Converting Linear Temporal Logic to Deterministic (Generalized) Rabin Automata

Salomon Sickert

December 14, 2021

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*:
$\mathit{finite}\ S \Longrightarrow (\forall x \in S.\ \forall_\infty i.\ P\ x\ (i{::}nat)) = (\forall_\infty i.\ \forall x \in S.\ P\ x\ i)$
$\langle \mathit{proof} \rangle$

lemma *almost-all-commutative′*:
$\mathit{finite}\ S \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow \forall_\infty i.\ P\ x\ (i{::}nat)) \Longrightarrow (\forall_\infty i.\ \forall x \in S.\ P\ x\ i)$
$\langle \mathit{proof} \rangle$

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

lemma *index-properties*:
fixes $i :: \mathit{nat}$
shows $\mathit{index}\ P = \mathit{Some}\ i \Longrightarrow 0 < i \Longrightarrow \neg\ P\ (i - 1)$
and $\mathit{index}\ P = \mathit{Some}\ i \Longrightarrow j \geq i \Longrightarrow P\ j$
$\langle \mathit{proof} \rangle$

end

2 Auxiliary Map Facts

theory *Map2*
imports *Main*
begin

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

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

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

lemma *map-equal*:
$\mathit{dom}\ m = \mathit{dom}\ m' \Longrightarrow (\bigwedge x.\ x \in \mathit{dom}\ m \Longrightarrow m\ x = m'\ x) \Longrightarrow m = m'$
$\langle \mathit{proof} \rangle$

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

end

3 Auxiliary Mapping Facts

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

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

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

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

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

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

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

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 $(k\#ks) = \{Mapping.update\ k\ v\ m \mid v\ m.\ \ v \in set\ (V\ k) \wedge m \in mapping\text{-}generator\ V\ ks\}$

fun *mapping-generator-list* :: $('a \Rightarrow 'b\ list) \Rightarrow 'a\ list \Rightarrow ('a, 'b)\ mapping\ list$
where
mapping-generator-list V $[] = [Mapping.empty]$
| *mapping-generator-list* V $(k\#ks) = concat\ (map\ (\lambda m.\ map\ (\lambda v.\ Mapping.update\ k\ v\ m)\ (V\ k))\ (mapping\text{-}generator\text{-}list\ V\ ks))$

lemma *mapping-generator-code* [*code*]:
mapping-generator $V\ K = set\ (mapping\text{-}generator\text{-}list\ V\ K)$
$\langle proof \rangle$

lemma *mapping-generator-set-eq*:
mapping-generator $V\ K = \{m.\ Mapping.keys\ m = set\ K \wedge (\forall k \in (set\ K).\ the\ (Mapping.lookup\ m\ k) \in set\ (V\ k))\}$
$\langle proof \rangle$

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 $('a, 'b)\ DTS = 'a \Rightarrow 'b \Rightarrow 'a$
type-synonym $('a, 'b)\ transition = ('a \times 'b \times 'a)$

4.1 Infinite Runs

fun *run* :: $('q,'a)\ DTS \Rightarrow 'q \Rightarrow 'a\ word \Rightarrow 'q\ word$
where
$run\ \delta\ q_0\ w\ 0 = q_0$
| $run\ \delta\ q_0\ w\ (Suc\ i) = \delta\ (run\ \delta\ q_0\ w\ i)\ (w\ i)$

fun run_t :: $('q,'a)\ DTS \Rightarrow 'q \Rightarrow 'a\ word \Rightarrow ('q * 'a * 'q)\ word$
where
$run_t\ \delta\ q_0\ w\ i = (run\ \delta\ q_0\ w\ i,\ w\ i,\ run\ \delta\ q_0\ w\ (Suc\ i))$

lemma *run-foldl*:
$run\ \Delta\ q_0\ w\ i = foldl\ \Delta\ q_0\ (map\ w\ [0..<i])$

$\langle proof \rangle$

lemma *run$_t$-foldl*:
$run_t\ \Delta\ q_0\ w\ i = (foldl\ \Delta\ q_0\ (map\ w\ [0..<i]),\ w\ i,\ foldl\ \Delta\ q_0\ (map\ w\ [0..<Suc\ i]))$
$\langle proof \rangle$

4.2 Reachable States and Transitions

definition $reach :: 'a\ set \Rightarrow ('b, 'a)\ DTS \Rightarrow 'b \Rightarrow 'b\ set$
where
$reach\ \Sigma\ \delta\ q_0 = \{run\ \delta\ q_0\ w\ n \mid w\ n.\ range\ w \subseteq \Sigma\}$

definition $reach_t :: 'a\ set \Rightarrow ('b, 'a)\ DTS \Rightarrow 'b \Rightarrow ('b, 'a)\ transition\ set$
where
$reach_t\ \Sigma\ \delta\ q_0 = \{run_t\ \delta\ q_0\ w\ n \mid w\ n.\ range\ w \subseteq \Sigma\}$

lemma *reach-foldl-def*:
assumes $\Sigma \neq \{\}$
shows $reach\ \Sigma\ \delta\ q_0 = \{foldl\ \delta\ q_0\ w \mid w.\ set\ w \subseteq \Sigma\}$
$\langle proof \rangle$

lemma *reach$_t$-foldl-def*:
$reach_t\ \Sigma\ \delta\ q_0 = \{(foldl\ \delta\ q_0\ w,\ \nu,\ foldl\ \delta\ q_0\ (w@[\nu])) \mid w\ \nu.\ set\ w \subseteq \Sigma \wedge \nu \in \Sigma\}$ (**is** *?lhs* = *?rhs*)
$\langle proof \rangle$

lemma *reach-card-0*:
assumes $\Sigma \neq \{\}$
shows $infinite\ (reach\ \Sigma\ \delta\ q_0) \longleftrightarrow card\ (reach\ \Sigma\ \delta\ q_0) = 0$
$\langle proof \rangle$

lemma *reach$_t$-card-0*:
assumes $\Sigma \neq \{\}$
shows $infinite\ (reach_t\ \Sigma\ \delta\ q_0) \longleftrightarrow card\ (reach_t\ \Sigma\ \delta\ q_0) = 0$
$\langle proof \rangle$

4.2.1 Relation to runs

lemma *run-subseteq-reach*:
assumes $range\ w \subseteq \Sigma$
shows $range\ (run\ \delta\ q_0\ w) \subseteq reach\ \Sigma\ \delta\ q_0$
and $range\ (run_t\ \delta\ q_0\ w) \subseteq reach_t\ \Sigma\ \delta\ q_0$
$\langle proof \rangle$

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

lemma run_t*-finite*:
  assumes $finite\ (reach\ \Sigma\ \delta\ q_0)$
  assumes $finite\ \Sigma$
  assumes $range\ w \subseteq \Sigma$
  defines $r \equiv run_t\ \delta\ q_0\ w$
  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
  $Q_L\ \Sigma\ \delta\ q_0 = (if\ \Sigma \neq []\ then\ gen\text{-}dfs\ (\lambda q.\ map\ (\delta\ q)\ \Sigma)\ Set.insert\ (\in)\ \{\}\ [q_0]\ else\ \{\})$

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
  $list\text{-}dfs\ succ\ S\ start \equiv gen\text{-}dfs\ succ\ List.insert\ (\lambda x\ xs.\ x \in set\ xs)\ S\ start$

definition δ_L :: $'a\ list \Rightarrow ('b,\ 'a)\ DTS \Rightarrow\ 'b \Rightarrow ('b,\ 'a)\ transition\ set$
where
  $\delta_L\ \Sigma\ \delta\ q_0 = set\ ($
    let
      $start = map\ (\lambda\nu.\ (q_0,\ \nu,\ \delta\ q_0\ \nu))\ \Sigma;$
      $succ = \lambda(\text{-},\ \text{-},\ q).\ (map\ (\lambda\nu.\ (q,\ \nu,\ \delta\ q\ \nu))\ \Sigma)$
    in
      $(list\text{-}dfs\ succ\ []\ start))$

lemma Q_L*-reach*:
  assumes $finite\ (reach\ (set\ \Sigma)\ \delta\ q_0)$
  shows $Q_L\ \Sigma\ \delta\ q_0 = reach\ (set\ \Sigma)\ \delta\ q_0$
⟨*proof*⟩

lemma δ_L*-reach*:
  assumes $finite\ (reach_t\ (set\ \Sigma)\ \delta\ q_0)$
  shows $\delta_L\ \Sigma\ \delta\ q_0 = reach_t\ (set\ \Sigma)\ \delta\ q_0$

$\langle proof \rangle$

lemma *reach-reach$_t$-fst*:
  $reach\ \Sigma\ \delta\ q_0 = fst\ `\ reach_t\ \Sigma\ \delta\ q_0$
  $\langle proof \rangle$

lemma *finite-reach*:
  $finite\ (reach_t\ \Sigma\ \delta\ q_0) \Longrightarrow finite\ (reach\ \Sigma\ \delta\ q_0)$
  $\langle proof \rangle$

lemma *finite-reach$_t$*:
  assumes $finite\ (reach\ \Sigma\ \delta\ q_0)$
  assumes $finite\ \Sigma$
  shows $finite\ (reach_t\ \Sigma\ \delta\ q_0)$
$\langle proof \rangle$

lemma *Q_L-eq-δ_L*:
  assumes $finite\ (reach_t\ (set\ \Sigma)\ \delta\ q_0)$
  shows $Q_L\ \Sigma\ \delta\ q_0 = fst\ `\ (\delta_L\ \Sigma\ \delta\ q_0)$
  $\langle proof \rangle$

4.3 Product of DTS

fun *simple-product* :: $('a, 'c)\ DTS \Rightarrow ('b, 'c)\ DTS \Rightarrow ('a \times 'b, 'c)\ DTS$ (-
$\times$ -)
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 $\varrho\ i = (\varrho_1\ i, \varrho_2\ i)$
  $\langle proof \rangle$

theorem *finite-reach-simple-product*:
  assumes $finite\ (reach\ \Sigma\ \delta_1\ q_1)$
  assumes $finite\ (reach\ \Sigma\ \delta_2\ q_2)$
  shows $finite\ (reach\ \Sigma\ (\delta_1 \times \delta_2)\ (q_1, q_2))$
$\langle proof \rangle$

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 $\varrho_1 \equiv run_t\ \delta_1\ q_1\ w$
defines $\varrho_2 \equiv run_t\ \delta_2\ q_2\ w$
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* $(\upharpoonleft_{-})$
where
$\upharpoonleft_x\ (q,\ \nu,\ q') = \{(m,\ \nu,\ m') \mid m\ m'.\ m\ x = Some\ q \wedge m'\ x = Some\ q'\}$

fun *project-transition* :: $'a \Rightarrow ('a \rightharpoonup 'b,\ 'c)$ *transition* $\Rightarrow ('b,\ 'c)$ *transition* $(\downharpoonleft_{-})$
where
$\downharpoonleft_x\ (m,\ \nu,\ m') = (the\ (m\ x),\ \nu,\ the\ (m'\ x))$

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*$)$ $(\upharpoonleft_{-})$
where
$\upharpoonleft_x\ (S,\ S') = (\bigcup (\upharpoonleft_x\ `\ S),\ \bigcup (\upharpoonleft_x\ `\ S'))$

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*$)$ $(\downharpoonleft_{-})$
where
$\downharpoonleft_x\ (S,\ S') = (\downharpoonleft_x\ `\ S,\ \downharpoonleft_x\ `\ S')$

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

$\langle proof \rangle$

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*: $\downharpoonright_x\ (\varrho\ i) = \varrho'\ i$
and *product-run-embed*: $\varrho\ i \in \upharpoonright_x\ (\varrho'\ i)$
$\langle proof \rangle$

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* $(range\ \varrho)$
shows *product-run-project-limit*: $\downharpoonright_x$ ‘ $limit\ \varrho = limit\ \varrho'$
and *product-run-embed-limit*: $limit\ \varrho \subseteq \bigcup\ (\upharpoonright_x$ ‘ $(limit\ \varrho'))$
$\langle proof \rangle$

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* $(range\ \varrho)$
shows $limit\ \varrho \cap (\bigcup\ (\upharpoonright_k$ ‘ $S)) = \{\} \longleftrightarrow limit\ \varrho' \cap S = \{\}$
(**is** $?lhs \longleftrightarrow ?rhs$)
$\langle proof \rangle$

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*
$\langle proof \rangle$

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* :: $('a \Rightarrow ('b, 'c)\ \mathit{DTS}) \Rightarrow (('a, 'b)\ \mathit{mapping}, 'c)\ \mathit{DTS}$ ($\uparrow\Delta_\times$) **is** *product*
parametric *product-parametric* ⟨*proof*⟩

lemma *product-abs-run-None*:
$\mathit{Mapping.lookup}\ \iota_m\ k = \mathit{None} \Longrightarrow \mathit{Mapping.lookup}\ (\mathit{run}\ (\uparrow\Delta_\times\ \delta_m)\ \iota_m\ w\ i)\ k = \mathit{None}$
⟨*proof*⟩

lemma *product-abs-run-Some*:
$\mathit{Mapping.lookup}\ \iota_m\ k = \mathit{Some}\ q_0 \Longrightarrow \mathit{Mapping.lookup}\ (\mathit{run}\ (\uparrow\Delta_\times\ \delta_m)\ \iota_m\ w\ i)\ k = \mathit{Some}\ (\mathit{run}\ (\delta_m\ k)\ q_0\ w\ i)$
⟨*proof*⟩

theorem *finite-reach-product-abs*:
assumes *finite* (*Mapping.keys* ι_m)
assumes $\bigwedge x.\ x \in (\mathit{Mapping.keys}\ \iota_m) \Longrightarrow \mathit{finite}\ (\mathit{reach}\ \Sigma\ (\delta_m\ x)\ (\mathit{the}\ (\mathit{Mapping.lookup}\ \iota_m\ x)))$
shows $\mathit{finite}\ (\mathit{reach}\ \Sigma\ (\uparrow\Delta_\times\ \delta_m)\ \iota_m)$
⟨*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
    $\Sigma :: 'a\ set$
  fixes
    — Transition Function
    $\delta :: ('b, 'a)\ DTS$
  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 $token\text{-}run :: nat \Rightarrow nat \Rightarrow 'b$
where
  $token\text{-}run\ x\ n = run\ \delta\ q_0\ (suffix\ x\ w)\ (n - x)$

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

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

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

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

fun *rank* :: *nat* ⇒ *nat* ⇒ *nat option*
where
rank x n =
($if\ x \le n \wedge \neg sink\ (token\text{-}run\ x\ n)\ then\ Some\ (card\ (older\text{-}seniors\ x\ n))\ else\ None$)

fun *senior-states* :: *'b* ⇒ *nat* ⇒ *'b set*
where
senior-states q n =
$\{p.\ \exists\, x\ y.\ oldest\text{-}token\ p\ n = Some\ y \wedge oldest\text{-}token\ q\ n = Some\ x \wedge y < x \wedge \neg\ sink\ p\}$

fun *state-rank* :: *'b* ⇒ *nat* ⇒ *nat option*
where
state-rank q n = ($if\ configuration\ q\ n \neq \{\} \wedge \neg sink\ q\ then\ Some\ (card\ (senior\text{-}states\ q\ n))\ else\ None$)

definition *max-rank* :: *nat*
where
max-rank = $card\ (reach\ \Sigma\ \delta\ q_0 - \{q.\ sink\ q\})$

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 = $\{i\ .\ \exists\, q'.\ r\ q' = Some\ i \wedge q = \delta\ q'\ \nu\} \cup (if\ q = q_0\ then\ \{max\text{-}rank\}\ else\ \{\})$

fun *step* :: (*'b* ⇒ *nat option*) ⇒ *'a* ⇒ (*'b* ⇒ *nat option*)
where
step r ν q = (
if
$\neg sink\ q \wedge pre\text{-}ranks\ r\ \nu\ q \neq \{\}$
then
$Some\ (card\ \{q'.\ \neg sink\ q' \wedge pre\text{-}ranks\ r\ \nu\ q' \neq \{\} \wedge Min\ (pre\text{-}ranks\ r\ \nu\ q') < Min\ (pre\text{-}ranks\ r\ \nu\ q)\})$
else
None)

5.1.2 Properties of Tokens

definition *token-squats* :: *nat* $\Rightarrow$ *bool*
where
  token-squats $x = (\forall\, n.\ \neg sink\ (token\text{-}run\ x\ n))$

