Markov Models

Johannes Hölzl and Tobias Nipkow

May 26, 2024

Abstract

This is a formalization of various Markov models in Isabelle/HOL. It builds on Isabelle's probability theory. The available models are currently discrete-time and continuous-time Markov chains as well as Markov decision processes. As application of these models we formalize probabilistic model checking of pCTL formulas, analysis of IPv4 address allocation in ZeroConf and an analysis of the anonymity of the Crowds protocol.

Contents

1 Introduction

This is a formalization of probabilistic models in Isabelle/HOL. It builds on Isabelle's probability theory (HOL-Probability). It provides formalizations for the following models:

- Discrete-time Markov processes with measurable state spaces [2]
- Markov decision processes on discrete spaces [5]
- Continuous-time Markov chains on discrete spaces [2]

As application of these models we formalize

- a probabilistic model checking of pCTL formulas [4],
- an analysis of IPv4 address allocation in ZeroConf [3],
- an analysis of the anonymity of the Crowds protocol [3],
- the reachability analysis on finite-state MDPs [5], and
- expected running-time semantics for pGCL [1].

The formalization of rewarded DTMCs and pCTL model checking is discussed in detail in our paper.

2 Auxiliary Theory

Parts of it should be moved to the Isabelle repository

theory *Markov-Models-Auxiliary*
imports
HOL−Probability.Probability
HOL−Library.Rewrite
HOL−Library.Linear-Temporal-Logic-on-Streams
Coinductive.Coinductive-Stream
Coinductive.Coinductive-Nat
begin

lemma *lfp-upperbound*: $(\bigwedge y.\ x \leq f\ y) \Longrightarrow x \leq lfp\ f$
⟨*proof*⟩

lemma *lfp-arg*: $(\lambda t.\ lfp\ (F\ t)) = lfp\ (\lambda x\ t.\ F\ t\ (x\ t))$
⟨*proof*⟩

lemma *lfp-pair*: $lfp\ (\lambda f\ (a,\ b).\ F\ (\lambda a\ b.\ f\ (a,\ b))\ a\ b)\ (a,\ b) = lfp\ F\ a\ b$
⟨*proof*⟩

lemma *all-Suc-split*: $(\forall i.\ P\ i) \longleftrightarrow (P\ 0 \wedge (\forall i.\ P\ (Suc\ i)))$
⟨*proof*⟩

definition *with* $P\ f\ d = (if\ \exists x.\ P\ x\ then\ f\ (SOME\ x.\ P\ x)\ else\ d)$

lemma *withI*[*case-names default exists*]:
$((\bigwedge x.\ \neg\ P\ x) \Longrightarrow Q\ d) \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ (f\ x)) \Longrightarrow Q\ (with\ P\ f\ d)$
⟨*proof*⟩

context *order*
begin

definition
maximal f S = {*x*∈*S*. ∀ *y*∈*S*. *f y* ≤ *f x*}

lemma *maximalI*: *x* ∈ *S* ⟹ (⋀*y*. *y* ∈ *S* ⟹ *f y* ≤ *f x*) ⟹ *x* ∈ *maximal f S*
⟨*proof*⟩

lemma *maximalI-trans*: *x* ∈ *maximal f S* ⟹ *f x* ≤ *f y* ⟹ *y* ∈ *S* ⟹ *y* ∈ *maximal f S*
⟨*proof*⟩

lemma *maximalD1*: *x* ∈ *maximal f S* ⟹ *x* ∈ *S*
⟨*proof*⟩

lemma *maximalD2*: *x* ∈ *maximal f S* ⟹ *y* ∈ *S* ⟹ *f y* ≤ *f x*
⟨*proof*⟩

lemma *maximal-inject*: *x* ∈ *maximal f S* ⟹ *y* ∈ *maximal f S* ⟹ *f x* = *f y*
⟨*proof*⟩

