Oldest Token

5.4.1 Properties

lemma *oldest-token-always-def*:
$\exists i.\ i \leq x \wedge$ *oldest-token* (*token-run x n*) *n* = *Some i*
⟨*proof*⟩

lemma *oldest-token-bounded*:
oldest-token q n = *Some x* $\Longrightarrow$ $x \leq n$
⟨*proof*⟩

lemma *oldest-token-distinct*:
$q \neq q' \Longrightarrow$ *oldest-token q n* = *Some i* $\Longrightarrow$ *oldest-token q' n* = *Some j* $\Longrightarrow$ $i \neq j$
⟨*proof*⟩

lemma *oldest-token-equal*:
oldest-token q n = *Some i* $\Longrightarrow$ *oldest-token q' n* = *Some i* $\Longrightarrow$ $q = q'$
⟨*proof*⟩

5.4.2 Monotonicity

lemma *oldest-token-monotonic-Suc*:
assumes $x \leq n$
assumes *oldest-token* (*token-run x n*) *n* = *Some i*
assumes *oldest-token* (*token-run x* (*Suc n*)) (*Suc n*) = *Some j*
shows $i \geq j$
⟨*proof*⟩

5.4.3 Pull-Up and Push-Down

lemma *push-down-oldest-token-configuration*:
oldest-token q n = *Some x* $\Longrightarrow$ $x \in$ *configuration q n*
⟨*proof*⟩

lemma *push-down-oldest-token-token-run*:
oldest-token q n = *Some x* $\Longrightarrow$ *token-run x n* = *q*
⟨*proof*⟩

5.5 Senior Token

5.5.1 Properties

lemma *senior-le-token*:
senior x n $\leq x$

⟨*proof*⟩

lemma *senior-token-run*:
senior x n = *senior y n* ⟷ *token-run x n* = *token-run y n*
⟨*proof*⟩

The senior of a token is always in the same state

lemma *senior-same-state*:
token-run (*senior x n*) *n* = *token-run x n*
⟨*proof*⟩

lemma *senior-senior*:
senior (*senior x n*) *n* = *senior x n*
⟨*proof*⟩

5.5.2 Monotonicity

lemma *senior-monotonic-Suc*:
x ≤ *n* ⟹ *senior x n* ≥ *senior x* (*Suc n*)
⟨*proof*⟩

5.5.3 Pull-Up and Push-Down

lemma *pull-up-configuration-senior*:
⟦*x* ∈ *configuration q n*; *y* ∈ *configuration q n*⟧ ⟹ *senior x n* = *senior y n*
⟨*proof*⟩

lemma *push-down-senior-tokens*:
⟦*x* ≤ *n*; *y* ≤ *n*; *senior x n* = *senior y n*⟧ ⟹ ∃ *q*. *x* ∈ *configuration q n* ∧ *y* ∈ *configuration q n*
⟨*proof*⟩

5.6 Set of Older Seniors

5.6.1 Properties

lemma *older-seniors-cases-subseteq* [*case-names le ge*]:
assumes *older-seniors x n* ⊆ *older-seniors y n* ⟹ *P*
assumes *older-seniors x n* ⊇ *older-seniors y n* ⟹ *P*
shows *P* ⟨*proof*⟩

lemma *older-seniors-cases-subset* [*case-names less equal greater*]:
assumes *older-seniors x n* ⊂ *older-seniors y n* ⟹ *P*
assumes *older-seniors x n* = *older-seniors y n* ⟹ *P*
assumes *older-seniors x n* ⊃ *older-seniors y n* ⟹ *P*

shows P ⟨*proof*⟩

lemma *older-seniors-finite*:
finite (*older-seniors x n*)
⟨*proof*⟩

lemma *older-seniors-older*:
$y \in$ *older-seniors x n* $\Longrightarrow y < x$
⟨*proof*⟩

lemma *older-seniors-senior-simp*:
older-seniors (*senior x n*) *n* = *older-seniors x n*
⟨*proof*⟩

lemma *older-seniors-not-self-referential*:
senior x n $\notin$ *older-seniors x n*
⟨*proof*⟩

lemma *older-seniors-not-self-referential-2*:
$x \notin$ *older-seniors x n*
⟨*proof*⟩

lemma *older-seniors-subset*:
$y \in$ *older-seniors x n* $\Longrightarrow$ *older-seniors y n* $\subset$ *older-seniors x n*
⟨*proof*⟩

lemma *older-seniors-subset-2*:
assumes ¬ *sink* (*token-run x n*)
assumes *older-seniors x n* $\subset$ *older-seniors y n*
shows *senior x n* $\in$ *older-seniors y n*
⟨*proof*⟩

lemmas *older-seniors-Max-in* = *Max-in*[*OF older-seniors-finite*]
lemmas *older-seniors-Min-in* = *Min-in*[*OF older-seniors-finite*]
lemmas *older-seniors-Max-coboundedI* = *Max.coboundedI*[*OF older-seniors-finite*]
lemmas *older-seniors-Min-coboundedI* = *Min.coboundedI*[*OF older-seniors-finite*]
lemmas *older-seniors-card-mono* = *card-mono*[*OF older-seniors-finite*]
lemmas *older-seniors-psubset-card-mono* = *psubset-card-mono*[*OF older-seniors-finite*]

lemma *older-seniors-recursive*:
fixes *x n*
defines *os* ≡ *older-seniors x n*
assumes *os* ≠ {}
shows *os* = {*Max os*} ∪ *older-seniors* (*Max os*) *n*

(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *older-seniors-recursive-card*:
fixes *x n*
defines *os* ≡ *older-seniors x n*
assumes *os* ≠ {}
shows *card os* = *Suc* (*card* (*older-seniors* (*Max os*) *n*))
⟨*proof*⟩

lemma *older-seniors-card*:
card (*older-seniors x n*) = *card* (*older-seniors y n*) ⟷ *older-seniors x n* = *older-seniors y n*
⟨*proof*⟩

lemma *older-seniors-card-le*:
card (*older-seniors x n*) < *card* (*older-seniors y n*) ⟷ *older-seniors x n* ⊂ *older-seniors y n*
⟨*proof*⟩

lemma *older-seniors-card-less*:
card (*older-seniors x n*) ≤ *card* (*older-seniors y n*) ⟷ *older-seniors x n* ⊆ *older-seniors y n*
⟨*proof*⟩

5.6.2 Monotonicity

lemma *older-seniors-monotonic-Suc*:
assumes *x* ≤ *n*
shows *older-seniors x n* ⊇ *older-seniors x* (*Suc n*)
⟨*proof*⟩

lemma *older-seniors-monotonic*:
x ≤ *n* ⟹ *older-seniors x n* ⊇ *older-seniors x* (*n* + *m*)
⟨*proof*⟩

lemma *older-seniors-stable*:
x ≤ *n* ⟹ *older-seniors x n* = *older-seniors x* (*n* + *m* + *m′*) ⟹ *older-seniors x n* = *older-seniors x* (*n* + *m*)
⟨*proof*⟩

lemma *card-older-seniors-monotonic*:
x ≤ *n* ⟹ *card* (*older-seniors x n*) ≥ *card* (*older-seniors x* (*n* + *m*))
⟨*proof*⟩

5.6.3 Pull-Up and Push-Down

lemma *pull-up-senior-older-seniors*:
senior x n = senior y n ⟹ older-seniors x n = older-seniors y n
⟨*proof*⟩

lemma *pull-up-senior-older-seniors-less*:
senior x n < senior y n ⟹ older-seniors x n ⊆ older-seniors y n
⟨*proof*⟩

lemma *pull-up-senior-older-seniors-less-2*:
assumes ¬ *sink* (*token-run x n*)
assumes *senior x n < senior y n*
shows *older-seniors x n ⊂ older-seniors y n*
⟨*proof*⟩

lemma *pull-up-senior-older-seniors-le*:
senior x n ≤ senior y n ⟹ older-seniors x n ⊆ older-seniors y n
⟨*proof*⟩

lemma *push-down-older-seniors-senior*:
assumes ¬ *sink* (*token-run x n*)
assumes ¬ *sink* (*token-run y n*)
assumes *older-seniors x n = older-seniors y n*
shows *senior x n = senior y n*
⟨*proof*⟩

5.6.4 Tower Lemma

lemma *older-seniors-tower''*:
assumes *x ≤ n*
assumes *y ≤ n*
assumes ¬*sink* (*token-run x n*)
assumes ¬*sink* (*token-run y n*)
assumes *older-seniors x n = older-seniors x* (*Suc n*)
assumes *older-seniors y n ⊆ older-seniors x n*
shows *older-seniors y n = older-seniors y* (*Suc n*)
⟨*proof*⟩

lemma *older-seniors-tower''2*:
assumes *x ≤ n*
assumes *y ≤ n*
assumes ¬*sink* (*token-run x* (*n + m*))
assumes ¬*sink* (*token-run y* (*n + m*))

assumes *older-seniors x n* = *older-seniors x* (*n* + *m*)
assumes *older-seniors y n* ⊆ *older-seniors x n*
shows *older-seniors y n* = *older-seniors y* (*n* + *m*)
⟨*proof*⟩

lemma *older-seniors-tower′*:
assumes *y* ∈ *older-seniors x n*
assumes *older-seniors x n* = *older-seniors x* (*Suc n*)
shows *older-seniors y n* = *older-seniors y* (*Suc n*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *older-seniors-tower*:
⟦*x* ≤ *n*; *y* ∈ *older-seniors x n*; *older-seniors x n* = *older-seniors x* (*n* + *m*)⟧ ⟹ *older-seniors y n* = *older-seniors y* (*n* + *m*)
⟨*proof*⟩

5.7 Rank

5.7.1 Properties

lemma *rank-None-before*:
x > *n* ⟹ *rank x n* = *None*
⟨*proof*⟩

lemma *rank-None-Suc*:
assumes *x* ≤ *n*
assumes *rank x n* = *None*
shows *rank x* (*Suc n*) = *None*
⟨*proof*⟩

lemma *rank-Some-time*:
rank x n = *Some j* ⟹ *x* ≤ *n*
⟨*proof*⟩

lemma *rank-Some-sink*:
rank x n = *Some j* ⟹ ¬*sink* (*token-run x n*)
⟨*proof*⟩

lemma *rank-Some-card*:
rank x n = *Some j* ⟹ *card* (*older-seniors x n*) = *j*
⟨*proof*⟩

lemma *rank-initial*:

$\exists\, i.\ rank\ x\ x = Some\ i$
⟨*proof*⟩

lemma *rank-continuous*:
assumes $rank\ x\ n = Some\ i$
assumes $rank\ x\ (n + m) = Some\ j$
assumes $m' \leq m$
shows $\exists\, k.\ rank\ x\ (n + m') = Some\ k$
⟨*proof*⟩

lemma *rank-token-squats*:
$token\text{-}squats\ x \Longrightarrow x \leq n \Longrightarrow \exists\, i.\ rank\ x\ n = Some\ i$
⟨*proof*⟩

lemma *rank-older-seniors-bounded*:
assumes $y \in older\text{-}seniors\ x\ n$
assumes $rank\ x\ n = Some\ j$
shows $\exists\, j' < j.\ rank\ y\ n = Some\ j'$
⟨*proof*⟩

5.7.2 Bounds

lemma *max-rank-lowerbound*:
$0 < max\text{-}rank$
⟨*proof*⟩

lemma *older-seniors-card-bounded*:
assumes $\neg sink\ (token\text{-}run\ x\ n)$ **and** $x \leq n$
shows $card\ (older\text{-}seniors\ x\ n) < card\ (reach\ \Sigma\ \delta\ q_0 - \{q.\ sink\ q\})$
(**is** *card ?S4* < *card ?S0*)
⟨*proof*⟩

lemma *rank-upper-bound*:
$rank\ x\ n = Some\ i \Longrightarrow i < max\text{-}rank$
⟨*proof*⟩

lemma *rank-range*:
$\exists\, i.\ range\ (rank\ x) \subseteq \{None\} \cup Some\ `\ \{0..<i\}$
⟨*proof*⟩

5.7.3 Monotonicity

lemma *rank-monotonic*:
$[\![rank\ x\ n = Some\ i;\ rank\ x\ (n + m) = Some\ j]\!] \Longrightarrow i \geq j$

⟨*proof*⟩

5.7.4 Pull-Up and Push-Down

lemma *pull-up-senior-rank*:
⟦$x \leq n$; $y \leq n$; *senior x n* = *senior y n*⟧ ⟹ *rank x n* = *rank y n*
⟨*proof*⟩

lemma *pull-up-configuration-rank*:
⟦$x \in$ *configuration q n*; $y \in$ *configuration q n*⟧ ⟹ *rank x n* = *rank y n*
⟨*proof*⟩

lemma *push-down-rank-older-seniors*:
⟦*rank x n* = *rank y n*; *rank x n* = *Some i*⟧ ⟹ *older-seniors x n* = *older-seniors y n*
⟨*proof*⟩

lemma *push-down-rank-senior*:
⟦*rank x n* = *rank y n*; *rank x n* = *Some i*⟧ ⟹ *senior x n* = *senior y n*
⟨*proof*⟩

lemma *push-down-rank-tokens*:
⟦*rank x n* = *rank y n*; *rank x n* = *Some i*⟧ ⟹ (∃ *q*. $x \in$ *configuration q n* ∧ $y \in$ *configuration q n*)
⟨*proof*⟩

5.7.5 Pulled-Up Lemmas

lemma *rank-senior-senior*:
$x \leq n$ ⟹ *rank* (*senior x n*) *n* = *rank x n*
⟨*proof*⟩

5.7.6 Stable Rank

definition *stable-rank* :: *nat* ⇒ *nat* ⇒ *bool*
where
stable-rank x i = (∀$_\infty$*n*. *rank x n* = *Some i*)

lemma *stable-rank-unique*:
assumes *stable-rank x i*
assumes *stable-rank x j*
shows *i* = *j*
⟨*proof*⟩

lemma *stable-rank-equiv-token-squats*:

token-squats x = (∃ *i*. *stable-rank x i*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *stable-rank-same-tokens*:
assumes *stable-rank x i*
assumes *stable-rank y j*
assumes *x* ∈ *configuration q n*
assumes *y* ∈ *configuration q n*
shows *i* = *j*
⟨*proof*⟩

5.7.7 Tower Lemma

lemma *rank-tower*:
assumes *i* ≤ *j*
assumes *rank x n* = *Some j*
assumes *rank x* (*n* + *m*) = *Some j*
assumes *rank y n* = *Some i*
shows *rank y* (*n* + *m*) = *Some i*
⟨*proof*⟩

lemma *stable-rank-alt-def*:
rank x n = *Some j* ∧ *stable-rank x j* ⟷ (∀ *m* ≥ *n*. *rank x m* = *Some j*)
(**is** *?rhs* ⟷ *?lhs*)
⟨*proof*⟩

lemma *stable-rank-tower*:
assumes *j* ≤ *i*
assumes *rank x n* = *Some j*
assumes *rank y n* = *Some i*
assumes *stable-rank y i*
shows *stable-rank x j*
⟨*proof*⟩

5.8 Senior States

lemma *senior-states-initial*:
senior-states q 0 = {}
⟨*proof*⟩

lemma *senior-states-cases-subseteq* [*case-names le ge*]:
assumes *senior-states p n* ⊆ *senior-states q n* ⟹ *P*
assumes *senior-states p n* ⊇ *senior-states q n* ⟹ *P*

shows P ⟨*proof*⟩

lemma *senior-states-cases-subset* [*case-names less equal greater*]:
assumes *senior-states p n* ⊂ *senior-states q n* ⟹ P
assumes *senior-states p n* = *senior-states q n* ⟹ P
assumes *senior-states p n* ⊃ *senior-states q n* ⟹ P
shows P ⟨*proof*⟩

lemma *senior-states-finite*:
finite (*senior-states q n*)
⟨*proof*⟩

lemmas *senior-states-card-mono* = *card-mono*[*OF senior-states-finite*]
lemmas *senior-states-psubset-card-mono* = *psubset-card-mono*[*OF senior-states-finite*]

lemma *senior-states-card*:
card (*senior-states p n*) = *card* (*senior-states q n*) ⟷ *senior-states p n*
= *senior-states q n*
⟨*proof*⟩

lemma *senior-states-card-le*:
card (*senior-states p n*) < *card* (*senior-states q n*) ⟷ *senior-states p n*
⊂ *senior-states q n*
⟨*proof*⟩

lemma *senior-states-card-less*:
card (*senior-states p n*) ≤ *card* (*senior-states q n*) ⟷ *senior-states p n*
⊆ *senior-states q n*
⟨*proof*⟩

lemma *senior-states-older-seniors*:
(λ*y*. *token-run y n*) ‘ *older-seniors x n* = *senior-states* (*token-run x n*) *n*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *card-older-senior-senior-states*:
assumes *x* ∈ *configuration q n*
shows *card* (*older-seniors x n*) = *card* (*senior-states q n*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

5.9 Rank of States

5.9.1 Alternative Definitions

lemma *state-rank-eq-rank*:
state-rank q n = (*case oldest-token q n of None* ⇒ *None* | *Some t* ⇒ *rank t n*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *state-rank-eq-rank-SOME*:
state-rank q n = (*if configuration q n* ≠ {} *then rank* (*SOME x*. *x* ∈ *configuration q n*) *n else None*)
⟨*proof*⟩

lemma *rank-eq-state-rank*:
x ≤ *n* ⟹ *rank x n* = *state-rank* (*token-run x n*) *n*
⟨*proof*⟩

5.9.2 Pull-Up and Push-Down

lemma *pull-up-configuration-state-rank*:
configuration q n = {} ⟹ *state-rank q n* = *None*
⟨*proof*⟩

lemma *push-down-state-rank-tokens*:
state-rank q n = *Some i* ⟹ *configuration q n* ≠ {}
⟨*proof*⟩

lemma *push-down-state-rank-configuration-None*:
state-rank q n = *None* ⟹ ¬*sink q* ⟹ *configuration q n* = {}
⟨*proof*⟩

lemma *push-down-state-rank-oldest-token*:
state-rank q n = *Some i* ⟹ ∃ *x*. *oldest-token q n* = *Some x*
⟨*proof*⟩

lemma *push-down-state-rank-token-run*:
state-rank q n = *Some i* ⟹ ∃ *x*. *token-run x n* = *q* ∧ *x* ≤ *n*
⟨*proof*⟩

5.9.3 Properties

lemma *state-rank-distinct*:
assumes *distinct*: *p* ≠ *q*

assumes *ranked-1*: *state-rank p n = Some i*
assumes *ranked-2*: *state-rank q n = Some j*
shows $i \neq j$
⟨*proof*⟩

lemma *state-rank-initial-state*:
obtains *i* **where** *state-rank* q_0 *n = Some i*
⟨*proof*⟩

lemma *state-rank-sink*:
sink q $\Longrightarrow$ *state-rank q n = None*
⟨*proof*⟩

lemma *state-rank-upper-bound*:
state-rank q n = Some i $\Longrightarrow$ $i <$ *max-rank*
⟨*proof*⟩

lemma *state-rank-range*:
state-rank q n $\in \{None\} \cup Some$ ‘ $\{0..<max\text{-}rank\}$
⟨*proof*⟩

lemma *state-rank-None*:
$\neg$*sink q* $\Longrightarrow$ *state-rank q n = None* $\longleftrightarrow$ *oldest-token q n = None*
⟨*proof*⟩

lemma *state-rank-Some*:
$\neg$*sink q* $\Longrightarrow$ $(\exists\, i.$ *state-rank q n = Some i*$) \longleftrightarrow (\exists\, j.$ *oldest-token q n = Some j*$)$
⟨*proof*⟩

lemma *state-rank-oldest-token*:
assumes *state-rank p n = Some i*
assumes *state-rank q n = Some j*
assumes *oldest-token p n = Some x*
assumes *oldest-token q n = Some y*
shows $i < j \longleftrightarrow x < y$
⟨*proof*⟩

lemma *state-rank-oldest-token-le*:
assumes *state-rank p n = Some i*
assumes *state-rank q n = Some j*
assumes *oldest-token p n = Some x*
assumes *oldest-token q n = Some y*
shows $i \leq j \longleftrightarrow x \leq y$

⟨*proof*⟩

lemma *state-rank-in-function-set*:
shows $(\lambda q.\ \textit{state-rank}\ q\ t) \in \{f.\ (\forall x.\ x \notin \textit{reach}\ \Sigma\ \delta\ q_0 \longrightarrow f\ x = \textit{None})$
$\wedge$
$(\forall x.\ x \in \textit{reach}\ \Sigma\ \delta\ q_0 \longrightarrow f\ x \in \{\textit{None}\} \cup \textit{Some}\ `\ \{0..<\textit{max-rank}\})\}$
⟨*proof*⟩

5.10 Step Function

fun *pre-oldest-tokens* :: $'b \Rightarrow \textit{nat} \Rightarrow \textit{nat set}$
where
$\textit{pre-oldest-tokens}\ q\ n = \{x.\ \exists q'.\ \textit{oldest-token}\ q'\ n = \textit{Some}\ x \wedge q = \delta\ q'\ (w\ n)\} \cup (\textit{if}\ q = q_0\ \textit{then}\ \{\textit{Suc}\ n\}\ \textit{else}\ \{\})$

lemma *pre-oldest-configuration-range*:
$\textit{pre-oldest-tokens}\ q\ n \subseteq \{0..\textit{Suc}\ n\}$
⟨*proof*⟩

lemma *pre-oldest-configuration-finite*:
finite (*pre-oldest-tokens q n*)
⟨*proof*⟩

lemmas *pre-oldest-configuration-Min-in* = *Min-in*[*OF pre-oldest-configuration-finite*]

lemma *pre-oldest-configuration-obtain*:
assumes $x \in \textit{pre-oldest-tokens}\ q\ n - \{\textit{Suc}\ n\}$
obtains q' **where** $\textit{oldest-token}\ q'\ n = \textit{Some}\ x$ **and** $q = \delta\ q'\ (w\ n)$
⟨*proof*⟩

lemma *pre-oldest-configuration-element*:
assumes $\textit{oldest-token}\ q'\ n = \textit{Some}\ ot$
assumes $q = \delta\ q'\ (w\ n)$
shows $ot \in \textit{pre-oldest-tokens}\ q\ n$
⟨*proof*⟩

lemma *pre-oldest-configuration-initial-state*:
$\textit{Suc}\ n \in \textit{pre-oldest-tokens}\ q\ n \Longrightarrow q = q_0$
⟨*proof*⟩

lemma *pre-oldest-configuration-initial-state-2*:
$q = q_0 \Longrightarrow \textit{Suc}\ n \in \textit{pre-oldest-tokens}\ q\ n$
⟨*proof*⟩

lemma *pre-oldest-configuration-tokens*:
pre-oldest-tokens q n $\neq \{\}$ $\longleftrightarrow$ *configuration q* (*Suc n*) $\neq \{\}$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *oldest-token-rec*:
oldest-token q (*Suc n*) = (*if pre-oldest-tokens q n* $\neq \{\}$ *then Some* (*Min* (*pre-oldest-tokens q n*)) *else None*)
⟨*proof*⟩

lemma *pre-ranks-range*:
pre-ranks (λq. *state-rank q n*) ν *q* $\subseteq \{0..max\text{-}rank\}$
⟨*proof*⟩

lemma *pre-ranks-finite*:
finite (*pre-ranks* (λq. *state-rank q n*) ν *q*)
⟨*proof*⟩

lemmas *pre-ranks-Min-in* = *Min-in*[*OF pre-ranks-finite*]

lemma *pre-ranks-state-obtain*:
assumes $r_q \in$ *pre-ranks r* ν *q* $- \{max\text{-}rank\}$
obtains q' **where** *r q'* = *Some* r_q **and** *q* = δ *q'* ν
⟨*proof*⟩

lemma *pre-ranks-element*:
assumes *state-rank q' n* = *Some r*
assumes *q* = δ *q'* (*w n*)
shows *r* $\in$ *pre-ranks* (λq. *state-rank q n*) (*w n*) *q*
⟨*proof*⟩

lemma *pre-ranks-initial-state*:
max-rank $\in$ *pre-ranks* (λq. *state-rank q n*) ν *q* $\Longrightarrow$ *q* = q_0
⟨*proof*⟩

lemma *pre-ranks-initial-state-2*:
q = q_0 $\Longrightarrow$ *max-rank* $\in$ *pre-ranks r* ν *q*
⟨*proof*⟩

lemma *pre-ranks-tokens*:
assumes $\neg$*sink q*
shows *pre-ranks* (λq. *state-rank q n*) (*w n*) *q* $\neq \{\}$ $\longleftrightarrow$ *configuration q* (*Suc n*) $\neq \{\}$
(**is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *pre-ranks-pre-oldest-token-Min-state-special*:
assumes ¬*sink q*
assumes *configuration q* (*Suc n*) ≠ {}
shows *Min* (*pre-ranks* (λ*q*. *state-rank q n*) (*w n*) *q*) = *max-rank* ⟷ *Min* (*pre-oldest-tokens q n*) = *Suc n*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *pre-ranks-pre-oldest-token-Min-state*:
assumes ¬*sink q*
assumes *q* = *δ q*′ (*w n*)
assumes *configuration q* (*Suc n*) ≠ {}
defines *min-r* ≡ *Min* (*pre-ranks* (λ*q*. *state-rank q n*) (*w n*) *q*)
defines *min-ot* ≡ *Min* (*pre-oldest-tokens q n*)
shows *state-rank q*′ *n* = *Some min-r* ⟷ *oldest-token q*′ *n* = *Some min-ot*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *Min-pre-ranks-pre-oldest-tokens*:
fixes *n*
defines *r* ≡ (λ*q*. *state-rank q n*)
assumes *configuration p* (*Suc n*) ≠ {}
and *configuration q* (*Suc n*) ≠ {}
assumes ¬*sink q*
and ¬*sink p*
shows *Min* (*pre-ranks r* (*w n*) *p*) < *Min* (*pre-ranks r* (*w n*) *q*) ⟷ *Min* (*pre-oldest-tokens p n*) < *Min* (*pre-oldest-tokens q n*)
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

5.10.1 Definition of initial and step

lemma *state-rank-initial*:
state-rank q 0 = *initial q*
⟨*proof*⟩

lemma *state-rank-step*:
state-rank q (*Suc n*) = *step* (λ*q*. *state-rank q n*) (*w n*) *q*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *state-rank-step-foldl*:

$(\lambda q.\ \textit{state-rank}\ q\ n) = \textit{foldl}\ \textit{step}\ \textit{initial}\ (\textit{map}\ w\ [0..<n])$
⟨*proof*⟩

end

end

6 Mojmir Automata

theory *Mojmir*
imports *Main Semi-Mojmir*
begin

6.1 Definitions

locale *mojmir-def* = *semi-mojmir-def* +
fixes
— Final States
$F :: 'b\ set$
begin

definition *token-succeeds* :: $nat \Rightarrow bool$
where
$\textit{token-succeeds}\ x = (\exists\, n.\ \textit{token-run}\ x\ n \in F)$

definition *token-fails* :: $nat \Rightarrow bool$
where
$\textit{token-fails}\ x = (\exists\, n.\ \textit{sink}\ (\textit{token-run}\ x\ n) \wedge \textit{token-run}\ x\ n \notin F)$

definition *accept* :: *bool* ($accept_M$)
where
$\textit{accept} \longleftrightarrow (\forall_\infty x.\ \textit{token-succeeds}\ x)$

definition *fail* :: *nat set*
where
$\textit{fail} = \{x.\ \textit{token-fails}\ x\}$

definition *merge* :: $nat \Rightarrow (nat \times nat)\ set$
where
$\textit{merge}\ i = \{(x,\ y) \mid x\ y\ n\ j.\ j < i$
$\wedge\ (\textit{token-run}\ x\ n \neq \textit{token-run}\ y\ n \wedge \textit{rank}\ y\ n \neq \textit{None} \vee y = \textit{Suc}\ n)$
$\wedge\ \textit{token-run}\ x\ (\textit{Suc}\ n) = \textit{token-run}\ y\ (\textit{Suc}\ n)$
$\wedge\ \textit{token-run}\ x\ (\textit{Suc}\ n) \notin F$
$\wedge\ \textit{rank}\ x\ n = \textit{Some}\ j\}$

definition *succeed* :: *nat* ⇒ *nat set*
where
succeed i = {*x*. ∃ *n*. *rank x n* = *Some i*
∧ *token-run x n* ∉ *F* − {q_0}
∧ *token-run x* (*Suc n*) ∈ *F*}

definition *smallest-accepting-rank* :: *nat option*
where
smallest-accepting-rank ≡ (*if accept then*
Some (*LEAST i*. *finite fail* ∧ *finite* (*merge i*) ∧ *infinite* (*succeed i*)) *else None*)

definition *fail-t* :: *nat set*
where
fail-t = {*n*. ∃ *q q'*. *state-rank q n* ≠ *None* ∧ *q'* = *δ q* (*w n*) ∧ *q'* ∉ *F* ∧ *sink q'*}

definition *merge-t* :: *nat* ⇒ *nat set*
where
merge-t i = {*n*. ∃ *q q' j*. *state-rank q n* = *Some j* ∧ *j* < *i* ∧ *q'* = *δ q* (*w n*) ∧ *q'* ∉ *F* ∧
(∃ *q''*. *q''* ≠ *q* ∧ *q'* = *δ q''* (*w n*) ∧ *state-rank q'' n* ≠ *None*) ∨ *q'* = q_0)}

definition *succeed-t* :: *nat* ⇒ *nat set*
where
succeed-t i = {*n*. ∃ *q*. *state-rank q n* = *Some i* ∧ *q* ∉ *F* − {q_0} ∧ *δ q* (*w n*) ∈ *F*}

fun $\mathcal{S}$
where
$\mathcal{S}$ *n* = *F* ∪ {*q*. (∃ *j* ≥ *the smallest-accepting-rank*. *state-rank q n* = *Some j*)}

end

locale *mojmir* = *semi-mojmir* + *mojmir-def* +
assumes
— All states reachable from final states are also final
wellformed-F: ⋀*q ν*. *q* ∈ *F* ⟹ *δ q ν* ∈ *F*
begin

lemma *token-stays-in-final-states*:
token-run x n ∈ *F* ⟹ *token-run x* (*n* + *m*) ∈ *F*

⟨*proof*⟩

lemma *token-run-enter-final-states*:
assumes *token-run x n* ∈ *F*
shows ∃ *m* ≥ *x*. *token-run x m* ∉ *F* − {q_0} ∧ *token-run x* (*Suc m*) ∈ *F*
⟨*proof*⟩

6.2 Token Properties

6.2.1 Alternative Definitions

lemma *token-succeeds-alt-def*:
token-succeeds x = (∀$_\infty$*n*. *token-run x n* ∈ *F*)
⟨*proof*⟩

lemma *token-fails-alt-def*:
token-fails x = (∀$_\infty$*n*. *sink* (*token-run x n*) ∧ *token-run x n* ∉ *F*)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *token-fails-alt-def-2*:
token-fails x ⟷ ¬*token-succeeds x* ∧ ¬*token-squats x*
⟨*proof*⟩

6.2.2 Properties

lemma *token-succeeds-run-merge*:
x ≤ *n* ⟹ *y* ≤ *n* ⟹ *token-run x n* = *token-run y n* ⟹ *token-succeeds x* ⟹ *token-succeeds y*
⟨*proof*⟩

lemma *token-squats-run-merge*:
x ≤ *n* ⟹ *y* ≤ *n* ⟹ *token-run x n* = *token-run y n* ⟹ *token-squats x* ⟹ *token-squats y*
⟨*proof*⟩

6.2.3 Pulled-Up Lemmas

lemma *configuration-token-succeeds*:
⟦*x* ∈ *configuration q n*; *y* ∈ *configuration q n*⟧ ⟹ *token-succeeds x* = *token-succeeds y*
⟨*proof*⟩

lemma *configuration-token-squats*:

$[\![x \in \mathit{configuration}\ q\ n;\ y \in \mathit{configuration}\ q\ n]\!] \Longrightarrow \mathit{token\text{-}squats}\ x = \mathit{token\text{-}squats}\ y$
⟨*proof*⟩

6.3 Mojmir Acceptance

lemma *Mojmir-reject*:
$\neg\ \mathit{accept} \longleftrightarrow (\exists_\infty x.\ \neg \mathit{token\text{-}succeeds}\ x)$
⟨*proof*⟩

lemma *mojmir-accept-alt-def*:
$\mathit{accept} \longleftrightarrow \mathit{finite}\ \{x.\ \neg \mathit{token\text{-}succeeds}\ x\}$
⟨*proof*⟩

lemma *mojmir-accept-initial*:
$q_0 \in F \Longrightarrow \mathit{accept}$
⟨*proof*⟩

6.4 Equivalent Acceptance Conditions

6.4.1 Token-Based Definitions

lemma *merge-token-succeeds*:
assumes $(x,\ y) \in \mathit{merge}\ i$
shows $\mathit{token\text{-}succeeds}\ x \longleftrightarrow \mathit{token\text{-}succeeds}\ y$
⟨*proof*⟩

lemma *merge-subset*:
$i \leq j \Longrightarrow \mathit{merge}\ i \subseteq \mathit{merge}\ j$
⟨*proof*⟩

lemma *merge-finite*:
$i \leq j \Longrightarrow \mathit{finite}\ (\mathit{merge}\ j) \Longrightarrow \mathit{finite}\ (\mathit{merge}\ i)$
⟨*proof*⟩

lemma *merge-finite′*:
$i < j \Longrightarrow \mathit{finite}\ (\mathit{merge}\ j) \Longrightarrow \mathit{finite}\ (\mathit{merge}\ i)$
⟨*proof*⟩

lemma *succeed-membership*:
$\mathit{token\text{-}succeeds}\ x \longleftrightarrow (\exists\, i.\ x \in \mathit{succeed}\ i)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *stable-rank-succeed*:

assumes *infinite* (*succeed i*)
and $x \in$ *succeed i*
and $q_0 \notin F$
shows ¬*stable-rank x i*
⟨*proof*⟩

lemma *stable-rank-bounded*:
assumes *stable*: *stable-rank x j*
assumes *inf*: *infinite* (*succeed i*)
assumes $q_0 \notin F$
shows $j < i$
⟨*proof*⟩

lemma *mojmir-accept-token-set-def1*:
assumes *accept*
shows ∃ *i* < *max-rank*. *finite fail* ∧ *finite* (*merge i*) ∧ *infinite* (*succeed i*) ∧ (∀ *j* < *i*. *finite* (*succeed j*))
⟨*proof*⟩

lemma *mojmir-accept-token-set-def2*:
assumes *finite fail*
and *finite* (*merge i*)
and *infinite* (*succeed i*)
shows *accept*
⟨*proof*⟩

theorem *mojmir-accept-iff-token-set-accept*:
accept ⟷ (∃ *i* < *max-rank*. *finite fail* ∧ *finite* (*merge i*) ∧ *infinite* (*succeed i*))
⟨*proof*⟩

theorem *mojmir-accept-iff-token-set-accept2*:
accept ⟷ (∃ *i* < *max-rank*. *finite fail* ∧ *finite* (*merge i*) ∧ *infinite* (*succeed i*) ∧ (∀ *j* < *i*. *finite* (*merge j*) ∧ *finite* (*succeed j*)))
⟨*proof*⟩

6.4.2 Time-Based Definitions

lemma *finite-monotonic-image*:
fixes *A B* :: *nat set*
assumes ⋀*i*. *i* ∈ *A* ⟹ *i* ≤ *f i*
assumes *f* ‘ *A* = *B*
shows *finite A* ⟷ *finite B*
⟨*proof*⟩

lemma *finite-monotonic-image-pairs*:
fixes $A :: (nat \times nat)\ set$
fixes $B :: nat\ set$
assumes $\bigwedge i.\ i \in A \Longrightarrow (fst\ i) \leq f\ i + c$
assumes $\bigwedge i.\ i \in A \Longrightarrow (snd\ i) \leq f\ i + d$
assumes $f\ `\ A = B$
shows *finite* $A \longleftrightarrow$ *finite* B
⟨*proof*⟩

lemma *token-time-finite-rule*:
fixes $A\ B :: nat\ set$
assumes *unique*: $\bigwedge x\ y\ z.\ P\ x\ y \Longrightarrow P\ x\ z \Longrightarrow y = z$
and *existsA*: $\bigwedge x.\ x \in A \Longrightarrow (\exists\, y.\ P\ x\ y)$
and *existsB*: $\bigwedge y.\ y \in B \Longrightarrow (\exists\, x.\ P\ x\ y)$
and *inA*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow x \in A$
and *inB*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow y \in B$
and *mono*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow x \leq y$
shows *finite* $A \longleftrightarrow$ *finite* B
⟨*proof*⟩

lemma *token-time-finite-pair-rule*:
fixes $A :: (nat \times nat)\ set$
fixes $B :: nat\ set$
assumes *unique*: $\bigwedge x\ y\ z.\ P\ x\ y \Longrightarrow P\ x\ z \Longrightarrow y = z$
and *existsA*: $\bigwedge x.\ x \in A \Longrightarrow (\exists\, y.\ P\ x\ y)$
and *existsB*: $\bigwedge y.\ y \in B \Longrightarrow (\exists\, x.\ P\ x\ y)$
and *inA*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow x \in A$
and *inB*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow y \in B$
and *mono*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow fst\ x \leq y + c \wedge snd\ x \leq y + d$
shows *finite* $A \longleftrightarrow$ *finite* B
⟨*proof*⟩

lemma *fail-t-inclusion*:
assumes $x \leq n$
assumes $\neg sink\ (token\text{-}run\ x\ n)$
assumes $sink\ (token\text{-}run\ x\ (Suc\ n))$
assumes $token\text{-}run\ x\ (Suc\ n) \notin F$
shows $n \in fail\text{-}t$
⟨*proof*⟩

lemma *merge-t-inclusion*:
assumes $x \leq n$
assumes $(\exists\, j'.\ token\text{-}run\ x\ n \neq token\text{-}run\ y\ n \wedge y \leq n \wedge state\text{-}rank$

$(\textit{token-run}\ y\ n)\ n = \textit{Some}\ j') \vee y = \textit{Suc}\ n$
assumes $\textit{token-run}\ x\ (\textit{Suc}\ n) = \textit{token-run}\ y\ (\textit{Suc}\ n)$
assumes $\textit{token-run}\ x\ (\textit{Suc}\ n) \notin F$
assumes $\textit{state-rank}\ (\textit{token-run}\ x\ n)\ n = \textit{Some}\ j$
assumes $j < i$
shows $n \in \textit{merge-t}\ i$
⟨*proof*⟩

lemma *succeed-t-inclusion*:
assumes $\textit{rank}\ x\ n = \textit{Some}\ i$
assumes $\textit{token-run}\ x\ n \notin F - \{q_0\}$
assumes $\textit{token-run}\ x\ (\textit{Suc}\ n) \in F$
shows $n \in \textit{succeed-t}\ i$
⟨*proof*⟩

lemma *finite-fail-t*:
$\textit{finite}\ \textit{fail} = \textit{finite}\ \textit{fail-t}$
⟨*proof*⟩

lemma *finite-succeed-t'*:
assumes $q_0 \notin F$
shows $\textit{finite}\ (\textit{succeed}\ i) = \textit{finite}\ (\textit{succeed-t}\ i)$
⟨*proof*⟩

lemma *initial-in-F-token-run*:
assumes $q_0 \in F$
shows $\textit{token-run}\ x\ y \in F$
⟨*proof*⟩

lemma *finite-succeed-t''*:
assumes $q_0 \in F$
shows $\textit{finite}\ (\textit{succeed}\ i) = \textit{finite}\ (\textit{succeed-t}\ i)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *finite-succeed-t*:
$\textit{finite}\ (\textit{succeed}\ i) = \textit{finite}\ (\textit{succeed-t}\ i)$
⟨*proof*⟩

lemma *finite-merge-t*:
$\textit{finite}\ (\textit{merge}\ i) = \textit{finite}\ (\textit{merge-t}\ i)$
⟨*proof*⟩

6.4.3 Relation to Mojmir Acceptance

lemma *token-iff-time-accept*:
shows (*finite fail* $\wedge$ *finite* (*merge i*) $\wedge$ *infinite* (*succeed i*) $\wedge$ ($\forall\, j < i$. *finite* (*succeed j*)))
$=$ (*finite fail-t* $\wedge$ *finite* (*merge-t i*) $\wedge$ *infinite* (*succeed-t i*) $\wedge$ ($\forall\, j < i$. *finite* (*succeed-t j*)))
⟨*proof*⟩

6.5 Succeeding Tokens (Alternative Definition)

definition *stable-rank-at* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
where
stable-rank-at x n $\equiv \exists\, i.\ \forall\, m \geq n.$ *rank x m* $=$ *Some i*

lemma *stable-rank-at-ge*:
$n \leq m \Longrightarrow$ *stable-rank-at x n* $\Longrightarrow$ *stable-rank-at x m*
⟨*proof*⟩

lemma *stable-rank-equiv*:
($\exists\, i.$ *stable-rank x i*) $=$ ($\exists\, n.$ *stable-rank-at x n*)
⟨*proof*⟩

lemma *smallest-accepting-rank-properties*:
assumes *smallest-accepting-rank* $=$ *Some i*
shows *accept finite fail finite* (*merge i*) *infinite* (*succeed i*) $\forall\, j < i$. *finite* (*succeed j*) $i <$ *max-rank*
⟨*proof*⟩

lemma *token-smallest-accepting-rank*:
assumes *smallest-accepting-rank* $=$ *Some i*
shows $\forall_\infty n.\ \forall\, x.$ *token-succeeds x* $\longleftrightarrow$ ($x >$ *n* $\vee$ ($\exists\, j \geq i.$ *rank x n* $=$ *Some j*) $\vee$ *token-run x n* $\in F$)
⟨*proof*⟩

lemma *succeeding-states*:
assumes *smallest-accepting-rank* $=$ *Some i*
shows $\forall_\infty n.\ \forall\, q.$ (($\exists\, x \in$ *configuration q n. token-succeeds x*) $\longrightarrow$ $q \in \mathcal{S}$ *n*) $\wedge$ ($q \in \mathcal{S}$ *n* $\longrightarrow$ ($\forall\, x \in$ *configuration q n. token-succeeds x*))
⟨*proof*⟩

end

end

7 (Generalized) Rabin Automata

theory *Rabin*
imports *Main DTS*
begin

type-synonym $('a, 'b)$ *rabin-pair* $= (('a, 'b)$ *transition set* $\times$ $('a, 'b)$ *transition set*$)$
type-synonym $('a, 'b)$ *generalized-rabin-pair* $= (('a, 'b)$ *transition set* $\times$ $('a, 'b)$ *transition set set*$)$

type-synonym $('a, 'b)$ *rabin-condition* $= ('a, 'b)$ *rabin-pair set*
type-synonym $('a, 'b)$ *generalized-rabin-condition* $= ('a, 'b)$ *generalized-rabin-pair set*

type-synonym $('a, 'b)$ *rabin-automaton* $= ('a, 'b)$ *DTS* $\times$ $'a$ $\times$ $('a, 'b)$ *rabin-condition*
type-synonym $('a, 'b)$ *generalized-rabin-automaton* $= ('a, 'b)$ *DTS* $\times$ $'a$ $\times$ $('a, 'b)$ *generalized-rabin-condition*

definition $accepting\text{-}pair_R :: ('a, 'b)$ *DTS* $\Rightarrow$ $'a \Rightarrow ('a, 'b)$ *rabin-pair* $\Rightarrow$ $'b$ *word* $\Rightarrow$ *bool*
where
$accepting\text{-}pair_R\ \delta\ q_0\ P\ w \equiv limit\ (run_t\ \delta\ q_0\ w) \cap fst\ P = \{\} \wedge limit\ (run_t\ \delta\ q_0\ w) \cap snd\ P \neq \{\}$

definition $accept_R :: ('a, 'b)$ *rabin-automaton* $\Rightarrow$ $'b$ *word* $\Rightarrow$ *bool*
where
$accept_R\ R\ w \equiv (\exists P \in (snd\ (snd\ R)).\ accepting\text{-}pair_R\ (fst\ R)\ (fst\ (snd\ R))\ P\ w)$

definition $accepting\text{-}pair_{GR} :: ('a, 'b)$ *DTS* $\Rightarrow$ $'a \Rightarrow ('a, 'b)$ *generalized-rabin-pair* $\Rightarrow$ $'b$ *word* $\Rightarrow$ *bool*
where
$accepting\text{-}pair_{GR}\ \delta\ q_0\ P\ w \equiv limit\ (run_t\ \delta\ q_0\ w) \cap fst\ P = \{\} \wedge (\forall I \in snd\ P.\ limit\ (run_t\ \delta\ q_0\ w) \cap I \neq \{\})$

definition $accept_{GR} :: ('a, 'b)$ *generalized-rabin-automaton* $\Rightarrow$ $'b$ *word* $\Rightarrow$ *bool*
where
$accept_{GR}\ R\ w \equiv (\exists (Fin, Inf) \in (snd\ (snd\ R)).\ accepting\text{-}pair_{GR}\ (fst\ R)\ (fst\ (snd\ R))\ (Fin, Inf)\ w)$

declare $accepting\text{-}pair_R\text{-}def[simp]$

declare *accepting-pair$_{GR}$-def*[*simp*]

lemma *accepting-pair$_R$-simp*[*simp*]:
accepting-pair$_R$ δ q_0 (F,I) $w \equiv$ *limit* (*run$_t$* δ q_0 w) $\cap$ $F = \{\} \wedge$ *limit* (*run$_t$* δ q_0 w) $\cap$ $I \neq \{\}$
⟨*proof*⟩

lemma *accepting-pair$_{GR}$-simp*[*simp*]:
accepting-pair$_{GR}$ δ q_0 (F, $\mathcal{I}$) $w \equiv$ *limit* (*run$_t$* δ q_0 w) $\cap$ $F = \{\} \wedge (\forall I \in \mathcal{I}.$ *limit* (*run$_t$* δ q_0 w) $\cap$ $I \neq \{\})$
⟨*proof*⟩

lemma *accept$_R$-simp*[*simp*]:
accept$_R$ (δ, q_0, α) $w = (\exists (Fin, Inf) \in \alpha.$ *limit* (*run$_t$* δ q_0 w) $\cap$ $Fin = \{\}$ $\wedge$ *limit* (*run$_t$* δ q_0 w) $\cap$ $Inf \neq \{\})$
⟨*proof*⟩

lemma *accept$_{GR}$-simp*[*simp*]:
accept$_{GR}$ (δ, q_0, α) $w \longleftrightarrow (\exists (Fin, Inf) \in \alpha.$ *limit* (*run$_t$* δ q_0 w) $\cap$ $Fin = \{\} \wedge (\forall I \in Inf.$ *limit* (*run$_t$* δ q_0 w) $\cap$ $I \neq \{\}))$
⟨*proof*⟩

lemma *accept$_{GR}$-simp2*:
accept$_{GR}$ (δ, q_0, α) $w \longleftrightarrow (\exists P \in \alpha.$ *accepting-pair$_{GR}$* δ q_0 P $w)$
⟨*proof*⟩

type-synonym ('a, 'b) *LTS* = ('a, 'b) *transition set*

definition *LTS-is-inf-run* :: ('q,'a) *LTS* ⇒ 'a *word* ⇒ 'q *word* ⇒ *bool*
where
LTS-is-inf-run Δ w $r \longleftrightarrow (\forall i.$ (r i, w i, r (*Suc* i)) $\in \Delta)$

fun *accept$_R$-LTS* :: (('a, 'b) *LTS* × 'a × ('a, 'b) *rabin-condition*) ⇒ 'b *word* ⇒ *bool*
where
accept$_R$-LTS (δ, q_0, α) $w \longleftrightarrow (\exists (Fin, Inf) \in \alpha. \exists r.$
LTS-is-inf-run δ w $r \wedge$ r 0 $= q_0$
$\wedge$ *limit* ($\lambda i.$ (r i, w i, r (*Suc* i))) $\cap$ $Fin = \{\}$
$\wedge$ *limit* ($\lambda i.$ (r i, w i, r (*Suc* i))) $\cap$ $Inf \neq \{\})$

definition *accepting-pair$_{GR}$-LTS* :: ('a, 'b) *LTS* ⇒ 'a ⇒ ('a, 'b) *generalized-rabin-pair* ⇒ 'b *word* ⇒ *bool*
where
accepting-pair$_{GR}$-LTS δ q_0 P $w \equiv \exists r.$ *LTS-is-inf-run* δ w $r \wedge$ r 0 $= q_0$

$\wedge$ *limit* ($\lambda i.$ (*r i*, *w i*, *r* (*Suc i*))) $\cap$ *fst P* = {}
$\wedge$ ($\forall I \in$ *snd P*. *limit* ($\lambda i.$ (*r i*, *w i*, *r* (*Suc i*))) $\cap$ *I* $\neq$ {})

fun *accept*$_{GR}$*-LTS* :: (('*a*, '*b*) *LTS* $\times$ '*a* $\times$ ('*a*, '*b*) *generalized-rabin-condition*) $\Rightarrow$ '*b word* $\Rightarrow$ *bool*
where
accept$_{GR}$*-LTS* (δ, q_0, α) *w* = ($\exists$ (*Fin*, *Inf*) $\in \alpha$. *accepting-pair*$_{GR}$*-LTS* δ q_0 (*Fin*, *Inf*) *w*)

lemma *accept*$_{GR}$*-LTS-E*:
assumes *accept*$_{GR}$*-LTS R w*
obtains *F I* **where** (*F*, *I*) $\in$ *snd* (*snd R*)
and *accepting-pair*$_{GR}$*-LTS* (*fst R*) (*fst* (*snd R*)) (*F*, *I*) *w*
⟨*proof*⟩

lemma *accept*$_{GR}$*-LTS-I*:
accepting-pair$_{GR}$*-LTS* δ q_0 (*F*, $\mathcal{I}$) *w* $\Longrightarrow$ (*F*, $\mathcal{I}$) $\in \alpha \Longrightarrow$ *accept*$_{GR}$*-LTS* (δ, q_0, α) *w*
⟨*proof*⟩

lemma *accept*$_{GR}$*-I*:
accepting-pair$_{GR}$ δ q_0 (*F*, $\mathcal{I}$) *w* $\Longrightarrow$ (*F*, $\mathcal{I}$) $\in \alpha \Longrightarrow$ *accept*$_{GR}$ (δ, q_0, α) *w*
⟨*proof*⟩

lemma *transfer-accept*:
accepting-pair$_R$ δ q_0 (*F*, *I*) *w* $\longleftrightarrow$ *accepting-pair*$_{GR}$ δ q_0 (*F*, {*I*}) *w*
accept$_R$ (δ, q_0, α) *w* $\longleftrightarrow$ *accept*$_{GR}$ (δ, q_0, (λ(*F*, *I*). (*F*, {*I*})) ‘ α) *w*
⟨*proof*⟩

7.1 Restriction Lemmas

lemma *accepting-pair*$_{GR}$*-restrict*:
assumes *range w* $\subseteq \Sigma$
shows *accepting-pair*$_{GR}$ δ q_0 (*F*, $\mathcal{I}$) *w* = *accepting-pair*$_{GR}$ δ q_0 (*F* $\cap$ *reach*$_t$ Σ δ q_0, ($\lambda I.$ *I* $\cap$ *reach*$_t$ Σ δ q_0) ‘ $\mathcal{I}$) *w*
⟨*proof*⟩

lemma *accept*$_{GR}$*-restrict*:
assumes *range w* $\subseteq \Sigma$
shows *accept*$_{GR}$ (δ, q_0, {(*f x*, *g x*) | *x*. *P x*}) *w* = *accept*$_{GR}$ (δ, q_0, {(*f x* $\cap$ *reach*$_t$ Σ δ q_0, ($\lambda I.$ *I* $\cap$ *reach*$_t$ Σ δ q_0) ‘ *g x*) | *x*. *P x*}) *w*
⟨*proof*⟩

lemma *accepting-pair*$_R$*-restrict*:

assumes *range* $w \subseteq \Sigma$
shows *accepting-pair*$_R$ δ q_0 (F, I) w = *accepting-pair*$_R$ δ q_0 $(F \cap reach_t\ \Sigma\ \delta\ q_0, I \cap reach_t\ \Sigma\ \delta\ q_0)$ w
⟨*proof*⟩

lemma *accept*$_R$*-restrict*:
assumes *range* $w \subseteq \Sigma$
shows *accept*$_R$ $(\delta, q_0, \{(f\ x, g\ x) \mid x.\ P\ x\})$ w = *accept*$_R$ $(\delta, q_0, \{(f\ x \cap reach_t\ \Sigma\ \delta\ q_0, g\ x \cap reach_t\ \Sigma\ \delta\ q_0) \mid x.\ P\ x\})$ w
⟨*proof*⟩

7.2 Abstraction Lemmas

lemma *accepting-pair*$_{GR}$*-abstract*:
assumes *finite* $(reach_t\ \Sigma\ \delta\ q_0)$
and *finite* $(reach_t\ \Sigma\ \delta'\ q_0{}')$
assumes *range* $w \subseteq \Sigma$
assumes $run_t\ \delta\ q_0\ w = f\ o\ (run_t\ \delta'\ q_0{}'\ w)$
assumes $\bigwedge t.\ t \in reach_t\ \Sigma\ \delta'\ q_0{}' \Longrightarrow f\ t \in F \longleftrightarrow t \in F'$
assumes $\bigwedge t\ i.\ i \in \mathcal{I} \Longrightarrow t \in reach_t\ \Sigma\ \delta'\ q_0{}' \Longrightarrow f\ t \in I\ i \longleftrightarrow t \in I'\ i$
shows *accepting-pair*$_{GR}$ δ q_0 $(F, \{I\ i \mid i.\ i \in \mathcal{I}\})$ $w \longleftrightarrow$ *accepting-pair*$_{GR}$ δ' $q_0{}'$ $(F', \{I'\ i \mid i.\ i \in \mathcal{I}\})$ w
⟨*proof*⟩

lemma *accepting-pair*$_R$*-abstract*:
assumes *finite* $(reach_t\ \Sigma\ \delta\ q_0)$
and *finite* $(reach_t\ \Sigma\ \delta'\ q_0{}')$
assumes *range* $w \subseteq \Sigma$
assumes $run_t\ \delta\ q_0\ w\ = f\ o\ (run_t\ \delta'\ q_0{}'\ w)$
assumes $\bigwedge t.\ t \in reach_t\ \Sigma\ \delta'\ q_0{}' \Longrightarrow f\ t \in F \longleftrightarrow t \in F'$
assumes $\bigwedge t.\ t \in reach_t\ \Sigma\ \delta'\ q_0{}' \Longrightarrow f\ t \in I \longleftrightarrow t \in I'$
shows *accepting-pair*$_R$ δ q_0 (F, I) $w \longleftrightarrow$ *accepting-pair*$_R$ δ' $q_0{}'$ (F', I') w
⟨*proof*⟩