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* $\Sigma\ \delta\ q_0$)
  assumes
    — w only contains letters from the alphabet
    bounded-w: *range* $w \subseteq \Sigma$
begin

lemma *nonempty-*Σ: $\Sigma \neq \{\}$
  ⟨*proof*⟩

lemma *bounded-w'*: $w\ i \in \Sigma$
  ⟨*proof*⟩

lemma *sink-rev-step*:
  $\neg sink\ q \Longrightarrow q = \delta\ q'\ \nu \Longrightarrow \nu \in \Sigma \Longrightarrow \neg sink\ q'$
  $\neg sink\ q \Longrightarrow q = \delta\ q'\ (w\ i) \Longrightarrow \neg sink\ q'$
  ⟨*proof*⟩

5.2 Token Run

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

lemma *token-is-not-in-sink*:
  $token\text{-}run\ x\ n \notin A \Longrightarrow token\text{-}run\ x\ (Suc\ n) \in A \Longrightarrow \neg sink\ (token\text{-}run\ x\ n)$
  ⟨*proof*⟩

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

lemma *token-run-P*:
  assumes $\neg$ P (*token-run* x n)
  assumes P (*token-run* x (*Suc* (n + m)))
  shows $\exists\, m' \leq m$. $\neg$ P (*token-run* x (n + m')) $\wedge$ P (*token-run* x (*Suc* (n + m')))
  ⟨*proof*⟩

lemma *token-run-merge-Suc*:
  assumes $x \leq n$
  assumes $y \leq n$
  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*:
  $[\![x \leq n;\ y \leq n;$ *token-run* x n = *token-run* y $n]\!] \Longrightarrow$ *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** $x \leq$ (*Suc* m) **and** $y \leq$ (*Suc* m)
    and y = *Suc* m $\vee$ *token-run* x m $\neq$ *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 $x \leq n$
  assumes *token-run* x n = q'
  assumes $q = \delta\ q'$ (w n)
  shows *token-run* x (*Suc* n) = q
  ⟨*proof*⟩

lemma *token-run-step'*:
  $x \leq n \Longrightarrow$ *token-run* x (*Suc* n) = δ (*token-run* x n) (w n)
  ⟨*proof*⟩

5.3 Configuration

5.3.1 Properties

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

lemma *configuration-finite*:
$\mathit{finite}\ (\mathit{configuration}\ q\ n)$
$\langle \mathit{proof} \rangle$

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

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

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*:
$x \leq n \Longrightarrow \mathit{configuration}\ (\mathit{token\text{-}run}\ x\ n)\ n \subseteq \mathit{configuration}\ (\mathit{token\text{-}run}\ x\ (\mathit{Suc}\ n))\ (\mathit{Suc}\ n)$
$\langle \mathit{proof} \rangle$

5.3.3 Pull-Up and Push-Down

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

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

5.3.4 Step Lemmas

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

$\langle proof \rangle$

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

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** $x \in configuration\ q'\ n$
$\langle proof \rangle$

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

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

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** $x \in configuration\ q'\ n$
$\langle proof \rangle$

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

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

5.4 Oldest Token

5.4.1 Properties

lemma *oldest-token-always-def*:
$\exists\, i.\ i \le x \wedge \mathit{oldest\text{-}token}\ (\mathit{token\text{-}run}\ x\ n)\ n = \mathit{Some}\ i$
$\langle \mathit{proof} \rangle$

lemma *oldest-token-bounded*:
$\mathit{oldest\text{-}token}\ q\ n = \mathit{Some}\ x \Longrightarrow x \le n$
$\langle \mathit{proof} \rangle$

lemma *oldest-token-distinct*:
$q \ne q' \Longrightarrow \mathit{oldest\text{-}token}\ q\ n = \mathit{Some}\ i \Longrightarrow \mathit{oldest\text{-}token}\ q'\ n = \mathit{Some}\ j \Longrightarrow i \ne j$
$\langle \mathit{proof} \rangle$

lemma *oldest-token-equal*:
$\mathit{oldest\text{-}token}\ q\ n = \mathit{Some}\ i \Longrightarrow \mathit{oldest\text{-}token}\ q'\ n = \mathit{Some}\ i \Longrightarrow q = q'$
$\langle \mathit{proof} \rangle$

5.4.2 Monotonicity

lemma *oldest-token-monotonic-Suc*:
assumes $x \le n$
assumes $\mathit{oldest\text{-}token}\ (\mathit{token\text{-}run}\ x\ n)\ n = \mathit{Some}\ i$
assumes $\mathit{oldest\text{-}token}\ (\mathit{token\text{-}run}\ x\ (\mathit{Suc}\ n))\ (\mathit{Suc}\ n) = \mathit{Some}\ j$
shows $i \ge j$
$\langle \mathit{proof} \rangle$

5.4.3 Pull-Up and Push-Down

lemma *push-down-oldest-token-configuration*:
$\mathit{oldest\text{-}token}\ q\ n = \mathit{Some}\ x \Longrightarrow x \in \mathit{configuration}\ q\ n$
$\langle \mathit{proof} \rangle$

lemma *push-down-oldest-token-token-run*:
$\mathit{oldest\text{-}token}\ q\ n = \mathit{Some}\ x \Longrightarrow \mathit{token\text{-}run}\ x\ n = q$
$\langle \mathit{proof} \rangle$

5.5 Senior Token

5.5.1 Properties

lemma *senior-le-token*:
$\mathit{senior}\ x\ n \le x$

⟨*proof*⟩

lemma *senior-token-run*:
$senior\ x\ n = senior\ y\ n \longleftrightarrow token\text{-}run\ x\ n = token\text{-}run\ y\ n$
⟨*proof*⟩

The senior of a token is always in the same state

lemma *senior-same-state*:
$token\text{-}run\ (senior\ x\ n)\ n = token\text{-}run\ x\ n$
⟨*proof*⟩

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

5.5.2 Monotonicity

lemma *senior-monotonic-Suc*:
$x \le n \Longrightarrow senior\ x\ n \ge senior\ x\ (Suc\ n)$
⟨*proof*⟩

5.5.3 Pull-Up and Push-Down

lemma *pull-up-configuration-senior*:
$[\![x \in configuration\ q\ n;\ y \in configuration\ q\ n]\!] \Longrightarrow senior\ x\ n = senior\ y\ n$
⟨*proof*⟩

lemma *push-down-senior-tokens*:
$[\![x \le n;\ y \le n;\ senior\ x\ n = senior\ y\ n]\!] \Longrightarrow \exists\, q.\ x \in configuration\ q\ n \land y \in configuration\ q\ n$
⟨*proof*⟩

5.6 Set of Older Seniors

5.6.1 Properties

lemma *older-seniors-cases-subseteq* [*case-names le ge*]:
assumes $older\text{-}seniors\ x\ n \subseteq older\text{-}seniors\ y\ n \Longrightarrow P$
assumes $older\text{-}seniors\ x\ n \supseteq older\text{-}seniors\ y\ n \Longrightarrow P$
shows P ⟨*proof*⟩

lemma *older-seniors-cases-subset* [*case-names less equal greater*]:
assumes $older\text{-}seniors\ x\ n \subset older\text{-}seniors\ y\ n \Longrightarrow P$
assumes $older\text{-}seniors\ x\ n = older\text{-}seniors\ y\ n \Longrightarrow P$
assumes $older\text{-}seniors\ x\ n \supset older\text{-}seniors\ y\ n \Longrightarrow P$

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 $\neg$ *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 $os \equiv$ *older-seniors* x n
assumes $os \neq \{\}$
shows $os = \{Max\ os\} \cup$ *older-seniors* (*Max os*) n

(**is** $?lhs = ?rhs$)
$\langle proof \rangle$

lemma *older-seniors-recursive-card*:
fixes x n
defines $os \equiv$ *older-seniors* x n
assumes $os \neq \{\}$
shows *card os* $=$ *Suc* (*card* (*older-seniors* (*Max os*) n))
$\langle proof \rangle$

lemma *older-seniors-card*:
card (*older-seniors* x n) $=$ *card* (*older-seniors* y n) $\longleftrightarrow$ *older-seniors* x n $=$ *older-seniors* y n
$\langle proof \rangle$

lemma *older-seniors-card-le*:
card (*older-seniors* x n) $<$ *card* (*older-seniors* y n) $\longleftrightarrow$ *older-seniors* x n $\subset$ *older-seniors* y n
$\langle proof \rangle$

lemma *older-seniors-card-less*:
card (*older-seniors* x n) $\leq$ *card* (*older-seniors* y n) $\longleftrightarrow$ *older-seniors* x n $\subseteq$ *older-seniors* y n
$\langle proof \rangle$

5.6.2 Monotonicity

lemma *older-seniors-monotonic-Suc*:
assumes $x \leq n$
shows *older-seniors* x n $\supseteq$ *older-seniors* x (*Suc* n)
$\langle proof \rangle$

lemma *older-seniors-monotonic*:
$x \leq n \Longrightarrow$ *older-seniors* x n $\supseteq$ *older-seniors* x $(n + m)$
$\langle proof \rangle$

lemma *older-seniors-stable*:
$x \leq n \Longrightarrow$ *older-seniors* x n $=$ *older-seniors* x $(n + m + m')$ $\Longrightarrow$ *older-seniors* x n $=$ *older-seniors* x $(n + m)$
$\langle proof \rangle$

lemma *card-older-seniors-monotonic*:
$x \leq n \Longrightarrow$ *card* (*older-seniors* x n) $\geq$ *card* (*older-seniors* x $(n + m)$)
$\langle proof \rangle$

5.6.3 Pull-Up and Push-Down

lemma *pull-up-senior-older-seniors*:
$senior\ x\ n = senior\ y\ n \Longrightarrow older\text{-}seniors\ x\ n = older\text{-}seniors\ y\ n$
⟨*proof*⟩

lemma *pull-up-senior-older-seniors-less*:
$senior\ x\ n < senior\ y\ n \Longrightarrow older\text{-}seniors\ x\ n \subseteq older\text{-}seniors\ y\ n$
⟨*proof*⟩

lemma *pull-up-senior-older-seniors-less-2*:
assumes $\neg\ sink\ (token\text{-}run\ x\ n)$
assumes $senior\ x\ n < senior\ y\ n$
shows $older\text{-}seniors\ x\ n \subset older\text{-}seniors\ y\ n$
⟨*proof*⟩

lemma *pull-up-senior-older-seniors-le*:
$senior\ x\ n \leq senior\ y\ n \Longrightarrow older\text{-}seniors\ x\ n \subseteq older\text{-}seniors\ y\ n$
⟨*proof*⟩

lemma *push-down-older-seniors-senior*:
assumes $\neg\ sink\ (token\text{-}run\ x\ n)$
assumes $\neg\ sink\ (token\text{-}run\ y\ n)$
assumes $older\text{-}seniors\ x\ n = older\text{-}seniors\ y\ n$
shows $senior\ x\ n = senior\ y\ n$
⟨*proof*⟩

5.6.4 Tower Lemma

lemma *older-seniors-tower''*:
assumes $x \leq n$
assumes $y \leq n$
assumes $\neg sink\ (token\text{-}run\ x\ n)$
assumes $\neg sink\ (token\text{-}run\ y\ n)$
assumes $older\text{-}seniors\ x\ n = older\text{-}seniors\ x\ (Suc\ n)$
assumes $older\text{-}seniors\ y\ n \subseteq older\text{-}seniors\ x\ n$
shows $older\text{-}seniors\ y\ n = older\text{-}seniors\ y\ (Suc\ n)$
⟨*proof*⟩

lemma *older-seniors-tower''2*:
assumes $x \leq n$
assumes $y \leq n$
assumes $\neg sink\ (token\text{-}run\ x\ (n + m))$
assumes $\neg sink\ (token\text{-}run\ y\ (n + m))$

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

lemma *older-seniors-tower'*:
assumes $y \in$ *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*:
⟦$x \leq n$; $y \in$ *older-seniors* x n; *older-seniors* x n = *older-seniors* x $(n + m)$⟧ $\Longrightarrow$ *older-seniors* y n = *older-seniors* y $(n + m)$
⟨*proof*⟩

5.7 Rank

5.7.1 Properties

lemma *rank-None-before*:
$x > n \Longrightarrow$ *rank* x n = *None*
⟨*proof*⟩

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

lemma *rank-Some-time*:
rank x n = *Some* $j \Longrightarrow x \leq n$
⟨*proof*⟩

lemma *rank-Some-sink*:
rank x n = *Some* $j \Longrightarrow \neg$*sink* (*token-run* x n)
⟨*proof*⟩

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

lemma *rank-initial*:

$\exists\, i.\ rank\ x\ x = Some\ i$
$\langle proof \rangle$

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

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

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

5.7.2 Bounds

lemma *max-rank-lowerbound*:
$0 < max\text{-}rank$
$\langle proof \rangle$

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** $card\ ?S4 < card\ ?S0$)
$\langle proof \rangle$

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

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

5.7.3 Monotonicity

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

$\langle proof \rangle$

5.7.4 Pull-Up and Push-Down

lemma *pull-up-senior-rank*:
$[\![x \leq n;\ y \leq n;\ senior\ x\ n = senior\ y\ n]\!] \Longrightarrow rank\ x\ n = rank\ y\ n$
$\langle proof \rangle$

lemma *pull-up-configuration-rank*:
$[\![x \in configuration\ q\ n;\ y \in configuration\ q\ n]\!] \Longrightarrow rank\ x\ n = rank\ y\ n$
$\langle proof \rangle$

lemma *push-down-rank-older-seniors*:
$[\![rank\ x\ n = rank\ y\ n;\ rank\ x\ n = Some\ i]\!] \Longrightarrow older\text{-}seniors\ x\ n = older\text{-}seniors\ y\ n$
$\langle proof \rangle$

lemma *push-down-rank-senior*:
$[\![rank\ x\ n = rank\ y\ n;\ rank\ x\ n = Some\ i]\!] \Longrightarrow senior\ x\ n = senior\ y\ n$
$\langle proof \rangle$

lemma *push-down-rank-tokens*:
$[\![rank\ x\ n = rank\ y\ n;\ rank\ x\ n = Some\ i]\!] \Longrightarrow (\exists\, q.\ x \in configuration\ q\ n \wedge y \in configuration\ q\ n)$
$\langle proof \rangle$

5.7.5 Pulled-Up Lemmas

lemma *rank-senior-senior*:
$x \leq n \Longrightarrow rank\ (senior\ x\ n)\ n = rank\ x\ n$
$\langle proof \rangle$

5.7.6 Stable Rank

definition *stable-rank* :: $nat \Rightarrow nat \Rightarrow bool$
where
$stable\text{-}rank\ x\ i = (\forall_\infty n.\ rank\ x\ n = Some\ i)$

lemma *stable-rank-unique*:
assumes *stable-rank x i*
assumes *stable-rank x j*
shows $i = j$
$\langle proof \rangle$

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

$token\text{-}squats\ x = (\exists\, i.\ stable\text{-}rank\ x\ i)$
(**is** $?lhs = ?rhs$)
⟨*proof*⟩

lemma *stable-rank-same-tokens*:
assumes $stable\text{-}rank\ x\ i$
assumes $stable\text{-}rank\ y\ j$
assumes $x \in configuration\ q\ n$
assumes $y \in configuration\ q\ n$
shows $i = j$
⟨*proof*⟩

5.7.7 Tower Lemma

lemma *rank-tower*:
assumes $i \le j$
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*:
$rank\ x\ n = Some\ j \wedge stable\text{-}rank\ x\ j \longleftrightarrow (\forall\, m \ge n.\ rank\ x\ m = Some\ j)$
(**is** $?rhs \longleftrightarrow ?lhs$)
⟨*proof*⟩

lemma *stable-rank-tower*:
assumes $j \le i$
assumes $rank\ x\ n = Some\ j$
assumes $rank\ y\ n = Some\ i$
assumes $stable\text{-}rank\ y\ i$
shows $stable\text{-}rank\ x\ j$
⟨*proof*⟩