lemma *maximal-empty*[*simp*]: *maximal f* {} = {}
⟨*proof*⟩

lemma *maximal-singleton*[*simp*]: *maximal f* {*x*} = {*x*}
⟨*proof*⟩

lemma *maximal-in-S*: *maximal f S* ⊆ *S*
⟨*proof*⟩

end

context *linorder*
begin

lemma *maximal-ne*:
assumes *finite S S* ≠ {}
shows *maximal f S* ≠ {}
⟨*proof*⟩

end

lemma *mono-les*:
fixes *s S N* **and** *l1 l2* :: *'a* ⇒ *real* **and** *K* :: *'a* ⇒ *'a pmf*
defines Δ *x* ≡ *l2 x* − *l1 x*
assumes *s*: *s* ∈ *S* **and** *S*: (⋃*s*∈*S*. *set-pmf* (*K s*)) ⊆ *S* ∪ *N*
assumes *int-l1*[*simp*]: ⋀*s*. *s* ∈ *S* ⟹ *integrable* (*K s*) *l1*
assumes *int-l2*[*simp*]: ⋀*s*. *s* ∈ *S* ⟹ *integrable* (*K s*) *l2*
assumes *to-N*: ⋀*s*. *s* ∈ *S* ⟹ ∃ *t*∈*N*. (*s*, *t*) ∈ (*SIGMA s*:*UNIV*. *K s*)*
assumes *l1*: ⋀*s*. *s* ∈ *S* ⟹ (∫ *t*. *l1 t* ∂*K s*) + *c s* ≤ *l1 s*
assumes *l2*: ⋀*s*. *s* ∈ *S* ⟹ *l2 s* ≤ (∫ *t*. *l2 t* ∂*K s*) + *c s*

assumes *eq*: $\bigwedge s.\ s \in N \Longrightarrow l2\ s \leq l1\ s$
assumes *finitary*: *finite* $(\Delta\ `\ (S \cup N))$
shows $l2\ s \leq l1\ s$
⟨*proof*⟩

lemma *unique-les*:
fixes $s\ S\ N$ **and** $l1\ l2 :: 'a \Rightarrow real$ **and** $K :: 'a \Rightarrow 'a\ pmf$
defines $\Delta\ x \equiv l2\ x - l1\ x$
assumes *s*: $s \in S$ **and** *S*: $(\bigcup s{\in}S.\ set\text{-}pmf\ (K\ s)) \subseteq S \cup N$
assumes $\bigwedge s.\ s \in S \Longrightarrow integrable\ (K\ s)\ l1$
assumes $\bigwedge s.\ s \in S \Longrightarrow integrable\ (K\ s)\ l2$
assumes $\bigwedge s.\ s \in S \Longrightarrow \exists\, t{\in}N.\ (s,\ t) \in (SIGMA\ s{:}UNIV.\ K\ s)^*$
assumes $\bigwedge s.\ s \in S \Longrightarrow l1\ s = (\int t.\ l1\ t\ \partial K\ s) + c\ s$
assumes $\bigwedge s.\ s \in S \Longrightarrow l2\ s = (\int t.\ l2\ t\ \partial K\ s) + c\ s$
assumes $\bigwedge s.\ s \in N \Longrightarrow l2\ s = l1\ s$
assumes *1*: *finite* $(\Delta\ `\ (S \cup N))$
shows $l2\ s = l1\ s$
⟨*proof*⟩

lemma *inf-continuous-suntil-disj*[*order-continuous-intros*]:
assumes *Q*: *inf-continuous Q*
assumes *disj*: $\bigwedge x\ \omega.\ \neg\ (P\ \omega \wedge Q\ x\ \omega)$
shows *inf-continuous* $(\lambda x.\ P\ suntil\ Q\ x)$
⟨*proof*⟩

lemma *inf-continuous-nxt*[*order-continuous-intros*]: *inf-continuous P* $\Longrightarrow$ *inf-continuous* $(\lambda x.\ nxt\ (P\ x)\ \omega)$
⟨*proof*⟩

lemma *sup-continuous-nxt*[*order-continuous-intros*]: *sup-continuous P* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ nxt\ (P\ x)\ \omega)$
⟨*proof*⟩

lemma *mcont-ennreal-of-enat*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ *ennreal-of-enat*
⟨*proof*⟩