7.3 LTS Lemmas

lemma *accepting-pair*$_{GR}$*-LTS*:
assumes *range* $w \subseteq \Sigma$
shows *accepting-pair*$_{GR}$ δ q_0 $(F, \mathcal{I})$ $w \longleftrightarrow$ *accepting-pair*$_{GR}$*-LTS* $(reach_t\ \Sigma\ \delta\ q_0)$ q_0 $(F, \mathcal{I})$ w
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

lemma *accept$_{GR}$-LTS*:
assumes *range* $w \subseteq \Sigma$
shows *accept$_{GR}$* $(\delta,\ q_0,\ \alpha)\ w \longleftrightarrow$ *accept$_{GR}$-LTS* $(\mathit{reach}_t\ \Sigma\ \delta\ q_0,\ q_0,\ \alpha)\ w$

⟨*proof*⟩

lemma *accept$_R$-LTS*:
assumes *range* $w \subseteq \Sigma$
shows *accept$_R$* $(\delta,\ q_0,\ \alpha)\ w \longleftrightarrow$ *accept$_R$-LTS* $(\mathit{reach}_t\ \Sigma\ \delta\ q_0,\ q_0,\ \alpha)\ w$
⟨*proof*⟩

7.4 Combination Lemmas

lemma *combine-rabin-pairs*:
$(\bigwedge x.\ x \in I \Longrightarrow$ *accepting-pair$_R$* $\delta\ q_0\ (f\ x,\ g\ x)\ w) \Longrightarrow$ *accepting-pair$_{GR}$* $\delta\ q_0\ (\bigcup\{f\ x \mid x.\ x \in I\},\ \{g\ x \mid x.\ x \in I\})\ w$
⟨*proof*⟩

lemma *combine-rabin-pairs-UNIV*:
accepting-pair$_R$ $\delta\ q_0\ (\mathit{fin},\ \mathit{UNIV})\ w \Longrightarrow$ *accepting-pair$_{GR}$* $\delta\ q_0\ (\mathit{fin}',\ \mathit{inf}')\ w \Longrightarrow$ *accepting-pair$_{GR}$* $\delta\ q_0\ (\mathit{fin} \cup \mathit{fin}',\ \mathit{inf}')\ w$
⟨*proof*⟩

end

8 Auxiliary List Facts

theory *List2*
imports *Main HOL−Library.Omega-Words-Fun List−Index.List-Index*
begin

8.1 remdups_fwd

fun *remdups-fwd-acc*
where
remdups-fwd-acc Acc [] = []
| *remdups-fwd-acc Acc* (*x*#*xs*) = (*if x* ∈ *Acc then* [] *else* [*x*]) @ *remdups-fwd-acc* (*insert x Acc*) *xs*

lemma *remdups-fwd-acc-append*[*simp*]:
remdups-fwd-acc Acc (*xs*@*ys*) = (*remdups-fwd-acc Acc xs*) @ (*remdups-fwd-acc* (*Acc* ∪ *set xs*) *ys*)
⟨*proof*⟩

lemma *remdups-fwd-acc-set*[*simp*]:
set (*remdups-fwd-acc Acc xs*) = *set xs* − *Acc*
⟨*proof*⟩

lemma *remdups-fwd-acc-distinct*:
distinct (*remdups-fwd-acc Acc xs*)
⟨*proof*⟩

lemma *remdups-fwd-acc-empty*:
set xs ⊆ *Acc* ⟷ *remdups-fwd-acc Acc xs* = []
⟨*proof*⟩

lemma *remdups-fwd-acc-drop*:
set ys ⊆ *Acc* ∪ *set xs* ⟹ *remdups-fwd-acc Acc* (*xs* @ *ys* @ *zs*) = *remdups-fwd-acc Acc* (*xs* @ *zs*)
⟨*proof*⟩

lemma *remdups-fwd-acc-filter*:
remdups-fwd-acc Acc (*filter P xs*) = *filter P* (*remdups-fwd-acc Acc xs*)
⟨*proof*⟩

fun *remdups-fwd*
where
remdups-fwd xs = *remdups-fwd-acc* {} *xs*

lemma *remdups-fwd-eq*:
remdups-fwd xs = (*rev o remdups o rev*) *xs*
⟨*proof*⟩

lemma *remdups-fwd-set*[*simp*]:
set (*remdups-fwd xs*) = *set xs*
⟨*proof*⟩

lemma *remdups-fwd-distinct*:
distinct (*remdups-fwd xs*)
⟨*proof*⟩

lemma *remdups-fwd-filter*:
remdups-fwd (*filter P xs*) = *filter P* (*remdups-fwd xs*)
⟨*proof*⟩

8.2 Split Lemmas

lemma *map-splitE*:

assumes *map f xs* = *ys* @ *zs*
obtains *us vs* **where** *xs* = *us* @ *vs* **and** *map f us* = *ys* **and** *map f vs* = *zs*
⟨*proof*⟩

lemma *filter-split′*:
filter P xs = *ys* @ *zs* ⟹ ∃ *us vs*. *xs* = *us* @ *vs* ∧ *filter P us* = *ys* ∧ *filter P vs* = *zs*
⟨*proof*⟩

lemma *filter-splitE*:
assumes *filter P xs* = *ys* @ *zs*
obtains *us vs* **where** *xs* = *us* @ *vs* **and** *filter P us* = *ys* **and** *filter P vs* = *zs*
⟨*proof*⟩

lemma *filter-map-splitE*:
assumes *filter P* (*map f xs*) = *ys* @ *zs*
obtains *us vs* **where** *xs* = *us* @ *vs* **and** *filter P* (*map f us*) = *ys* **and** *filter P* (*map f vs*) = *zs*
⟨*proof*⟩

lemma *filter-map-split-iff*:
filter P (*map f xs*) = *ys* @ *zs* ⟷ (∃ *us vs*. *xs* = *us* @ *vs* ∧ *filter P* (*map f us*) = *ys* ∧ *filter P* (*map f vs*) = *zs*)
⟨*proof*⟩

lemma *list-empty-prefix*:
xs @ *y* # *zs* = *y* # *us* ⟹ *y* ∉ *set xs* ⟹ *xs* = []
⟨*proof*⟩

lemma *remdups-fwd-split*:
remdups-fwd-acc Acc xs = *ys* @ *zs* ⟹ ∃ *us vs*. *xs* = *us* @ *vs* ∧ *remdups-fwd-acc Acc us* = *ys* ∧ *remdups-fwd-acc* (*Acc* ∪ *set ys*) *vs* = *zs*
⟨*proof*⟩

lemma *remdups-fwd-split-exact*:
assumes *remdups-fwd-acc Acc xs* = *ys* @ *x* # *zs*
shows ∃ *us vs*. *xs* = *us* @ *x* # *vs* ∧ *x* ∉ *Acc* ∧ *x* ∉ *set ys* ∧ *remdups-fwd-acc Acc us* = *ys* ∧ *remdups-fwd-acc* (*Acc* ∪ *set ys* ∪ {*x*}) *vs* = *zs*
⟨*proof*⟩

lemma *remdups-fwd-split-exactE*:
assumes *remdups-fwd-acc Acc xs* = *ys* @ *x* # *zs*

obtains *us vs* **where** $xs = us \mathbin{@} x \# vs$ **and** $x \notin set\ us$ **and** *remdups-fwd-acc Acc us* $= ys$ **and** *remdups-fwd-acc* $(Acc \cup set\ ys \cup \{x\})\ vs = zs$
⟨*proof*⟩

lemma *remdups-fwd-split-exact-iff*:
remdups-fwd-acc Acc xs $= ys \mathbin{@} x \# zs \longleftrightarrow$
$(\exists\, us\ vs.\ xs = us \mathbin{@} x \# vs \wedge x \notin Acc \wedge x \notin set\ us \wedge$ *remdups-fwd-acc Acc us* $= ys \wedge$ *remdups-fwd-acc* $(Acc \cup set\ ys \cup \{x\})\ vs = zs)$
⟨*proof*⟩

lemma *sorted-pre*:
$(\bigwedge x\ y\ xs\ ys.\ zs = xs \mathbin{@} [x, y] \mathbin{@} ys \Longrightarrow x \leq y) \Longrightarrow sorted\ zs$
⟨*proof*⟩

lemma *sorted-list*:
assumes $x \in set\ xs$ **and** $y \in set\ xs$
assumes *sorted* $(map\ f\ xs)$ **and** $f\ x < f\ y$
shows $\exists\, xs'\ xs''\ xs'''.\ xs = xs' \mathbin{@} x \# xs'' \mathbin{@} y \# xs'''$
⟨*proof*⟩

lemma *takeWhile-foo*:
$x \notin set\ ys \Longrightarrow ys = takeWhile\ (\lambda y.\ y \neq x)\ (ys \mathbin{@} x \# zs)$
⟨*proof*⟩

lemma *takeWhile-split*:
$x \in set\ xs \Longrightarrow y \in set\ (takeWhile\ (\lambda y.\ y \neq x)\ xs) \Longrightarrow \exists\, xs'\ xs''\ xs'''.\ xs = xs' \mathbin{@} y \# xs'' \mathbin{@} x \# xs'''$
⟨*proof*⟩

lemma *takeWhile-distinct*:
$distinct\ (xs' \mathbin{@} x \# xs'') \Longrightarrow y \in set\ (takeWhile\ (\lambda y.\ y \neq x)\ (xs' \mathbin{@} x \# xs'')) \longleftrightarrow y \in set\ xs'$
⟨*proof*⟩

lemma *finite-lists-length-eqE*:
assumes *finite A*
shows *finite* $\{xs.\ set\ xs = A \wedge length\ xs = n\}$
⟨*proof*⟩

lemma *finite-set2*:
assumes *finite A*
shows *finite* $\{xs.\ set\ xs = A \wedge distinct\ xs\}$
⟨*proof*⟩

lemma *set-list*:
assumes *finite* (*set* ‘ *XS*)
assumes $\bigwedge xs.\ xs \in XS \Longrightarrow$ *distinct xs*
shows *finite XS*
⟨*proof*⟩

lemma *set-foldl-append*:
set (*foldl* (@) *i xs*) = *set i* ∪ $\bigcup$ {*set x* | *x*. *x* ∈ *set xs*}
⟨*proof*⟩

8.3 Short-circuited Version of *foldl*

fun *foldl-break* :: ('*b* ⇒ '*a* ⇒ '*b*) ⇒ ('*b* ⇒ *bool*) ⇒ '*b* ⇒ '*a list* ⇒ '*b*
where
foldl-break f s a [] = *a*
| *foldl-break f s a* (*x* # *xs*) = (*if s a then a else foldl-break f s* (*f a x*) *xs*)

lemma *foldl-break-append*:
foldl-break f s a (*xs* @ *ys*) = (*if s* (*foldl-break f s a xs*) *then foldl-break f s a xs else* (*foldl-break f s* (*foldl-break f s a xs*) *ys*))
⟨*proof*⟩

8.4 Suffixes

fun *suffixes*
where
suffixes [] = []
| *suffixes* (*x*#*xs*) = (*suffixes xs*) @ [*x*#*xs*]

lemma *suffixes-append*:
suffixes (*xs* @ *ys*) = (*suffixes ys*) @ (*map* (λ*zs*. *zs* @ *ys*) (*suffixes xs*))
⟨*proof*⟩

lemma *suffixes-alt-def*:
suffixes xs = *rev* (*prefix* (*length xs*) (λ*i*. *drop i xs*))
⟨*proof*⟩

end

9 Translation to Deterministic Transition-Based Rabin Automata

theory *Mojmir-Rabin*
imports *Main Mojmir Rabin Auxiliary/List2*

begin

locale *mojmir-to-rabin-def* = *mojmir-def*
begin

definition $fail_R :: ('b \Rightarrow nat\ option, 'a)\ transition\ set$
where
$fail_R = \{(r, \nu, s) \mid r\ \nu\ s\ q\ q'.\ r\ q \neq None \wedge q' = \delta\ q\ \nu \wedge q' \notin F \wedge sink\ q'\}$

definition $succeed_R :: nat \Rightarrow ('b \Rightarrow nat\ option, 'a)\ transition\ set$
where
$succeed_R\ i = \{(r, \nu, s) \mid r\ \nu\ s\ q.\ r\ q = Some\ i \wedge q \notin F - \{q_0\} \wedge (\delta\ q\ \nu) \in F\}$

definition $merge_R :: nat \Rightarrow ('b \Rightarrow nat\ option, 'a)\ transition\ set$
where
$merge_R\ i = \{(r, \nu, s) \mid r\ \nu\ s\ q\ q'\ j.\ r\ q = Some\ j \wedge j < i \wedge q' = \delta\ q\ \nu \wedge ((\exists q''.\ q' = \delta\ q''\ \nu \wedge r\ q'' \neq None \wedge q'' \neq q) \vee q' = q_0) \wedge q' \notin F\}$

abbreviation Q_R
where
$Q_R \equiv reach\ \Sigma\ step\ initial$

abbreviation $q_{\mathcal{R}}$
where
$q_{\mathcal{R}} \equiv initial$

abbreviation $\delta_{\mathcal{R}}$
where
$\delta_{\mathcal{R}} \equiv step$

abbreviation $Acc_{\mathcal{R}}$
where
$Acc_{\mathcal{R}}\ j \equiv (fail_R \cup merge_R\ j, succeed_R\ j)$

abbreviation $\mathcal{R}$
where
$\mathcal{R} \equiv (\delta_{\mathcal{R}}, q_{\mathcal{R}}, \{Acc_{\mathcal{R}}\ j \mid j.\ j < max\text{-}rank\})$

end

9.1 Well-formedness

lemma *function-set-finite*:
assumes *finite R*
assumes *finite A*
shows *finite* $\{f.\ (\forall x.\ x \notin R \longrightarrow f\ x = c) \wedge (\forall x.\ x \in R \longrightarrow f\ x \in A)\}$
(**is** *finite ?F*)
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *wellformed-*$\mathcal{R}$:
shows *finite* (*reach* Σ *step initial*)
⟨*proof*⟩

locale *mojmir-to-rabin* = *mojmir* + *mojmir-to-rabin-def* **begin**

9.2 Correctness

lemma *imp-and-not-imp-eq*:
assumes $P \Longrightarrow Q$
assumes $\neg P \Longrightarrow \neg Q$
shows $P = Q$
⟨*proof*⟩

lemma *finite-limit-intersection*:
assumes *finite* (*range f*)
assumes $\bigwedge x{::}nat.\ x \in A \longleftrightarrow (f\ x) \in B$
shows *finite* $A \longleftrightarrow$ *limit* $f \cap B = \{\}$
⟨*proof*⟩

lemma *finite-range-run*:
defines $r \equiv run_t\ \delta_{\mathcal{R}}\ q_{\mathcal{R}}\ w$
shows *finite* (*range r*)
⟨*proof*⟩

theorem *mojmir-accept-iff-rabin-accept-rank*:
shows (*finite* (*fail*) $\wedge$ *finite* (*merge i*) $\wedge$ *infinite* (*succeed i*))
$\longleftrightarrow$ *accepting-pair*$_R$ $\delta_{\mathcal{R}}$ $q_{\mathcal{R}}$ ($Acc_{\mathcal{R}}$ *i*) *w*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

theorem *mojmir-accept-iff-rabin-accept*:
accept $\longleftrightarrow$ *accept*$_R$ $\mathcal{R}$ *w*
⟨*proof*⟩

definition *smallest-accepting-rank*$_{\mathcal{R}}$ :: *nat option*
where
smallest-accepting-rank$_{\mathcal{R}}$ ≡ (*if accept*$_R$ $\mathcal{R}$ *w then*
Some (*LEAST i. accepting-pair*$_R$ $\delta_{\mathcal{R}}$ $q_{\mathcal{R}}$ (*Acc*$_{\mathcal{R}}$ *i*) *w*) *else None*)

theorem *Mojmir-rabin-smallest-accepting-rank*:
smallest-accepting-rank = *smallest-accepting-rank*$_{\mathcal{R}}$
⟨*proof*⟩

lemma *smallest-accepting-rank*$_{\mathcal{R}}$*-properties*:
smallest-accepting-rank$_{\mathcal{R}}$ = *Some i* ⟹ *accepting-pair*$_R$ $\delta_{\mathcal{R}}$ $q_{\mathcal{R}}$ (*Acc*$_{\mathcal{R}}$ *i*) *w*
⟨*proof*⟩

end

end

10 LTL (in Negation-Normal-Form, FGXU-Syntax)

theory *LTL-FGXU*
imports *Main HOL−Library.Omega-Words-Fun*
begin

Inspired/Based on schimpf/LTL

10.1 Syntax

datatype (*vars*: *'a*) *ltl* =
LTLTrue (*true*)
| *LTLFalse* (*false*)
| *LTLProp 'a* (*p'(-')*)
| *LTLPropNeg 'a* (*np'(-')* [*86*] *85*)
| *LTLAnd 'a ltl 'a ltl* (*- and -* [*83*,*83*] *82*)
| *LTLOr 'a ltl 'a ltl* (*- or -* [*82*,*82*] *81*)
| *LTLNext 'a ltl* (*X -* [*88*] *87*)
| *LTLGlobal* (*theG*: *'a ltl*) (*G -* [*85*] *84*)
| *LTLFinal 'a ltl* (*F -* [*84*] *83*)
| *LTLUntil 'a ltl 'a ltl* (*- U -* [*87*,*87*] *86*)

10.2 Semantics

fun *ltl-semantics* :: [*'a set word*, *'a ltl*] ⇒ *bool* (**infix** ⊨ *80*)
where

$w \models \mathit{true} = \mathit{True}$
| $w \models \mathit{false} = \mathit{False}$
| $w \models p(q) = (q \in w\ 0)$
| $w \models \mathit{np}(q) = (q \notin w\ 0)$
| $w \models \varphi\ \mathit{and}\ \psi = (w \models \varphi \wedge w \models \psi)$
| $w \models \varphi\ \mathit{or}\ \psi = (w \models \varphi \vee w \models \psi)$
| $w \models X\ \varphi = (\mathit{suffix}\ 1\ w \models \varphi)$
| $w \models G\ \varphi = (\forall k.\ \mathit{suffix}\ k\ w \models \varphi)$
| $w \models F\ \varphi = (\exists k.\ \mathit{suffix}\ k\ w \models \varphi)$
| $w \models \varphi\ U\ \psi = (\exists k.\ \mathit{suffix}\ k\ w \models \psi \wedge (\forall j < k.\ \mathit{suffix}\ j\ w \models \varphi))$

fun *ltl-prop-entailment* :: [*'a ltl set*, *'a ltl*] ⇒ *bool* (**infix** $\models_P$ *80*)
where
$\mathcal{A} \models_P \mathit{true} = \mathit{True}$
| $\mathcal{A} \models_P \mathit{false} = \mathit{False}$
| $\mathcal{A} \models_P \varphi\ \mathit{and}\ \psi = (\mathcal{A} \models_P \varphi \wedge \mathcal{A} \models_P \psi)$
| $\mathcal{A} \models_P \varphi\ \mathit{or}\ \psi = (\mathcal{A} \models_P \varphi \vee \mathcal{A} \models_P \psi)$
| $\mathcal{A} \models_P \varphi = (\varphi \in \mathcal{A})$

10.2.1 Properties

lemma *LTL-G-one-step-unfolding*:
$w \models G\ \varphi \longleftrightarrow (w \models \varphi \wedge w \models X\ (G\ \varphi))$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *LTL-F-one-step-unfolding*:
$w \models F\ \varphi \longleftrightarrow (w \models \varphi \vee w \models X\ (F\ \varphi))$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *LTL-U-one-step-unfolding*:
$w \models \varphi\ U\ \psi \longleftrightarrow (w \models \psi \vee (w \models \varphi \wedge w \models X\ (\varphi\ U\ \psi)))$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *LTL-GF-infinitely-many-suffixes*:
$w \models G\ (F\ \varphi) = (\exists_\infty i.\ \mathit{suffix}\ i\ w \models \varphi)$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *LTL-FG-almost-all-suffixes*:
$w \models F\ G\ \varphi = (\forall_\infty i.\ \mathit{suffix}\ i\ w \models \varphi)$
(**is** *?lhs* = *?rhs*)

⟨*proof*⟩

lemma *LTL-FG-suffix*:
(*suffix i w*) ⊨ *F* (*G* φ) = *w* ⊨ *F* (*G* φ)
⟨*proof*⟩

lemma *LTL-GF-suffix*:
(*suffix i w*) ⊨ *G* (*F* φ) = *w* ⊨ *G* (*F* φ)
⟨*proof*⟩

lemma *LTL-suffix-G*:
w ⊨ *G* φ ⟹ *suffix i w* ⊨ *G* φ
⟨*proof*⟩

lemma *LTL-prop-entailment-monotonI*[*intro*]:
S ⊨*P* φ ⟹ *S* ⊆ *S'* ⟹ *S'* ⊨*P* φ
⟨*proof*⟩

lemma *ltl-models-equiv-prop-entailment*:
w ⊨ φ = {χ. *w* ⊨ χ} ⊨*P* φ
⟨*proof*⟩

10.2.2 Limit Behaviour of the G-operator

abbreviation *Only-G*
where
Only-G S ≡ ∀ *x* ∈ *S*. ∃ *y*. *x* = *G y*

lemma *ltl-G-stabilize*:
assumes *finite* 𝒢
assumes *Only-G* 𝒢
obtains *i* **where** ⋀χ *j*. χ ∈ 𝒢 ⟹ *suffix i w* ⊨ χ = *suffix* (*i* + *j*) *w* ⊨ χ
⟨*proof*⟩

lemma *ltl-G-stabilize-property*:
assumes *finite* 𝒢
assumes *Only-G* 𝒢
assumes ⋀χ *j*. χ ∈ 𝒢 ⟹ *suffix i w* ⊨ χ = *suffix* (*i* + *j*) *w* ⊨ χ
assumes *G* ψ ∈ 𝒢 ∩ {χ. *w* ⊨ *F* χ}
shows *suffix i w* ⊨ *G* ψ
⟨*proof*⟩

10.3 Subformulae

10.3.1 Propositions

fun *propos* :: *'a ltl* ⇒*'a ltl set*
where
  propos true = {}
| *propos false* = {}
| *propos* (*φ and ψ*) = *propos φ* ∪ *propos ψ*
| *propos* (*φ or ψ*) = *propos φ* ∪ *propos ψ*
| *propos φ* = {*φ*}

fun *nested-propos* :: *'a ltl* ⇒*'a ltl set*
where
  nested-propos true = {}
| *nested-propos false* = {}
| *nested-propos* (*φ and ψ*) = *nested-propos φ* ∪ *nested-propos ψ*
| *nested-propos* (*φ or ψ*) = *nested-propos φ* ∪ *nested-propos ψ*
| *nested-propos* (*F φ*) = {*F φ*} ∪ *nested-propos φ*
| *nested-propos* (*G φ*) = {*G φ*} ∪ *nested-propos φ*
| *nested-propos* (*X φ*) = {*X φ*} ∪ *nested-propos φ*
| *nested-propos* (*φ U ψ*) = {*φ U ψ*} ∪ *nested-propos φ* ∪ *nested-propos ψ*
| *nested-propos φ* = {*φ*}

lemma *finite-propos*:
  finite (*propos φ*) *finite* (*nested-propos φ*)
  ⟨*proof*⟩

lemma *propos-subset*:
  propos φ ⊆ *nested-propos φ*
  ⟨*proof*⟩

lemma *LTL-prop-entailment-restrict-to-propos*:
  S ⊨*P φ* = (*S* ∩ *propos φ*) ⊨*P φ*
  ⟨*proof*⟩

lemma *propos-foldl*:
  assumes ⋀*x y*. *propos* (*f x y*) = *propos x* ∪ *propos y*
  shows ⋃{*propos y* |*y*. *y* = *i* ∨ *y* ∈ *set xs*} = *propos* (*foldl f i xs*)
⟨*proof*⟩

10.3.2 G-Subformulae

Notation for paper: mathdsG

fun *G-nested-propos* :: *'a ltl* ⇒*'a ltl set* (**G**)
where
  G (*φ and ψ*) = **G** *φ* ∪ **G** *ψ*
| **G** (*φ or ψ*) = **G** *φ* ∪ **G** *ψ*
| **G** (*F φ*) = **G** *φ*
| **G** (*G φ*) = **G** *φ* ∪ {*G φ*}
| **G** (*X φ*) = **G** *φ*
| **G** (*φ U ψ*) = **G** *φ* ∪ **G** *ψ*
| **G** *φ* = {}

lemma *G-nested-finite*:
  finite (**G** *φ*)
  ⟨*proof*⟩

lemma *G-nested-propos-alt-def*:
  G *φ* = *nested-propos φ* ∩ {*ψ*. (∃ *x*. *ψ* = *G x*)}
  ⟨*proof*⟩

lemma *G-nested-propos-Only-G*:
  Only-G (**G** *φ*)
  ⟨*proof*⟩

lemma *G-not-in-G*:
  G φ ∉ **G** *φ*
⟨*proof*⟩

lemma *G-subset-G*:
  ψ ∈ **G** *φ* ⟹ **G** *ψ* ⊆ **G** *φ*
  G ψ ∈ **G** *φ* ⟹ **G** *ψ* ⊆ **G** *φ*
  ⟨*proof*⟩

lemma *𝒢-properties*:
  assumes *𝒢* ⊆ **G** *φ*
  shows *𝒢-finite*: *finite 𝒢* **and** *𝒢-elements*: *Only-G 𝒢*
  ⟨*proof*⟩

10.4 Propositional Implication and Equivalence

definition *ltl-prop-implies* :: ['*a ltl*, '*a ltl*] ⇒ *bool* (**infix** ⟶$_P$ 75)
where
  φ ⟶$_P$ *ψ* ≡ ∀ *𝒜*. *𝒜* ⊨$_P$ *φ* ⟶ *𝒜* ⊨$_P$ *ψ*

definition *ltl-prop-equiv* :: ['*a ltl*, '*a ltl*] ⇒ *bool* (**infix** ≡$_P$ 75)
where