5.8 Senior States

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

lemma *senior-states-cases-subseteq* [*case-names le ge*]:
assumes $senior\text{-}states\ p\ n \subseteq senior\text{-}states\ q\ n \Longrightarrow P$
assumes $senior\text{-}states\ p\ n \supseteq senior\text{-}states\ q\ n \Longrightarrow P$

shows P ⟨*proof*⟩

lemma *senior-states-cases-subset* [*case-names less equal greater*]:
assumes *senior-states p n* $\subset$ *senior-states q n* $\Longrightarrow$ P
assumes *senior-states p n* $=$ *senior-states q n* $\Longrightarrow$ P
assumes *senior-states p n* $\supset$ *senior-states q n* $\Longrightarrow$ 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*) $\longleftrightarrow$ *senior-states p n* = *senior-states q n*
⟨*proof*⟩

lemma *senior-states-card-le*:
card (*senior-states p n*) $<$ *card* (*senior-states q n*) $\longleftrightarrow$ *senior-states p n* $\subset$ *senior-states q n*
⟨*proof*⟩

lemma *senior-states-card-less*:
card (*senior-states p n*) $\leq$ *card* (*senior-states q n*) $\longleftrightarrow$ *senior-states p n* $\subseteq$ *senior-states q n*
⟨*proof*⟩

lemma *senior-states-older-seniors*:
(λy. *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 $x \in$ *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*:
$state\text{-}rank\ q\ n = (case\ oldest\text{-}token\ q\ n\ of\ None \Rightarrow None \mid Some\ t \Rightarrow rank\ t\ n)$
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *state-rank-eq-rank-SOME*:
$state\text{-}rank\ q\ n = (if\ configuration\ q\ n \neq \{\}\ then\ rank\ (SOME\ x.\ x \in configuration\ q\ n)\ n\ else\ None)$
⟨*proof*⟩

lemma *rank-eq-state-rank*:
$x \leq n \Longrightarrow rank\ x\ n = state\text{-}rank\ (token\text{-}run\ x\ n)\ n$
⟨*proof*⟩

5.9.2 Pull-Up and Push-Down

lemma *pull-up-configuration-state-rank*:
$configuration\ q\ n = \{\} \Longrightarrow state\text{-}rank\ q\ n = None$
⟨*proof*⟩

lemma *push-down-state-rank-tokens*:
$state\text{-}rank\ q\ n = Some\ i \Longrightarrow configuration\ q\ n \neq \{\}$
⟨*proof*⟩

lemma *push-down-state-rank-configuration-None*:
$state\text{-}rank\ q\ n = None \Longrightarrow \neg sink\ q \Longrightarrow configuration\ q\ n = \{\}$
⟨*proof*⟩

lemma *push-down-state-rank-oldest-token*:
$state\text{-}rank\ q\ n = Some\ i \Longrightarrow \exists\, x.\ oldest\text{-}token\ q\ n = Some\ x$
⟨*proof*⟩

lemma *push-down-state-rank-token-run*:
$state\text{-}rank\ q\ n = Some\ i \Longrightarrow \exists\, x.\ token\text{-}run\ x\ n = q \wedge x \leq n$
⟨*proof*⟩

5.9.3 Properties

lemma *state-rank-distinct*:
assumes *distinct*: $p \neq q$

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-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 $i \leq j \longleftrightarrow x \leq y$

$\langle proof \rangle$

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

5.10 Step Function

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

lemma *pre-oldest-configuration-range*:
$\mathit{pre\text{-}oldest\text{-}tokens}\ q\ n \subseteq \{0..\mathit{Suc}\ n\}$
$\langle proof \rangle$

lemma *pre-oldest-configuration-finite*:
$\mathit{finite}\ (\mathit{pre\text{-}oldest\text{-}tokens}\ q\ n)$
$\langle proof \rangle$

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

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

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

lemma *pre-oldest-configuration-initial-state*:
$\mathit{Suc}\ n \in \mathit{pre\text{-}oldest\text{-}tokens}\ q\ n \Longrightarrow q = q_0$
$\langle proof \rangle$

lemma *pre-oldest-configuration-initial-state-2*:
$q = q_0 \Longrightarrow \mathit{Suc}\ n \in \mathit{pre\text{-}oldest\text{-}tokens}\ q\ n$
$\langle proof \rangle$

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

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

lemma *pre-ranks-range*:
$\textit{pre-ranks } (\lambda q.\ \textit{state-rank } q\ n)\ \nu\ q \subseteq \{0..\textit{max-rank}\}$
$\langle \textit{proof} \rangle$

lemma *pre-ranks-finite*:
$\textit{finite } (\textit{pre-ranks } (\lambda q.\ \textit{state-rank } q\ n)\ \nu\ q)$
$\langle \textit{proof} \rangle$

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

lemma *pre-ranks-state-obtain*:
assumes $r_q \in \textit{pre-ranks } r\ \nu\ q - \{\textit{max-rank}\}$
obtains q' **where** $r\ q' = \textit{Some } r_q$ **and** $q = \delta\ q'\ \nu$
$\langle \textit{proof} \rangle$

lemma *pre-ranks-element*:
assumes $\textit{state-rank } q'\ n = \textit{Some } r$
assumes $q = \delta\ q'\ (w\ n)$
shows $r \in \textit{pre-ranks } (\lambda q.\ \textit{state-rank } q\ n)\ (w\ n)\ q$
$\langle \textit{proof} \rangle$

lemma *pre-ranks-initial-state*:
$\textit{max-rank} \in \textit{pre-ranks } (\lambda q.\ \textit{state-rank } q\ n)\ \nu\ q \Longrightarrow q = q_0$
$\langle \textit{proof} \rangle$

lemma *pre-ranks-initial-state-2*:
$q = q_0 \Longrightarrow \textit{max-rank} \in \textit{pre-ranks } r\ \nu\ q$
$\langle \textit{proof} \rangle$

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

$\langle proof \rangle$

lemma *pre-ranks-pre-oldest-token-Min-state-special*:
  assumes $\neg$*sink q*
  assumes *configuration q* (*Suc n*) $\neq$ {}
  shows *Min* (*pre-ranks* (λ*q*. *state-rank q n*) (*w n*) *q*) = *max-rank* $\longleftrightarrow$ *Min* (*pre-oldest-tokens q n*) = *Suc n*
  (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
$\langle proof \rangle$

lemma *pre-ranks-pre-oldest-token-Min-state*:
  assumes $\neg$*sink q*
  assumes $q = \delta\ q'\ (w\ n)$
  assumes *configuration q* (*Suc n*) $\neq$ {}
  defines *min-r* $\equiv$ *Min* (*pre-ranks* (λ*q*. *state-rank q n*) (*w n*) *q*)
  defines *min-ot* $\equiv$ *Min* (*pre-oldest-tokens q n*)
  shows *state-rank* q' *n* = *Some min-r* $\longleftrightarrow$ *oldest-token* q' *n* = *Some min-ot*
  (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
$\langle proof \rangle$

lemma *Min-pre-ranks-pre-oldest-tokens*:
  fixes *n*
  defines *r* $\equiv$ (λ*q*. *state-rank q n*)
  assumes *configuration p* (*Suc n*) $\neq$ {}
    and *configuration q* (*Suc n*) $\neq$ {}
  assumes $\neg$*sink q*
    and $\neg$*sink p*
  shows *Min* (*pre-ranks r* (*w n*) *p*) < *Min* (*pre-ranks r* (*w n*) *q*) $\longleftrightarrow$ *Min* (*pre-oldest-tokens p n*) < *Min* (*pre-oldest-tokens q n*)
  (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
$\langle proof \rangle$

5.10.1 Definition of initial and step

lemma *state-rank-initial*:
  state-rank q 0 = *initial q*
  $\langle proof \rangle$

lemma *state-rank-step*:
  state-rank q (*Suc n*) = *step* (λ*q*. *state-rank q n*) (*w n*) *q*
  (**is** *?lhs* = *?rhs*)
$\langle proof \rangle$

lemma *state-rank-step-foldl*:

$(\lambda q.\ \textit{state-rank}\ q\ n) = \textit{foldl step initial}\ (\textit{map}\ w\ [0..<n])$
⟨*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
$F :: {}'b\ \textit{set}$
begin

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

definition *token-fails* :: *nat* ⇒ *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* ($\textit{accept}_M$)
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* ⇒ (*nat* × *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 $i = \{x.\ \exists n.\ rank\ x\ n = Some\ i$
$\wedge$ *token-run* $x\ n \notin F - \{q_0\}$
$\wedge$ *token-run* $x\ (Suc\ n) \in F\}$

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

definition *fail-t* :: *nat set*
where
fail-t $= \{n.\ \exists q\ q'.\ state\text{-}rank\ q\ n \neq None \wedge q' = \delta\ q\ (w\ n) \wedge q' \notin F \wedge sink\ q'\}$

definition *merge-t* :: *nat* ⇒ *nat set*
where
merge-t $i = \{n.\ \exists q\ q'\ j.\ state\text{-}rank\ q\ n = Some\ j \wedge j < i \wedge q' = \delta\ q\ (w\ n) \wedge q' \notin F \wedge$
$((\exists q''.\ q'' \neq q \wedge q' = \delta\ q''\ (w\ n) \wedge state\text{-}rank\ q''\ n \neq None) \vee q' = q_0)\}$

definition *succeed-t* :: *nat* ⇒ *nat set*
where
succeed-t $i = \{n.\ \exists q.\ state\text{-}rank\ q\ n = Some\ i \wedge q \notin F - \{q_0\} \wedge \delta\ q\ (w\ n) \in F\}$

fun $\mathcal{S}$
where
$\mathcal{S}\ n = F \cup \{q.\ (\exists j \geq the\ smallest\text{-}accepting\text{-}rank.\ state\text{-}rank\ q\ n = Some\ j)\}$

end

locale *mojmir* = *semi-mojmir* + *mojmir-def* +
assumes
— All states reachable from final states are also final
wellformed-F: $\bigwedge q\ \nu.\ q \in F \Longrightarrow \delta\ q\ \nu \in F$
begin

lemma *token-stays-in-final-states*:
token-run $x\ n \in F \Longrightarrow$ *token-run* $x\ (n + m) \in F$

$\langle proof \rangle$

lemma *token-run-enter-final-states*:
  assumes $token\text{-}run\ x\ n \in F$
  shows $\exists\, m \geq x.\ token\text{-}run\ x\ m \notin F - \{q_0\} \wedge token\text{-}run\ x\ (Suc\ m) \in F$
$\langle proof \rangle$

6.2 Token Properties

6.2.1 Alternative Definitions

lemma *token-succeeds-alt-def*:
  $token\text{-}succeeds\ x = (\forall_\infty n.\ token\text{-}run\ x\ n \in F)$
  $\langle proof \rangle$

lemma *token-fails-alt-def*:
  $token\text{-}fails\ x = (\forall_\infty n.\ sink\ (token\text{-}run\ x\ n) \wedge token\text{-}run\ x\ n \notin F)$
  (**is** *?lhs = ?rhs*)
$\langle proof \rangle$

lemma *token-fails-alt-def-2*:
  $token\text{-}fails\ x \longleftrightarrow \neg token\text{-}succeeds\ x \wedge \neg token\text{-}squats\ x$
  $\langle proof \rangle$

6.2.2 Properties

lemma *token-succeeds-run-merge*:
  $x \leq n \Longrightarrow y \leq n \Longrightarrow token\text{-}run\ x\ n = token\text{-}run\ y\ n \Longrightarrow token\text{-}succeeds\ x \Longrightarrow token\text{-}succeeds\ y$
  $\langle proof \rangle$

lemma *token-squats-run-merge*:
  $x \leq n \Longrightarrow y \leq n \Longrightarrow token\text{-}run\ x\ n = token\text{-}run\ y\ n \Longrightarrow token\text{-}squats\ x \Longrightarrow token\text{-}squats\ y$
  $\langle proof \rangle$

6.2.3 Pulled-Up Lemmas

lemma *configuration-token-succeeds*:
  $[\![x \in configuration\ q\ n;\ y \in configuration\ q\ n]\!] \Longrightarrow token\text{-}succeeds\ x = token\text{-}succeeds\ y$
  $\langle proof \rangle$

lemma *configuration-token-squats*:

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

6.3 Mojmir Acceptance

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

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

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

6.4 Equivalent Acceptance Conditions

6.4.1 Token-Based Definitions

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

lemma *merge-subset*:
$i \le j \Longrightarrow \mathit{merge}\ i \subseteq \mathit{merge}\ j$
$\langle \mathit{proof} \rangle$

lemma *merge-finite*:
$i \le j \Longrightarrow \mathit{finite}\ (\mathit{merge}\ j) \Longrightarrow \mathit{finite}\ (\mathit{merge}\ i)$
$\langle \mathit{proof} \rangle$

lemma *merge-finite'*:
$i < j \Longrightarrow \mathit{finite}\ (\mathit{merge}\ j) \Longrightarrow \mathit{finite}\ (\mathit{merge}\ i)$
$\langle \mathit{proof} \rangle$

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

lemma *stable-rank-succeed*:

assumes *infinite* (*succeed* i)
and $x \in$ *succeed* i
and $q_0 \notin F$
shows $\neg$*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 $\exists\, i <$ *max-rank*. *finite fail* $\wedge$ *finite* (*merge* i) $\wedge$ *infinite* (*succeed* i) $\wedge$ ($\forall\, j < i$. *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 $\longleftrightarrow$ ($\exists\, i <$ *max-rank*. *finite fail* $\wedge$ *finite* (*merge* i) $\wedge$ *infinite* (*succeed* i))
⟨*proof*⟩

theorem *mojmir-accept-iff-token-set-accept2*:
accept $\longleftrightarrow$ ($\exists\, i <$ *max-rank*. *finite fail* $\wedge$ *finite* (*merge* i) $\wedge$ *infinite* (*succeed* i) $\wedge$ ($\forall\, j < i$. *finite* (*merge* j) $\wedge$ *finite* (*succeed* j)))
⟨*proof*⟩

6.4.2 Time-Based Definitions

lemma *finite-monotonic-image*:
fixes A B :: *nat set*
assumes $\bigwedge i.\ i \in A \Longrightarrow i \leq f\ i$
assumes f ‘ $A = B$
shows *finite* $A \longleftrightarrow$ *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-run* x n)
  assumes *sink* (*token-run* x (*Suc* n))
  assumes *token-run* x (*Suc* n) $\notin F$
  shows $n \in$ *fail-t*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

6.4.3 Relation to Mojmir Acceptance

lemma *token-iff-time-accept*:
shows $(finite\ fail \wedge finite\ (merge\ i) \wedge infinite\ (succeed\ i) \wedge (\forall j < i.\ finite\ (succeed\ j)))$
$= (finite\ fail\text{-}t \wedge finite\ (merge\text{-}t\ i) \wedge infinite\ (succeed\text{-}t\ i) \wedge (\forall j < i.\ finite\ (succeed\text{-}t\ j)))$
$\langle proof \rangle$

6.5 Succeeding Tokens (Alternative Definition)

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

lemma *stable-rank-at-ge*:
$n \leq m \Longrightarrow stable\text{-}rank\text{-}at\ x\ n \Longrightarrow stable\text{-}rank\text{-}at\ x\ m$
$\langle proof \rangle$

lemma *stable-rank-equiv*:
$(\exists i.\ stable\text{-}rank\ x\ i) = (\exists n.\ stable\text{-}rank\text{-}at\ x\ n)$
$\langle proof \rangle$

lemma *smallest-accepting-rank-properties*:
assumes $smallest\text{-}accepting\text{-}rank = Some\ i$
shows $accept\ finite\ fail\ finite\ (merge\ i)\ infinite\ (succeed\ i)\ \forall j < i.\ finite\ (succeed\ j)\ i < max\text{-}rank$
$\langle proof \rangle$

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

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

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

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

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

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

declare $\mathit{accepting\text{-}pair}_R\text{-}\mathit{def}[\mathit{simp}]$

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

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

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

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

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

lemma *accept$_{GR}$-simp2*:
$accept_{GR}\ (\delta,\ q_0,\ \alpha)\ w \longleftrightarrow (\exists P \in \alpha.\ accepting\text{-}pair_{GR}\ \delta\ q_0\ P\ w)$
⟨*proof*⟩

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

definition $LTS\text{-}is\text{-}inf\text{-}run :: ('q,'a)\ LTS \Rightarrow 'a\ word \Rightarrow 'q\ word \Rightarrow bool$
where
$LTS\text{-}is\text{-}inf\text{-}run\ \Delta\ w\ r \longleftrightarrow (\forall i.\ (r\ i,\ w\ i,\ r\ (Suc\ i)) \in \Delta)$

fun $accept_R\text{-}LTS :: (('a,\ 'b)\ LTS \times 'a \times ('a,\ 'b)\ rabin\text{-}condition) \Rightarrow 'b\ word \Rightarrow bool$
where
$accept_R\text{-}LTS\ (\delta,\ q_0,\ \alpha)\ w \longleftrightarrow (\exists (Fin,\ Inf) \in \alpha.\ \exists r.$
$LTS\text{-}is\text{-}inf\text{-}run\ \delta\ w\ r \wedge r\ 0 = q_0$
$\wedge\ limit\ (\lambda i.\ (r\ i,\ w\ i,\ r\ (Suc\ i))) \cap Fin = \{\}$
$\wedge\ limit\ (\lambda i.\ (r\ i,\ w\ i,\ r\ (Suc\ i))) \cap Inf \neq \{\})$