lemma *mcont2mcont-ennreal-of-enat*[*cont-intro*]:
mcont lub ord Sup $(\leq)$ *f* $\Longrightarrow$ *mcont lub ord Sup* $(\leq)$ $(\lambda x.\ ennreal\text{-}of\text{-}enat\ (f\ x))$
⟨*proof*⟩

declare *stream.exhaust*[*cases type*: *stream*]

lemma *scount-eq-emeasure*: *scount P* ω = *emeasure* (*count-space UNIV*) $\{i.\ P\ (sdrop\ i\ \omega)\}$
⟨*proof*⟩

lemma *measurable-scount*[*measurable*]:
assumes [*measurable*]: *Measurable.pred* (*stream-space M*) *P*
shows *scount P* $\in$ *measurable* (*stream-space M*) (*count-space UNIV*)

⟨*proof*⟩

lemma *measurable-sfirst2*:
assumes [*measurable*]: *Measurable.pred* $(N \bigotimes_M$ *stream-space* $M)$ $(\lambda(x, \omega).\ P\ x\ \omega)$
shows $(\lambda(x, \omega).$ *sfirst* $(P\ x)\ \omega) \in$ *measurable* $(N \bigotimes_M$ *stream-space* $M)$ (*count-space UNIV*)
⟨*proof*⟩

lemma *measurable-sfirst2'*[*measurable* (*raw*)]:
assumes [*measurable* (*raw*)]: $f \in N \to_M$ *stream-space* M *Measurable.pred* $(N \bigotimes_M$ *stream-space* $M)$ $(\lambda x.\ P$ (*fst* x) (*snd* x))
shows $(\lambda x.$ *sfirst* $(P\ x)\ (f\ x)) \in$ *measurable* N (*count-space UNIV*)
⟨*proof*⟩

lemma *measurable-sfirst*[*measurable*]:
assumes [*measurable*]: *Measurable.pred* (*stream-space* M) P
shows *sfirst* $P \in$ *measurable* (*stream-space* M) (*count-space UNIV*)
⟨*proof*⟩

lemma *measurable-epred*[*measurable*]: *epred* $\in$ *count-space UNIV* $\to_M$ *count-space UNIV*
⟨*proof*⟩

lemma *nn-integral-stretch*:
$f \in$ *borel* $\to_M$ *borel* $\Longrightarrow c \neq 0 \Longrightarrow (\int^+ x.\ f\ (c * x)\ \partial$*lborel*$) = (1\ /\ |c|$::*real*$) * (\int^+ x.\ f\ x\ \partial$*lborel*$)$
⟨*proof*⟩

lemma *prod-sum-distrib*:
fixes $f\ g ::\ 'a \Rightarrow\ 'b \Rightarrow\ 'c$::*comm-semiring-1*
assumes *finite* I **shows** $(\bigwedge i.\ i \in I \Longrightarrow$ *finite* $(J\ i)) \Longrightarrow (\prod i{\in}I.\ \sum j{\in}J\ i.\ f\ i\ j) = (\sum m{\in}Pi_E\ I\ J.\ \prod i{\in}I.\ f\ i\ (m\ i))$
⟨*proof*⟩

lemma *prod-add-distrib*:
fixes $f\ g ::\ 'a \Rightarrow\ 'b$::*comm-semiring-1*
assumes *finite* I **shows** $(\prod i{\in}I.\ f\ i + g\ i) = (\sum J{\in}Pow\ I.\ (\prod i{\in}J.\ f\ i) * (\prod i{\in}I - J.\ g\ i))$
⟨*proof*⟩

subclass (**in** *linordered-nonzero-semiring*) *ordered-semiring-0*
⟨*proof*⟩

lemma (**in** *linordered-nonzero-semiring*) *prod-nonneg*: $(\forall\, a{\in}A.\ 0 \le f\ a) \Longrightarrow 0 \le$ *prod* $f\ A$
⟨*proof*⟩

lemma (**in** *linordered-nonzero-semiring*) *prod-mono*:

$\forall i{\in}A.\ 0 \le f\ i \wedge f\ i \le g\ i \Longrightarrow prod\ f\ A \le prod\ g\ A$
⟨*proof*⟩

lemma (**in** *linordered-nonzero-semiring*) *prod-mono2*:
assumes *finite J I* ⊆ *J* ⋀*i*. *i* ∈ *I* ⟹ *0* ≤ *g i* ∧ *g i* ≤ *f i* (⋀*i*. *i* ∈ *J* − *I* ⟹ *1* ≤ *f i*)
shows *prod g I* ≤ *prod f J*
⟨*proof*⟩

lemma (**in** *linordered-nonzero-semiring*) *prod-mono3*:
assumes *finite J I* ⊆ *J* ⋀*i*. *i* ∈ *J* ⟹ *0* ≤ *g i* ⋀*i*. *i* ∈ *I* ⟹ *g i* ≤ *f i* (⋀*i*. *i* ∈ *J* − *I* ⟹ *g i* ≤ *1*)
shows *prod g J* ≤ *prod f I*
⟨*proof*⟩

lemma (**in** *linordered-nonzero-semiring*) *one-le-prod*: (⋀*i*. *i* ∈ *I* ⟹ *1* ≤ *f i*) ⟹ *1* ≤ *prod f I*
⟨*proof*⟩

lemma *sum-plus-one-le-prod-plus-one*:
fixes *p* :: *'a* ⇒ *'b*::*linordered-nonzero-semiring*
assumes ⋀*i*. *i* ∈ *I* ⟹ *0* ≤ *p i*
shows (∑ *i*∈*I*. *p i*) + *1* ≤ (∏ *i*∈*I*. *p i* + *1*)
⟨*proof*⟩

lemma *summable-iff-convergent-prod*:
fixes *p* :: *nat* ⇒ *real* **assumes** *p*: ⋀*i*. *0* ≤ *p i*
shows *summable p* ⟷ *convergent* (λ*n*. ∏ *i*<*n*. *p i* + *1*)
⟨*proof*⟩

primrec *eexp* :: *ereal* ⇒ *ennreal*
where
eexp MInfty = *0*
| *eexp* (*ereal r*) = *ennreal* (*exp r*)
| *eexp PInfty* = *top*

lemma
shows *eexp-minus-infty*[*simp*]: *eexp* (−∞) = *0*
and *eexp-infty*[*simp*]: *eexp* ∞ = *top*
⟨*proof*⟩

lemma *eexp-0*[*simp*]: *eexp 0* = *1*
⟨*proof*⟩

lemma *eexp-inj*[*simp*]: *eexp x* = *eexp y* ⟷ *x* = *y*
⟨*proof*⟩

lemma *eexp-mono*[*simp*]: *eexp x* ≤ *eexp y* ⟷ *x* ≤ *y*
⟨*proof*⟩

lemma *eexp-strict-mono*[*simp*]: *eexp x* < *eexp y* ⟷ *x* < *y*
⟨*proof*⟩

lemma *exp-eq-0-iff*[*simp*]: *eexp x* = *0* ⟷ *x* = −∞
⟨*proof*⟩

lemma *eexp-surj*: *range eexp* = *UNIV*
⟨*proof*⟩

lemma *continuous-on-eexp'*: *continuous-on UNIV eexp*
⟨*proof*⟩

lemma *continuous-on-eexp*[*continuous-intros*]: *continuous-on A f* ⟹ *continuous-on A* (λ*x*. *eexp* (*f x*))
⟨*proof*⟩

lemma *tendsto-eexp*[*tendsto-intros*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. *eexp* (*f x*)) ⟶ *eexp x*) *F*
⟨*proof*⟩

lemma *measurable-eexp*[*measurable*]: *eexp* ∈ *borel* →$_M$ *borel*
⟨*proof*⟩

lemma *eexp-add*: ¬ ((*x* = ∞ ∧ *y* = −∞) ∨ (*x* = −∞ ∧ *y* = ∞)) ⟹ *eexp* (*x* + *y*) = *eexp x* ∗ *eexp y*
⟨*proof*⟩

lemma *sum-Pinfty*:
fixes *f* :: *'a* ⇒ *ereal*
shows *sum f I* = ∞ ⟷ (*finite I* ∧ (∃ *i*∈*I*. *f i* = ∞))
⟨*proof*⟩