$\varphi \equiv_P \psi \equiv \forall \mathcal{A}.\ \mathcal{A} \models_P \varphi \longleftrightarrow \mathcal{A} \models_P \psi$

lemma *ltl-prop-implies-equiv*:
$\varphi \longrightarrow_P \psi \wedge \psi \longrightarrow_P \varphi \longleftrightarrow \varphi \equiv_P \psi$
⟨*proof*⟩

lemma *ltl-prop-equiv-equivp*:
equivp $(\equiv_P)$
⟨*proof*⟩

lemma [*trans*]:
$\varphi \equiv_P \psi \Longrightarrow \psi \equiv_P \chi \Longrightarrow \varphi \equiv_P \chi$
⟨*proof*⟩

10.4.1 Quotient Type for Propositional Equivalence

quotient-type *'a ltl-prop-equiv-quotient* = *'a ltl* / $(\equiv_P)$
morphisms *Rep Abs*
⟨*proof*⟩

type-synonym *'a* ltl_P = *'a ltl-prop-equiv-quotient*

instantiation *ltl-prop-equiv-quotient* :: (*type*) *equal* **begin**

lift-definition *ltl-prop-equiv-quotient-eq-test* :: *'a* ltl_P ⇒ *'a* ltl_P ⇒ *bool* **is** $\lambda x\ y.\ x \equiv_P y$
⟨*proof*⟩

definition
eq: *equal-class.equal* ≡ *ltl-prop-equiv-quotient-eq-test*

instance
⟨*proof*⟩

end

lemma ltl_P*-abs-rep*: *Abs* (*Rep* φ) = φ
⟨*proof*⟩

lift-definition *ltl-prop-entails-abs* :: *'a ltl set* ⇒ *'a* ltl_P ⇒ *bool* (- ↑$\models_P$ -) **is** $(\models_P)$
⟨*proof*⟩

lift-definition *ltl-prop-implies-abs* :: *'a* ltl_P ⇒ *'a* ltl_P ⇒ *bool* (- ↑$\longrightarrow_P$ -)

is $(\longrightarrow_P)$
⟨*proof*⟩

10.4.2 Propositional Equivalence implies LTL Equivalence

lemma *ltl-prop-implication-implies-ltl-implication*:
$w \models \varphi \Longrightarrow \varphi \longrightarrow_P \psi \Longrightarrow w \models \psi$
⟨*proof*⟩

lemma *ltl-prop-equiv-implies-ltl-equiv*:
$\varphi \equiv_P \psi \Longrightarrow w \models \varphi = w \models \psi$
⟨*proof*⟩

10.5 Substitution

fun *subst* :: *'a ltl* ⇒ (*'a ltl* ⇀ *'a ltl*) ⇒ *'a ltl*
where
subst true m = true
| *subst false m = false*
| *subst* (φ *and* ψ) *m = subst* φ *m and subst* ψ *m*
| *subst* (φ *or* ψ) *m = subst* φ *m or subst* ψ *m*
| *subst* φ *m* = (*case m* φ *of Some* φ' ⇒ φ' | *None* ⇒ φ)

Based on Uwe Schoening's Translation Lemma (Logic for CS, p. 54)

lemma *ltl-prop-equiv-subst-S*:
$S \models_P$ *subst* φ *m* = ((S − *dom m*) ∪ {χ | χ χ'. $\chi \in$ *dom m* ∧ *m* χ = *Some* χ' ∧ $S \models_P \chi'$}) $\models_P \varphi$
⟨*proof*⟩

lemma *subst-respects-ltl-prop-entailment*:
$\varphi \longrightarrow_P \psi \Longrightarrow$ *subst* φ *m* $\longrightarrow_P$ *subst* ψ *m*
$\varphi \equiv_P \psi \Longrightarrow$ *subst* φ *m* $\equiv_P$ *subst* ψ *m*
⟨*proof*⟩

lemma *subst-respects-ltl-prop-entailment-generalized*:
($\bigwedge\mathcal{A}$. ($\bigwedge x$. $x \in S \Longrightarrow \mathcal{A} \models_P x$) $\Longrightarrow \mathcal{A} \models_P y$) $\Longrightarrow$ ($\bigwedge x$. $x \in S \Longrightarrow \mathcal{A} \models_P$ *subst x m*) $\Longrightarrow \mathcal{A} \models_P$ *subst y m*
⟨*proof*⟩

lemma *decomposable-function-subst*:
⟦*f true = true*; *f false = false*; $\bigwedge\varphi$ ψ. *f* (φ *and* ψ) = *f* φ *and f* ψ; $\bigwedge\varphi$ ψ. *f* (φ *or* ψ) = *f* φ *or f* ψ⟧ $\Longrightarrow$ *f* φ = *subst* φ ($\lambda\chi$. *Some* (*f* χ))
⟨*proof*⟩

10.6 Additional Operators

10.6.1 And

lemma *foldl-LTLAnd-prop-entailment*:
$S \models_P foldl\ LTLAnd\ i\ xs = (S \models_P i \wedge (\forall y \in set\ xs.\ S \models_P y))$
⟨*proof*⟩

fun $And :: 'a\ ltl\ list \Rightarrow 'a\ ltl$
where
$And\ [] = true$
| $And\ (x\#xs) = foldl\ LTLAnd\ x\ xs$

lemma *And-prop-entailment*:
$S \models_P And\ xs = (\forall x \in set\ xs.\ S \models_P x)$
⟨*proof*⟩

lemma *And-propos*:
$propos\ (And\ xs) = \bigcup \{propos\ x |\ x.\ x \in set\ xs\}$
⟨*proof*⟩

lemma *And-semantics*:
$w \models And\ xs = (\forall x \in set\ xs.\ w \models x)$
⟨*proof*⟩

lemma *And-append-syntactic*:
$xs \neq [] \Longrightarrow And\ (xs\ @\ ys) = And\ ((And\ xs)\#ys)$
⟨*proof*⟩

lemma *And-append-S*:
$S \models_P And\ (xs\ @\ ys) = S \models_P And\ xs\ and\ And\ ys$
⟨*proof*⟩

lemma *And-append*:
$And\ (xs\ @\ ys) \equiv_P And\ xs\ and\ And\ ys$
⟨*proof*⟩

10.6.2 Lifted Variant

lift-definition $and\text{-}abs :: 'a\ ltl_P \Rightarrow 'a\ ltl_P \Rightarrow 'a\ ltl_P$ (- ↑*and* -) **is** $\lambda x\ y.\ x\ and\ y$
⟨*proof*⟩

fun $And\text{-}abs :: 'a\ ltl_P\ list \Rightarrow 'a\ ltl_P$ (↑*And*)
where

$\uparrow And\ xs = foldl\ and\text{-}abs\ (Abs\ true)\ xs$

lemma *foldl-LTLAnd-prop-entailment-abs*:
$S \uparrow\models_P foldl\ and\text{-}abs\ i\ xs = (S \uparrow\models_P i \wedge (\forall y{\in}set\ xs.\ S \uparrow\models_P y))$
⟨*proof*⟩

lemma *And-prop-entailment-abs*:
$S \uparrow\models_P \uparrow And\ xs = (\forall x \in set\ xs.\ S \uparrow\models_P x)$
⟨*proof*⟩

lemma *and-abs-conjunction*:
$S \uparrow\models_P \varphi \uparrow and\ \psi \longleftrightarrow S \uparrow\models_P \varphi \wedge S \uparrow\models_P \psi$
⟨*proof*⟩

10.6.3 Or

lemma *foldl-LTLOr-prop-entailment*:
$S \models_P foldl\ LTLOr\ i\ xs = (S \models_P i \vee (\exists y \in set\ xs.\ S \models_P y))$
⟨*proof*⟩

fun $Or :: 'a\ ltl\ list \Rightarrow 'a\ ltl$
where
$Or\ [] = false$
$|\ Or\ (x\#xs) = foldl\ LTLOr\ x\ xs$

lemma *Or-prop-entailment*:
$S \models_P Or\ xs = (\exists x \in set\ xs.\ S \models_P x)$
⟨*proof*⟩

lemma *Or-propos*:
$propos\ (Or\ xs) = \bigcup \{propos\ x|\ x.\ x \in set\ xs\}$
⟨*proof*⟩

lemma *Or-semantics*:
$w \models Or\ xs = (\exists x \in set\ xs.\ w \models x)$
⟨*proof*⟩

lemma *Or-append-syntactic*:
$xs \neq [] \Longrightarrow Or\ (xs\ @\ ys) = Or\ ((Or\ xs)\#ys)$
⟨*proof*⟩

lemma *Or-append-S*:
$S \models_P Or\ (xs\ @\ ys) = S \models_P Or\ xs\ or\ Or\ ys$
⟨*proof*⟩

lemma *Or-append*:
Or (*xs* @ *ys*) $\equiv_P$ *Or xs or Or ys*
⟨*proof*⟩

10.6.4 $eval_G$

fun $eval_G$
where
$eval_G\ S\ (\varphi\ and\ \psi) = eval_G\ S\ \varphi\ and\ eval_G\ S\ \psi$
| $eval_G\ S\ (\varphi\ or\ \psi) = eval_G\ S\ \varphi\ or\ eval_G\ S\ \psi$
| $eval_G\ S\ (G\ \varphi) = (if\ G\ \varphi \in S\ then\ true\ else\ false)$
| $eval_G\ S\ \varphi = \varphi$

— Syntactic Properties

lemma $eval_G$-*And-map*:
$eval_G\ S\ (And\ xs) = And\ (map\ (eval_G\ S)\ xs)$
⟨*proof*⟩

lemma $eval_G$-*Or-map*:
$eval_G\ S\ (Or\ xs) = Or\ (map\ (eval_G\ S)\ xs)$
⟨*proof*⟩

lemma $eval_G$-*G-nested*:
G $(eval_G\ \mathcal{G}\ \varphi) \subseteq$ **G** φ
⟨*proof*⟩

lemma $eval_G$-*subst*:
$eval_G\ S\ \varphi = subst\ \varphi\ (\lambda\chi.\ Some\ (eval_G\ S\ \chi))$
⟨*proof*⟩

lemma $eval_G$-*prop-entailment*:
$S \models_P eval_G\ S\ \varphi \longleftrightarrow S \models_P \varphi$
⟨*proof*⟩

lemma $eval_G$-*respectfulness*:
$\varphi \longrightarrow_P \psi \Longrightarrow eval_G\ S\ \varphi \longrightarrow_P eval_G\ S\ \psi$
$\varphi \equiv_P \psi \Longrightarrow eval_G\ S\ \varphi \equiv_P eval_G\ S\ \psi$
⟨*proof*⟩

lemma $eval_G$-*respectfulness-generalized*:
$(\bigwedge\mathcal{A}.\ (\bigwedge x.\ x \in S \Longrightarrow \mathcal{A} \models_P x) \Longrightarrow \mathcal{A} \models_P y) \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow \mathcal{A} \models_P eval_G\ P\ x) \Longrightarrow \mathcal{A} \models_P eval_G\ P\ y$

⟨*proof*⟩

lift-definition $eval_G$-*abs* :: *'a ltl set* ⇒ *'a* ltl_P ⇒ *'a* ltl_P (↑$eval_G$) **is** $eval_G$
⟨*proof*⟩

10.7 Finite Quotient Set

If we restrict formulas to a finite set of propositions, the set of quotients of these is finite

lemma *Rep-Abs-prop-entailment*[*simp*]:
$A \models_P$ *Rep* (*Abs* φ) = $A \models_P \varphi$
⟨*proof*⟩

fun *sat-models* :: *'a ltl-prop-equiv-quotient* ⇒ *'a ltl set set*
where
sat-models a = {*A*. $A \models_P$ *Rep*(*a*)}

lemma *sat-models-invariant*:
$A \in$ *sat-models* (*Abs* φ) = $A \models_P \varphi$
⟨*proof*⟩

lemma *sat-models-inj*:
inj sat-models
⟨*proof*⟩

lemma *sat-models-finite-image*:
assumes *finite P*
shows *finite* (*sat-models* ‘ {*Abs* φ | φ. *nested-propos* $\varphi \subseteq P$})
⟨*proof*⟩

lemma *ltl-prop-equiv-quotient-restricted-to-P-finite*:
assumes *finite P*
shows *finite* {*Abs* φ | φ. *nested-propos* $\varphi \subseteq P$}
⟨*proof*⟩

locale *lift-ltl-transformer* =
fixes
f :: *'a ltl* ⇒ *'b* ⇒ *'a ltl*
assumes
respectfulness: $\varphi \equiv_P \psi \Longrightarrow f\ \varphi\ \nu \equiv_P f\ \psi\ \nu$
assumes
nested-propos-bounded: *nested-propos* (*f* φ ν) ⊆ *nested-propos* φ
begin

lift-definition *f-abs* :: *'a ltl$_P$* ⇒ *'b* ⇒ *'a ltl$_P$* **is** *f*
⟨*proof*⟩

lift-definition *f-foldl-abs* :: *'a ltl$_P$* ⇒ *'b list* ⇒ *'a ltl$_P$* **is** *foldl f*
⟨*proof*⟩

lemma *f-foldl-abs-alt-def*:
f-foldl-abs (*Abs* φ) *w* = *foldl f-abs* (*Abs* φ) *w*
⟨*proof*⟩

definition *abs-reach* :: *'a ltl-prop-equiv-quotient* ⇒ *'a ltl-prop-equiv-quotient set*
where
abs-reach Φ = {*foldl f-abs* Φ *w* |*w*. *True*}

lemma *finite-abs-reach*:
finite (*abs-reach* (*Abs* φ))
⟨*proof*⟩

end

end

11 af - Unfolding Functions

theory *af*
imports *Main LTL-FGXU Auxiliary/List2*
begin

11.1 af

fun *af-letter* :: *'a ltl* ⇒ *'a set* ⇒ *'a ltl*
where
af-letter true ν = *true*
| *af-letter false* ν = *false*
| *af-letter p*(*a*) ν = (*if a* ∈ ν *then true else false*)
| *af-letter* (*np*(*a*)) ν = (*if a* ∉ ν *then true else false*)
| *af-letter* (φ *and* ψ) ν = (*af-letter* φ ν) *and* (*af-letter* ψ ν)
| *af-letter* (φ *or* ψ) ν = (*af-letter* φ ν) *or* (*af-letter* ψ ν)
| *af-letter* (*X* φ) ν = φ
| *af-letter* (*G* φ) ν = (*G* φ) *and* (*af-letter* φ ν)
| *af-letter* (*F* φ) ν = (*F* φ) *or* (*af-letter* φ ν)
| *af-letter* (φ *U* ψ) ν = (φ *U* ψ *and* (*af-letter* φ ν)) *or* (*af-letter* ψ ν)

abbreviation *af* :: *'a ltl* ⇒ *'a set list* ⇒ *'a ltl* (*af*)
where
af φ *w* ≡ *foldl af-letter* φ *w*

lemma *af-decompose*:
af (φ *and* ψ) *w* = (*af* φ *w*) *and* (*af* ψ *w*)
af (φ *or* ψ) *w* = (*af* φ *w*) *or* (*af* ψ *w*)
⟨*proof*⟩

lemma *af-simps*[*simp*]:
af true w = *true*
af false w = *false*
af (*X* φ) (*x*#*xs*) = *af* φ (*xs*)
⟨*proof*⟩

lemma *af-F*:
af (*F* φ) *w* = *Or* (*F* φ # *map* (*af* φ) (*suffixes w*))
⟨*proof*⟩

lemma *af-G*:
af (*G* φ) *w* = *And* (*G* φ # *map* (*af* φ) (*suffixes w*))
⟨*proof*⟩

lemma *af-U*:
af (φ *U* ψ) (*x*#*xs*) = (*af* (φ *U* ψ) *xs and af* φ (*x*#*xs*)) *or af* ψ (*x*#*xs*)
⟨*proof*⟩

lemma *af-respectfulness*:
φ ⟶$_P$ ψ ⟹ *af-letter* φ ν ⟶$_P$ *af-letter* ψ ν
φ ≡$_P$ ψ ⟹ *af-letter* φ ν ≡$_P$ *af-letter* ψ ν
⟨*proof*⟩

lemma *af-respectfulness'*:
φ ⟶$_P$ ψ ⟹ *af* φ *w* ⟶$_P$ *af* ψ *w*
φ ≡$_P$ ψ ⟹ *af* φ *w* ≡$_P$ *af* ψ *w*
⟨*proof*⟩

lemma *af-nested-propos*:
nested-propos (*af-letter* φ ν) ⊆ *nested-propos* φ
⟨*proof*⟩

11.2 af_G

fun *af-G-letter* :: *'a ltl* ⇒ *'a set* ⇒ *'a ltl*
where
af-G-letter true ν = true
| *af-G-letter false ν = false*
| *af-G-letter p(a) ν = (if a ∈ ν then true else false)*
| *af-G-letter (np(a)) ν = (if a ∉ ν then true else false)*
| *af-G-letter (φ and ψ) ν = (af-G-letter φ ν) and (af-G-letter ψ ν)*
| *af-G-letter (φ or ψ) ν = (af-G-letter φ ν) or (af-G-letter ψ ν)*
| *af-G-letter (X φ) ν = φ*
| *af-G-letter (G φ) ν = (G φ)*
| *af-G-letter (F φ) ν = (F φ) or (af-G-letter φ ν)*
| *af-G-letter (φ U ψ) ν = (φ U ψ and (af-G-letter φ ν)) or (af-G-letter ψ ν)*

abbreviation af_G :: *'a ltl* ⇒ *'a set list* ⇒ *'a ltl*
where
af_G *φ w* ≡ (*foldl af-G-letter φ w*)

lemma af_G*-decompose*:
af_G *(φ and ψ) w = (*af_G *φ w) and (*af_G *ψ w)*
af_G *(φ or ψ) w = (*af_G *φ w) or (*af_G *ψ w)*
⟨*proof*⟩

lemma af_G*-simps*[*simp*]:
af_G *true w = true*
af_G *false w = false*
af_G *(G φ) w = G φ*
af_G *(X φ) (x#xs) =* af_G *φ (xs)*
⟨*proof*⟩

lemma af_G*-F*:
af_G *(F φ) w = Or (F φ # map (*af_G *φ) (suffixes w))*
⟨*proof*⟩

lemma af_G*-U*:
af_G *(φ U ψ) (x#xs) = (*af_G *(φ U ψ) xs and* af_G *φ (x#xs)) or* af_G *ψ (x#xs)*
⟨*proof*⟩

lemma af_G*-subsequence-U*:
af_G *(φ U ψ) (w [0 → Suc n]) = (*af_G *(φ U ψ) (w [1 → Suc n]) and* af_G *φ (w [0 → Suc n])) or* af_G *ψ (w [0 → Suc n])*

⟨*proof*⟩

lemma *af-G-respectfulness*:
$\varphi \longrightarrow_P \psi \Longrightarrow$ *af-G-letter* $\varphi\ \nu \longrightarrow_P$ *af-G-letter* $\psi\ \nu$
$\varphi \equiv_P \psi \Longrightarrow$ *af-G-letter* $\varphi\ \nu \equiv_P$ *af-G-letter* $\psi\ \nu$
⟨*proof*⟩

lemma *af-G-respectfulness'*:
$\varphi \longrightarrow_P \psi \Longrightarrow af_G\ \varphi\ w \longrightarrow_P af_G\ \psi\ w$
$\varphi \equiv_P \psi \Longrightarrow af_G\ \varphi\ w \equiv_P af_G\ \psi\ w$
⟨*proof*⟩

lemma *af-G-nested-propos*:
nested-propos (*af-G-letter* $\varphi\ \nu$) $\subseteq$ *nested-propos* φ
⟨*proof*⟩

lemma *af-G-letter-sat-core*:
Only-G $\mathcal{G} \Longrightarrow \mathcal{G} \models_P \varphi \Longrightarrow \mathcal{G} \models_P$ *af-G-letter* $\varphi\ \nu$
⟨*proof*⟩

lemma af_G*-sat-core*:
Only-G $\mathcal{G} \Longrightarrow \mathcal{G} \models_P \varphi \Longrightarrow \mathcal{G} \models_P af_G\ \varphi\ w$
⟨*proof*⟩

lemma af_G*-sat-core-generalized*:
Only-G $\mathcal{G} \Longrightarrow i \leq j \Longrightarrow \mathcal{G} \models_P af_G\ \varphi\ (w\ [0 \rightarrow i]) \Longrightarrow \mathcal{G} \models_P af_G\ \varphi\ (w\ [0 \rightarrow j])$
⟨*proof*⟩

lemma af_G*-eval*$_G$:
Only-G $\mathcal{G} \Longrightarrow \mathcal{G} \models_P af_G\ (eval_G\ \mathcal{G}\ \varphi)\ w \longleftrightarrow \mathcal{G} \models_P eval_G\ \mathcal{G}\ (af_G\ \varphi\ w)$
⟨*proof*⟩

lemma af_G*-keeps-F-and-S*:
assumes $ys \neq []$
assumes $S \models_P af_G\ \varphi\ ys$
shows $S \models_P af_G\ (F\ \varphi)\ (xs\ @\ ys)$
⟨*proof*⟩

11.3 G-Subformulae Simplification

lemma *G-af-simp*[*simp*]:
G $(af\ \varphi\ w) =$ **G** φ
⟨*proof*⟩

lemma *G-af$_G$-simp*[*simp*]:
G (*af$_G$* φ *w*) = **G** φ
⟨*proof*⟩

11.4 Relation between af and af_G

lemma *af-G-letter-free-F*:
G φ = {} ⟹ **G** (*af-letter* φ ν) = {}
G φ = {} ⟹ **G** (*af-G-letter* φ ν) = {}
⟨*proof*⟩

lemma *af-G-free*:
assumes **G** φ = {}
shows *af* φ *w* = *af$_G$* φ *w*
⟨*proof*⟩

lemma *af-equals-af$_G$-base-cases*:
af true w = *af$_G$ true w*
af false w = *af$_G$ false w*
af p(*a*) *w* = *af$_G$ p*(*a*) *w*
af (*np*(*a*)) *w* = *af$_G$* (*np*(*a*)) *w*
⟨*proof*⟩

lemma *af-implies-af$_G$*:
S ⊨$_P$ *af* φ *w* ⟹ *S* ⊨$_P$ *af$_G$* φ *w*
⟨*proof*⟩

lemma *af-implies-af$_G$-2*:
w ⊨ *af* φ *xs* ⟹ *w* ⊨ *af$_G$* φ *xs*
⟨*proof*⟩

lemma *af$_G$-implies-af-eval$_G$′*:
assumes *S* ⊨$_P$ *eval$_G$* $\mathcal{G}$ (*af$_G$* φ *w*)
assumes ⋀ψ. *G* ψ ∈ $\mathcal{G}$ ⟹ *S* ⊨$_P$ *G* ψ
assumes ⋀ψ *i*. *G* ψ ∈ $\mathcal{G}$ ⟹ *i* < *length w* ⟹ *S* ⊨$_P$ *eval$_G$* $\mathcal{G}$ (*af$_G$* ψ (*drop i w*))
shows *S* ⊨$_P$ *af* φ *w*
⟨*proof*⟩

lemma *af$_G$-implies-af-eval$_G$*:
assumes *S* ⊨$_P$ *eval$_G$* $\mathcal{G}$ (*af$_G$* φ (*w* [0→*j*]))
assumes ⋀ψ. *G* ψ ∈ $\mathcal{G}$ ⟹ *S* ⊨$_P$ *G* ψ
assumes ⋀ψ *i*. *G* ψ ∈ $\mathcal{G}$ ⟹ *i* ≤ *j* ⟹ *S* ⊨$_P$ *eval$_G$* $\mathcal{G}$ (*af$_G$* ψ (*w* [*i* →

$j]))$
shows $S \models_P af\ \varphi\ (w\ [0{\rightarrow}j])$
⟨*proof*⟩

11.5 Continuation

lemma *af-ltl-continuation*:
$(w \frown w') \models \varphi \longleftrightarrow w' \models af\ \varphi\ w$
⟨*proof*⟩

lemma *af-ltl-continuation-suffix*:
$w \models \varphi \longleftrightarrow suffix\ i\ w \models af\ \varphi\ (w[0 \rightarrow i])$
⟨*proof*⟩

lemma *af-G-ltl-continuation*:
$\forall \psi \in \mathbf{G}\ \varphi.\ w' \models \psi = (w \frown w') \models \psi \Longrightarrow (w \frown w') \models \varphi \longleftrightarrow w' \models af_G\ \varphi\ w$
⟨*proof*⟩

lemma *af$_G$-ltl-continuation-suffix*:
$\forall \psi \in \mathbf{G}\ \varphi.\ w \models \psi = (suffix\ i\ w) \models \psi \Longrightarrow w \models \varphi \longleftrightarrow suffix\ i\ w \models af_G\ \varphi\ (w\ [0 \rightarrow i])$
⟨*proof*⟩

11.6 Eager Unfolding *af* and *af$_G$*

fun $Unf :: 'a\ ltl \Rightarrow 'a\ ltl$
where
$Unf\ (F\ \varphi) = F\ \varphi\ or\ Unf\ \varphi$
| $Unf\ (G\ \varphi) = G\ \varphi\ and\ Unf\ \varphi$
| $Unf\ (\varphi\ U\ \psi) = (\varphi\ U\ \psi\ and\ Unf\ \varphi)\ or\ Unf\ \psi$
| $Unf\ (\varphi\ and\ \psi) = Unf\ \varphi\ and\ Unf\ \psi$
| $Unf\ (\varphi\ or\ \psi) = Unf\ \varphi\ or\ Unf\ \psi$
| $Unf\ \varphi = \varphi$

fun $Unf_G :: 'a\ ltl \Rightarrow 'a\ ltl$
where
$Unf_G\ (F\ \varphi) = F\ \varphi\ or\ Unf_G\ \varphi$
| $Unf_G\ (G\ \varphi) = G\ \varphi$
| $Unf_G\ (\varphi\ U\ \psi) = (\varphi\ U\ \psi\ and\ Unf_G\ \varphi)\ or\ Unf_G\ \psi$
| $Unf_G\ (\varphi\ and\ \psi) = Unf_G\ \varphi\ and\ Unf_G\ \psi$
| $Unf_G\ (\varphi\ or\ \psi) = Unf_G\ \varphi\ or\ Unf_G\ \psi$
| $Unf_G\ \varphi = \varphi$

fun *step* :: *'a ltl* ⇒ *'a set* ⇒ *'a ltl*
where
  step p(a) ν = (if a ∈ ν then true else false)
| *step (np(a)) ν = (if a ∉ ν then true else false)*
| *step (X φ) ν = φ*
| *step (φ and ψ) ν = step φ ν and step ψ ν*
| *step (φ or ψ) ν = step φ ν or step ψ ν*
| *step φ ν = φ*

fun *af-letter-opt*
where
  af-letter-opt φ ν = Unf (step φ ν)

fun *af-G-letter-opt*
where
  af-G-letter-opt φ ν = Unf$_G$ (step φ ν)

abbreviation *af-opt* :: *'a ltl* ⇒ *'a set list* ⇒ *'a ltl* ($af_{\mathfrak{U}}$)
where
  $af_{\mathfrak{U}}$ *φ w* ≡ (*foldl af-letter-opt φ w*)

abbreviation *af-G-opt* :: *'a ltl* ⇒ *'a set list* ⇒ *'a ltl* ($af_{G\mathfrak{U}}$)
where
  $af_{G\mathfrak{U}}$ *φ w* ≡ (*foldl af-G-letter-opt φ w*)

lemma *af-letter-alt-def*:
  af-letter φ ν = step (Unf φ) ν
  af-G-letter φ ν = step (Unf$_G$ φ) ν
  ⟨*proof*⟩

lemma *af-to-af-opt*:
  Unf (af φ w) = $af_{\mathfrak{U}}$ *(Unf φ) w*
  Unf$_G$ (af$_G$ φ w) = $af_{G\mathfrak{U}}$ *(Unf$_G$ φ) w*
  ⟨*proof*⟩

lemma *af-equiv*:
  af φ (w @ [ν]) = step ($af_{\mathfrak{U}}$ *(Unf φ) w) ν*
  ⟨*proof*⟩

lemma *af-equiv'*:
  af φ (w [0 → Suc i]) = step ($af_{\mathfrak{U}}$ *(Unf φ) (w [0 → i])) (w i)*
  ⟨*proof*⟩