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

$\wedge$ *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* :: $(('a,\ 'b)\ LTS \times 'a \times ('a,\ 'b)\ generalized\text{-}rabin\text{-}condition) \Rightarrow 'b\ word \Rightarrow bool$
where
$accept_{GR}\text{-}LTS\ (\delta,\ q_0,\ \alpha)\ w = (\exists (Fin,\ Inf) \in \alpha.\ accepting\text{-}pair_{GR}\text{-}LTS\ \delta\ q_0\ (Fin,\ Inf)\ w)$

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

lemma $accept_{GR}$-*LTS-I*:
$accepting\text{-}pair_{GR}\text{-}LTS\ \delta\ q_0\ (F,\ \mathcal{I})\ w \Longrightarrow (F,\ \mathcal{I}) \in \alpha \Longrightarrow accept_{GR}\text{-}LTS\ (\delta,\ q_0,\ \alpha)\ w$
⟨*proof*⟩

lemma $accept_{GR}$-*I*:
$accepting\text{-}pair_{GR}\ \delta\ q_0\ (F, \mathcal{I})\ w \Longrightarrow (F, \mathcal{I}) \in \alpha \Longrightarrow accept_{GR}\ (\delta,\ q_0,\ \alpha)\ w$
⟨*proof*⟩

lemma *transfer-accept*:
$accepting\text{-}pair_R\ \delta\ q_0\ (F,\ I)\ w \longleftrightarrow accepting\text{-}pair_{GR}\ \delta\ q_0\ (F,\ \{I\})\ w$
$accept_R\ (\delta,\ q_0,\ \alpha)\ w \longleftrightarrow accept_{GR}\ (\delta,\ q_0,\ (\lambda(F,\ I).\ (F,\ \{I\}))\ `\ \alpha)\ w$
⟨*proof*⟩

7.1 Restriction Lemmas

lemma $accepting\text{-}pair_{GR}$-*restrict*:
assumes $range\ w \subseteq \Sigma$
shows $accepting\text{-}pair_{GR}\ \delta\ q_0\ (F,\ \mathcal{I})\ w = accepting\text{-}pair_{GR}\ \delta\ q_0\ (F \cap reach_t\ \Sigma\ \delta\ q_0,\ (\lambda I.\ I \cap reach_t\ \Sigma\ \delta\ q_0)\ `\ \mathcal{I})\ w$
⟨*proof*⟩

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

lemma $accepting\text{-}pair_R$-*restrict*:

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

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
$\langle proof \rangle$

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
$\langle proof \rangle$

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
$\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma $accept_{GR}$-*LTS*:
assumes *range* $w \subseteq \Sigma$
shows $accept_{GR}$ (δ, q_0, α) $w \longleftrightarrow$ $accept_{GR}$-*LTS* $(reach_t\ \Sigma\ \delta\ q_0, q_0, \alpha)$ w

$\langle proof \rangle$

lemma $accept_R$-*LTS*:
assumes *range* $w \subseteq \Sigma$
shows $accept_R$ (δ, q_0, α) $w \longleftrightarrow$ $accept_R$-*LTS* $(reach_t\ \Sigma\ \delta\ q_0, q_0, \alpha)$ w
$\langle proof \rangle$

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*$_{GR}$ δ q_0 $(\bigcup\{f\ x \mid x.\ x \in I\}, \{g\ x \mid x.\ x \in I\})$ w
$\langle proof \rangle$

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

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* $x \in Acc$ *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* $(Acc \cup set\ xs)$ *ys*)
$\langle proof \rangle$

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** $map\ f\ vs = zs$
$\langle proof \rangle$

lemma *filter-split'*:
$filter\ P\ xs = ys\ @\ zs \Longrightarrow \exists\, us\ vs.\ xs = us\ @\ vs \wedge filter\ P\ us = ys \wedge filter\ P\ vs = zs$
$\langle proof \rangle$

lemma *filter-splitE*:
assumes $filter\ P\ xs = ys\ @\ zs$
obtains $us\ vs$ **where** $xs = us\ @\ vs$ **and** $filter\ P\ us = ys$ **and** $filter\ P\ vs = zs$
$\langle proof \rangle$

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** $filter\ P\ (map\ f\ vs) = zs$
$\langle proof \rangle$

lemma *filter-map-split-iff*:
$filter\ P\ (map\ f\ xs) = ys\ @\ zs \longleftrightarrow (\exists\, us\ vs.\ xs = us\ @\ vs \wedge filter\ P\ (map\ f\ us) = ys \wedge filter\ P\ (map\ f\ vs) = zs)$
$\langle proof \rangle$

lemma *list-empty-prefix*:
$xs\ @\ y\ \#\ zs = y\ \#\ us \Longrightarrow y \notin set\ xs \Longrightarrow xs = []$
$\langle proof \rangle$

lemma *remdups-fwd-split*:
$remdups\text{-}fwd\text{-}acc\ Acc\ xs = ys\ @\ zs \Longrightarrow \exists\, us\ vs.\ xs = us\ @\ vs \wedge remdups\text{-}fwd\text{-}acc\ Acc\ us = ys \wedge remdups\text{-}fwd\text{-}acc\ (Acc \cup set\ ys)\ vs = zs$
$\langle proof \rangle$

lemma *remdups-fwd-split-exact*:
assumes $remdups\text{-}fwd\text{-}acc\ Acc\ xs = ys\ @\ x\ \#\ zs$
shows $\exists\, us\ vs.\ xs = us\ @\ x\ \#\ vs \wedge x \notin Acc \wedge x \notin set\ ys \wedge remdups\text{-}fwd\text{-}acc\ Acc\ us = ys \wedge remdups\text{-}fwd\text{-}acc\ (Acc \cup set\ ys \cup \{x\})\ vs = zs$
$\langle proof \rangle$

lemma *remdups-fwd-split-exactE*:
assumes $remdups\text{-}fwd\text{-}acc\ Acc\ xs = ys\ @\ x\ \#\ zs$

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

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

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

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

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

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

lemma *finite-set2*:
assumes $card\ A = n$ **and** *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*:
  $set\ (foldl\ (@)\ i\ xs) = set\ i \cup \bigcup\{set\ x \mid x.\ x \in set\ xs\}$
  ⟨*proof*⟩

8.3 Short-circuited Version of *foldl*

fun *foldl-break* :: $('b \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'b \Rightarrow 'a\ list \Rightarrow 'b$
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* ($\lambda zs.\ zs$ @ *ys*) (*suffixes xs*))
  ⟨*proof*⟩

lemma *suffixes-alt-def*:
  suffixes xs = *rev* (*prefix* (*length xs*) ($\lambda i.\ drop\ i\ xs$))
⟨*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* $\{f.\ (\forall x.\ x \notin R \longrightarrow f\ x = c) \land (\forall x.\ x \in R \longrightarrow f\ x \in A)\}$
(**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*) $\land$ *finite* (*merge i*) $\land$ *infinite* (*succeed i*))
    $\longleftrightarrow$ $accepting\text{-}pair_R\ \delta_{\mathcal{R}}\ q_{\mathcal{R}}\ (Acc_{\mathcal{R}}\ i)\ w$
  (**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

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

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

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* $\longleftrightarrow$ *?rhs*)
$\langle proof \rangle$

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

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* $\longleftrightarrow$ *?rhs*)
$\langle proof \rangle$

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

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

$\langle proof \rangle$

lemma *LTL-FG-suffix*:
$(suffix\ i\ w) \models F\ (G\ \varphi) = w \models F\ (G\ \varphi)$
$\langle proof \rangle$

lemma *LTL-GF-suffix*:
$(suffix\ i\ w) \models G\ (F\ \varphi) = w \models G\ (F\ \varphi)$
$\langle proof \rangle$

lemma *LTL-suffix-G*:
$w \models G\ \varphi \Longrightarrow suffix\ i\ w \models G\ \varphi$
$\langle proof \rangle$

lemma *LTL-prop-entailment-monotonI*[*intro*]:
$S \models_P \varphi \Longrightarrow S \subseteq S' \Longrightarrow S' \models_P \varphi$
$\langle proof \rangle$

lemma *ltl-models-equiv-prop-entailment*:
$w \models \varphi = \{\chi.\ w \models \chi\} \models_P \varphi$
$\langle proof \rangle$

10.2.2 Limit Behaviour of the G-operator

abbreviation *Only-G*
where
$Only\text{-}G\ S \equiv \forall\, x \in S.\ \exists\, y.\ x = G\ y$

lemma *ltl-G-stabilize*:
assumes *finite* $\mathcal{G}$
assumes *Only-G* $\mathcal{G}$
obtains i **where** $\bigwedge\chi\ j.\ \chi \in \mathcal{G} \Longrightarrow suffix\ i\ w \models \chi = suffix\ (i + j)\ w \models \chi$
$\langle proof \rangle$

lemma *ltl-G-stabilize-property*:
assumes *finite* $\mathcal{G}$
assumes *Only-G* $\mathcal{G}$
assumes $\bigwedge\chi\ j.\ \chi \in \mathcal{G} \Longrightarrow suffix\ i\ w \models \chi = suffix\ (i + j)\ w \models \chi$
assumes $G\ \psi \in \mathcal{G} \cap \{\chi.\ w \models F\ \chi\}$
shows $suffix\ i\ w \models G\ \psi$
$\langle proof \rangle$

10.3 Subformulae

10.3.1 Propositions

fun *propos* :: $'a\ ltl \Rightarrow 'a\ ltl\ set$
where
  $propos\ true = \{\}$
| $propos\ false = \{\}$
| $propos\ (\varphi\ and\ \psi) = propos\ \varphi \cup propos\ \psi$
| $propos\ (\varphi\ or\ \psi) = propos\ \varphi \cup propos\ \psi$
| $propos\ \varphi = \{\varphi\}$

fun *nested-propos* :: $'a\ ltl \Rightarrow 'a\ ltl\ set$
where
  $nested\text{-}propos\ true = \{\}$
| $nested\text{-}propos\ false = \{\}$
| $nested\text{-}propos\ (\varphi\ and\ \psi) = nested\text{-}propos\ \varphi \cup nested\text{-}propos\ \psi$
| $nested\text{-}propos\ (\varphi\ or\ \psi) = nested\text{-}propos\ \varphi \cup nested\text{-}propos\ \psi$
| $nested\text{-}propos\ (F\ \varphi) = \{F\ \varphi\} \cup nested\text{-}propos\ \varphi$
| $nested\text{-}propos\ (G\ \varphi) = \{G\ \varphi\} \cup nested\text{-}propos\ \varphi$
| $nested\text{-}propos\ (X\ \varphi) = \{X\ \varphi\} \cup nested\text{-}propos\ \varphi$
| $nested\text{-}propos\ (\varphi\ U\ \psi) = \{\varphi\ U\ \psi\} \cup nested\text{-}propos\ \varphi \cup nested\text{-}propos\ \psi$
| $nested\text{-}propos\ \varphi = \{\varphi\}$

lemma *finite-propos*:
  $finite\ (propos\ \varphi)\ finite\ (nested\text{-}propos\ \varphi)$
  ⟨*proof*⟩

lemma *propos-subset*:
  $propos\ \varphi \subseteq nested\text{-}propos\ \varphi$
  ⟨*proof*⟩

lemma *LTL-prop-entailment-restrict-to-propos*:
  $S \models_P \varphi = (S \cap propos\ \varphi) \models_P \varphi$
  ⟨*proof*⟩

lemma *propos-foldl*:
  assumes $\bigwedge x\ y.\ propos\ (f\ x\ y) = propos\ x \cup propos\ y$
  shows $\bigcup \{propos\ y\ | y.\ y = i \vee y \in set\ xs\} = propos\ (foldl\ f\ i\ xs)$
⟨*proof*⟩

10.3.2 G-Subformulae

Notation for paper: mathdsG

fun *G-nested-propos* :: $'a\ ltl \Rightarrow 'a\ ltl\ set$ (**G**)
where
$\mathbf{G}\ (\varphi\ \mathit{and}\ \psi) = \mathbf{G}\ \varphi \cup \mathbf{G}\ \psi$
$|\ \mathbf{G}\ (\varphi\ \mathit{or}\ \psi) = \mathbf{G}\ \varphi \cup \mathbf{G}\ \psi$
$|\ \mathbf{G}\ (F\ \varphi) = \mathbf{G}\ \varphi$
$|\ \mathbf{G}\ (G\ \varphi) = \mathbf{G}\ \varphi \cup \{G\ \varphi\}$
$|\ \mathbf{G}\ (X\ \varphi) = \mathbf{G}\ \varphi$
$|\ \mathbf{G}\ (\varphi\ U\ \psi) = \mathbf{G}\ \varphi \cup \mathbf{G}\ \psi$
$|\ \mathbf{G}\ \varphi = \{\}$

lemma *G-nested-finite*:
finite ($\mathbf{G}\ \varphi$)
$\langle proof \rangle$

lemma *G-nested-propos-alt-def*:
$\mathbf{G}\ \varphi = \mathit{nested\text{-}propos}\ \varphi \cap \{\psi.\ (\exists x.\ \psi = G\ x)\}$
$\langle proof \rangle$

lemma *G-nested-propos-Only-G*:
Only-G ($\mathbf{G}\ \varphi$)
$\langle proof \rangle$

lemma *G-not-in-G*:
$G\ \varphi \notin \mathbf{G}\ \varphi$
$\langle proof \rangle$

lemma *G-subset-G*:
$\psi \in \mathbf{G}\ \varphi \Longrightarrow \mathbf{G}\ \psi \subseteq \mathbf{G}\ \varphi$
$G\ \psi \in \mathbf{G}\ \varphi \Longrightarrow \mathbf{G}\ \psi \subseteq \mathbf{G}\ \varphi$
$\langle proof \rangle$

lemma $\mathcal{G}$*-properties*:
assumes $\mathcal{G} \subseteq \mathbf{G}\ \varphi$
shows $\mathcal{G}$*-finite*: *finite* $\mathcal{G}$ **and** $\mathcal{G}$*-elements*: *Only-G* $\mathcal{G}$
$\langle proof \rangle$

10.4 Propositional Implication and Equivalence

definition *ltl-prop-implies* :: $['a\ ltl,\ 'a\ ltl] \Rightarrow bool$ (**infix** $\longrightarrow_P$ *75*)
where
$\varphi \longrightarrow_P \psi \equiv \forall \mathcal{A}.\ \mathcal{A} \models_P \varphi \longrightarrow \mathcal{A} \models_P \psi$

definition *ltl-prop-equiv* :: $['a\ ltl,\ 'a\ ltl] \Rightarrow bool$ (**infix** $\equiv_P$ *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*:
$\varphi \longrightarrow_P \psi \land \psi \longrightarrow_P \varphi \longleftrightarrow \varphi \equiv_P \psi$
$\langle proof \rangle$

lemma *ltl-prop-equiv-equivp*:
$equivp\ (\equiv_P)$
$\langle proof \rangle$

lemma [*trans*]:
$\varphi \equiv_P \psi \Longrightarrow \psi \equiv_P \chi \Longrightarrow \varphi \equiv_P \chi$
$\langle proof \rangle$

10.4.1 Quotient Type for Propositional Equivalence

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

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* :: $'a\ ltl_P \Rightarrow {}'a\ ltl_P \Rightarrow bool$ **is**
$\lambda x\ y.\ x \equiv_P y$
$\langle proof \rangle$

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

instance
$\langle proof \rangle$

end

lemma ltl_P*-abs-rep*: $Abs\ (Rep\ \varphi) = \varphi$
$\langle proof \rangle$

lift-definition *ltl-prop-entails-abs* :: $'a\ ltl\ set \Rightarrow {}'a\ ltl_P \Rightarrow bool$ (- $\uparrow\models_P$ -)
is $(\models_P)$
$\langle proof \rangle$

lift-definition *ltl-prop-implies-abs* :: $'a\ ltl_P \Rightarrow {}'a\ ltl_P \Rightarrow bool$ (- $\uparrow\longrightarrow_P$ -)

is $(\longrightarrow_P)$
$\langle proof \rangle$

10.4.2 Propositional Equivalence implies LTL Equivalence

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

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

10.5 Substitution

fun $subst :: 'a\ ltl \Rightarrow ('a\ ltl \rightharpoonup 'a\ ltl) \Rightarrow 'a\ ltl$
where
$subst\ true\ m = true$
$|\ subst\ false\ m = false$
$|\ subst\ (\varphi\ and\ \psi)\ m = subst\ \varphi\ m\ and\ subst\ \psi\ m$
$|\ subst\ (\varphi\ or\ \psi)\ m = subst\ \varphi\ m\ or\ subst\ \psi\ m$
$|\ subst\ \varphi\ m = (case\ m\ \varphi\ of\ Some\ \varphi' \Rightarrow \varphi'\ |\ None \Rightarrow \varphi)$

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

lemma *ltl-prop-equiv-subst-S*:
$S \models_P subst\ \varphi\ m = ((S - dom\ m) \cup \{\chi\ |\ \chi\ \chi'.\ \chi \in dom\ m \land m\ \chi = Some\ \chi' \land S \models_P \chi'\}) \models_P \varphi$
$\langle proof \rangle$

lemma *subst-respects-ltl-prop-entailment*:
$\varphi \longrightarrow_P \psi \Longrightarrow subst\ \varphi\ m \longrightarrow_P subst\ \psi\ m$
$\varphi \equiv_P \psi \Longrightarrow subst\ \varphi\ m \equiv_P subst\ \psi\ m$
$\langle proof \rangle$

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

lemma *decomposable-function-subst*:
$[\![f\ true = true;\ f\ false = false;\ \bigwedge\varphi\ \psi.\ f\ (\varphi\ and\ \psi) = f\ \varphi\ and\ f\ \psi;\ \bigwedge\varphi\ \psi.\ f\ (\varphi\ or\ \psi) = f\ \varphi\ or\ f\ \psi]\!] \Longrightarrow f\ \varphi = subst\ \varphi\ (\lambda\chi.\ Some\ (f\ \chi))$
$\langle proof \rangle$

10.6 Additional Operators

10.6.1 And

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

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

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

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

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

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

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

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

10.6.2 Lifted Variant

lift-definition *and-abs* :: $'a\ ltl_P \Rightarrow 'a\ ltl_P \Rightarrow 'a\ ltl_P$ (- $\uparrow and$ -) **is** $\lambda x\ y.\ x\ and\ y$
  $\langle proof \rangle$

fun *And-abs* :: $'a\ ltl_P\ list \Rightarrow 'a\ ltl_P$ ($\uparrow And$)
where

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

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

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

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

10.6.3 Or

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

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

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

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

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

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

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

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

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*:
$eval_G\ S\ (And\ xs) = And\ (map\ (eval_G\ S)\ xs)$
$\langle proof \rangle$

lemma $eval_G$*-Or-map*:
$eval_G\ S\ (Or\ xs) = Or\ (map\ (eval_G\ S)\ xs)$
$\langle proof \rangle$

lemma $eval_G$*-G-nested*:
$\mathbf{G}\ (eval_G\ \mathcal{G}\ \varphi) \subseteq \mathbf{G}\ \varphi$
$\langle proof \rangle$

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

lemma $eval_G$*-prop-entailment*:
$S \models_P eval_G\ S\ \varphi \longleftrightarrow S \models_P \varphi$
$\langle proof \rangle$

lemma $eval_G$*-respectfulness*:
$\varphi \longrightarrow_P \psi \Longrightarrow eval_G\ S\ \varphi \longrightarrow_P eval_G\ S\ \psi$
$\varphi \equiv_P \psi \Longrightarrow eval_G\ S\ \varphi \equiv_P eval_G\ S\ \psi$
$\langle proof \rangle$

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

$\langle proof \rangle$

lift-definition $eval_G\text{-}abs :: {'a}\ ltl\ set \Rightarrow {'a}\ ltl_P \Rightarrow {'a}\ ltl_P$ $(\uparrow eval_G)$ **is** $eval_G$
$\langle proof \rangle$

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*]:
$A \models_P Rep\ (Abs\ \varphi) = A \models_P \varphi$
$\langle proof \rangle$

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

lemma *sat-models-invariant*:
$A \in sat\text{-}models\ (Abs\ \varphi) = A \models_P \varphi$
$\langle proof \rangle$

lemma *sat-models-inj*:
inj sat-models
$\langle proof \rangle$

lemma *sat-models-finite-image*:
assumes *finite P*
shows $finite\ (sat\text{-}models\ `\ \{Abs\ \varphi \mid \varphi.\ nested\text{-}propos\ \varphi \subseteq P\})$
$\langle proof \rangle$

lemma *ltl-prop-equiv-quotient-restricted-to-P-finite*:
assumes *finite P*
shows $finite\ \{Abs\ \varphi \mid \varphi.\ nested\text{-}propos\ \varphi \subseteq P\}$
$\langle proof \rangle$

locale *lift-ltl-transformer* =
fixes
$f :: {'a}\ ltl \Rightarrow {'b} \Rightarrow {'a}\ ltl$
assumes
respectfulness: $\varphi \equiv_P \psi \Longrightarrow f\ \varphi\ \nu \equiv_P f\ \psi\ \nu$
assumes
nested-propos-bounded: $nested\text{-}propos\ (f\ \varphi\ \nu) \subseteq nested\text{-}propos\ \varphi$
begin

lift-definition *f-abs* :: $'a\ ltl_P \Rightarrow 'b \Rightarrow 'a\ ltl_P$ **is** *f*
⟨*proof*⟩

lift-definition *f-foldl-abs* :: $'a\ ltl_P \Rightarrow 'b\ list \Rightarrow 'a\ ltl_P$ **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* $\Rightarrow$ *'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* $\Rightarrow$ *'a set* $\Rightarrow$ *'a ltl*
where
af-letter true ν = *true*
| *af-letter false* ν = *false*
| *af-letter p*(*a*) ν = (*if a* $\in \nu$ *then true else false*)
| *af-letter* (*np*(*a*)) ν = (*if a* $\notin \nu$ *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 :: $'a\ ltl \Rightarrow 'a\ set\ list \Rightarrow 'a\ ltl$ (af)
where
$af\ \varphi\ w \equiv foldl\ af\text{-}letter\ \varphi\ w$

lemma *af-decompose*:
$af\ (\varphi\ and\ \psi)\ w = (af\ \varphi\ w)\ and\ (af\ \psi\ w)$
$af\ (\varphi\ or\ \psi)\ w = (af\ \varphi\ w)\ or\ (af\ \psi\ w)$
⟨*proof*⟩

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

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

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

lemma *af-U*:
$af\ (\varphi\ U\ \psi)\ (x\#xs) = (af\ (\varphi\ U\ \psi)\ xs\ and\ af\ \varphi\ (x\#xs))\ or\ af\ \psi\ (x\#xs)$
⟨*proof*⟩

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

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

lemma *af-nested-propos*:
$nested\text{-}propos\ (af\text{-}letter\ \varphi\ \nu) \subseteq nested\text{-}propos\ \varphi$
⟨*proof*⟩

11.2 af_G

fun *af-G-letter* :: $'a$ *ltl* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *ltl*
where
af-G-letter true ν = *true*
| *af-G-letter false* ν = *false*
| *af-G-letter* $p(a)$ ν = (*if* $a \in \nu$ *then true else false*)
| *af-G-letter* $(np(a))$ ν = (*if* $a \notin \nu$ *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* $\Rightarrow$ $'a$ *set list* $\Rightarrow$ $'a$ *ltl*
where
af_G φ w $\equiv$ (*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 $\rightarrow$ *Suc* n]) = (af_G (φ U ψ) (w [1 $\rightarrow$ *Suc* n]) *and* af_G φ (w [0 $\rightarrow$ *Suc* n])) *or* af_G ψ (w [0 $\rightarrow$ *Suc* n])

$\langle proof \rangle$

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* $\psi\ \nu$
$\langle proof \rangle$

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

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

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

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

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

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

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

11.3 G-Subformulae Simplification

lemma *G-af-simp*[*simp*]:
$\mathbf{G}\ (af\ \varphi\ w) = \mathbf{G}\ \varphi$
$\langle proof \rangle$

lemma *G-af$_G$-simp*[*simp*]:
$\mathbf{G}\ (af_G\ \varphi\ w) = \mathbf{G}\ \varphi$
⟨*proof*⟩

11.4 Relation between af and af_G

lemma *af-G-letter-free-F*:
$\mathbf{G}\ \varphi = \{\} \Longrightarrow \mathbf{G}\ (\textit{af-letter}\ \varphi\ \nu) = \{\}$
$\mathbf{G}\ \varphi = \{\} \Longrightarrow \mathbf{G}\ (\textit{af-G-letter}\ \varphi\ \nu) = \{\}$
⟨*proof*⟩

lemma *af-G-free*:
assumes $\mathbf{G}\ \varphi = \{\}$
shows $af\ \varphi\ w = af_G\ \varphi\ w$
⟨*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$_G$*:
$S \models_P af\ \varphi\ w \Longrightarrow S \models_P af_G\ \varphi\ w$
⟨*proof*⟩

lemma *af-implies-af$_G$-2*:
$w \models af\ \varphi\ xs \Longrightarrow w \models af_G\ \varphi\ xs$
⟨*proof*⟩

lemma *af$_G$-implies-af-eval$_G$'*:
assumes $S \models_P eval_G\ \mathcal{G}\ (af_G\ \varphi\ w)$
assumes $\bigwedge\psi.\ G\ \psi \in \mathcal{G} \Longrightarrow S \models_P G\ \psi$
assumes $\bigwedge\psi\ i.\ G\ \psi \in \mathcal{G} \Longrightarrow i < \textit{length}\ w \Longrightarrow S \models_P eval_G\ \mathcal{G}\ (af_G\ \psi$
$(\textit{drop}\ i\ w))$
shows $S \models_P af\ \varphi\ w$
⟨*proof*⟩

lemma *af$_G$-implies-af-eval$_G$*:
assumes $S \models_P eval_G\ \mathcal{G}\ (af_G\ \varphi\ (w\ [0 \rightarrow j]))$
assumes $\bigwedge\psi.\ G\ \psi \in \mathcal{G} \Longrightarrow S \models_P G\ \psi$
assumes $\bigwedge\psi\ i.\ G\ \psi \in \mathcal{G} \Longrightarrow i \le j \Longrightarrow S \models_P eval_G\ \mathcal{G}\ (af_G\ \psi\ (w\ [i \rightarrow$

$j]))$
shows $S \models_P af\ \varphi\ (w\ [0 \to j])$
$\langle proof \rangle$

11.5 Continuation

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

lemma *af-ltl-continuation-suffix*:
$w \models \varphi \longleftrightarrow suffix\ i\ w \models af\ \varphi\ (w[0 \to i])$
$\langle proof \rangle$

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

lemma af_G*-ltl-continuation-suffix*:
$\forall \psi \in \mathbf{G}\ \varphi.\ w \models \psi = (suffix\ i\ w) \models \psi \Longrightarrow w \models \varphi \longleftrightarrow suffix\ i\ w \models af_G$
$\varphi\ (w\ [0 \to i])$
$\langle proof \rangle$

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* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *ltl*
where
  step $p(a)$ ν = (*if* $a \in \nu$ *then true else false*)
| *step* ($np(a)$) ν = (*if* $a \notin \nu$ *then true else false*)
| *step* ($X\ \varphi$) ν = φ
| *step* (φ *and* ψ) ν = *step* $\varphi\ \nu$ *and step* $\psi\ \nu$
| *step* (φ *or* ψ) ν = *step* $\varphi\ \nu$ *or step* $\psi\ \nu$
| *step* $\varphi\ \nu = \varphi$

fun *af-letter-opt*
where
  af-letter-opt $\varphi\ \nu$ = *Unf* (*step* $\varphi\ \nu$)

fun *af-G-letter-opt*
where
  af-G-letter-opt $\varphi\ \nu$ = Unf_G (*step* $\varphi\ \nu$)

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

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

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

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

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

lemma *af-equiv'*:
  af φ (w [$0 \rightarrow$ *Suc* i]) = *step* ($af_{\mathfrak{U}}$ (*Unf* φ) (w [$0 \rightarrow i$])) ($w\ i$)
  ⟨*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*:
$Only\text{-}G\ \mathcal{G} \Longrightarrow \mathcal{G} \models_P \varphi \Longrightarrow \mathcal{G} \models_P af\text{-}G\text{-}letter\text{-}opt\ \varphi\ \nu$
⟨*proof*⟩

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

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

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

lemma *Rep-Abs-equiv*:
$Rep\ (Abs\ \varphi) \equiv_P \varphi$
$\langle proof \rangle$

lemma *Rep-step*:
$Rep\ (\uparrow step\ \Phi\ \nu) \equiv_P step\ (Rep\ \Phi)\ \nu$
$\langle proof \rangle$

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

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

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 $\mathcal{G}_{FG}$ :: $'a\ ltl \Rightarrow 'a\ set\ word \Rightarrow 'a\ ltl\ set$
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*:
$\mathcal{G}_{FG}\ \varphi\ w = \{G\ \psi \mid \psi.\ G\ \psi \in \mathbf{G}\ \varphi \wedge w \models F\ (G\ \psi)\}$
$\langle proof \rangle$

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

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

12.2 Eventually Provable and Almost All Eventually Provable

abbreviation $\mathfrak{P}$
where
$\mathfrak{P}\ \varphi\ \mathcal{G}\ w\ i \equiv \exists j.\ \mathcal{G} \models_P \mathit{af}_G\ \varphi\ (w\ [i \to j])$

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*]:
$\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \Longrightarrow \mathcal{G} \subseteq \mathcal{G}' \Longrightarrow \mathfrak{P}_\infty\ \varphi\ \mathcal{G}'\ w$
$\langle proof \rangle$

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

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

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

lemma *threshold-suffix*:
assumes $\textit{threshold}\ \varphi\ w\ \mathcal{G} = \textit{Some}\ k$
assumes $\textit{threshold}\ \varphi\ (\textit{suffix}\ i\ w)\ \mathcal{G} = \textit{Some}\ k'$
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

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

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

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 $Only\text{-}G\ S$
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 $threshold\ \psi\ (suffix\ l\ w)\ \mathcal{G}' = Some\ 0$
⟨*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
  $accept_M\ \varphi\ \mathcal{G}\ w \equiv (\forall_\infty j.\ \forall\, S.\ (\forall\, \psi.\ G\ \psi \in \mathcal{G} \longrightarrow S \models_P G\ \psi \wedge S \models_P eval_G\ \mathcal{G}\ (\mathcal{F}\ \psi\ w\ \mathcal{G}\ j)) \longrightarrow S \models_P af\ \varphi\ (w\ [0 \to j]))$

lemma *lemmaD*:
  assumes $w \models \varphi$
  assumes $\bigwedge\psi.\ G\ \psi \in \mathcal{G}_{FG}\ \varphi\ w \Longrightarrow$ *threshold* $\psi\ w\ (\mathcal{G}_{FG}\ \varphi\ w) =$ *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* ⟷ *?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* ⟷ *?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*:
  $run \uparrow af_G\ (Abs\ \varphi)\ w\ i = Abs\ (run\ af\text{-}G\text{-}letter\ \varphi\ w\ i)$
  $\langle proof \rangle$

lemma *finite-reach-af*:
  $finite\ (reach\ \Sigma \uparrow af\ (Abs\ \varphi))$
$\langle proof \rangle$

lemma *ltl-semi-mojmir*:
  assumes $finite\ \Sigma$
  assumes $range\ w \subseteq \Sigma$
  shows $semi\text{-}mojmir\ \Sigma \uparrow af_G\ (Abs\ \psi)\ w$
$\langle proof \rangle$

13.2 Single Secondary Automaton

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

sublocale $mojmir\text{-}to\text{-}rabin\text{-}def\ \Sigma \uparrow af_G\ Abs\ \varphi\ w\ \{q.\ \mathcal{G} \models_P Rep\ q\}$ $\langle proof \rangle$
abbreviation $\delta_R \equiv step$
abbreviation $q_R \equiv initial$
abbreviation $Acc_R\ j \equiv (fail_R \cup merge_R\ j,\ succeed_R\ j)$
abbreviation $max\text{-}rank_R \equiv max\text{-}rank$
abbreviation $smallest\text{-}accepting\text{-}rank_R \equiv smallest\text{-}accepting\text{-}rank$
abbreviation $accept_R{}' \equiv accept$
abbreviation $\mathcal{S}_R \equiv \mathcal{S}$

lemma *Rep-token-run-af*:
  $Rep\ (token\text{-}run\ x\ n) \equiv_P af_G\ \varphi\ (w\ [x \to n])$
$\langle proof \rangle$