11.7 Lifted Functions

lemma *respectfulness*:
$\varphi \longrightarrow_P \psi \Longrightarrow$ *af-letter-opt* $\varphi\ \nu \longrightarrow_P$ *af-letter-opt* $\psi\ \nu$
$\varphi \equiv_P \psi \Longrightarrow$ *af-letter-opt* $\varphi\ \nu \equiv_P$ *af-letter-opt* $\psi\ \nu$
$\varphi \longrightarrow_P \psi \Longrightarrow$ *af-G-letter-opt* $\varphi\ \nu \longrightarrow_P$ *af-G-letter-opt* $\psi\ \nu$
$\varphi \equiv_P \psi \Longrightarrow$ *af-G-letter-opt* $\varphi\ \nu \equiv_P$ *af-G-letter-opt* $\psi\ \nu$
$\varphi \longrightarrow_P \psi \Longrightarrow$ *step* $\varphi\ \nu \longrightarrow_P$ *step* $\psi\ \nu$
$\varphi \equiv_P \psi \Longrightarrow$ *step* $\varphi\ \nu \equiv_P$ *step* $\psi\ \nu$
$\varphi \longrightarrow_P \psi \Longrightarrow$ *Unf* $\varphi \longrightarrow_P$ *Unf* ψ
$\varphi \equiv_P \psi \Longrightarrow$ *Unf* $\varphi \equiv_P$ *Unf* ψ
$\varphi \longrightarrow_P \psi \Longrightarrow$ $Unf_G\ \varphi \longrightarrow_P Unf_G\ \psi$
$\varphi \equiv_P \psi \Longrightarrow$ $Unf_G\ \varphi \equiv_P Unf_G\ \psi$
⟨*proof*⟩

lemma *nested-propos*:
nested-propos (*step* $\varphi\ \nu$) $\subseteq$ *nested-propos* φ
nested-propos (*Unf* φ) $\subseteq$ *nested-propos* φ
nested-propos ($Unf_G\ \varphi$) $\subseteq$ *nested-propos* φ
nested-propos (*af-letter-opt* $\varphi\ \nu$) $\subseteq$ *nested-propos* φ
nested-propos (*af-G-letter-opt* $\varphi\ \nu$) $\subseteq$ *nested-propos* φ
⟨*proof*⟩

Lift functions and bind to new names

interpretation *af-abs*: *lift-ltl-transformer af-letter*
⟨*proof*⟩

definition *af-letter-abs* ($\uparrow af$)
where
$\uparrow af \equiv$ *af-abs.f-abs*

interpretation *af-G-abs*: *lift-ltl-transformer af-G-letter*
⟨*proof*⟩

definition *af-G-letter-abs* ($\uparrow af_G$)
where
$\uparrow af_G \equiv$ *af-G-abs.f-abs*

interpretation *af-abs-opt*: *lift-ltl-transformer af-letter-opt*
⟨*proof*⟩

definition *af-letter-abs-opt* ($\uparrow af_{\mathfrak{U}}$)
where
$\uparrow af_{\mathfrak{U}} \equiv$ *af-abs-opt.f-abs*

interpretation *af-G-abs-opt*: *lift-ltl-transformer af-G-letter-opt*
⟨*proof*⟩

definition *af-G-letter-abs-opt* ($\uparrow af_{G\mathfrak{U}}$)
where
$\uparrow af_{G\mathfrak{U}} \equiv$ *af-G-abs-opt.f-abs*

lift-definition *step-abs* :: $'a\ ltl_P \Rightarrow 'a\ set \Rightarrow 'a\ ltl_P$ ($\uparrow step$) **is** *step*
⟨*proof*⟩

lift-definition *Unf-abs* :: $'a\ ltl_P \Rightarrow 'a\ ltl_P$ ($\uparrow Unf$) **is** *Unf*
⟨*proof*⟩

lift-definition Unf_G*-abs* :: $'a\ ltl_P \Rightarrow 'a\ ltl_P$ ($\uparrow Unf_G$) **is** Unf_G
⟨*proof*⟩

11.7.1 Properties

lemma *af-G-letter-opt-sat-core*:
Only-G $\mathcal{G} \Longrightarrow \mathcal{G} \models_P \varphi \Longrightarrow \mathcal{G} \models_P$ *af-G-letter-opt* $\varphi\ \nu$
⟨*proof*⟩

lemma *af-G-letter-sat-core-lifted*:
Only-G $\mathcal{G} \Longrightarrow \mathcal{G} \models_P$ *Rep* $\varphi \Longrightarrow \mathcal{G} \models_P$ *Rep* (*af-G-letter-abs* $\varphi\ \nu$)
⟨*proof*⟩

lemma *af-G-letter-opt-sat-core-lifted*:
Only-G $\mathcal{G} \Longrightarrow \mathcal{G} \models_P$ *Rep* $\varphi \Longrightarrow \mathcal{G} \models_P$ *Rep* ($\uparrow af_{G\mathfrak{U}}\ \varphi\ \nu$)
⟨*proof*⟩

lemma *af-G-letter-abs-opt-split*:
$\uparrow Unf_G$ ($\uparrow step\ \Phi\ \nu$) = $\uparrow af_{G\mathfrak{U}}\ \Phi\ \nu$
⟨*proof*⟩

lemma *af-unfold*:
$\uparrow af = (\lambda\varphi\ \nu.\ \uparrow step\ (\uparrow Unf\ \varphi)\ \nu)$
⟨*proof*⟩

lemma *af-opt-unfold*:
$\uparrow af_{\mathfrak{U}} = (\lambda\varphi\ \nu.\ \uparrow Unf\ (\uparrow step\ \varphi\ \nu))$
⟨*proof*⟩

lemma *af-abs-equiv*:

foldl $\uparrow af\ \psi\ (xs\ @\ [x]) = \uparrow step\ (foldl\ \uparrow af_{\mathfrak{U}}\ (\uparrow Unf\ \psi)\ xs)\ x$
⟨*proof*⟩

lemma *Rep-Abs-equiv*:
$Rep\ (Abs\ \varphi) \equiv_P \varphi$
⟨*proof*⟩

lemma *Rep-step*:
$Rep\ (\uparrow step\ \Phi\ \nu) \equiv_P step\ (Rep\ \Phi)\ \nu$
⟨*proof*⟩

lemma *step-*$\mathcal{G}$:
$Only\text{-}G\ \mathcal{G} \Longrightarrow \mathcal{G} \models_P \varphi \Longrightarrow \mathcal{G} \models_P step\ \varphi\ \nu$
⟨*proof*⟩

lemma Unf_G*-*$\mathcal{G}$:
$Only\text{-}G\ \mathcal{G} \Longrightarrow \mathcal{G} \models_P \varphi \Longrightarrow \mathcal{G} \models_P Unf_G\ \varphi$
⟨*proof*⟩

hide-fact (**open**) *respectfulness nested-propos*

end

12 Logical Characterization Theorems

theory *Logical-Characterization*
imports *Main af Auxiliary/Preliminaries2*
begin

12.1 Eventually True G-Subformulae

fun $\mathcal{G}_{FG}$ :: *'a ltl* ⇒ *'a set word* ⇒ *'a ltl set*
where
$\mathcal{G}_{FG}\ true\ w = \{\}$
| $\mathcal{G}_{FG}\ (false)\ w = \{\}$
| $\mathcal{G}_{FG}\ (p(a))\ w = \{\}$
| $\mathcal{G}_{FG}\ (np(a))\ w = \{\}$
| $\mathcal{G}_{FG}\ (\varphi_1\ and\ \varphi_2)\ w = \mathcal{G}_{FG}\ \varphi_1\ w \cup \mathcal{G}_{FG}\ \varphi_2\ w$
| $\mathcal{G}_{FG}\ (\varphi_1\ or\ \varphi_2)\ w = \mathcal{G}_{FG}\ \varphi_1\ w \cup \mathcal{G}_{FG}\ \varphi_2\ w$
| $\mathcal{G}_{FG}\ (F\ \varphi)\ w = \mathcal{G}_{FG}\ \varphi\ w$
| $\mathcal{G}_{FG}\ (G\ \varphi)\ w = (if\ w \models F\ G\ \varphi\ then\ \{G\ \varphi\} \cup \mathcal{G}_{FG}\ \varphi\ w\ else\ \mathcal{G}_{FG}\ \varphi\ w)$
| $\mathcal{G}_{FG}\ (X\ \varphi)\ w = \mathcal{G}_{FG}\ \varphi\ w$
| $\mathcal{G}_{FG}\ (\varphi\ U\ \psi)\ w = \mathcal{G}_{FG}\ \varphi\ w \cup \mathcal{G}_{FG}\ \psi\ w$

lemma $\mathcal{G}_{FG}$*-alt-def*:
$\mathcal{G}_{FG}\ \varphi\ w = \{G\ \psi \mid \psi.\ G\ \psi \in \mathbf{G}\ \varphi \wedge w \models F\ (G\ \psi)\}$
⟨*proof*⟩

lemma $\mathcal{G}_{FG}$*-Only-G*:
Only-G ($\mathcal{G}_{FG}\ \varphi\ w$)
⟨*proof*⟩

lemma $\mathcal{G}_{FG}$*-suffix*[*simp*]:
$\mathcal{G}_{FG}\ \varphi$ (*suffix i w*) = $\mathcal{G}_{FG}\ \varphi\ w$
⟨*proof*⟩

12.2 Eventually Provable and Almost All Eventually Provable

abbreviation $\mathfrak{P}$
where
$\mathfrak{P}\ \varphi\ \mathcal{G}\ w\ i \equiv \exists j.\ \mathcal{G} \models_P af_G\ \varphi\ (w\ [i \rightarrow j])$

definition *almost-all-eventually-provable* :: *'a ltl* ⇒ *'a ltl set* ⇒ *'a set word* ⇒ *bool* ($\mathfrak{P}_\infty$)
where
$\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \equiv \forall_\infty i.\ \mathfrak{P}\ \varphi\ \mathcal{G}\ w\ i$

12.2.1 Proof Rules

lemma *almost-all-eventually-provable-monotonI*[*intro*]:
$\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \Longrightarrow \mathcal{G} \subseteq \mathcal{G}' \Longrightarrow \mathfrak{P}_\infty\ \varphi\ \mathcal{G}'\ w$
⟨*proof*⟩

lemma *almost-all-eventually-provable-restrict-to-G*:
$\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \Longrightarrow$ *Only-G* $\mathcal{G} \Longrightarrow \mathfrak{P}_\infty\ \varphi\ (\mathcal{G} \cap \mathbf{G}\ \varphi)\ w$
⟨*proof*⟩

fun *G-depth* :: *'a ltl* ⇒ *nat*
where
G-depth (φ *and* ψ) = *max* (*G-depth* φ) (*G-depth* ψ)
| *G-depth* (φ *or* ψ) = *max* (*G-depth* φ) (*G-depth* ψ)
| *G-depth* ($F\ \varphi$) = *G-depth* φ
| *G-depth* ($G\ \varphi$) = *G-depth* φ + *1*
| *G-depth* ($X\ \varphi$) = *G-depth* φ
| *G-depth* ($\varphi\ U\ \psi$) = *max* (*G-depth* φ) (*G-depth* ψ)
| *G-depth* φ = *0*

lemma *almost-all-eventually-provable-restrict-to-G-depth*:
assumes $\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w$
assumes *Only-G* $\mathcal{G}$
shows $\mathfrak{P}_\infty\ \varphi\ (\mathcal{G} \cap \{\psi.\ \textit{G-depth}\ \psi \leq \textit{G-depth}\ \varphi\})\ w$
⟨*proof*⟩

lemma *almost-all-eventually-provable-suffix*:
$\mathfrak{P}_\infty\ \varphi\ \mathcal{G}'\ w \Longrightarrow \mathfrak{P}_\infty\ \varphi\ \mathcal{G}'\ (\textit{suffix}\ i\ w)$
⟨*proof*⟩

12.2.2 Threshold

The first index, such that the formula is eventually provable from this time on

fun *threshold* :: *'a ltl* ⇒ *'a set word* ⇒ *'a ltl set* ⇒ *nat option*
where
threshold $\varphi\ w\ \mathcal{G} = \textit{index}\ (\lambda j.\ \mathfrak{P}\ \varphi\ \mathcal{G}\ w\ j)$

lemma *threshold-properties*:
threshold $\varphi\ w\ \mathcal{G} = \textit{Some}\ i \Longrightarrow 0 < i \Longrightarrow \neg\ \mathcal{G} \models_P \textit{af}_G\ \varphi\ (w\ [(i - 1) \rightarrow k])$
threshold $\varphi\ w\ \mathcal{G} = \textit{Some}\ i \Longrightarrow j \geq i \Longrightarrow \exists k.\ \mathcal{G} \models_P \textit{af}_G\ \varphi\ (w\ [j \rightarrow k])$
⟨*proof*⟩

lemma *threshold-suffix*:
assumes *threshold* $\varphi\ w\ \mathcal{G} = \textit{Some}\ k$
assumes *threshold* $\varphi\ (\textit{suffix}\ i\ w)\ \mathcal{G} = \textit{Some}\ k'$
shows $k \leq k' + i$
⟨*proof*⟩

12.2.3 Relation to LTL semantics

lemma *ltl-implies-provable*:
$w \models \varphi \Longrightarrow \mathfrak{P}\ \varphi\ (\mathcal{G}_{FG}\ \varphi\ w)\ w\ 0$
⟨*proof*⟩

lemma *ltl-implies-provable-almost-all*:
$w \models \varphi \Longrightarrow \forall_\infty i.\ \mathcal{G}_{FG}\ \varphi\ w \models_P \textit{af}_G\ \varphi\ (w\ [0 \rightarrow i])$
⟨*proof*⟩

12.2.4 Closed Sets

abbreviation *closed*
where

closed $\mathcal{G}$ $w \equiv$ *finite* $\mathcal{G} \wedge$ *Only-G* $\mathcal{G} \wedge (\forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow \mathfrak{P}_\infty\ \psi\ \mathcal{G}\ w)$

lemma *closed-FG*:
assumes *closed* $\mathcal{G}$ w
assumes $G\ \psi \in \mathcal{G}$
shows $w \models F\ G\ \psi$
⟨*proof*⟩

lemma *closed-*$\mathcal{G}_{FG}$:
closed $(\mathcal{G}_{FG}\ \varphi\ w)\ w$
⟨*proof*⟩

12.2.5 Conjunction of Eventually Provable Formulas

definition $\mathcal{F}$
where
$\mathcal{F}\ \varphi\ w\ \mathcal{G}\ j =$ *And* (*map* $(\lambda i.\ af_G\ \varphi\ (w\ [i \rightarrow j]))$ [*the* (*threshold* $\varphi\ w\ \mathcal{G}$) ..< *Suc* j])

lemma *almost-all-suffixes-model-*$\mathcal{F}$:
assumes *closed* $\mathcal{G}$ w
assumes $G\ \varphi \in \mathcal{G}$
shows $\forall_\infty j.$ *suffix* $j\ w \models eval_G\ \mathcal{G}\ (\mathcal{F}\ \varphi\ w\ \mathcal{G}\ j)$
⟨*proof*⟩

lemma *almost-all-commutative''*:
assumes *finite* S
assumes *Only-G* S
assumes $\bigwedge x.\ G\ x \in S \Longrightarrow \forall_\infty i.\ P\ x\ (i::nat)$
shows $\forall_\infty i.\ \forall x.\ G\ x \in S \longrightarrow P\ x\ i$
⟨*proof*⟩

lemma *almost-all-suffixes-model-*$\mathcal{F}$*-generalized*:
assumes *closed* $\mathcal{G}$ w
shows $\forall_\infty j.\ \forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow$ *suffix* $j\ w \models eval_G\ \mathcal{G}\ (\mathcal{F}\ \psi\ w\ \mathcal{G}\ j)$
⟨*proof*⟩

12.3 Technical Lemmas

lemma *threshold-suffix-2*:
assumes *threshold* $\psi\ w\ \mathcal{G}' =$ *Some* k
assumes $k \leq l$
shows *threshold* ψ (*suffix* $l\ w$) $\mathcal{G}' =$ *Some 0*
⟨*proof*⟩

lemma *threshold-closed*:
assumes *closed* $\mathcal{G}$ w
shows $\exists k.\ \forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow$ *threshold* ψ (*suffix* k w) $\mathcal{G}$ = *Some* 0
⟨*proof*⟩

lemma $\mathcal{F}$*-drop*:
assumes $\mathfrak{P}_\infty\ \varphi\ \mathcal{G}'\ w$
assumes $S \models_P \mathcal{F}\ \varphi\ w\ \mathcal{G}'\ (i + j)$
shows $S \models_P \mathcal{F}\ \varphi$ (*suffix* i w) $\mathcal{G}'\ j$
⟨*proof*⟩

12.4 Main Results

definition $accept_M$
where
$accept_M\ \varphi\ \mathcal{G}\ w \equiv (\forall_\infty j.\ \forall S.\ (\forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow S \models_P G\ \psi \wedge S \models_P eval_G\ \mathcal{G}\ (\mathcal{F}\ \psi\ w\ \mathcal{G}\ j)) \longrightarrow S \models_P af\ \varphi\ (w\ [0 \rightarrow j]))$

lemma *lemmaD*:
assumes $w \models \varphi$
assumes $\bigwedge \psi.\ G\ \psi \in \mathcal{G}_{FG}\ \varphi\ w \Longrightarrow$ *threshold* ψ w ($\mathcal{G}_{FG}\ \varphi\ w$) = *Some* 0
shows $accept_M\ \varphi\ (\mathcal{G}_{FG}\ \varphi\ w)\ w$
⟨*proof*⟩

theorem *ltl-FG-logical-characterization*:
$w \models F\ G\ \varphi \longleftrightarrow (\exists \mathcal{G} \subseteq \mathbf{G}\ (F\ G\ \varphi).\ G\ \varphi \in \mathcal{G} \wedge$ *closed* $\mathcal{G}$ $w)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

theorem *ltl-logical-characterization*:
$w \models \varphi \longleftrightarrow (\exists \mathcal{G} \subseteq \mathbf{G}\ \varphi.\ accept_M\ \varphi\ \mathcal{G}\ w \wedge$ *closed* $\mathcal{G}$ $w)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

end

13 Translation from LTL to (Deterministic Transitions-Based) Generalised Rabin Automata

theory *LTL-Rabin*
imports *Main Mojmir-Rabin Logical-Characterization*
begin

13.1 Preliminary Facts

lemma *run-af-G-letter-abs-eq-Abs-af-G-letter*:
run $\uparrow af_G$ (*Abs* φ) w i = *Abs* (*run af-G-letter* φ w i)
⟨*proof*⟩

lemma *finite-reach-af*:
finite (*reach* Σ $\uparrow af$ (*Abs* φ))
⟨*proof*⟩

lemma *ltl-semi-mojmir*:
assumes *finite* Σ
assumes *range* $w \subseteq \Sigma$
shows *semi-mojmir* Σ $\uparrow af_G$ (*Abs* ψ) w
⟨*proof*⟩

13.2 Single Secondary Automaton

locale *ltl-FG-to-rabin-def* =
fixes
Σ :: $'a$ *set set*
fixes
φ :: $'a$ *ltl*
fixes
$\mathcal{G}$ :: $'a$ *ltl set*
fixes
w :: $'a$ *set word*
begin

sublocale *mojmir-to-rabin-def* Σ $\uparrow af_G$ *Abs* φ w {q. $\mathcal{G}$ $\models_P$ *Rep* q} ⟨*proof*⟩
abbreviation $\delta_R \equiv$ *step*
abbreviation $q_R \equiv$ *initial*
abbreviation Acc_R $j \equiv$ (*fail*$_R$ ∪ *merge*$_R$ j, *succeed*$_R$ j)
abbreviation *max-rank*$_R \equiv$ *max-rank*
abbreviation *smallest-accepting-rank*$_R \equiv$ *smallest-accepting-rank*
abbreviation *accept*$_R{}' \equiv$ *accept*
abbreviation $\mathcal{S}_R \equiv \mathcal{S}$

lemma *Rep-token-run-af*:
Rep (*token-run* x n) $\equiv_P$ af_G φ (w [$x \rightarrow n$])
⟨*proof*⟩

end

locale *ltl-FG-to-rabin* = *ltl-FG-to-rabin-def* +
  assumes
    wellformed-$\mathcal{G}$: *Only-G* $\mathcal{G}$
  assumes
    bounded-w: *range* $w \subseteq \Sigma$
  assumes
    finite-Σ: *finite* Σ
begin

sublocale *mojmir-to-rabin* Σ $\uparrow af_G$ *Abs* φ w $\{q.\ \mathcal{G} \models_P$ *Rep* $q\}$
⟨*proof*⟩

lemma *ltl-to-rabin-correct-exposed'*:
  $\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \longleftrightarrow$ *accept*
⟨*proof*⟩

lemma *ltl-to-rabin-correct-exposed*:
  $\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \longleftrightarrow$ *accept*$_R$ (δ_R, q_R, $\{Acc_R\ i \mid i.\ i <$ *max-rank*$_R\}$) w
  ⟨*proof*⟩
lemmas *max-rank-lowerbound* = *max-rank-lowerbound*
lemmas *state-rank-step-foldl* = *state-rank-step-foldl*
lemmas *smallest-accepting-rank-properties* = *smallest-accepting-rank-properties*

lemmas *wellformed-*$\mathcal{R}$ = *wellformed-*$\mathcal{R}$

end

fun *ltl-to-rabin*
where
  ltl-to-rabin $\Sigma\ \varphi\ \mathcal{G}$ = (*ltl-FG-to-rabin-def*.$\delta_R\ \Sigma\ \varphi$, *ltl-FG-to-rabin-def*.$q_R$ φ, $\{$*ltl-FG-to-rabin-def*.$Acc_R\ \Sigma\ \varphi\ \mathcal{G}\ i \mid i.\ i <$ *ltl-FG-to-rabin-def*.*max-rank*$_R$ $\Sigma\ \varphi\}$)

context
  fixes
    Σ :: *'a set set*
  assumes
    finite-Σ: *finite* Σ
begin

lemma *ltl-to-rabin-correct*:
  assumes *range* $w \subseteq \Sigma$
  shows $w \models F\ G\ \varphi = (\exists \mathcal{G} \subseteq \mathbf{G}\ (G\ \varphi).\ G\ \varphi \in \mathcal{G} \wedge (\forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow$ *accept*$_R$ (*ltl-to-rabin* $\Sigma\ \psi\ \mathcal{G}$) $w))$

⟨*proof*⟩

end

13.2.1 LTL-to-Mojmir Lemmas

lemma $\mathcal{F}$-*eq*-$\mathcal{S}$:
assumes *finite*-Σ: *finite* Σ
assumes *bounded-w*: *range* $w \subseteq \Sigma$
assumes *closed* $\mathcal{G}$ w
assumes $G\ \psi \in \mathcal{G}$
shows $\forall_\infty j.\ (\forall S.\ (S \models_P \mathcal{F}\ \psi\ w\ \mathcal{G}\ j \wedge \mathcal{G} \subseteq S) \longleftrightarrow (\forall q.\ q \in (\textit{ltl-FG-to-rabin-def}.\mathcal{S}_R\ \Sigma\ \psi\ \mathcal{G}\ w\ j) \longrightarrow S \models_P \textit{Rep}\ q))$
⟨*proof*⟩

lemma $\mathcal{F}$-*eq*-$\mathcal{S}$-*generalized*:
assumes *finite*-Σ: *finite* Σ
assumes *bounded-w*: *range* $w \subseteq \Sigma$
assumes *closed* $\mathcal{G}$ w
shows $\forall_\infty j.\ \forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow (\forall S.\ (S \models_P \mathcal{F}\ \psi\ w\ \mathcal{G}\ j \wedge \mathcal{G} \subseteq S) \longleftrightarrow (\forall q.\ q \in (\textit{ltl-FG-to-rabin-def}.\mathcal{S}_R\ \Sigma\ \psi\ \mathcal{G})\ w\ j \longrightarrow S \models_P \textit{Rep}\ q))$
⟨*proof*⟩