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*:
$\mathfrak{P}_\infty\ \varphi\ \mathcal{G}\ w \longleftrightarrow accept_R\ (\delta_R,\ q_R,\ \{Acc_R\ i \mid i.\ i < max\text{-}rank_R\})\ w$
⟨*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*.δ_R Σ φ, *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*.$max\text{-}rank_R$ $\Sigma\ \varphi\})$

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

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

$\langle proof \rangle$

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 $\forall_\infty j.\ (\forall S.\ (S \models_P \mathcal{F}\ \psi\ w\ \mathcal{G}\ j \wedge \mathcal{G} \subseteq S) \longleftrightarrow (\forall q.\ q \in (\textit{ltl-FG-to-rabin-def}.\mathcal{S}_R$ $\Sigma\ \psi\ \mathcal{G}\ w\ j) \longrightarrow S \models_P \textit{Rep}\ q))$
$\langle proof \rangle$

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 $\forall_\infty j.\ \forall \psi.\ G\ \psi \in \mathcal{G} \longrightarrow (\forall S.\ (S \models_P \mathcal{F}\ \psi\ w\ \mathcal{G}\ j \wedge \mathcal{G} \subseteq S) \longleftrightarrow$ $(\forall q.\ q \in (\textit{ltl-FG-to-rabin-def}.\mathcal{S}_R\ \Sigma\ \psi\ \mathcal{G})\ w\ j \longrightarrow S \models_P \textit{Rep}\ q))$
$\langle proof \rangle$

13.3 Product of Secondary Automata

context
  fixes
    Σ :: *'a set set*
begin

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

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

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

where
  combine-pairs' $P = (\bigcup (fst \text{ ` } P),\ snd \text{ ` } P)$

lemma *combine-pairs-prop*:
  $(\forall P \in \mathcal{P}.\ accepting\text{-}pair_R\ \delta\ q_0\ P\ w) = accepting\text{-}pair_{GR}\ \delta\ q_0\ (combine\text{-}pairs\ \mathcal{P})\ w$
  $\langle proof \rangle$

lemma *combine-pairs2*:
  $combine\text{-}pairs\ \mathcal{P} \in \alpha \Longrightarrow (\bigwedge P.\ P \in \mathcal{P} \Longrightarrow accepting\text{-}pair_R\ \delta\ q_0\ P\ w\ ) \Longrightarrow accept_{GR}\ (\delta,\ q_0,\ \alpha)\ w$
  $\langle proof \rangle$

lemma *combine-pairs'-prop*:
  $(\forall P \in \mathcal{P}.\ accepting\text{-}pair_R\ \delta\ q_0\ P\ w) = accepting\text{-}pair_{GR}\ \delta\ q_0\ (combine\text{-}pairs'\ \mathcal{P})\ w$
  $\langle proof \rangle$

fun *ltl-FG-to-generalized-rabin* :: $'a\ ltl \Rightarrow ('a\ ltl \rightharpoonup 'a\ ltl_P \rightharpoonup nat,\ 'a\ set)$ *generalized-rabin-automaton* $(\mathcal{P})$
where
  $ltl\text{-}FG\text{-}to\text{-}generalized\text{-}rabin\ \varphi = ($
    $\Delta_\times\ (\lambda\chi.\ ltl\text{-}FG\text{-}to\text{-}rabin\text{-}def.\delta_R\ \Sigma\ (theG\ \chi)),$
    $\iota_\times\ (\mathbf{G}\ (G\ \varphi))\ (\lambda\chi.\ ltl\text{-}FG\text{-}to\text{-}rabin\text{-}def.q_R\ (theG\ \chi)),$
    $\{combine\text{-}pairs'\ \{embed\text{-}pair\ \chi\ (ltl\text{-}FG\text{-}to\text{-}rabin\text{-}def.Acc_R\ \Sigma\ (theG\ \chi)\ \mathcal{G}\ (\pi\ \chi))\ |\ \chi.\ \chi \in \mathcal{G}\}$
    $|\ \mathcal{G}\ \pi.\ \mathcal{G} \subseteq \mathbf{G}\ (G\ \varphi) \wedge G\ \varphi \in \mathcal{G} \wedge (\forall\chi.\ \pi\ \chi < ltl\text{-}FG\text{-}to\text{-}rabin\text{-}def.max\text{-}rank_R\ \Sigma\ (theG\ \chi))\})$

context
  assumes
    finite-Σ: *finite* Σ
begin

lemma *ltl-FG-to-generalized-rabin-wellformed*:
  $finite\ (reach\ \Sigma\ (fst\ (\mathcal{P}\ \varphi))\ (fst\ (snd\ (\mathcal{P}\ \varphi))))$
$\langle proof \rangle$

theorem *ltl-FG-to-generalized-rabin-correct*:
  assumes $range\ w \subseteq \Sigma$
  shows $w \models F\ G\ \varphi = accept_{GR}\ (\mathcal{P}\ \varphi)\ w$
  (**is** *?lhs* = *?rhs*)
$\langle proof \rangle$

end

end

13.4 Automaton Template

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

— Transition Function and Initial State

fun *delta*
where
  $delta\ \Sigma = \delta \times \Delta_\times\ (semi\text{-}mojmir\text{-}def.step\ \Sigma\ \delta_M\ o\ q_{0M}\ o\ theG)$

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

— Acceptance Condition

definition *max-rank-of*
where
  $max\text{-}rank\text{-}of\ \Sigma\ \psi \equiv semi\text{-}mojmir\text{-}def.max\text{-}rank\ \Sigma\ \delta_M\ (q_{0M}\ (theG\ \psi))$

fun *Acc-fin*
where
  $Acc\text{-}fin\ \Sigma\ \pi\ \chi = \bigcup (embed\text{-}transition\text{-}snd\ `\ \bigcup (embed\text{-}transition\ \chi\ `$
    $(mojmir\text{-}to\text{-}rabin\text{-}def.fail_R\ \Sigma\ \delta_M\ (q_{0M}\ (theG\ \chi))\ \{q.\ dom\ \pi \uparrow\models_P q\}$
    $\cup\ mojmir\text{-}to\text{-}rabin\text{-}def.merge_R\ \delta_M\ (q_{0M}\ (theG\ \chi))\ \{q.\ dom\ \pi \uparrow\models_P q\}$
$(the\ (\pi\ \chi)))))$

fun *Acc-inf*

where
  Acc-inf π χ = $\bigcup$(*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 χ $\equiv$ (*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* π $\cup$ $\bigcup${*Acc-fin* Σ π χ | χ. $\chi \in$ *dom* π}, {*Acc-inf* π χ | χ. $\chi \in$ *dom* π})
    | π. *dom* $\pi \subseteq$ **G** φ $\wedge$ ($\forall\chi \in$ *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* π' $\Longrightarrow$ ($\bigwedge\chi$. $\chi \in$ *dom* π $\Longrightarrow$ *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*) $\neq$ {}
⟨*proof*⟩

lemma *run-properties*:
fixes φ
defines $r \equiv$ *run* (*delta* Σ) (*initial* φ) *w*
shows *fst* (*r i*) = *foldl* δ (q_0 φ) (*w* [*0* $\to$ *i*])
and $\bigwedge\chi$ *q*. $\chi \in$ **G** $\varphi \Longrightarrow$ *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 $accept_{GR}$ ($\mathcal{A}$ Σ φ) *w*
obtains π **where** *dom* $\pi \subseteq$ **G** φ
and $\bigwedge\chi$. $\chi \in$ *dom* $\pi \Longrightarrow$ *the* (π χ) < *max-rank-of* Σ χ
and *accepting-pair*$_R$ (*delta* Σ) (*initial* φ) (*M-fin* π, *UNIV*) *w*
and $\bigwedge\chi$. $\chi \in$ *dom* $\pi \Longrightarrow$ *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* $\rightharpoonup$ *nat*
assumes
G $\psi \in$ *dom* π
assumes
dom $\pi \subseteq$ **G** φ
begin

interpretation $\mathfrak{M}$: *mojmir-to-rabin* Σ δ_M q_{0M} ψ *w* {*q*. *dom* $\pi \uparrow\models_P q$}

$\langle proof \rangle$

lemma *Acc-property*:
$accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi\ (G\ \psi))\ w \longleftrightarrow accepting\text{-}pair_R\ \mathfrak{M}.\delta_{\mathcal{R}}\ \mathfrak{M}.q_{\mathcal{R}}\ (\mathfrak{M}.Acc_{\mathcal{R}}\ (the\ (\pi\ (G\ \psi))))\ w$
(**is** $?Acc = ?Acc_{\mathcal{R}}$)
$\langle proof \rangle$

lemma *Acc-to-rabin-accept*:
$[\![accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi\ (G\ \psi))\ w;\ the\ (\pi\ (G\ \psi)) < \mathfrak{M}.max\text{-}rank]\!] \Longrightarrow accept_R\ \mathfrak{M}.\mathcal{R}\ w$
$\langle proof \rangle$

lemma *Acc-to-mojmir-accept*:
$[\![accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi\ (G\ \psi))\ w;\ the\ (\pi\ (G\ \psi)) < \mathfrak{M}.max\text{-}rank]\!] \Longrightarrow \mathfrak{M}.accept$
$\langle proof \rangle$

lemma *rabin-accept-to-Acc*:
$[\![accept_R\ \mathfrak{M}.\mathcal{R}\ w;\ \pi\ (G\ \psi) = \mathfrak{M}.smallest\text{-}accepting\text{-}rank]\!] \Longrightarrow accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi\ (G\ \psi))\ w$
$\langle proof \rangle$

lemma *mojmir-accept-to-Acc*:
$[\![\mathfrak{M}.accept;\ \pi\ (G\ \psi) = \mathfrak{M}.smallest\text{-}accepting\text{-}rank]\!] \Longrightarrow accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi\ (G\ \psi))\ w$
$\langle proof \rangle$

end

lemma *normalize-π*:
assumes *dom-subset*: $dom\ \pi \subseteq \mathbf{G}\ \varphi$
assumes $\bigwedge\chi.\ \chi \in dom\ \pi \Longrightarrow the\ (\pi\ \chi) < max\text{-}rank\text{-}of\ \Sigma\ \chi$
assumes $accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (M\text{-}fin\ \pi,\ UNIV)\ w$
assumes $\bigwedge\chi.\ \chi \in dom\ \pi \Longrightarrow accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi\ \chi)\ w$
obtains $\pi_{\mathcal{A}}$ **where** $dom\ \pi = dom\ \pi_{\mathcal{A}}$
and $\bigwedge\chi.\ \chi \in dom\ \pi_{\mathcal{A}} \Longrightarrow \pi_{\mathcal{A}}\ \chi = mojmir\text{-}def.smallest\text{-}accepting\text{-}rank\ \Sigma\ \delta_M\ (q_{0M}\ (theG\ \chi))\ w\ \{q.\ dom\ \pi_{\mathcal{A}} \uparrow\models_P q\}$
and $accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (M\text{-}fin\ \pi_{\mathcal{A}},\ UNIV)\ w$
and $\bigwedge\chi.\ \chi \in dom\ \pi_{\mathcal{A}} \Longrightarrow accepting\text{-}pair_R\ (delta\ \Sigma)\ (initial\ \varphi)\ (Acc\ \Sigma\ \pi_{\mathcal{A}}\ \chi)\ w$
$\langle proof \rangle$

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
 $M\text{-}fin\ \pi = \{((\varphi', m), \nu, p).$
  $\neg(\forall S.\ (\forall \chi \in dom\ \pi.\ S \uparrow\models_P Abs\ \chi \wedge (\forall q.\ (\exists j \geq the\ (\pi\ \chi).\ the\ (m\ \chi)\ q = Some\ j) \longrightarrow S \uparrow\models_P \uparrow eval_G\ (dom\ \pi)\ q)) \longrightarrow S \uparrow\models_P \varphi')\}$

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 $F_{\mathcal{A}} \equiv rabin\text{-}pairs$
abbreviation $\mathcal{A} \equiv ltl\text{-}to\text{-}generalized\text{-}rabin$

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-to-generalized-rabin-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*:
  $\mathit{finite}\ (\mathit{reach}\ \Sigma\ \uparrow\!\mathit{af}_{\mathfrak{U}}\ (\mathit{Abs}\ \varphi))$
⟨*proof*⟩

lemma *finite-reach-af-G-opt*:
  $\mathit{finite}\ (\mathit{reach}\ \Sigma\ \uparrow\!\mathit{af}_{G\mathfrak{U}}\ (\mathit{Abs}\ \varphi))$
⟨*proof*⟩

lemma *wellformed-mojmir-opt*:
  assumes $\mathit{Only\text{-}G}\ \mathcal{G}$
  assumes $\mathit{finite}\ \Sigma$
  assumes $\mathit{range}\ w \subseteq \Sigma$
  shows $\mathit{mojmir}\ \Sigma\ \uparrow\!\mathit{af}_{G\mathfrak{U}}\ (\mathit{Abs}\ \varphi)\ w\ \{q.\ \mathcal{G} \models_P \mathit{Rep}\ q\}$
⟨*proof*⟩

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

sublocale *mojmir-to-rabin-def* $\Sigma\ \uparrow\!\mathit{af}_{G\mathfrak{U}}\ \mathit{Abs}\ (\mathit{Unf}_G\ \varphi)\ w\ \{q.\ \mathcal{G} \models_P \mathit{Rep}\ q\}$
⟨*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* $q\}$
$\langle proof \rangle$

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
  $\langle proof \rangle$
interpretation $\mathfrak{U}$: *ltl-FG-to-rabin-opt* Σ φ $\mathcal{G}$ w
  $\langle proof \rangle$

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*)
$\langle proof \rangle$

lemma *unfold-token-succeeds-eq*:
$\mathfrak{M}$.*token-succeeds* x = $\mathfrak{U}$.*token-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
$M_{\mathfrak{U}}$-*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-pairs*
abbreviation $Acc_{\mathfrak{U}} \equiv$ *Acc*
abbreviation $\mathcal{A}_{\mathfrak{U}} \equiv$ *ltl-to-generalized-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* $ltl\text{-}to\text{-}generalized\text{-}rabin\text{-}af_{\mathfrak{U}}$-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* = $true_n$ | $false_n$ | *Prop-ltln 'a* | *Nprop-ltln 'a* | *And-ltln ('a ltln) ('a ltln)* | *Or-ltln ('a ltln) ('a ltln)* | *Next-ltln ('a ltln)* | *Until-ltln ('a ltln) ('a ltln)* | *Release-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* $\Rightarrow$ $'a$ *ltl* $\Rightarrow$ $'a$ *ltl* (- *R* - [*87*,*87*] *86*)
where
$\varphi\ R\ \psi \equiv (G\ \psi)\ or\ (\psi\ U\ (\varphi\ and\ \psi))$

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

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

fun *ltln-to-ltl* :: $'a$ *ltln* $\Rightarrow$ $'a$ *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* $\varphi = true_n$ *then F* (*ltln-to-ltl* ψ) *else* (*ltln-to-ltl* φ) *U* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* $(\varphi\ R_n\ \psi)$ = (*if* $\varphi = false_n$ *then G* (*ltln-to-ltl* ψ) *else* (*ltln-to-ltl* φ) *R* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* $(\varphi\ W_n\ \psi)$ = (*if* $\psi = false_n$ *then G* (*ltln-to-ltl* φ) *else* (*ltln-to-ltl* φ) *W* (*ltln-to-ltl* ψ))
| *ltln-to-ltl* $(\varphi\ M_n\ \psi)$ = (*if* $\psi = true_n$ *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* $\Rightarrow$ $'a$ *list*
where

$atoms\text{-}list\ (\varphi\ and_n\ \psi) = List.union\ (atoms\text{-}list\ \varphi)\ (atoms\text{-}list\ \psi)$
$|\ atoms\text{-}list\ (\varphi\ or_n\ \psi) = List.union\ (atoms\text{-}list\ \varphi)\ (atoms\text{-}list\ \psi)$
$|\ atoms\text{-}list\ (\varphi\ U_n\ \psi) = List.union\ (atoms\text{-}list\ \varphi)\ (atoms\text{-}list\ \psi)$
$|\ atoms\text{-}list\ (\varphi\ R_n\ \psi) = List.union\ (atoms\text{-}list\ \varphi)\ (atoms\text{-}list\ \psi)$
$|\ atoms\text{-}list\ (\varphi\ W_n\ \psi) = List.union\ (atoms\text{-}list\ \varphi)\ (atoms\text{-}list\ \psi)$
$|\ atoms\text{-}list\ (\varphi\ M_n\ \psi) = List.union\ (atoms\text{-}list\ \varphi)\ (atoms\text{-}list\ \psi)$
$|\ atoms\text{-}list\ (X_n\ \varphi) = atoms\text{-}list\ \varphi$
$|\ atoms\text{-}list\ (prop_n(a)) = [a]$
$|\ atoms\text{-}list\ (nprop_n(a)) = [a]$
$|\ atoms\text{-}list\ \text{-} = []$

lemma *atoms-list-correct*:
$set\ (atoms\text{-}list\ \varphi) = atoms\text{-}ltln\ \varphi$
⟨*proof*⟩

lemma *atoms-list-distinct*:
$distinct\ (atoms\text{-}list\ \varphi)$
⟨*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 $G\text{-}list :: 'a\ ltl \Rightarrow 'a\ ltl\ list$
where
$G\text{-}list\ (\varphi\ and\ \psi) = List.union\ (G\text{-}list\ \varphi)\ (G\text{-}list\ \psi)$
$|\ G\text{-}list\ (\varphi\ or\ \psi) = List.union\ (G\text{-}list\ \varphi)\ (G\text{-}list\ \psi)$
$|\ G\text{-}list\ (F\ \varphi) = G\text{-}list\ \varphi$
$|\ G\text{-}list\ (G\ \varphi) = List.insert\ (G\ \varphi)\ (G\text{-}list\ \varphi)$
$|\ G\text{-}list\ (X\ \varphi) = G\text{-}list\ \varphi$
$|\ G\text{-}list\ (\varphi\ U\ \psi) = List.union\ (G\text{-}list\ \varphi)\ (G\text{-}list\ \psi)$
$|\ G\text{-}list\ \varphi = []$

lemma *G-eq-G-list*:
$\mathbf{G}\ \varphi = set\ (G\text{-}list\ \varphi)$

⟨*proof*⟩

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

16.2 Propositional Equivalence

fun *ifex-of-ltl* :: $'a$ *ltl* $\Rightarrow$ $'a$ *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 ($\lambda y\ s.\ \{x.\ s\ x\} \models_P y$)
⟨*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* $\Rightarrow$ $'a$ *ltl*
where
remove-constants$_P$ (φ *and* ψ) = (

case (*remove-constants*$_P$ φ) *of*
  false $\Rightarrow$ *false*
  | *true* $\Rightarrow$ *remove-constants*$_P$ ψ
  | φ' $\Rightarrow$ (*case remove-constants*$_P$ ψ *of*
    false $\Rightarrow$ *false*
    | *true* $\Rightarrow$ φ'
    | ψ' $\Rightarrow$ φ' *and* ψ'))
| *remove-constants*$_P$ (φ *or* ψ) = (
  case (*remove-constants*$_P$ φ) *of*
    true $\Rightarrow$ *true*
  | *false* $\Rightarrow$ *remove-constants*$_P$ ψ
  | φ' $\Rightarrow$ (*case remove-constants*$_P$ ψ *of*
    true $\Rightarrow$ *true*
    | *false* $\Rightarrow$ φ'
    | ψ' $\Rightarrow$ φ' *or* ψ'))
| *remove-constants*$_P$ φ = φ

lemma *remove-constants-correct*:
  $S \models_P \varphi \longleftrightarrow S \models_P$ *remove-constants*$_P$ φ
  ⟨*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* $\Rightarrow$ *false* | *true* $\Rightarrow$ f y
  | x' $\Rightarrow$ (*case* f y *of false* $\Rightarrow$ *false* | *true* $\Rightarrow$ x'
  | y' $\Rightarrow$ *if in-and* x' y' *then* y' *else if in-and* y' x' *then* x' *else* x' *and* y'))

definition *mk-and'*
where
$\textit{mk-and}'\ x\ y \equiv \textit{case}\ y\ \textit{of false} \Rightarrow \textit{false} \mid \textit{true} \Rightarrow x \mid \text{-} \Rightarrow x\ \textit{and}\ y$

definition *mk-or*
where
$\textit{mk-or}\ f\ x\ y = (\textit{case}\ f\ x\ \textit{of true} \Rightarrow \textit{true} \mid \textit{false} \Rightarrow f\ y$
$\mid x' \Rightarrow (\textit{case}\ f\ y\ \textit{of true} \Rightarrow \textit{true} \mid \textit{false} \Rightarrow x'$
$\mid y' \Rightarrow \textit{if in-or}\ x'\ y'\ \textit{then}\ y'\ \textit{else if in-or}\ y'\ x'\ \textit{then}\ x'\ \textit{else}\ x'\ \textit{or}\ y'))$

definition *mk-or'*
where
$\textit{mk-or}'\ x\ y \equiv \textit{case}\ y\ \textit{of true} \Rightarrow \textit{true} \mid \textit{false} \Rightarrow x \mid \text{-} \Rightarrow x\ \textit{or}\ y$

lemma *mk-and-correct*:
$S \models_P \textit{mk-and}\ f\ x\ y \longleftrightarrow S \models_P f\ x\ \textit{and}\ f\ y$
⟨*proof*⟩

lemma *mk-and'-correct*:
$S \models_P \textit{mk-and}'\ x\ y \longleftrightarrow S \models_P x\ \textit{and}\ y$
⟨*proof*⟩

lemma *mk-or-correct*:
$S \models_P \textit{mk-or}\ f\ x\ y \longleftrightarrow S \models_P f\ x\ \textit{or}\ f\ y$
⟨*proof*⟩

lemma *mk-or'-correct*:
$S \models_P \textit{mk-or}'\ x\ y \longleftrightarrow S \models_P x\ \textit{or}\ y$
⟨*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 $(z\ or\ y) = (case\ z\ of$
    $(((z'\ and\ x')\ or\ y')\ and\ x) \Rightarrow if\ x = x' \wedge y = y'\ then\ ((z'\ and\ x')\ or\ y')\ else\ remove\text{-}and\text{-}or\ z\ or\ remove\text{-}and\text{-}or\ y$
    $\mid \text{-} \Rightarrow remove\text{-}and\text{-}or\ z\ or\ remove\text{-}and\text{-}or\ y)$
$\mid remove\text{-}and\text{-}or\ (x\ and\ y) = remove\text{-}and\text{-}or\ x\ and\ remove\text{-}and\text{-}or\ y$
$\mid remove\text{-}and\text{-}or\ x = x$

lemma *remove-and-or-correct*:
  $S \models_P remove\text{-}and\text{-}or\ x \longleftrightarrow S \models_P x$
$\langle proof \rangle$

17.2 Optimized Equations

fun *af-letter-simp*
where
  $af\text{-}letter\text{-}simp\ true\ \nu = true$
$\mid af\text{-}letter\text{-}simp\ false\ \nu = false$
$\mid af\text{-}letter\text{-}simp\ p(a)\ \nu = (if\ a \in \nu\ then\ true\ else\ false)$
$\mid af\text{-}letter\text{-}simp\ (np(a))\ \nu\ = (if\ a \notin \nu\ then\ true\ else\ false)$
$\mid af\text{-}letter\text{-}simp\ (\varphi\ and\ \psi)\ \nu = (case\ \varphi\ of$
    $true \Rightarrow af\text{-}letter\text{-}simp\ \psi\ \nu$
  $\mid false \Rightarrow false$
  $\mid p(a) \Rightarrow if\ a \in \nu\ then\ af\text{-}letter\text{-}simp\ \psi\ \nu\ else\ false$
  $\mid np(a) \Rightarrow if\ a \notin \nu\ then\ af\text{-}letter\text{-}simp\ \psi\ \nu\ else\ false$
  $\mid G\ \varphi' \Rightarrow$
    $(let$
      $\varphi'' = af\text{-}letter\text{-}simp\ \varphi'\ \nu;$
      $\psi'' = af\text{-}letter\text{-}simp\ \psi\ \nu$
     in
      $(if\ \varphi'' = \psi''\ then\ mk\text{-}and'\ (G\ \varphi')\ \varphi''\ else\ mk\text{-}and\ id\ (mk\text{-}and'\ (G\ \varphi')\ \varphi'')\ \psi''))$
  $\mid \text{-} \Rightarrow mk\text{-}and\ id\ (af\text{-}letter\text{-}simp\ \varphi\ \nu)\ (af\text{-}letter\text{-}simp\ \psi\ \nu))$
$\mid af\text{-}letter\text{-}simp\ (\varphi\ or\ \psi)\ \nu = (case\ \varphi\ of$
    $true \Rightarrow true$
  $\mid false \Rightarrow af\text{-}letter\text{-}simp\ \psi\ \nu$
  $\mid p(a) \Rightarrow if\ a \in \nu\ then\ true\ else\ af\text{-}letter\text{-}simp\ \psi\ \nu$
  $\mid np(a) \Rightarrow if\ a \notin \nu\ then\ true\ else\ af\text{-}letter\text{-}simp\ \psi\ \nu$
  $\mid F\ \varphi' \Rightarrow$
    $(let$
      $\varphi'' = af\text{-}letter\text{-}simp\ \varphi'\ \nu;$

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

lemma *af-letter-simp-correct*:
$S \models_P af\text{-}letter\ \varphi\ \nu \longleftrightarrow S \models_P af\text{-}letter\text{-}simp\ \varphi\ \nu$
$\langle proof \rangle$

fun *af-G-letter-simp*
where
$af\text{-}G\text{-}letter\text{-}simp\ true\ \nu = true$
$\mid af\text{-}G\text{-}letter\text{-}simp\ false\ \nu = false$
$\mid af\text{-}G\text{-}letter\text{-}simp\ p(a)\ \nu = (if\ a \in \nu\ then\ true\ else\ false)$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (np(a))\ \nu\ = (if\ a \notin \nu\ then\ true\ else\ false)$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (\varphi\ and\ \psi)\ \nu = (case\ \varphi\ of$
$true \Rightarrow af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu$
$\mid false \Rightarrow false$
$\mid p(a) \Rightarrow if\ a \in \nu\ then\ af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu\ else\ false$
$\mid np(a) \Rightarrow if\ a \notin \nu\ then\ af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu\ else\ false$
$\mid$ - $\Rightarrow mk\text{-}and\ id\ (af\text{-}G\text{-}letter\text{-}simp\ \varphi\ \nu)\ (af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu))$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (\varphi\ or\ \psi)\ \nu = (case\ \varphi\ of$
$true \Rightarrow true$
$\mid false \Rightarrow af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu$
$\mid p(a) \Rightarrow if\ a \in \nu\ then\ true\ else\ af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu$
$\mid np(a) \Rightarrow if\ a \notin \nu\ then\ true\ else\ af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu$
$\mid F\ \varphi' \Rightarrow$
$(let$
$\varphi'' = af\text{-}G\text{-}letter\text{-}simp\ \varphi'\ \nu;$
$\psi'' = af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu$
in
$(if\ \varphi'' = \psi''\ then\ mk\text{-}or'\ (F\ \varphi')\ \varphi''\ else\ mk\text{-}or\ id\ (mk\text{-}or'\ (F\ \varphi')\ \varphi'')\ \psi''))$
$\mid$ - $\Rightarrow mk\text{-}or\ id\ (af\text{-}G\text{-}letter\text{-}simp\ \varphi\ \nu)\ (af\text{-}G\text{-}letter\text{-}simp\ \psi\ \nu))$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (X\ \varphi)\ \nu = \varphi$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (G\ \varphi)\ \nu = G\ \varphi$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (F\ \varphi)\ \nu = mk\text{-}or'\ (F\ \varphi)\ (af\text{-}G\text{-}letter\text{-}simp\ \varphi\ \nu)$
$\mid af\text{-}G\text{-}letter\text{-}simp\ (\varphi\ U\ \psi)\ \nu = mk\text{-}or'\ (mk\text{-}and'\ (\varphi\ U\ \psi)\ (af\text{-}G\text{-}letter\text{-}simp$

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

lemma *af-G-letter-simp-correct*:
$S \models_P \textit{af-G-letter}\ \varphi\ \nu \longleftrightarrow S \models_P \textit{af-G-letter-simp}\ \varphi\ \nu$
$\langle \textit{proof} \rangle$

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

lemma *step-simp-correct*:
$S \models_P \textit{step}\ \varphi\ \nu \longleftrightarrow S \models_P \textit{step-simp}\ \varphi\ \nu$
$\langle \textit{proof} \rangle$

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

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
| $F\ \varphi'$ ⇒
(*let*
φ'' = *Unf*$_G$*-simp* φ'; ψ'' = *Unf*$_G$*-simp* ψ
in
(*if* $\varphi'' = \psi''$ *then mk-or'* ($F\ \varphi'$) φ'' *else mk-or id* (*mk-or'* ($F\ \varphi'$) φ'')
ψ''))
| - ⇒ *mk-or id* (*Unf*$_G$*-simp* φ) (*Unf*$_G$*-simp* ψ))
| *Unf*$_G$*-simp* ($F\ \varphi$) = *mk-or'* ($F\ \varphi$) (*Unf*$_G$*-simp* φ)
| *Unf*$_G$*-simp* ($\varphi\ U\ \psi$) = *mk-or'* (*mk-and'* ($\varphi\ U\ \psi$) (*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*
| $G\ \varphi'$ ⇒
(*let*
φ'' = *Unf-simp* φ';
ψ'' = *af-letter-opt-simp* ψ ν

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

lemma *af-letter-opt-simp-correct*:
$S \models_P$ *af-letter-opt* $\varphi\ \nu \longleftrightarrow S \models_P$ *af-letter-opt-simp* $\varphi\ \nu$
⟨*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* $a \in \nu$ *then true else false*)
| *af-G-letter-opt-simp* ($np(a)$) ν = (*if* $a \notin \nu$ *then true else false*)
| *af-G-letter-opt-simp* (φ *and* ψ) ν = (*case* φ *of*
true ⇒ *af-G-letter-opt-simp* $\psi\ \nu$
| *false* ⇒ *false*
| $p(a)$ ⇒ *if* $a \in \nu$ *then af-G-letter-opt-simp* $\psi\ \nu$ *else false*
| $np(a)$ ⇒ *if* $a \notin \nu$ *then af-G-letter-opt-simp* $\psi\ \nu$ *else false*
| - ⇒ *mk-and id* (*af-G-letter-opt-simp* $\varphi\ \nu$) (*af-G-letter-opt-simp* $\psi\ \nu$))
| *af-G-letter-opt-simp* (φ *or* ψ) ν = (*case* φ *of*
true ⇒ *true*
| *false* ⇒ *af-G-letter-opt-simp* $\psi\ \nu$
| $p(a)$ ⇒ *if* $a \in \nu$ *then true else af-G-letter-opt-simp* $\psi\ \nu$
| $np(a)$ ⇒ *if* $a \notin \nu$ *then true else af-G-letter-opt-simp* $\psi\ \nu$

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

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

17.3 Register Code Equations

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

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

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

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

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

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

lemma [*code*]:
  $\uparrow af_{G\mathfrak{U}}$ (*Abs* φ) ν = *Abs* (*remove-and-or* (*af-G-letter-opt-simp* φ ν))
  ⟨*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* ⇒ *'a list*
where
  init q_0 = [q_0]

fun *nxt* :: *'b set* ⇒ (*'a*, *'b*) *DTS* ⇒ *'a* ⇒ (*'a list*, *'b*) *DTS*
where
  nxt Σ δ q_0 = (λ*qs* ν. *remdups-fwd* ((*filter* (λ*q*. ¬*semi-mojmir-def.sink* Σ δ q_0 *q*) (*map* (λ*q*. δ *q* ν) *qs*)) @ [q_0]))

— Recompute the rank from the list

fun *rk* :: *'a list* ⇒ *'a* ⇒ *nat option*
where
  rk qs q = (*let i* = *index qs q in if i* ≠ *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* ⇒ (*'a*, *'b*) *DTS* ⇒ *'a* ⇒ (*'a* ⇒ *bool*) ⇒ (*'a list*, *'b*) *transition* ⇒ *bool*
where
  fail-filt Σ δ q_0 *F* (*r*, ν, -) = (∃ *q* ∈ *set r*. *let q'* = δ *q* ν *in* (¬*F q'*) ∧ *semi-mojmir-def.sink* Σ δ q_0 *q'*)

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

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

$\langle proof \rangle$

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

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

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

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 $q \in set\ (r\ n) \longleftrightarrow \neg sink\ q \wedge configuration\ q\ n \neq \{\}$
$\langle proof \rangle$

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

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*)
$\langle proof \rangle$

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

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

$state\text{-}rank\ q\ n = rk\ (foldl\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ (map\ w\ [0..<n]))\ q$
⟨*proof*⟩

lemma (**in** *semi-mojmir*) *nxt-run-step-run*:
$run\ step\ initial\ w = rk\ o\ (run\ (nxt\ \Sigma\ \delta\ q_0)\ (init\ q_0)\ w)$
⟨*proof*⟩