13.3 Product of Secondary Automata

context
fixes
$\Sigma :: 'a\ set\ set$
begin

fun *product-initial-state* :: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \rightharpoonup 'b)$ $(\iota_\times)$
where
$\iota_\times\ K\ q_m = (\lambda k.\ \textit{if}\ k \in K\ \textit{then}\ \textit{Some}\ (q_m\ k)\ \textit{else}\ \textit{None})$

fun *combine-pairs* :: $(('a, 'b)\ transition\ set \times ('a, 'b)\ transition\ set)\ set \Rightarrow (('a, 'b)\ transition\ set \times ('a, 'b)\ transition\ set\ set)$
where
$\textit{combine-pairs}\ P = (\bigcup(\textit{fst}\ `\ P), \textit{snd}\ `\ P)$

fun *combine-pairs'* :: $((('a\ ltl \Rightarrow ('a\ ltl\text{-}prop\text{-}equiv\text{-}quotient \Rightarrow nat\ option)\ option, 'a\ set)\ transition\ set \times ('a\ ltl \Rightarrow ('a\ ltl\text{-}prop\text{-}equiv\text{-}quotient \Rightarrow nat\ option)\ option, 'a\ set)\ transition\ set)\ set \Rightarrow (('a\ ltl \Rightarrow ('a\ ltl\text{-}prop\text{-}equiv\text{-}quotient \Rightarrow nat\ option)\ option, 'a\ set)\ transition\ set \times ('a\ ltl \Rightarrow ('a\ ltl\text{-}prop\text{-}equiv\text{-}quotient \Rightarrow nat\ option)\ option, 'a\ set)\ transition\ set\ set)$

where
combine-pairs′ P = (⋃(*fst* ‘ *P*), *snd* ‘ *P*)

lemma *combine-pairs-prop*:
(∀ *P* ∈ $\mathcal{P}$. *accepting-pair*$_R$ *δ q*$_0$ *P w*) = *accepting-pair*$_{GR}$ *δ q*$_0$ (*combine-pairs* $\mathcal{P}$) *w*
⟨*proof*⟩

lemma *combine-pairs2*:
combine-pairs $\mathcal{P}$ ∈ *α* ⟹ (⋀*P*. *P* ∈ $\mathcal{P}$ ⟹ *accepting-pair*$_R$ *δ q*$_0$ *P w*) ⟹ *accept*$_{GR}$ (*δ*, *q*$_0$, *α*) *w*
⟨*proof*⟩

lemma *combine-pairs′-prop*:
(∀ *P* ∈ $\mathcal{P}$. *accepting-pair*$_R$ *δ q*$_0$ *P w*) = *accepting-pair*$_{GR}$ *δ q*$_0$ (*combine-pairs′* $\mathcal{P}$) *w*
⟨*proof*⟩

fun *ltl-FG-to-generalized-rabin* :: *'a ltl* ⇒ (*'a ltl* ⇀ *'a ltl*$_P$ ⇀ *nat*, *'a set*) *generalized-rabin-automaton* ($\mathcal{P}$)
where
ltl-FG-to-generalized-rabin φ = (
Δ$_\times$ (λχ. *ltl-FG-to-rabin-def*.*δ*$_R$ Σ (*theG* χ)),
ι$_\times$ (**G** (*G φ*)) (λχ. *ltl-FG-to-rabin-def*.*q*$_R$ (*theG* χ)),
{*combine-pairs′* {*embed-pair* χ (*ltl-FG-to-rabin-def*.*Acc*$_R$ Σ (*theG* χ) $\mathcal{G}$ (*π* χ)) | χ. χ ∈ $\mathcal{G}$}
| $\mathcal{G}$ *π*. $\mathcal{G}$ ⊆ **G** (*G φ*) ∧ *G φ* ∈ $\mathcal{G}$ ∧ (∀ χ. *π* χ < *ltl-FG-to-rabin-def*.*max-rank*$_R$ Σ (*theG* χ))})

context
assumes
finite-Σ: *finite* Σ
begin

lemma *ltl-FG-to-generalized-rabin-wellformed*:
finite (*reach* Σ (*fst* ($\mathcal{P}$ *φ*)) (*fst* (*snd* ($\mathcal{P}$ *φ*))))
⟨*proof*⟩

theorem *ltl-FG-to-generalized-rabin-correct*:
assumes *range w* ⊆ Σ
shows *w* ⊨ *F G φ* = *accept*$_{GR}$ ($\mathcal{P}$ *φ*) *w*
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

end

13.4 Automaton Template

locale *ltl-to-rabin-base-def* =
  fixes
    δ :: $'a$ ltl_P ⇒ $'a$ *set* ⇒ $'a$ ltl_P
  fixes
    δ_M :: $'a$ ltl_P ⇒ $'a$ *set* ⇒ $'a$ ltl_P
  fixes
    q_0 :: $'a$ *ltl* ⇒ $'a$ ltl_P
  fixes
    q_{0M} :: $'a$ *ltl* ⇒ $'a$ ltl_P
  fixes
     M-fin :: ($'a$ *ltl* ⇀ *nat*) ⇒ ($'a$ ltl_P × ($'a$ *ltl* ⇀ $'a$ ltl_P ⇀ *nat*), $'a$ *set*) *transition set*
begin

— Transition Function and Initial State

fun *delta*
where
  delta Σ = δ × $\Delta_\times$ (*semi-mojmir-def.step* Σ δ_M *o* q_{0M} *o* *theG*)

fun *initial*
where
  initial φ = (q_0 φ, $\iota_\times$ (**G** φ) (*semi-mojmir-def.initial o* q_{0M} *o* *theG*))

— Acceptance Condition

definition *max-rank-of*
where
  max-rank-of Σ ψ ≡ *semi-mojmir-def.max-rank* Σ δ_M (q_{0M} (*theG* ψ))

fun *Acc-fin*
where
  Acc-fin Σ π χ = ⋃(*embed-transition-snd* ‘ ⋃(*embed-transition* χ ‘
    (*mojmir-to-rabin-def.fail*$_R$ Σ δ_M (q_{0M} (*theG* χ)) {q. *dom* π ↑⊨$_P$ q}
    ∪ *mojmir-to-rabin-def.merge*$_R$ δ_M (q_{0M} (*theG* χ)) {q. *dom* π ↑⊨$_P$ q}
(*the* (π χ)))))

fun *Acc-inf*

where
Acc-inf $\pi\ \chi = \bigcup$(*embed-transition-snd* ' $\bigcup$(*embed-transition* χ '
(*mojmir-to-rabin-def.succeed*$_R$ δ_M (q_{0M} (*theG* χ)) {q. *dom* π $\uparrow\models_P$ q}
(*the* (π χ)))))

abbreviation *Acc*
where
Acc Σ π $\chi \equiv$ (*Acc-fin* Σ π χ, *Acc-inf* π χ)

fun *rabin-pairs* :: $'a$ *set set* $\Rightarrow$ $'a$ *ltl* $\Rightarrow$ ($'a$ *ltl*$_P$ $\times$ ($'a$ *ltl* $\rightharpoonup$ $'a$ *ltl*$_P$ $\rightharpoonup$ *nat*), $'a$ *set*) *generalized-rabin-condition*
where
rabin-pairs Σ φ = {(*M-fin* $\pi \cup \bigcup$ {*Acc-fin* Σ π χ | χ. $\chi \in$ *dom* π}, {*Acc-inf* π χ | χ. $\chi \in$ *dom* π})
| π. *dom* $\pi \subseteq$ **G** $\varphi \wedge$ ($\forall \chi \in$ *dom* π. *the* (π χ) $<$ *max-rank-of* Σ χ)}

fun *ltl-to-generalized-rabin* :: $'a$ *set set* $\Rightarrow$ $'a$ *ltl* $\Rightarrow$ ($'a$ *ltl*$_P$ $\times$ ($'a$ *ltl* $\rightharpoonup$ $'a$ *ltl*$_P$ $\rightharpoonup$ *nat*), $'a$ *set*) *generalized-rabin-automaton* ($\mathcal{A}$)
where
$\mathcal{A}$ Σ φ = (*delta* Σ, *initial* φ, *rabin-pairs* Σ φ)

end

locale *ltl-to-rabin-base* = *ltl-to-rabin-base-def* +
fixes
Σ :: $'a$ *set set*
fixes
w :: $'a$ *set word*
assumes
finite-Σ: *finite* Σ
assumes
bounded-w: *range* $w \subseteq \Sigma$
assumes
M-fin-monotonic: *dom* π = *dom* $\pi' \Longrightarrow$ ($\bigwedge\chi$. $\chi \in$ *dom* $\pi \Longrightarrow$ *the* (π χ) $\leq$ *the* (π' χ)) $\Longrightarrow$ *M-fin* $\pi \subseteq$ *M-fin* π'
assumes
finite-reach': *finite* (*reach* Σ δ (q_0 φ))
assumes
mojmir-to-rabin: *Only-G* $\mathcal{G} \Longrightarrow$ *mojmir-to-rabin* Σ δ_M (q_{0M} ψ) w {q. $\mathcal{G}$ $\uparrow\models_P$ q}
begin

lemma *semi-mojmir*:
semi-mojmir Σ δ_M (q_{0M} ψ) w

⟨*proof*⟩

lemma *finite-reach*:
finite (*reach* Σ (*delta* Σ) (*initial* φ))
⟨*proof*⟩

lemma *run-limit-not-empty*:
limit (*run*$_t$ (*delta* Σ) (*initial* φ) *w*) ≠ {}
⟨*proof*⟩

lemma *run-properties*:
fixes φ
defines *r* ≡ *run* (*delta* Σ) (*initial* φ) *w*
shows *fst* (*r i*) = *foldl* δ (*q*$_0$ φ) (*w* [*0* → *i*])
and ⋀χ *q*. χ ∈ **G** φ ⟹ *the* (*snd* (*r i*) χ) *q* = *semi-mojmir-def.state-rank* Σ δ$_M$ (*q*$_{0M}$ (*theG* χ)) *w q i*
⟨*proof*⟩

lemma *accept*$_{GR}$*-I*:
assumes *accept*$_{GR}$ (*A* Σ φ) *w*
obtains π **where** *dom* π ⊆ **G** φ
and ⋀χ. χ ∈ *dom* π ⟹ *the* (π χ) < *max-rank-of* Σ χ
and *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*M-fin* π, *UNIV*) *w*
and ⋀χ. χ ∈ *dom* π ⟹ *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π χ) *w*
⟨*proof*⟩

context
fixes
φ :: 'a *ltl*
begin

context
fixes
ψ :: 'a *ltl*
fixes
π :: 'a *ltl* ⇀ *nat*
assumes
G ψ ∈ *dom* π
assumes
dom π ⊆ **G** φ
begin

interpretation 𝔐: *mojmir-to-rabin* Σ δ$_M$ *q*$_{0M}$ ψ *w* {*q*. *dom* π ↑⊨$_P$ *q*}

⟨*proof*⟩

lemma *Acc-property*:
accepting-pair$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π (*G* ψ)) $w \longleftrightarrow$ *accepting-pair*$_R$ $\mathfrak{M}.\delta_{\mathcal{R}}$ $\mathfrak{M}.q_{\mathcal{R}}$ ($\mathfrak{M}.Acc_{\mathcal{R}}$ (*the* (π (*G* ψ)))) w
(**is** *?Acc* = *?Acc*$_{\mathcal{R}}$)
⟨*proof*⟩

lemma *Acc-to-rabin-accept*:
⟦*accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π (*G* ψ)) w; *the* (π (*G* ψ)) < $\mathfrak{M}$.*max-rank*⟧ ⟹ *accept*$_R$ $\mathfrak{M}.\mathcal{R}$ w
⟨*proof*⟩

lemma *Acc-to-mojmir-accept*:
⟦*accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π (*G* ψ)) w; *the* (π (*G* ψ)) < $\mathfrak{M}$.*max-rank*⟧ ⟹ $\mathfrak{M}$.*accept*
⟨*proof*⟩

lemma *rabin-accept-to-Acc*:
⟦*accept*$_R$ $\mathfrak{M}.\mathcal{R}$ w; π (*G* ψ) = $\mathfrak{M}$.*smallest-accepting-rank*⟧ ⟹ *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π (*G* ψ)) w
⟨*proof*⟩

lemma *mojmir-accept-to-Acc*:
⟦$\mathfrak{M}$.*accept*; π (*G* ψ) = $\mathfrak{M}$.*smallest-accepting-rank*⟧ ⟹ *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π (*G* ψ)) w
⟨*proof*⟩

end

lemma *normalize-*π:
assumes *dom-subset*: *dom* $\pi \subseteq$ **G** φ
assumes $\bigwedge\chi$. $\chi \in$ *dom* $\pi \Longrightarrow$ *the* (π χ) < *max-rank-of* Σ χ
assumes *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*M-fin* π, *UNIV*) w
assumes $\bigwedge\chi$. $\chi \in$ *dom* $\pi \Longrightarrow$ *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ π χ) w
obtains $\pi_{\mathcal{A}}$ **where** *dom* π = *dom* $\pi_{\mathcal{A}}$
and $\bigwedge\chi$. $\chi \in$ *dom* $\pi_{\mathcal{A}} \Longrightarrow \pi_{\mathcal{A}}$ χ = *mojmir-def.smallest-accepting-rank* Σ δ_M (q_{0M} (*theG* χ)) w {q. *dom* $\pi_{\mathcal{A}}$ ↑⊨$_P$ q}
and *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*M-fin* $\pi_{\mathcal{A}}$, *UNIV*) w
and $\bigwedge\chi$. $\chi \in$ *dom* $\pi_{\mathcal{A}} \Longrightarrow$ *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*Acc* Σ $\pi_{\mathcal{A}}$ χ) w
⟨*proof*⟩

end

end

13.5 Generalized Deterministic Rabin Automaton

13.5.1 Definition

fun *M-fin* :: $('a\ ltl \rightharpoonup nat) \Rightarrow ('a\ ltl_P \times ('a\ ltl \rightharpoonup 'a\ ltl_P \rightharpoonup nat), 'a\ set)$ *transition set*
where
M-fin $\pi = \{((\varphi', m), \nu, p).$
$\neg(\forall S.\ (\forall \chi \in dom\ \pi.\ S \uparrow\models_P Abs\ \chi \wedge (\forall q.\ (\exists j \geq the\ (\pi\ \chi).\ the\ (m\ \chi)\ q = Some\ j) \longrightarrow S \uparrow\models_P \uparrow eval_G\ (dom\ \pi)\ q)) \longrightarrow S \uparrow\models_P \varphi')\}$

locale *ltl-to-rabin-af* = *ltl-to-rabin-base* $\uparrow af\ \uparrow af_G$ *Abs Abs M-fin* **begin**

abbreviation $\delta_{\mathcal{A}} \equiv$ *delta*
abbreviation $\iota_{\mathcal{A}} \equiv$ *initial*
abbreviation $Acc_{\mathcal{A}} \equiv$ *Acc*
abbreviation $F_{\mathcal{A}} \equiv$ *rabin-pairs*
abbreviation $\mathcal{A} \equiv$ *ltl-to-generalized-rabin*

13.5.2 Correctness Theorem

theorem *ltl-to-generalized-rabin-correct*:
$w \models \varphi = accept_{GR}\ (ltl\text{-}to\text{-}generalized\text{-}rabin\ \Sigma\ \varphi)\ w$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

end

fun *ltl-to-generalized-rabin-af*
where
ltl-to-generalized-rabin-af $\Sigma\ \varphi$ = *ltl-to-rabin-base-def.ltl-to-generalized-rabin* $\uparrow af\ \uparrow af_G$ *Abs Abs M-fin* $\Sigma\ \varphi$

lemma *ltl-to-generalized-rabin-af-wellformed*:
finite $\Sigma \Longrightarrow$ *range* $w \subseteq \Sigma \Longrightarrow$ *ltl-to-rabin-af* $\Sigma\ w$
⟨*proof*⟩

theorem *ltl-to-generalized-rabin-af-correct*:
assumes *finite* Σ
assumes *range* $w \subseteq \Sigma$
shows $w \models \varphi = accept_{GR}\ (ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af\ \Sigma\ \varphi)\ w$

⟨*proof*⟩

thm *ltl-to-generalized-rabin-af-correct ltl-FG-to-generalized-rabin-correct*

end

14 Eager Unfolding Optimisation

theory *LTL-Rabin-Unfold-Opt*
imports *Main LTL-Rabin*
begin

14.1 Preliminary Facts

lemma *finite-reach-af-opt*:
finite (*reach* Σ $\uparrow af_{\mathfrak{U}}$ (*Abs* φ))
⟨*proof*⟩

lemma *finite-reach-af-G-opt*:
finite (*reach* Σ $\uparrow af_{G\mathfrak{U}}$ (*Abs* φ))
⟨*proof*⟩

lemma *wellformed-mojmir-opt*:
assumes *Only-G* $\mathcal{G}$
assumes *finite* Σ
assumes *range* $w \subseteq \Sigma$
shows *mojmir* Σ $\uparrow af_{G\mathfrak{U}}$ (*Abs* φ) w $\{q.\ \mathcal{G} \models_P Rep\ q\}$
⟨*proof*⟩

locale *ltl-FG-to-rabin-opt-def* =
fixes
Σ :: *'a set set*
fixes
φ :: *'a ltl*
fixes
$\mathcal{G}$ :: *'a ltl set*
fixes
w :: *'a set word*
begin

sublocale *mojmir-to-rabin-def* Σ $\uparrow af_{G\mathfrak{U}}$ *Abs* (*Unf*$_G$ φ) w $\{q.\ \mathcal{G} \models_P Rep\ q\}$
⟨*proof*⟩

end

locale *ltl-FG-to-rabin-opt* = *ltl-FG-to-rabin-opt-def* +
  assumes
    wellformed-$\mathcal{G}$: *Only-G* $\mathcal{G}$
  assumes
    bounded-w: *range* $w \subseteq \Sigma$
  assumes
    finite-Σ: *finite* Σ
begin

sublocale *mojmir-to-rabin* Σ $\uparrow af_{G\mathfrak{U}}$ *Abs* (*Unf*$_G$ φ) w $\{q.\ \mathcal{G} \models_P Rep\ q\}$
⟨*proof*⟩

end

14.2 Equivalences between the standard and the eager Mojmir construction

context
  fixes
    Σ :: *'a set set*
  fixes
    φ :: *'a ltl*
  fixes
    $\mathcal{G}$ :: *'a ltl set*
  fixes
    w :: *'a set word*
  assumes
    context-assms: *Only-G* $\mathcal{G}$ *finite* Σ *range* $w \subseteq \Sigma$
begin

— Create an interpretation of the mojmir locale for the standard construction
interpretation $\mathfrak{M}$: *ltl-FG-to-rabin* Σ φ $\mathcal{G}$ w
  ⟨*proof*⟩
interpretation $\mathfrak{U}$: *ltl-FG-to-rabin-opt* Σ φ $\mathcal{G}$ w
  ⟨*proof*⟩

lemma *unfold-token-run-eq*:
  assumes $x \leq n$
  shows $\mathfrak{M}$.*token-run* x (*Suc* n) = $\uparrow$*step* ($\mathfrak{U}$.*token-run* x n) (w n)
  (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *unfold-token-succeeds-eq*:
$\mathfrak{M}.token\text{-}succeeds\ x = \mathfrak{U}.token\text{-}succeeds\ x$
⟨*proof*⟩

lemma *unfold-accept-eq*:
$\mathfrak{M}.accept = \mathfrak{U}.accept$
⟨*proof*⟩

lemma *unfold-$\mathcal{S}$-eq*:
assumes $\mathfrak{M}.accept$
shows $\forall_\infty n.\ \mathfrak{M}.\mathcal{S}\ (Suc\ n) = (\lambda q.\ step\text{-}abs\ q\ (w\ n))\ `\ (\mathfrak{U}.\mathcal{S}\ n) \cup \{Abs\ \varphi\} \cup \{q.\ \mathcal{G} \models_P Rep\ q\}$
⟨*proof*⟩

end

14.3 Automaton Definition

fun $M_\mathfrak{U}$-*fin* :: $('a\ ltl \rightharpoonup nat) \Rightarrow ('a\ ltl_P \times ('a\ ltl \rightharpoonup 'a\ ltl_P \rightharpoonup nat),\ 'a\ set)\ transition\ set$
where
$M_\mathfrak{U}\text{-}fin\ \pi = \{((\varphi',\ m),\ \nu,\ p).\ \neg(\forall S.\ (\forall \chi \in (dom\ \pi).\ S \uparrow\models_P Abs\ \chi \wedge S \uparrow\models_P \uparrow eval_G\ (dom\ \pi)\ (Abs\ (theG\ \chi)) \wedge (\forall q.\ (\exists j \geq the\ (\pi\ \chi).\ the\ (m\ \chi)\ q = Some\ j) \longrightarrow S \uparrow\models_P \uparrow eval_G\ (dom\ \pi)\ (\uparrow step\ q\ \nu))) \longrightarrow S \uparrow\models_P (\uparrow step\ \varphi'\ \nu))\}$

locale *ltl-to-rabin-af-unf* = *ltl-to-rabin-base* $\uparrow af_\mathfrak{U}$ $\uparrow af_{G\mathfrak{U}}$ *Abs o Unf Abs o Unf*$_G$ $M_\mathfrak{U}$-*fin* **begin**

abbreviation $\delta_\mathfrak{U} \equiv delta$
abbreviation $\iota_\mathfrak{U} \equiv initial$
abbreviation $Acc_\mathfrak{U}$-*fin* $\equiv$ *Acc-fin*
abbreviation $Acc_\mathfrak{U}$-*inf* $\equiv$ *Acc-inf*
abbreviation $F_\mathfrak{U} \equiv rabin\text{-}pairs$
abbreviation $Acc_\mathfrak{U} \equiv Acc$
abbreviation $\mathcal{A}_\mathfrak{U} \equiv ltl\text{-}to\text{-}generalized\text{-}rabin$

14.4 Properties

14.5 Correctness Theorem

lemma *unfold-optimisation-correct-M*:
assumes $dom\ \pi_\mathcal{A} \subseteq \mathbf{G}\ \varphi$
assumes $dom\ \pi_\mathfrak{U} = dom\ \pi_\mathcal{A}$

assumes $\bigwedge\chi.\ \chi \in dom\ \pi_{\mathcal{A}} \Longrightarrow \pi_{\mathcal{A}}\ \chi = mojmir\text{-}def.smallest\text{-}accepting\text{-}rank\ \Sigma\ \uparrow af_G\ (Abs\ (theG\ \chi))\ w\ \{q.\ dom\ \pi_{\mathcal{A}}\ \uparrow\models_P\ q\}$
assumes $\bigwedge\chi.\ \chi \in dom\ \pi_{\mathfrak{U}} \Longrightarrow \pi_{\mathfrak{U}}\ \chi = mojmir\text{-}def.smallest\text{-}accepting\text{-}rank\ \Sigma\ af\text{-}G\text{-}letter\text{-}abs\text{-}opt\ (Abs\ (Unf_G\ (theG\ \chi)))\ w\ \{q.\ dom\ \pi_{\mathfrak{U}}\ \uparrow\models_P\ q\}$
shows $accepting\text{-}pair_R\ (ltl\text{-}to\text{-}rabin\text{-}af.\delta_{\mathcal{A}}\ \Sigma)\ (ltl\text{-}to\text{-}rabin\text{-}af.\iota_{\mathcal{A}}\ \varphi)\ (M\text{-}fin\ \pi_{\mathcal{A}},\ UNIV)\ w \longleftrightarrow accepting\text{-}pair_R\ (\delta_{\mathfrak{U}}\ \Sigma)\ (\iota_{\mathfrak{U}}\ \varphi)\ (M_{\mathfrak{U}}\text{-}fin\ \pi_{\mathfrak{U}},\ UNIV)\ w$
⟨*proof*⟩

theorem *ltl-to-generalized-rabin-correct*:
$w \models \varphi \longleftrightarrow accept_{GR}\ (\mathcal{A}_{\mathfrak{U}}\ \Sigma\ \varphi)\ w$
(**is** - $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

end

fun $ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af_{\mathfrak{U}}$
where
$ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af_{\mathfrak{U}}\ \Sigma\ \varphi = ltl\text{-}to\text{-}rabin\text{-}base\text{-}def.ltl\text{-}to\text{-}generalized\text{-}rabin\ \uparrow af_{\mathfrak{U}}\ \uparrow af_{G\mathfrak{U}}\ (Abs \circ Unf)\ (Abs \circ Unf_G)\ M_{\mathfrak{U}}\text{-}fin\ \Sigma\ \varphi$

lemma $ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af_{\mathfrak{U}}\text{-}wellformed$:
$finite\ \Sigma \Longrightarrow range\ w \subseteq \Sigma \Longrightarrow ltl\text{-}to\text{-}rabin\text{-}af\text{-}unf\ \Sigma\ w$
⟨*proof*⟩

theorem $ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af_{\mathfrak{U}}\text{-}correct$:
assumes *finite* Σ
assumes $range\ w \subseteq \Sigma$
shows $w \models \varphi = accept_{GR}\ (ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af_{\mathfrak{U}}\ \Sigma\ \varphi)\ w$
⟨*proof*⟩

thm *ltl-FG-to-generalized-rabin-correct ltl-to-generalized-rabin-af-correct ltl-to-generalized-rabin-af*$_{\mathfrak{U}}$-*c*

end

15 LTL Translation Layer

theory *LTL-Compat*
imports *Main LTL.LTL ../LTL-FGXU*
begin

— The following infrastructure translates the generic **datatype** $'a\ ltln = true_n \mid false_n \mid Prop\text{-}ltln\ 'a \mid Nprop\text{-}ltln\ 'a \mid And\text{-}ltln\ ('a\ ltln)\ ('a\ ltln) \mid Or\text{-}ltln\ ('a\ ltln)\ ('a\ ltln) \mid Next\text{-}ltln\ ('a\ ltln) \mid Until\text{-}ltln\ ('a\ ltln)\ ('a\ ltln) \mid Release\text{-}ltln$

(*'a ltln*) (*'a ltln*) | *WeakUntil-ltln* (*'a ltln*) (*'a ltln*) | *StrongRelease-ltln* (*'a ltln*) (*'a ltln*) datatype to special structure used in this project

abbreviation *LTLRelease* :: *'a ltl* ⇒ *'a ltl* ⇒ *'a ltl* (- *R* - [*87*,*87*] *86*)
where
  $\varphi\ R\ \psi \equiv (G\ \psi)\ or\ (\psi\ U\ (\varphi\ and\ \psi))$

abbreviation *LTLWeakUntil* :: *'a ltl* ⇒ *'a ltl* ⇒ *'a ltl* (- *W* - [*87*,*87*] *86*)
where
  $\varphi\ W\ \psi \equiv (\varphi\ U\ \psi)\ or\ (G\ \varphi)$

abbreviation *LTLStrongRelease* :: *'a ltl* ⇒ *'a ltl* ⇒ *'a ltl* (- *M* - [*87*,*87*] *86*)
where
  $\varphi\ M\ \psi \equiv \psi\ U\ (\varphi\ and\ \psi)$

fun *ltln-to-ltl* :: *'a ltln* ⇒ *'a ltl*
where
  ltln-to-ltl $true_n$ = *true*
| *ltln-to-ltl* $false_n$ = *false*
| *ltln-to-ltl* $prop_n(q)$ = $p(q)$
| *ltln-to-ltl* $nprop_n(q)$ = $np(q)$
| *ltln-to-ltl* ($\varphi\ and_n\ \psi$) = *ltln-to-ltl* φ *and ltln-to-ltl* ψ
| *ltln-to-ltl* ($\varphi\ or_n\ \psi$) = *ltln-to-ltl* φ *or ltln-to-ltl* ψ
| *ltln-to-ltl* ($\varphi\ U_n\ \psi$) = (*if* φ = $true_n$ *then F* (*ltln-to-ltl* ψ) *else* (*ltln-to-ltl* φ) *U* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* ($\varphi\ R_n\ \psi$) = (*if* φ = $false_n$ *then G* (*ltln-to-ltl* ψ) *else* (*ltln-to-ltl* φ) *R* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* ($\varphi\ W_n\ \psi$) = (*if* ψ = $false_n$ *then G* (*ltln-to-ltl* φ) *else* (*ltln-to-ltl* φ) *W* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* ($\varphi\ M_n\ \psi$) = (*if* ψ = $true_n$ *then F* (*ltln-to-ltl* φ) *else* (*ltln-to-ltl* φ) *M* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* ($X_n\ \varphi$) = *X* (*ltln-to-ltl* φ)

lemma *ltln-to-ltl-semantics*:
  $w \models$ *ltln-to-ltl* $\varphi \longleftrightarrow w \models_n \varphi$
  ⟨*proof*⟩

lemma *ltln-to-ltl-atoms*:
  vars (*ltln-to-ltl* φ) = *atoms-ltln* φ
  ⟨*proof*⟩

fun *atoms-list* :: *'a ltln* ⇒*'a list*
where

atoms-list (φ *and*$_n$ ψ) = *List.union* (*atoms-list* φ) (*atoms-list* ψ)
| *atoms-list* (φ *or*$_n$ ψ) = *List.union* (*atoms-list* φ) (*atoms-list* ψ)
| *atoms-list* (φ U_n ψ) = *List.union* (*atoms-list* φ) (*atoms-list* ψ)
| *atoms-list* (φ R_n ψ) = *List.union* (*atoms-list* φ) (*atoms-list* ψ)
| *atoms-list* (φ W_n ψ) = *List.union* (*atoms-list* φ) (*atoms-list* ψ)
| *atoms-list* (φ M_n ψ) = *List.union* (*atoms-list* φ) (*atoms-list* ψ)
| *atoms-list* (X_n φ) = *atoms-list* φ
| *atoms-list* (*prop*$_n$(a)) = [a]
| *atoms-list* (*nprop*$_n$(a)) = [a]
| *atoms-list* - = []

lemma *atoms-list-correct*:
set (*atoms-list* φ) = *atoms-ltln* φ
⟨*proof*⟩

lemma *atoms-list-distinct*:
distinct (*atoms-list* φ)
⟨*proof*⟩

end

16 LTL Code Equations

theory *LTL-Impl*
imports *Main*
../LTL-FGXU
Boolean-Expression-Checkers.Boolean-Expression-Checkers
Boolean-Expression-Checkers.Boolean-Expression-Checkers-AList-Mapping
begin

16.1 Subformulae

fun *G-list* :: *'a ltl* ⇒*'a ltl list*
where
G-list (φ *and* ψ) = *List.union* (*G-list* φ) (*G-list* ψ)
| *G-list* (φ *or* ψ) = *List.union* (*G-list* φ) (*G-list* ψ)
| *G-list* (*F* φ) = *G-list* φ
| *G-list* (*G* φ) = *List.insert* (*G* φ) (*G-list* φ)
| *G-list* (*X* φ) = *G-list* φ
| *G-list* (φ *U* ψ) = *List.union* (*G-list* φ) (*G-list* ψ)
| *G-list* φ = []

lemma *G-eq-G-list*:
G φ = *set* (*G-list* φ)

⟨*proof*⟩

lemma *G-list-distinct*:
distinct (*G-list* φ)
⟨*proof*⟩

16.2 Propositional Equivalence

fun *ifex-of-ltl* :: *'a ltl* ⇒ *'a ltl ifex*
where
ifex-of-ltl true = *Trueif*
| *ifex-of-ltl false* = *Falseif*
| *ifex-of-ltl* (φ *and* ψ) = *normif Mapping.empty* (*ifex-of-ltl* φ) (*ifex-of-ltl* ψ) *Falseif*
| *ifex-of-ltl* (φ *or* ψ) = *normif Mapping.empty* (*ifex-of-ltl* φ) *Trueif* (*ifex-of-ltl* ψ)
| *ifex-of-ltl* φ = *IF* φ *Trueif Falseif*

lemma *val-ifex*:
val-ifex (*ifex-of-ltl b*) *s* = ($\models_P$) {*x*. *s x*} *b*
⟨*proof*⟩

lemma *reduced-ifex*:
reduced (*ifex-of-ltl b*) {}
⟨*proof*⟩

lemma *ifex-of-ltl-reduced-bdt-checker*:
reduced-bdt-checkers ifex-of-ltl (λ*y s*. {*x*. *s x*} $\models_P$ *y*)
⟨*proof*⟩

lemma [*code*]:
($\varphi \equiv_P \psi$) = *equiv-test ifex-of-ltl* φ ψ
⟨*proof*⟩

lemma [*code*]:
($\varphi \longrightarrow_P \psi$) = *impl-test ifex-of-ltl* φ ψ
⟨*proof*⟩
export-code ($\equiv_P$) ($\longrightarrow_P$) **checking**

16.3 Remove Constants

fun *remove-constants*$_P$ :: *'a ltl* ⇒ *'a ltl*
where
remove-constants$_P$ (φ *and* ψ) = (

```
    case (remove-constants_P φ) of
        false ⇒ false
      | true ⇒ remove-constants_P ψ
      | φ' ⇒ (case remove-constants_P ψ of
          false ⇒ false
        | true ⇒ φ'
        | ψ' ⇒ φ' and ψ'))
| remove-constants_P (φ or ψ) = (
    case (remove-constants_P φ) of
        true ⇒ true
      | false ⇒ remove-constants_P ψ
      | φ' ⇒ (case remove-constants_P ψ of
          true ⇒ true
        | false ⇒ φ'
        | ψ' ⇒ φ' or ψ'))
| remove-constants_P φ = φ
```

lemma *remove-constants-correct*:
$S \models_P \varphi \longleftrightarrow S \models_P$ *remove-constants*$_P\ \varphi$
⟨*proof*⟩

16.4 And/Or Constructors

fun *in-and*
where
in-and x (y *and* z) = (*in-and* $x\ y \vee$ *in-and* $x\ z$)
| *in-and* $x\ y = (x = y)$

fun *in-or*
where
in-or x (y *or* z) = (*in-or* $x\ y \vee$ *in-or* $x\ z$)
| *in-or* $x\ y = (x = y)$

lemma *in-entailment*:
in-and $x\ y \Longrightarrow S \models_P y \Longrightarrow S \models_P x$
in-or $x\ y \Longrightarrow S \models_P x \Longrightarrow S \models_P y$
⟨*proof*⟩

definition *mk-and*
where
mk-and $f\ x\ y$ = (*case* $f\ x$ *of false* ⇒ *false* | *true* ⇒ $f\ y$
| x' ⇒ (*case* $f\ y$ *of false* ⇒ *false* | *true* ⇒ x'
| y' ⇒ *if in-and* $x'\ y'$ *then* y' *else if in-and* $y'\ x'$ *then* x' *else* x' *and* y'))

definition *mk-and′*
where
mk-and′ x y ≡ *case y of false* ⇒ *false* | *true* ⇒ *x* | - ⇒ *x and y*

definition *mk-or*
where
mk-or f x y = (*case f x of true* ⇒ *true* | *false* ⇒ *f y*
| *x′* ⇒ (*case f y of true* ⇒ *true* | *false* ⇒ *x′*
| *y′* ⇒ *if in-or x′ y′ then y′ else if in-or y′ x′ then x′ else x′ or y′*))

definition *mk-or′*
where
mk-or′ x y ≡ *case y of true* ⇒ *true* | *false* ⇒ *x* | - ⇒ *x or y*

lemma *mk-and-correct*:
S ⊨$_P$ *mk-and f x y* ⟷ *S* ⊨$_P$ *f x and f y*
⟨*proof*⟩

lemma *mk-and′-correct*:
S ⊨$_P$ *mk-and′ x y* ⟷ *S* ⊨$_P$ *x and y*
⟨*proof*⟩

lemma *mk-or-correct*:
S ⊨$_P$ *mk-or f x y* ⟷ *S* ⊨$_P$ *f x or f y*
⟨*proof*⟩

lemma *mk-or′-correct*:
S ⊨$_P$ *mk-or′ x y* ⟷ *S* ⊨$_P$ *x or y*
⟨*proof*⟩

end

17 af - Unfolding Functions - Optimized Code Equations

theory *af-Impl*
imports *Main ../af LTL-Impl*
begin

Provide optimized code definitions for ↑*af* and other functions, which use heuristics to reduce the formula size

17.1 Helper Function

fun *remove-and-or*
where
remove-and-or (*z or y*) = (*case z of*
(((*z′ and x′*) *or y′*) *and x*) ⇒ *if x* = *x′* ∧ *y* = *y′ then* ((*z′ and x′*) *or y′*) *else remove-and-or z or remove-and-or y*
| - ⇒ *remove-and-or z or remove-and-or y*)
| *remove-and-or* (*x and y*) = *remove-and-or x and remove-and-or y*
| *remove-and-or x* = *x*

lemma *remove-and-or-correct*:
S ⊨*P remove-and-or x* ⟷ *S* ⊨*P x*
⟨*proof*⟩

17.2 Optimized Equations

fun *af-letter-simp*
where
af-letter-simp true ν = *true*
| *af-letter-simp false ν* = *false*
| *af-letter-simp p*(*a*) *ν* = (*if a* ∈ *ν then true else false*)
| *af-letter-simp* (*np*(*a*)) *ν* = (*if a* ∉ *ν then true else false*)
| *af-letter-simp* (*φ and ψ*) *ν* = (*case φ of*
true ⇒ *af-letter-simp ψ ν*
| *false* ⇒ *false*
| *p*(*a*) ⇒ *if a* ∈ *ν then af-letter-simp ψ ν else false*
| *np*(*a*) ⇒ *if a* ∉ *ν then af-letter-simp ψ ν else false*
| *G φ′* ⇒
(*let*
φ″ = *af-letter-simp φ′ ν*;
ψ″ = *af-letter-simp ψ ν*
in
(*if φ″* = *ψ″ then mk-and′* (*G φ′*) *φ″ else mk-and id* (*mk-and′* (*G φ′*) *φ″*) *ψ″*))
| - ⇒ *mk-and id* (*af-letter-simp φ ν*) (*af-letter-simp ψ ν*))
| *af-letter-simp* (*φ or ψ*) *ν* = (*case φ of*
true ⇒ *true*
| *false* ⇒ *af-letter-simp ψ ν*
| *p*(*a*) ⇒ *if a* ∈ *ν then true else af-letter-simp ψ ν*
| *np*(*a*) ⇒ *if a* ∉ *ν then true else af-letter-simp ψ ν*
| *F φ′* ⇒
(*let*
φ″ = *af-letter-simp φ′ ν*;

$\psi'' = \textit{af-letter-simp}\ \psi\ \nu$
in
(*if* $\varphi'' = \psi''$ *then mk-or'* (*F* φ') φ'' *else mk-or id* (*mk-or'* (*F* φ') φ'') ψ''))
| - ⇒ *mk-or id* (*af-letter-simp* $\varphi\ \nu$) (*af-letter-simp* $\psi\ \nu$))
| *af-letter-simp* (*X* φ) $\nu = \varphi$
| *af-letter-simp* (*G* φ) ν = *mk-and'* (*G* φ) (*af-letter-simp* $\varphi\ \nu$)
| *af-letter-simp* (*F* φ) ν = *mk-or'* (*F* φ) (*af-letter-simp* $\varphi\ \nu$)
| *af-letter-simp* (φ *U* ψ) ν = *mk-or'* (*mk-and'* (φ *U* ψ) (*af-letter-simp* $\varphi\ \nu$)) (*af-letter-simp* $\psi\ \nu$)

lemma *af-letter-simp-correct*:
$S \models_P$ *af-letter* $\varphi\ \nu \longleftrightarrow S \models_P$ *af-letter-simp* $\varphi\ \nu$
⟨*proof*⟩

fun *af-G-letter-simp*
where
af-G-letter-simp true ν = *true*
| *af-G-letter-simp false* ν = *false*
| *af-G-letter-simp p*(*a*) ν = (*if a* ∈ ν *then true else false*)
| *af-G-letter-simp* (*np*(*a*)) ν = (*if a* ∉ ν *then true else false*)
| *af-G-letter-simp* (φ *and* ψ) ν = (*case* φ *of*
true ⇒ *af-G-letter-simp* $\psi\ \nu$
| *false* ⇒ *false*
| *p*(*a*) ⇒ *if a* ∈ ν *then af-G-letter-simp* $\psi\ \nu$ *else false*
| *np*(*a*) ⇒ *if a* ∉ ν *then af-G-letter-simp* $\psi\ \nu$ *else false*
| - ⇒ *mk-and id* (*af-G-letter-simp* $\varphi\ \nu$) (*af-G-letter-simp* $\psi\ \nu$))
| *af-G-letter-simp* (φ *or* ψ) ν = (*case* φ *of*
true ⇒ *true*
| *false* ⇒ *af-G-letter-simp* $\psi\ \nu$
| *p*(*a*) ⇒ *if a* ∈ ν *then true else af-G-letter-simp* $\psi\ \nu$
| *np*(*a*) ⇒ *if a* ∉ ν *then true else af-G-letter-simp* $\psi\ \nu$
| *F* φ' ⇒
(*let*
$\varphi'' = \textit{af-G-letter-simp}\ \varphi'\ \nu$;
$\psi'' = \textit{af-G-letter-simp}\ \psi\ \nu$
in
(*if* $\varphi'' = \psi''$ *then mk-or'* (*F* φ') φ'' *else mk-or id* (*mk-or'* (*F* φ') φ'') ψ''))
| - ⇒ *mk-or id* (*af-G-letter-simp* $\varphi\ \nu$) (*af-G-letter-simp* $\psi\ \nu$))
| *af-G-letter-simp* (*X* φ) $\nu = \varphi$
| *af-G-letter-simp* (*G* φ) ν = *G* φ
| *af-G-letter-simp* (*F* φ) ν = *mk-or'* (*F* φ) (*af-G-letter-simp* $\varphi\ \nu$)
| *af-G-letter-simp* (φ *U* ψ) ν = *mk-or'* (*mk-and'* (φ *U* ψ) (*af-G-letter-simp*

$\varphi\ \nu))\ (af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu)$

lemma *af-G-letter-simp-correct*:
$S \models_P af\text{-}G\text{-}letter\ \varphi\ \nu \longleftrightarrow S \models_P af\text{-}G\text{-}letter\text{-}simp\ \varphi\ \nu$
⟨*proof*⟩

fun *step-simp*
where
$step\text{-}simp\ p(a)\ \nu = (if\ a \in \nu\ then\ true\ else\ false)$
$|\ step\text{-}simp\ (np(a))\ \nu = (if\ a \notin \nu\ then\ true\ else\ false)$
$|\ step\text{-}simp\ (\varphi\ and\ \psi)\ \nu = (mk\text{-}and\ id\ (step\text{-}simp\ \varphi\ \nu)\ (step\text{-}simp\ \psi\ \nu))$
$|\ step\text{-}simp\ (\varphi\ or\ \psi)\ \nu = (mk\text{-}or\ id\ (step\text{-}simp\ \varphi\ \nu)\ (step\text{-}simp\ \psi\ \nu))$
$|\ step\text{-}simp\ (X\ \varphi)\ \nu = remove\text{-}constants_P\ \varphi$
$|\ step\text{-}simp\ \varphi\ \nu = \varphi$

lemma *step-simp-correct*:
$S \models_P step\ \varphi\ \nu \longleftrightarrow S \models_P step\text{-}simp\ \varphi\ \nu$
⟨*proof*⟩

fun *Unf-simp*
where
$Unf\text{-}simp\ (\varphi\ and\ \psi) = (case\ \varphi\ of$
$\quad true \Rightarrow Unf\text{-}simp\ \psi$
$\quad |\ false \Rightarrow false$
$\quad |\ G\ \varphi' \Rightarrow$
$\quad (let$
$\quad\ \varphi'' = Unf\text{-}simp\ \varphi';\ \psi'' = Unf\text{-}simp\ \psi$
$\quad\ in$
$\quad\ (if\ \varphi'' = \psi''\ then\ mk\text{-}and'\ (G\ \varphi')\ \varphi''\ else\ mk\text{-}and\ id\ (mk\text{-}and'\ (G\ \varphi')$
$\varphi'')\ \psi''))$
$\quad |\ \text{-} \Rightarrow mk\text{-}and\ id\ (Unf\text{-}simp\ \varphi)\ (Unf\text{-}simp\ \psi))$
$|\ Unf\text{-}simp\ (\varphi\ or\ \psi) = (case\ \varphi\ of$
$\quad true \Rightarrow true$
$\quad |\ false \Rightarrow Unf\text{-}simp\ \psi$
$\quad |\ F\ \varphi' \Rightarrow$
$\quad (let$
$\quad\ \varphi'' = Unf\text{-}simp\ \varphi';\ \psi'' = Unf\text{-}simp\ \psi$
$\quad\ in$
$\quad\ (if\ \varphi'' = \psi''\ then\ mk\text{-}or'\ (F\ \varphi')\ \varphi''\ else\ mk\text{-}or\ id\ (mk\text{-}or'\ (F\ \varphi')\ \varphi'')$
$\psi''))$
$\quad |\ \text{-} \Rightarrow mk\text{-}or\ id\ (Unf\text{-}simp\ \varphi)\ (Unf\text{-}simp\ \psi))$
$|\ Unf\text{-}simp\ (G\ \varphi) = mk\text{-}and'\ (G\ \varphi)\ (Unf\text{-}simp\ \varphi)$
$|\ Unf\text{-}simp\ (F\ \varphi) = mk\text{-}or'\ (F\ \varphi)\ (Unf\text{-}simp\ \varphi)$
$|\ Unf\text{-}simp\ (\varphi\ U\ \psi) = mk\text{-}or'\ (mk\text{-}and'\ (\varphi\ U\ \psi)\ (Unf\text{-}simp\ \varphi))\ (Unf\text{-}simp$

ψ)
| *Unf-simp* $\varphi = \varphi$

lemma *Unf-simp-correct*:
$S \models_P$ *Unf* $\varphi \longleftrightarrow S \models_P$ *Unf-simp* φ
⟨*proof*⟩

fun Unf_G*-simp*
where
Unf_G*-simp* (φ *and* ψ) = *mk-and id* (Unf_G*-simp* φ) (Unf_G*-simp* ψ)
| Unf_G*-simp* (φ *or* ψ) = (*case* φ *of*
true ⇒ *true*
| *false* ⇒ Unf_G*-simp* ψ
| *F* φ' ⇒
(*let*
φ'' = Unf_G*-simp* φ'; ψ'' = Unf_G*-simp* ψ
in
(*if* $\varphi'' = \psi''$ *then mk-or'* (*F* φ') φ'' *else mk-or id* (*mk-or'* (*F* φ') φ'')
ψ''))
| - ⇒ *mk-or id* (Unf_G*-simp* φ) (Unf_G*-simp* ψ))
| Unf_G*-simp* (*F* φ) = *mk-or'* (*F* φ) (Unf_G*-simp* φ)
| Unf_G*-simp* (φ *U* ψ) = *mk-or'* (*mk-and'* (φ *U* ψ) (Unf_G*-simp* φ)) (Unf_G*-simp*
ψ)
| Unf_G*-simp* $\varphi = \varphi$

lemma Unf_G*-simp-correct*:
$S \models_P$ Unf_G $\varphi \longleftrightarrow S \models_P$ Unf_G*-simp* φ
⟨*proof*⟩

fun *af-letter-opt-simp*
where
af-letter-opt-simp true ν = *true*
| *af-letter-opt-simp false* ν = *false*
| *af-letter-opt-simp* $p(a)$ ν = (*if* $a \in \nu$ *then true else false*)
| *af-letter-opt-simp* ($np(a)$) ν = (*if* $a \notin \nu$ *then true else false*)
| *af-letter-opt-simp* (φ *and* ψ) ν = (*case* φ *of*
true ⇒ *af-letter-opt-simp* ψ ν
| *false* ⇒ *false*
| $p(a)$ ⇒ *if* $a \in \nu$ *then af-letter-opt-simp* ψ ν *else false*
| $np(a)$ ⇒ *if* $a \notin \nu$ *then af-letter-opt-simp* ψ ν *else false*
| *G* φ' ⇒
(*let*
φ'' = *Unf-simp* φ';
ψ'' = *af-letter-opt-simp* ψ ν

```
      in
       (if φ'' = ψ'' then mk-and' (G φ') φ'' else mk-and id (mk-and' (G φ')
φ'') ψ''))
   | - ⇒ mk-and id (af-letter-opt-simp φ ν) (af-letter-opt-simp ψ ν))
| af-letter-opt-simp (φ or ψ) ν = (case φ of
     true ⇒ true
   | false ⇒ af-letter-opt-simp ψ ν
   | p(a) ⇒ if a ∈ ν then true else af-letter-opt-simp ψ ν
   | np(a) ⇒ if a ∉ ν then true else af-letter-opt-simp ψ ν
   | F φ' ⇒
     (let
       φ'' = Unf-simp φ';
       ψ'' = af-letter-opt-simp ψ ν
      in
       (if φ'' = ψ'' then mk-or' (F φ') φ'' else mk-or id (mk-or' (F φ') φ'')
ψ''))
   | - ⇒ mk-or id (af-letter-opt-simp φ ν) (af-letter-opt-simp ψ ν))
| af-letter-opt-simp (X φ) ν = Unf-simp φ
| af-letter-opt-simp (G φ) ν = mk-and' (G φ) (Unf-simp φ)
| af-letter-opt-simp (F φ) ν = mk-or' (F φ) (Unf-simp φ)
| af-letter-opt-simp (φ U ψ) ν = mk-or' (mk-and' (φ U ψ) (Unf-simp φ))
(Unf-simp ψ)

lemma af-letter-opt-simp-correct:
  S ⊨P af-letter-opt φ ν ⟷ S ⊨P af-letter-opt-simp φ ν
⟨proof⟩

fun af-G-letter-opt-simp
where
  af-G-letter-opt-simp true ν = true
| af-G-letter-opt-simp false ν = false
| af-G-letter-opt-simp p(a) ν = (if a ∈ ν then true else false)
| af-G-letter-opt-simp (np(a)) ν  = (if a ∉ ν then true else false)
| af-G-letter-opt-simp (φ and ψ) ν = (case φ of
     true ⇒ af-G-letter-opt-simp ψ ν
   | false ⇒ false
   | p(a) ⇒ if a ∈ ν then af-G-letter-opt-simp ψ ν else false
   | np(a) ⇒ if a ∉ ν then af-G-letter-opt-simp ψ ν else false
   | - ⇒ mk-and id (af-G-letter-opt-simp φ ν) (af-G-letter-opt-simp ψ ν))
| af-G-letter-opt-simp (φ or ψ) ν = (case φ of
     true ⇒ true
   | false ⇒ af-G-letter-opt-simp ψ ν
   | p(a) ⇒ if a ∈ ν then true else af-G-letter-opt-simp ψ ν
   | np(a) ⇒ if a ∉ ν then true else af-G-letter-opt-simp ψ ν
```

$|\ F\ \varphi' \Rightarrow$
$(let$
$\varphi'' = Unf_G\text{-}simp\ \varphi';$
$\psi'' = af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ \psi\ \nu$
in
$(if\ \varphi'' = \psi''\ then\ mk\text{-}or'\ (F\ \varphi')\ \varphi''\ else\ mk\text{-}or\ id\ (mk\text{-}or'\ (F\ \varphi')\ \varphi'')\ \psi''))$
$|\ \text{-} \Rightarrow mk\text{-}or\ id\ (af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ \varphi\ \nu)\ (af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ \psi\ \nu))$
$|\ af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ (X\ \varphi)\ \nu = Unf_G\text{-}simp\ \varphi$
$|\ af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ (G\ \varphi)\ \nu = G\ \varphi$
$|\ af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ (F\ \varphi)\ \nu = mk\text{-}or'\ (F\ \varphi)\ (Unf_G\text{-}simp\ \varphi)$
$|\ af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ (\varphi\ U\ \psi)\ \nu = mk\text{-}or'\ (mk\text{-}and'\ (\varphi\ U\ \psi)\ (Unf_G\text{-}simp\ \varphi))\ (Unf_G\text{-}simp\ \psi)$

lemma *af-G-letter-opt-simp-correct*:
$S \models_P af\text{-}G\text{-}letter\text{-}opt\ \varphi\ \nu \longleftrightarrow S \models_P af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ \varphi\ \nu$
⟨*proof*⟩

17.3 Register Code Equations

lemma [*code*]:
$\uparrow af\ (Abs\ \varphi)\ \nu = Abs\ (remove\text{-}and\text{-}or\ (af\text{-}letter\text{-}simp\ \varphi\ \nu))$
⟨*proof*⟩

lemma [*code*]:
$\uparrow af_G\ (Abs\ \varphi)\ \nu = Abs\ (remove\text{-}and\text{-}or\ (af\text{-}G\text{-}letter\text{-}simp\ \varphi\ \nu))$
⟨*proof*⟩

lemma [*code*]:
$\uparrow step\ (Abs\ \varphi)\ \nu = Abs\ (step\text{-}simp\ \varphi\ \nu)$
⟨*proof*⟩

lemma [*code*]:
$\uparrow Unf\ (Abs\ \varphi) = Abs\ (remove\text{-}and\text{-}or\ (Unf\text{-}simp\ \varphi))$
⟨*proof*⟩

lemma [*code*]:
$\uparrow Unf_G\ (Abs\ \varphi) = Abs\ (remove\text{-}and\text{-}or\ (Unf_G\text{-}simp\ \varphi))$
⟨*proof*⟩

lemma [*code*]:
$\uparrow af_{\mathfrak{U}}\ (Abs\ \varphi)\ \nu = Abs\ (remove\text{-}and\text{-}or\ (af\text{-}letter\text{-}opt\text{-}simp\ \varphi\ \nu))$
⟨*proof*⟩

lemma [*code*]:
  $\uparrow af_{G\mathfrak{U}}\ (Abs\ \varphi)\ \nu = Abs\ (remove\text{-}and\text{-}or\ (af\text{-}G\text{-}letter\text{-}opt\text{-}simp\ \varphi\ \nu))$
  ⟨*proof*⟩

end

18 Executable Translation from Mojmir to Rabin Automata

theory *Mojmir-Rabin-Impl*
  imports *Main ../Mojmir-Rabin*
begin

— Ranking functions are stored as lists sorted ascending by the state rank

fun *init* :: $'a \Rightarrow 'a\ list$
where
  $init\ q_0 = [q_0]$

fun *nxt* :: $'b\ set \Rightarrow ('a, 'b)\ DTS \Rightarrow 'a \Rightarrow ('a\ list, 'b)\ DTS$
where
  $nxt\ \Sigma\ \delta\ q_0 = (\lambda qs\ \nu.\ remdups\text{-}fwd\ ((filter\ (\lambda q.\ \neg semi\text{-}mojmir\text{-}def.sink\ \Sigma\ \delta\ q_0\ q)\ (map\ (\lambda q.\ \delta\ q\ \nu)\ qs)) @ [q_0]))$

— Recompute the rank from the list

fun *rk* :: $'a\ list \Rightarrow 'a \Rightarrow nat\ option$
where
  $rk\ qs\ q = (let\ i = index\ qs\ q\ in\ if\ i \neq length\ qs\ then\ Some\ i\ else\ None)$

— Instead of computing the whole sets for fail, merge, and succeed, we define filters (a.k.a. characteristic functions)

fun *fail-filt* :: $'b\ set \Rightarrow ('a, 'b)\ DTS \Rightarrow 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a\ list, 'b)\ transition \Rightarrow bool$
where
  $fail\text{-}filt\ \Sigma\ \delta\ q_0\ F\ (r, \nu, \text{-}) = (\exists q \in set\ r.\ let\ q' = \delta\ q\ \nu\ in\ (\neg F\ q') \wedge semi\text{-}mojmir\text{-}def.sink\ \Sigma\ \delta\ q_0\ q')$

fun *merge-filt* :: $('a, 'b)\ DTS \Rightarrow 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow nat \Rightarrow ('a\ list, 'b)\ transition \Rightarrow bool$
where
  $merge\text{-}filt\ \delta\ q_0\ F\ i\ (r, \nu, \text{-}) = (\exists q \in set\ r.\ let\ q' = \delta\ q\ \nu\ in\ the\ (rk\ r\ q)$