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

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

lemma (**in** *mojmir-to-rabin-def*) *filt-equiv*:
$(rk\ x,\ \nu,\ y) \in fail_R \longleftrightarrow fail\text{-}filt\ \Sigma\ \delta\ q_0\ (\lambda x.\ x \in F)\ (x,\ \nu,\ y')$
$(rk\ x,\ \nu,\ y) \in succeed_R\ i \longleftrightarrow succeed\text{-}filt\ \delta\ q_0\ (\lambda x.\ x \in F)\ i\ (x,\ \nu,\ y')$
$(rk\ x,\ \nu,\ y) \in merge_R\ i \longleftrightarrow merge\text{-}filt\ \delta\ q_0\ (\lambda x.\ x \in F)\ i\ (x,\ \nu,\ y')$
⟨*proof*⟩

lemma *fail-filt-eq*:
$fail\text{-}filt\ \Sigma\ \delta\ q_0\ P\ (x,\ \nu,\ y) \longleftrightarrow (rk\ x,\ \nu,\ y') \in mojmir\text{-}to\text{-}rabin\text{-}def.fail_R$
$\Sigma\ \delta\ q_0\ \{x.\ P\ x\}$
⟨*proof*⟩

lemma *merge-filt-eq*:
$merge\text{-}filt\ \delta\ q_0\ P\ i\ (x,\ \nu,\ y) \longleftrightarrow (rk\ x,\ \nu,\ y') \in mojmir\text{-}to\text{-}rabin\text{-}def.merge_R$
$\delta\ q_0\ \{x.\ P\ x\}\ i$
⟨*proof*⟩

lemma *succeed-filt-eq*:
$succeed\text{-}filt\ \delta\ q_0\ P\ i\ (x,\ \nu,\ y) \longleftrightarrow (rk\ x,\ \nu,\ y') \in mojmir\text{-}to\text{-}rabin\text{-}def.succeed_R$
$\delta\ q_0\ \{x.\ P\ x\}\ i$
⟨*proof*⟩

theorem (**in** *mojmir-to-rabin*) *rabin-accept-iff-rabin-list-accept-rank*:
$accepting\text{-}pair_R\ \delta_{\mathcal{R}}\ q_{\mathcal{R}}\ (Acc_{\mathcal{R}}\ i)\ w \longleftrightarrow accepting\text{-}pair_R\ (nxt\ \Sigma\ \delta\ q_0)\ (init$
$q_0)\ (\{t.\ fail\text{-}filt\ \Sigma\ \delta\ q_0\ (\lambda x.\ x \in F)\ t\} \cup \{t.\ merge\text{-}filt\ \delta\ q_0\ (\lambda x.\ x \in F)\ i$
$t\},\ \{t.\ succeed\text{-}filt\ \delta\ q_0\ (\lambda x.\ x \in F)\ i\ t\})\ w$
(**is** $accepting\text{-}pair_R\ \delta_{\mathcal{R}}\ q_{\mathcal{R}}\ (?F,\ ?I)\ w \longleftrightarrow accepting\text{-}pair_R\ (nxt\ \Sigma\ \delta\ q_0)$
$(init\ q_0)\ (?F',\ ?I')\ w)$
⟨*proof*⟩

18.1 Compute Rabin Automata List Representation

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

18.2 Code Generation

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

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

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

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

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

theorem (**in** *mojmir-to-rabin*) *exec-correct*:
  assumes $\Sigma = set\ \Sigma'$
  shows $accept \longleftrightarrow accept_R\text{-}LTS\ (mojmir\text{-}to\text{-}rabin\text{-}exec\ \Sigma'\ \delta\ q_0\ (\lambda x.\ x \in$

F)) w (**is** ?*lhs* $\longleftrightarrow$?*rhs*)
$\langle proof \rangle$

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
$M\text{-}fin_C :: {}'a\ ltl \Rightarrow ({}'a\ ltl,\ nat)\ mapping \Rightarrow ({}'a\ ltl_P \times ({}'a\ ltl,\ {}'a\ ltl_P\ list)\ mapping,\ {}'a\ set)\ transition \Rightarrow bool$
begin

— Transition Function and Initial State

fun $delta_C$
where
$delta_C\ \Sigma = \delta \times \uparrow\Delta_\times\ (nxt\ \Sigma\ \delta_M\ o\ q_{0M}\ o\ theG)$

fun $initial_C$
where
$initial_C\ \varphi = (q_0\ \varphi,\ Mapping.tabulate\ (G\text{-}list\ \varphi)\ (init\ o\ q_{0M}\ o\ theG))$

— Acceptance Condition

definition $max\text{-}rank\text{-}of_C$
where
$max\text{-}rank\text{-}of_C\ \Sigma\ \psi = card\ (Set.filter\ (Not\ o\ semi\text{-}mojmir\text{-}def.sink\ (set\ \Sigma)\ \delta_M\ (q_{0M}\ (theG\ \psi)))\ (Q_L\ \Sigma\ \delta_M\ (q_{0M}\ (theG\ \psi))))$

fun $Acc\text{-}fin_C$
where
$Acc\text{-}fin_C\ \Sigma\ \pi\ \chi\ ((\text{-},\ m'),\ \nu,\ \text{-}) = ($
let

$t = (\mathit{the}\ (\mathit{Mapping.lookup}\ m'\ \chi),\ \nu,\ [])$; — Third element is unused. Hence it is safe to pass a dummy value.
$\mathcal{G} = \mathit{Mapping.keys}\ \pi$
in
$\mathit{fail\text{-}filt}\ \Sigma\ \delta_M\ (q_{0M}\ (\mathit{theG}\ \chi))\ (\mathit{ltl\text{-}prop\text{-}entails\text{-}abs}\ \mathcal{G})\ t$
$\vee\ \mathit{merge\text{-}filt}\ \delta_M\ (q_{0M}\ (\mathit{theG}\ \chi))\ (\mathit{ltl\text{-}prop\text{-}entails\text{-}abs}\ \mathcal{G})\ (\mathit{the}\ (\mathit{Mapping.lookup}\ \pi\ \chi))\ t)$

fun $\mathit{Acc\text{-}inf}_C$
where
$\mathit{Acc\text{-}inf}_C\ \pi\ \chi\ ((\text{-},\ m'),\ \nu,\ \text{-}) = ($
let
$t = (\mathit{the}\ (\mathit{Mapping.lookup}\ m'\ \chi),\ \nu,\ [])$; — Third element is unused. Hence it is safe to pass a dummy value.
$\mathcal{G} = \mathit{Mapping.keys}\ \pi$
in
$\mathit{succeed\text{-}filt}\ \delta_M\ (q_{0M}\ (\mathit{theG}\ \chi))\ (\mathit{ltl\text{-}prop\text{-}entails\text{-}abs}\ \mathcal{G})\ (\mathit{the}\ (\mathit{Mapping.lookup}\ \pi\ \chi))\ t)$

definition $\mathit{mappings}_C :: {'a}\ \mathit{set}\ \mathit{list} \Rightarrow {'a}\ \mathit{ltl} \Rightarrow ({'a}\ \mathit{ltl},\ \mathit{nat})\ \mathit{mapping}\ \mathit{set}$
where
$\mathit{mappings}_C\ \Sigma\ \varphi \equiv \{\pi.\ \mathit{Mapping.keys}\ \pi \subseteq \mathbf{G}\ \varphi \wedge (\forall \chi \in (\mathit{Mapping.keys}\ \pi).\ \mathit{the}\ (\mathit{Mapping.lookup}\ \pi\ \chi) < \mathit{max\text{-}rank\text{-}of}_C\ \Sigma\ \chi)\}$

definition $\mathit{reachable\text{-}transitions}_C$
where
$\mathit{reachable\text{-}transitions}_C\ \Sigma\ \varphi \equiv \delta_L\ \Sigma\ (\mathit{delta}_C\ (\mathit{set}\ \Sigma))\ (\mathit{initial}_C\ \varphi)$

fun $\mathit{ltl\text{-}to\text{-}generalized\text{-}rabin}_C$
where
$\mathit{ltl\text{-}to\text{-}generalized\text{-}rabin}_C\ \Sigma\ \varphi = ($
let
$\delta\text{-}\mathit{LTS} = \mathit{reachable\text{-}transitions}_C\ \Sigma\ \varphi;$
$\alpha\text{-}\mathit{fin\text{-}filter} = \lambda\pi\ t.\ \mathit{M\text{-}fin}_C\ \varphi\ \pi\ t \vee (\exists \chi \in \mathit{Mapping.keys}\ \pi.\ \mathit{Acc\text{-}fin}_C\ (\mathit{set}\ \Sigma)\ \pi\ \chi\ t);$
$\mathit{to\text{-}pair} = \lambda\pi.\ (\mathit{Set.filter}\ (\alpha\text{-}\mathit{fin\text{-}filter}\ \pi)\ \delta\text{-}\mathit{LTS},\ (\lambda\chi.\ \mathit{Set.filter}\ (\mathit{Acc\text{-}inf}_C\ \pi\ \chi)\ \delta\text{-}\mathit{LTS})\ `\ \mathit{Mapping.keys}\ \pi);$
$\alpha = \mathit{to\text{-}pair}\ `\ (\mathit{mappings}_C\ \Sigma\ \varphi)$ — Multi-thread here!, prove *mappings* $(\mathit{set}\ \ldots)$ equation
in
$(\delta\text{-}\mathit{LTS},\ \mathit{initial}_C\ \varphi,\ \alpha))$

lemma $\mathit{mappings}_C\text{-}\mathit{code}$:
$\mathit{mappings}_C\ \Sigma\ \varphi = ($

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*)
$\langle$*proof*$\rangle$

lemma *reach-delta-initial*:
assumes $(x, y) \in$ *reach* Σ (*delta*$_C$ Σ) (*initial*$_C$ φ)
assumes $\chi \in \mathbf{G}\ \varphi$
shows *Mapping.lookup y* $\chi \neq$ *None* (**is** *?t1*)
and *distinct* (*the* (*Mapping.lookup y* χ)) (**is** *?t2*)
$\langle$*proof*$\rangle$

end

19.1.2 Correctness

fun *abstract-state* :: $'x \times ('y, 'z\ list)\ mapping \Rightarrow 'x \times ('y \rightharpoonup 'z \rightharpoonup nat)$
where
abstract-state $(a, b) = (a,$ (*map-option rk*) *o* (*Mapping.lookup b*))

fun *abstract-transition*
where
abstract-transition $(q, \nu, q') =$ (*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*: $[\![t \in reach_t\ \Sigma\ (delta_C\ \Sigma)\ (initial_C\ \varphi);\ dom\ \pi \subseteq \mathbf{G}\ \varphi]\!] \Longrightarrow$
abstract-transition $t \in$ *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))
$\langle$*proof*$\rangle$

lemma *max-rank-of*$_C$*-eq*:
assumes $\Sigma =$ *set* Σ'
shows *max-rank-of*$_C$ Σ' ψ = *max-rank-of* Σ ψ
$\langle$*proof*$\rangle$

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 χ $\longleftrightarrow$ *Acc-fin*$_C$ Σ (*Mapping.Mapping* π) χ *t* (**is** *?t1*)
and *Acc-inf*$_C$*-correct*:
abstract-transition t $\in$ *Acc-inf* π χ $\longleftrightarrow$ *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* $\Rightarrow$ (*'a ltl*, *nat*) *mapping* $\Rightarrow$ (*'a ltl*, (*'a ltl*$_P$ *list*)) *mapping* $\Rightarrow$ *'a ltl*$_P$
where
M-fin$_C$*-af-lhs* φ π *m'* $\equiv$
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* (*Abs* χ) (*map* ($\uparrow$*eval*$_G$ $\mathcal{G}$) (*drop* (*the* (*Mapping.lookup* π χ)) (*the* (*Mapping.lookup* *m'* χ))))
in

$\uparrow$*And* (*map mk-conj* $\mathcal{G}_L$)

fun *M-fin*$_C$*-af* :: *'a ltl* $\Rightarrow$ (*'a ltl*, *nat*) *mapping* $\Rightarrow$ (*'a ltl*$_P$ $\times$ ((*'a ltl*, (*'a ltl*$_P$ *list*)) *mapping*), *'a set*) *transition* $\Rightarrow$ *bool*
where
M-fin$_C$*-af* φ π ((φ', *m'*), -) = *Not* ((*M-fin*$_C$*-af-lhs* φ π *m'*) $\uparrow\longrightarrow_P$ φ')

lemma *M-fin*$_C$*-af-correct*:
assumes *t* $\in$ *reach*$_t$ Σ (*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* $\pi \subseteq$ **G** φ
shows *abstract-transition t* $\in$ *M-fin* π = *M-fin*$_C$*-af* φ (*Mapping.Mapping* π) *t*
$\langle$*proof*$\rangle$

definition
ltl-to-generalized-rabin$_C$*-af* $\equiv$ *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* $\subseteq$ *set* Σ
shows *w* $\models \varphi \longleftrightarrow$ *accept*$_{GR}$*-LTS* (*ltl-to-generalized-rabin*$_C$*-af* Σ φ) *w*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
$\langle$*proof*$\rangle$

19.3 Generalized Deterministic Rabin Automaton (eager af)

definition *M-fin*$_C$*-af*$_{\mathfrak{U}}$*-lhs* :: *'a ltl* $\Rightarrow$ (*'a ltl*, *nat*) *mapping* $\Rightarrow$ (*'a ltl*, (*'a ltl*$_P$ *list*)) *mapping* $\Rightarrow$ *'a set* $\Rightarrow$ *'a ltl*$_P$
where
M-fin$_C$*-af*$_{\mathfrak{U}}$*-lhs* φ π *m'* ν $\equiv$
let
$\mathcal{G}$ = *Mapping.keys* π;
$\mathcal{G}_L$ = *filter* (λ*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* (λ*q.* $\uparrow$*step q* ν)) (*drop* (*the* (*Mapping.lookup* π χ)) (*the* (*Mapping.lookup m'* χ))))
in
$\uparrow$*And* (*map mk-conj* $\mathcal{G}_L$)

fun *M-fin*$_C$*-af*$_{\mathfrak{U}}$ :: *'a ltl* $\Rightarrow$ (*'a ltl*, *nat*) *mapping* $\Rightarrow$ (*'a ltl*$_P$ $\times$ ((*'a ltl*, (*'a ltl*$_P$ *list*)) *mapping*), *'a set*) *transition* $\Rightarrow$ *bool*
where
M-fin$_C$*-af*$_{\mathfrak{U}}$ φ π ((φ', *m'*), ν, -) = *Not* ((*M-fin*$_C$*-af*$_{\mathfrak{U}}$*-lhs* φ π *m'* ν) $\uparrow\longrightarrow_P$

($\uparrow$*step* φ' ν))

lemma *M-fin$_C$-af$_{\mathfrak{U}}$-correct*:
  assumes $t \in$ *reach$_t$* Σ (*ltl-to-rabin-base-code-def.delta$_C$* $\uparrow$*af$_{\mathfrak{U}}$* $\uparrow$*af$_{G\mathfrak{U}}$* (*Abs* $\circ$ *Unf$_G$*) Σ) (*ltl-to-rabin-base-code-def.initial$_C$* (*Abs* $\circ$ *Unf*) (*Abs* $\circ$ *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$_{\mathfrak{U}}$ $\equiv$ *ltl-to-rabin-base-code-def.ltl-to-generalized-rabin$_C$* $\uparrow$*af$_{\mathfrak{U}}$* $\uparrow$*af$_{G\mathfrak{U}}$* (*Abs* $\circ$ *Unf*) (*Abs* $\circ$ *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* $\longleftrightarrow$ *?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*) $\equiv$
    (*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 $(i, (q, \nu, p))$ $(q', \nu', p') = ($
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
$(i \mid ($*eq-q* & *eq-p* & $\nu = \nu'), q'', \nu, p''))$

fun *norm-fold* :: $('a, 'b)$ *transition* $\Rightarrow$ $('a, 'b)$ *transition list* $\Rightarrow$ (*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* $\Rightarrow$ $('a, 'b)$ *transition list* $\Rightarrow$ (*bool* $*$ $('a, 'b)$ *transition list*)
where
norm-insert x xs $\equiv$ *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*]:
$\uparrow\Delta_\times$ *f* (*AList-Mapping.Mapping xs*) *c* = *AList-Mapping.Mapping* (*map-ran* ($\lambda a\ b.\ f\ a\ b\ c$) *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.