$< i \wedge \neg F\ q' \wedge ((\exists\, q'' \in set\ r.\ q'' \neq q \wedge \delta\ q''\ \nu = q') \vee q' = q_0))$

fun *succeed-filt* :: $('a, 'b)\ DTS \Rightarrow 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow nat \Rightarrow ('a\ list, 'b)\ transition \Rightarrow bool$
where
succeed-filt $\delta\ q_0\ F\ i\ (r, \nu, \text{-}) = (\exists\, q \in set\ r.\ let\ q' = \delta\ q\ \nu\ in\ rk\ r\ q = Some\ i\ \wedge (\neg F\ q \vee q = q_0) \wedge F\ q')$

18.0.1 nxt Properties

lemma *nxt-run-distinct*:
distinct $(run\ (nxt\ \Sigma\ \Delta\ q_0)\ (init\ q_0)\ w\ n)$
⟨*proof*⟩

lemma *nxt-run-reverse-step*:
fixes $\Sigma\ \delta\ q_0\ w$
defines $r \equiv run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w$
assumes $q \in set\ (r\ (Suc\ n))$
assumes $q \neq q_0$
shows $\exists\, q' \in set\ (r\ n).\ \delta\ q'\ (w\ n) = q$
⟨*proof*⟩

lemma *nxt-run-sink-free*:
$q \in set\ (run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w\ n) \Longrightarrow \neg semi\text{-}mojmir\text{-}def.sink\ \Sigma\ \delta\ q_0\ q$
⟨*proof*⟩

18.0.2 rk Properties

lemma *rk-bounded*:
$rk\ xs\ x = Some\ i \Longrightarrow i < length\ xs$
⟨*proof*⟩

lemma *rk-facts*:
$x \in set\ xs \longleftrightarrow rk\ xs\ x \neq None$
$x \in set\ xs \longleftrightarrow (\exists\, i.\ rk\ xs\ x = Some\ i)$
⟨*proof*⟩

lemma *rk-split*:
$y \notin set\ xs \Longrightarrow rk\ (xs\ @\ y\ \#\ zs)\ y = Some\ (length\ xs)$
⟨*proof*⟩

lemma *rk-split-card*:
$y \notin set\ xs \Longrightarrow distinct\ xs \Longrightarrow rk\ (xs\ @\ y\ \#\ zs)\ y = Some\ (card\ (set\ xs))$

⟨*proof*⟩

lemma *rk-split-card-takeWhile*:
assumes $x \in set\ xs$
assumes *distinct xs*
shows $rk\ xs\ x = Some\ (card\ (set\ (takeWhile\ (\lambda y.\ y \neq x)\ xs)))$
⟨*proof*⟩

lemma *take-rk*:
assumes *distinct xs*
shows $set\ (take\ i\ xs) = \{q.\ \exists j < i.\ rk\ xs\ q = Some\ j\}$
(**is** *?rhs* = *?lhs*)
⟨*proof*⟩

lemma *drop-rk*:
assumes *distinct xs*
shows $set\ (drop\ i\ xs) = \{q.\ \exists j \geq i.\ rk\ xs\ q = Some\ j\}$
⟨*proof*⟩

18.0.3 Relation to (Semi) Mojmir Automata

lemma (**in** *semi-mojmir*) *nxt-run-configuration*:
defines $r \equiv run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w$
shows $q \in set\ (r\ n) \longleftrightarrow \neg sink\ q \wedge configuration\ q\ n \neq \{\}$
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *nxt-run-sorted*:
defines $r \equiv run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w$
shows $sorted\ (map\ (\lambda q.\ the\ (oldest\text{-}token\ q\ n))\ (r\ n))$
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *nxt-run-senior-states*:
defines $r \equiv run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w$
assumes $\neg sink\ q$
assumes $configuration\ q\ n \neq \{\}$
shows $senior\text{-}states\ q\ n = set\ (takeWhile\ (\lambda q'.\ q' \neq q)\ (r\ n))$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *nxt-run-state-rank*:
$state\text{-}rank\ q\ n = rk\ (run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w\ n)\ q$
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *nxt-foldl-state-rank*:

state-rank q n = rk (foldl (nxt Σ δ q₀) (init q₀) (map w [0..<n])) q
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *nxt-run-step-run*:
run step initial w = rk o (run (nxt Σ δ q₀) (init q₀) w)
⟨*proof*⟩

definition (**in** *semi-mojmir-def*) Q_E
where
$Q_E \equiv$ *reach* Σ (*nxt* Σ δ q_0) (*init* q_0)

lemma (**in** *semi-mojmir*) *finite-Q*:
finite Q_E
⟨*proof*⟩

lemma (**in** *mojmir-to-rabin-def*) *filt-equiv*:
(*rk x*, ν, *y*) ∈ $fail_R$ ⟷ *fail-filt* Σ δ q_0 (λ*x*. *x* ∈ *F*) (*x*, ν, *y*′)
(*rk x*, ν, *y*) ∈ $succeed_R$ *i* ⟷ *succeed-filt* δ q_0 (λ*x*. *x* ∈ *F*) *i* (*x*, ν, *y*′)
(*rk x*, ν, *y*) ∈ $merge_R$ *i* ⟷ *merge-filt* δ q_0 (λ*x*. *x* ∈ *F*) *i* (*x*, ν, *y*′)
⟨*proof*⟩

lemma *fail-filt-eq*:
fail-filt Σ δ q_0 *P* (*x*, ν, *y*) ⟷ (*rk x*, ν, *y*′) ∈ *mojmir-to-rabin-def*.$fail_R$ Σ δ q_0 {*x*. *P x*}
⟨*proof*⟩

lemma *merge-filt-eq*:
merge-filt δ q_0 *P i* (*x*, ν, *y*) ⟷ (*rk x*, ν, *y*′) ∈ *mojmir-to-rabin-def*.$merge_R$ δ q_0 {*x*. *P x*} *i*
⟨*proof*⟩

lemma *succeed-filt-eq*:
succeed-filt δ q_0 *P i* (*x*, ν, *y*) ⟷ (*rk x*, ν, *y*′) ∈ *mojmir-to-rabin-def*.$succeed_R$ δ q_0 {*x*. *P x*} *i*
⟨*proof*⟩

theorem (**in** *mojmir-to-rabin*) *rabin-accept-iff-rabin-list-accept-rank*:
$accepting\text{-}pair_R$ $\delta_{\mathcal{R}}$ $q_{\mathcal{R}}$ ($Acc_{\mathcal{R}}$ *i*) *w* ⟷ $accepting\text{-}pair_R$ (*nxt* Σ δ q_0) (*init* q_0) ({*t*. *fail-filt* Σ δ q_0 (λ*x*. *x* ∈ *F*) *t*} ∪ {*t*. *merge-filt* δ q_0 (λ*x*. *x* ∈ *F*) *i* *t*}, {*t*. *succeed-filt* δ q_0 (λ*x*. *x* ∈ *F*) *i* *t*}) *w*
(**is** $accepting\text{-}pair_R$ $\delta_{\mathcal{R}}$ $q_{\mathcal{R}}$ (*?F*, *?I*) *w* ⟷ $accepting\text{-}pair_R$ (*nxt* Σ δ q_0) (*init* q_0) (*?F*′, *?I*′) *w*)
⟨*proof*⟩

18.1 Compute Rabin Automata List Representation

fun *mojmir-to-rabin-exec*
where
mojmir-to-rabin-exec $\Sigma\ \delta\ q_0\ F$ = (
let
$q_0' = init\ q_0$;
$\delta' = \delta_L\ \Sigma\ (nxt\ (set\ \Sigma)\ \delta\ q_0)\ q_0'$;
max-rank = *card* (*Set.filter* (*Not o semi-mojmir-def.sink* (*set* Σ) $\delta\ q_0$) ($Q_L\ \Sigma\ \delta\ q_0$));
fail = *Set.filter* (*fail-filt* (*set* Σ) $\delta\ q_0\ F$) δ';
merge = (λi. *Set.filter* (*merge-filt* $\delta\ q_0\ F\ i$) δ');
succeed = (λi. *Set.filter* (*succeed-filt* $\delta\ q_0\ F\ i$) δ')
in
(δ', q_0', (λi. (*fail* $\cup$ (*merge i*), *succeed i*)) ‘ {*0*..<*max-rank*}))

18.2 Code Generation

declare *semi-mojmir-def.sink-def* [*code*]

— Drop computation of length by different code equation
fun *index-option* :: *nat* $\Rightarrow$ *'a list* $\Rightarrow$ *'a* $\Rightarrow$ *nat option*
where
index-option n [] *y* = *None*
| *index-option n* (*x* # *xs*) *y* = (*if x* = *y then Some n else index-option* (*Suc n*) *xs y*)

declare *rk.simps* [*code del*]

lemma *rk-eq-index-option* [*code*]:
rk xs x = *index-option 0 xs x*
⟨*proof*⟩
export-code *init nxt fail-filt succeed-filt merge-filt mojmir-to-rabin-exec* **checking**

lemma (**in** *mojmir*) *max-rank-card*:
assumes Σ = *set* Σ'
shows *max-rank* = *card* (*Set.filter* (*Not o semi-mojmir-def.sink* (*set* Σ') $\delta\ q_0$) ($Q_L\ \Sigma'\ \delta\ q_0$))
⟨*proof*⟩

theorem (**in** *mojmir-to-rabin*) *exec-correct*:
assumes Σ = *set* Σ'
shows *accept* $\longleftrightarrow$ *accept*$_R$*-LTS* (*mojmir-to-rabin-exec* $\Sigma'\ \delta\ q_0$ (λx. $x \in$

F)) *w* (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

end

19 Executable Translation from LTL to Rabin Automata

theory *LTL-Rabin-Impl*
imports *Main ../Auxiliary/Map2 ../LTL-Rabin ../LTL-Rabin-Unfold-Opt af-Impl Mojmir-Rabin-Impl*
begin

19.1 Template

19.1.1 Definition

locale *ltl-to-rabin-base-code-def* = *ltl-to-rabin-base-def* +
fixes
M-fin$_C$ :: *'a ltl* ⇒ (*'a ltl*, *nat*) *mapping* ⇒ (*'a ltl*$_P$ × (*'a ltl*, *'a ltl*$_P$ *list*) *mapping*, *'a set*) *transition* ⇒ *bool*
begin

— Transition Function and Initial State

fun *delta*$_C$
where
delta$_C$ Σ = δ × ↑Δ$_\times$ (*nxt* Σ δ$_M$ *o* *q*$_{0M}$ *o* *theG*)

fun *initial*$_C$
where
initial$_C$ φ = (*q*$_0$ φ, *Mapping.tabulate* (*G-list* φ) (*init o q*$_{0M}$ *o theG*))

— Acceptance Condition

definition *max-rank-of*$_C$
where
max-rank-of$_C$ Σ ψ = *card* (*Set.filter* (*Not o semi-mojmir-def.sink* (*set* Σ) δ$_M$ (*q*$_{0M}$ (*theG* ψ))) (*Q*$_L$ Σ δ$_M$ (*q*$_{0M}$ (*theG* ψ))))

fun *Acc-fin*$_C$
where
Acc-fin$_C$ Σ π χ ((-, *m'*), ν, -) = (
let

```
      t = (the (Mapping.lookup m' χ), ν, []); — Third element is unused.
Hence it is safe to pass a dummy value.
      𝒢 = Mapping.keys π
    in
      fail-filt Σ δ_M (q0_M (theG χ)) (ltl-prop-entails-abs 𝒢) t
  ∨ merge-filt δ_M (q0_M (theG χ)) (ltl-prop-entails-abs 𝒢) (the (Mapping.lookup
π χ)) t)

fun Acc-inf_C
where
  Acc-inf_C π χ ((-, m'), ν, -) = (
    let
      t = (the (Mapping.lookup m' χ), ν, []); — Third element is unused.
Hence it is safe to pass a dummy value.
      𝒢 = Mapping.keys π
    in
    succeed-filt δ_M (q0_M (theG χ)) (ltl-prop-entails-abs 𝒢) (the (Mapping.lookup
π χ)) t)

definition mappings_C :: 'a set list ⇒ 'a ltl ⇒ ('a ltl, nat) mapping set
where
  mappings_C Σ φ ≡ {π. Mapping.keys π ⊆ G φ ∧ (∀ χ ∈ (Mapping.keys
π). the (Mapping.lookup π χ) < max-rank-of_C Σ χ)}

definition reachable-transitions_C
where
  reachable-transitions_C Σ φ ≡ δ_L Σ (delta_C (set Σ)) (initial_C φ)

fun ltl-to-generalized-rabin_C
where
  ltl-to-generalized-rabin_C Σ φ = (
    let
      δ-LTS = reachable-transitions_C Σ φ;
      α-fin-filter = λπ t. M-fin_C φ π t ∨ (∃ χ ∈ Mapping.keys π. Acc-fin_C
(set Σ) π χ t);
     to-pair = λπ. (Set.filter (α-fin-filter π) δ-LTS, (λχ. Set.filter (Acc-inf_C
π χ) δ-LTS) ‘ Mapping.keys π);
      α = to-pair ‘ (mappings_C Σ φ) — Multi-thread here!, prove mappings
(set ...) equation
    in
      (δ-LTS, initial_C φ, α))

lemma mappings_C-code:
  mappings_C Σ φ = (
```

let
Gs = *G-list* φ;
max-rank = *Mapping.lookup* (*Mapping.tabulate Gs* (*max-rank-of*$_C$ Σ))
in
set (*concat* (*map* (*mapping-generator-list* (λx. [*0* ..< *the* (*max-rank* *x*)])) (*subseqs Gs*))))
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *reach-delta-initial*:
assumes (x, y) ∈ *reach* Σ (*delta*$_C$ Σ) (*initial*$_C$ φ)
assumes χ ∈ **G** φ
shows *Mapping.lookup* y χ ≠ *None* (**is** *?t1*)
and *distinct* (*the* (*Mapping.lookup* y χ)) (**is** *?t2*)
⟨*proof*⟩

end

19.1.2 Correctness

fun *abstract-state* :: $'x$ × ($'y$, $'z$ *list*) *mapping* ⇒ $'x$ × ($'y$ ⇀ $'z$ ⇀ *nat*)
where
abstract-state (a, b) = (a, (*map-option rk*) *o* (*Mapping.lookup* b))

fun *abstract-transition*
where
abstract-transition (q, ν, q') = (*abstract-state* q, ν, *abstract-state* q')

locale *ltl-to-rabin-base-code* = *ltl-to-rabin-base* + *ltl-to-rabin-base-code-def* +
assumes
M-fin$_C$*-correct*: ⟦t ∈ *reach*$_t$ Σ (*delta*$_C$ Σ) (*initial*$_C$ φ); *dom* π ⊆ **G** φ⟧ ⟹
abstract-transition t ∈ *M-fin* π = *M-fin*$_C$ φ (*Mapping.Mapping* π) t
begin

lemma *finite-reach*$_C$:
finite (*reach*$_t$ Σ (*delta*$_C$ Σ) (*initial*$_C$ φ))
⟨*proof*⟩

lemma *max-rank-of*$_C$*-eq*:
assumes Σ = *set* Σ'
shows *max-rank-of*$_C$ Σ' ψ = *max-rank-of* Σ ψ
⟨*proof*⟩

lemma *reachable-transitions$_C$-eq*:
assumes Σ = *set* Σ'
shows *reachable-transitions$_C$* Σ' φ = *reach$_t$* Σ (*delta$_C$* Σ) (*initial$_C$* φ)
⟨*proof*⟩

lemma *run-abstraction-correct*:
run (*delta* Σ) (*initial* φ) w = *abstract-state* o (*run* (*delta$_C$* Σ) (*initial$_C$* φ) w)
⟨*proof*⟩

lemma
assumes $t \in$ *reach$_t$* Σ (*delta$_C$* Σ) (*initial$_C$* φ)
assumes $\chi \in$ **G** φ
shows *Acc-fin$_C$-correct*:
abstract-transition $t \in$ *Acc-fin* Σ π $\chi \longleftrightarrow$ *Acc-fin$_C$* Σ (*Mapping.Mapping* π) χ t (**is** *?t1*)
and *Acc-inf$_C$-correct*:
abstract-transition $t \in$ *Acc-inf* π $\chi \longleftrightarrow$ *Acc-inf$_C$* (*Mapping.Mapping* π) χ t (**is** *?t2*)
⟨*proof*⟩

theorem *ltl-to-generalized-rabin$_C$-correct*:
assumes Σ = *set* Σ'
shows *accept$_{GR}$* (*ltl-to-generalized-rabin* Σ φ) $w \longleftrightarrow$ *accept$_{GR}$-LTS* (*ltl-to-generalized-rabin$_C$* Σ' φ) w
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
⟨*proof*⟩

end

19.2 Generalized Deterministic Rabin Automaton (af)

definition *M-fin$_C$-af-lhs* :: *'a ltl* ⇒ (*'a ltl*, *nat*) *mapping* ⇒ (*'a ltl*, (*'a ltl$_P$ list*)) *mapping* ⇒ *'a ltl$_P$*
where
M-fin$_C$-af-lhs φ π m' ≡
let
$\mathcal{G}$ = *Mapping.keys* π;
$\mathcal{G}_L$ = *filter* (λ*x*. $x \in \mathcal{G}$) (*G-list* φ);
mk-conj = λχ. *foldl and-abs* (*Abs* χ) (*map* (↑*eval$_G$* $\mathcal{G}$) (*drop* (*the* (*Mapping.lookup* π χ)) (*the* (*Mapping.lookup* m' χ))))
in

$\uparrow And\ (map\ mk\text{-}conj\ \mathcal{G}_L)$

fun *M-fin$_C$-af* :: *'a ltl* ⇒ (*'a ltl*, *nat*) *mapping* ⇒ (*'a ltl$_P$* × ((*'a ltl*, (*'a ltl$_P$ list*)) *mapping*), *'a set*) *transition* ⇒ *bool*
where
M-fin$_C$-af $\varphi\ \pi$ ((*φ'*, *m'*), -) = *Not* ((*M-fin$_C$-af-lhs* $\varphi\ \pi$ *m'*) $\uparrow\longrightarrow_P$ *φ'*)

lemma *M-fin$_C$-af-correct*:
assumes $t \in reach_t\ \Sigma$ (*ltl-to-rabin-base-code-def.delta$_C$* $\uparrow af\ \uparrow af_G$ *Abs* Σ) (*ltl-to-rabin-base-code-def.initial$_C$ Abs Abs* φ)
assumes *dom* $\pi \subseteq \mathbf{G}\ \varphi$
shows *abstract-transition t* ∈ *M-fin* π = *M-fin$_C$-af* φ (*Mapping.Mapping* π) *t*
⟨*proof*⟩

definition
ltl-to-generalized-rabin$_C$-af ≡ *ltl-to-rabin-base-code-def.ltl-to-generalized-rabin$_C$* $\uparrow af\ \uparrow af_G$ *Abs Abs M-fin$_C$-af*

theorem *ltl-to-generalized-rabin$_C$-af-correct*:
assumes *range w* ⊆ *set* Σ
shows *w* ⊨ $\varphi \longleftrightarrow$ *accept$_{GR}$-LTS* (*ltl-to-generalized-rabin$_C$-af* Σ φ) *w*
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

19.3 Generalized Deterministic Rabin Automaton (eager af)

definition *M-fin$_C$-af$_{\mathfrak{U}}$-lhs* :: *'a ltl* ⇒ (*'a ltl*, *nat*) *mapping* ⇒ (*'a ltl*, (*'a ltl$_P$ list*)) *mapping* ⇒ *'a set* ⇒ *'a ltl$_P$*
where
M-fin$_C$-af$_{\mathfrak{U}}$-lhs $\varphi\ \pi\ m'\ \nu$ ≡
let
$\mathcal{G}$ = *Mapping.keys* π;
$\mathcal{G}_L$ = *filter* ($\lambda x.\ x \in \mathcal{G}$) (*G-list* φ);
mk-conj = $\lambda\chi.$ *foldl and-abs* (*and-abs* (*Abs* χ) ($\uparrow eval_G\ \mathcal{G}$ (*Abs* (*theG* χ)))) (*map* ($\uparrow eval_G\ \mathcal{G}\ o\ (\lambda q.\ \uparrow step\ q\ \nu)$) (*drop* (*the* (*Mapping.lookup* $\pi\ \chi$)) (*the* (*Mapping.lookup m'* χ))))
in
$\uparrow And\ (map\ mk\text{-}conj\ \mathcal{G}_L)$

fun *M-fin$_C$-af$_{\mathfrak{U}}$* :: *'a ltl* ⇒ (*'a ltl*, *nat*) *mapping* ⇒ (*'a ltl$_P$* × ((*'a ltl*, (*'a ltl$_P$ list*)) *mapping*), *'a set*) *transition* ⇒ *bool*
where
M-fin$_C$-af$_{\mathfrak{U}}$ $\varphi\ \pi$ ((*φ'*, *m'*), ν, -) = *Not* ((*M-fin$_C$-af$_{\mathfrak{U}}$-lhs* $\varphi\ \pi\ m'\ \nu$) $\uparrow\longrightarrow_P$

(↑*step* φ' ν))

lemma *M-fin$_C$-af$_{\mathfrak{U}}$-correct*:
assumes $t \in$ *reach$_t$* Σ (*ltl-to-rabin-base-code-def.delta$_C$* ↑*af$_{\mathfrak{U}}$* ↑*af$_{G\mathfrak{U}}$* (*Abs* ∘ *Unf$_G$*) Σ) (*ltl-to-rabin-base-code-def.initial$_C$* (*Abs* ∘ *Unf*) (*Abs* ∘ *Unf$_G$*) φ)
assumes *dom* $\pi \subseteq$ **G** φ
shows *abstract-transition* $t \in$ *M$_{\mathfrak{U}}$-fin* π = *M-fin$_C$-af$_{\mathfrak{U}}$* φ (*Mapping.Mapping* π) t
⟨*proof*⟩

definition
ltl-to-generalized-rabin$_C$-af$_{\mathfrak{U}}$ ≡ *ltl-to-rabin-base-code-def.ltl-to-generalized-rabin$_C$* ↑*af$_{\mathfrak{U}}$* ↑*af$_{G\mathfrak{U}}$* (*Abs* ∘ *Unf*) (*Abs* ∘ *Unf$_G$*) *M-fin$_C$-af$_{\mathfrak{U}}$*

theorem *ltl-to-generalized-rabin$_C$-af$_{\mathfrak{U}}$-correct*:
assumes *range* $w \subseteq$ *set* Σ
shows $w \models \varphi \longleftrightarrow$ *accept$_{GR}$-LTS* (*ltl-to-generalized-rabin$_C$-af$_{\mathfrak{U}}$* Σ φ) w
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

end

20 Code Generation

theory *Export-Code*
imports *Main LTL-Compat LTL-Rabin-Impl*
HOL−Library.AList-Mapping
LTL.Rewriting
HOL−Library.Code-Target-Numeral
begin

20.1 External Interface

definition
ltlc-to-rabin eager mode (φ_c :: *String.literal ltlc*) ≡
(*let*
φ_n = *ltlc-to-ltln* φ_c;
Σ = *map set* (*subseqs* (*atoms-list* φ_n));
φ = *ltln-to-ltl* (*simplify mode* φ_n)
in
(*if eager then ltl-to-generalized-rabin$_C$-af$_{\mathfrak{U}}$* Σ φ *else ltl-to-generalized-rabin$_C$-af* Σ φ))

theorem *ltlc-to-rabin-exec-correct*:
assumes *range* $w \subseteq$ *Pow* (*atoms-ltlc* φ_c)
shows $w \models_c \varphi_c \longleftrightarrow$ *accept*$_{GR}$*-LTS* (*ltlc-to-rabin eager mode* φ_c) w
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

20.2 Normalize Equivalence Classes During DFS-Search

fun *norm-rep*
where
norm-rep $(i, (q, \nu, p))$ (q', ν', p') = (
let
eq-q = $(q = q')$; *eq-p* = $(p = p')$;
q'' = *if eq-q then* q' *else if* $q = p'$ *then* p' *else* q;
p'' = *if eq-p then* p' *else if* $p = q'$ *then* q' *else* p
in
$(i \mid (\textit{eq-q} \;\&\; \textit{eq-p} \;\&\; \nu = \nu'), q'', \nu, p''))$

fun *norm-fold* :: $('a, 'b)$ *transition* ⇒ $('a, 'b)$ *transition list* ⇒ (*bool* ∗ $'a$ ∗ $'b$ ∗ $'a$)
where
norm-fold (q, ν, p) *xs* = *foldl-break norm-rep fst* (*False*, q, ν, *if* $q = p$ *then* q *else* p) *xs*

definition *norm-insert* :: $('a, 'b)$ *transition* ⇒ $('a, 'b)$ *transition list* ⇒ (*bool* ∗ $('a, 'b)$ *transition list*)
where
norm-insert x xs ≡ *let* (i, x') = *norm-fold x xs in if i then* (i, xs) *else* $(i, x' \# xs)$

lemma *norm-fold*:
norm-fold (q, ν, p) *xs* = $((q, \nu, p) \in$ *set xs*, q, ν, $p)$
⟨*proof*⟩

lemma *norm-insert*:
norm-insert x xs = $(x \in$ *set xs*, *List.insert x xs*$)$
⟨*proof*⟩

declare *list-dfs-def* [*code del*]
declare *norm-insert-def* [*code-unfold*]

lemma *list-dfs-norm-insert* [*code*]:
list-dfs succ S [] = *S*
list-dfs succ S $(x \# xs)$ = (*let* $(memb, S')$ = *norm-insert x S in list-dfs*

succ S' (*if memb then xs else succ x @ xs*))
⟨*proof*⟩

20.3 Register Code Equations

lemma [*code*]:
↑$\Delta_\times$ f (*AList-Mapping.Mapping xs*) c = *AList-Mapping.Mapping* (*map-ran* (λa b. f a b c) *xs*)
⟨*proof*⟩

lemmas *ltl-to-rabin-base-code-export* [*code*, *code-unfold*] =
ltl-to-rabin-base-code-def.ltl-to-generalized-rabin$_C$*.simps*
ltl-to-rabin-base-code-def.reachable-transitions$_C$*-def*
ltl-to-rabin-base-code-def.mappings$_C$*-code*
ltl-to-rabin-base-code-def.delta$_C$*.simps*
ltl-to-rabin-base-code-def.initial$_C$*.simps*
ltl-to-rabin-base-code-def.Acc-inf$_C$*.simps*
ltl-to-rabin-base-code-def.Acc-fin$_C$*.simps*
ltl-to-rabin-base-code-def.max-rank-of$_C$*-def*

lemmas *M-fin*$_C$*-lhs* [*code del*, *code-unfold*] =
M-fin$_C$*-af*$_\mathfrak{U}$*-lhs-def M-fin*$_C$*-af-lhs-def*

— Test code export
export-code *true*$_c$ *Iff-ltlc Nop true Abs AList-Mapping.Mapping set ltlc-to-rabin* **checking**

— Export translator (and also constructors)
export-code *true*$_c$ *Iff-ltlc Nop true Abs AList-Mapping.Mapping set ltlc-to-rabin*

in *SML* **module-name** *LTL* **file** ‹../*Code*/*LTL-to-DRA-Translator.sml*›

end

References

[1] J. Esparza, J. Kretínský, and S. Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 49(3):219–271, 2016.