lemma *sum-Minfty*:
fixes *f* :: *'a* ⇒ *ereal*
shows *sum f I* = −∞ ⟷ (*finite I* ∧ ¬ (∃ *i*∈*I*. *f i* = ∞) ∧ (∃ *i*∈*I*. *f i* = −∞))
⟨*proof*⟩

lemma *eexp-sum*: ¬ (∃ *i*∈*I*. ∃ *j*∈*I*. *f i* = −∞ ∧ *f j* = ∞) ⟹ *eexp* (∑ *i*∈*I*. *f i*) = (∏ *i*∈*I*. *eexp* (*f i*))
⟨*proof*⟩

lemma *eexp-suminf*:
assumes *wf-f*: ¬ {−∞, ∞} ⊆ *range f* **and** *f*: *summable f*
shows (λ*n*. ∏ *i*<*n*. *eexp* (*f i*)) ⟶ *eexp* (∑ *i*. *f i*)
⟨*proof*⟩

lemma *continuous-onI-antimono*:
fixes *f* :: *'a*::*linorder-topology* ⇒ *'b*::{*dense-order*,*linorder-topology*}

assumes *open* (*f*‘*A*)
and *mono*: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \leq y \Longrightarrow f\ y \leq f\ x$
shows *continuous-on A f*
⟨*proof*⟩

lemma *minus-add-eq-ereal*: $\neg\ ((a = \infty \wedge b = -\infty) \vee (a = -\infty \wedge b = \infty)) \Longrightarrow -\ (a + b::ereal) = -a - b$
⟨*proof*⟩

lemma *setsum-negf-ereal*: $\neg\ \{-\infty, \infty\} \subseteq f`I \Longrightarrow (\sum i{\in}I.\ -\ f\ i) = -\ (\sum i{\in}I.\ f\ i::ereal)$
⟨*proof*⟩

lemma *convergent-minus-iff-ereal*: *convergent* $(\lambda x.\ -\ f\ x::ereal) \longleftrightarrow$ *convergent f*
⟨*proof*⟩

lemma *summable-minus-ereal*: $\neg\ \{-\infty, \infty\} \subseteq$ *range f* $\Longrightarrow$ *summable* $(\lambda n.\ f\ n) \Longrightarrow$ *summable* $(\lambda n.\ -\ f\ n::ereal)$
⟨*proof*⟩

lemma (**in** *product-prob-space*) *product-nn-integral-component*:
assumes $f \in$ *borel-measurable* $(M\ i) i \in I$
shows $integral^N\ (Pi_M\ I\ M)\ (\lambda x.\ f\ (x\ i)) = integral^N\ (M\ i)\ f$
⟨*proof*⟩

lemma *ennreal-inverse-le*[*simp*]: *inverse x* $\leq$ *inverse y* $\longleftrightarrow$ $y \leq (x::ennreal)$
⟨*proof*⟩

lemma *inverse-inverse-ennreal*[*simp*]: *inverse* (*inverse x*::*ennreal*) = *x*
⟨*proof*⟩

lemma *range-inverse-ennreal*: *range inverse* = (*UNIV*::*ennreal set*)
⟨*proof*⟩

lemma *continuous-on-inverse-ennreal*′: *continuous-on* (*UNIV* :: *ennreal set*) *inverse*
⟨*proof*⟩

lemma *sums-minus-ereal*: $\neg\ \{-\ \infty, \infty\} \subseteq f\ `\ UNIV \Longrightarrow (\lambda n.\ -\ f\ n::ereal)$ *sums* $x \Longrightarrow f$ *sums* $-\ x$
⟨*proof*⟩

lemma *suminf-minus-ereal*: $\neg\ \{-\ \infty, \infty\} \subseteq f\ `\ UNIV \Longrightarrow$ *summable f* $\Longrightarrow (\sum n.\ -\ f\ n :: ereal) = -\ suminf\ f$
⟨*proof*⟩

end

3 Discrete-Time Markov Chain

theory *Discrete-Time-Markov-Chain*
imports *Markov-Models-Auxiliary*
begin

Markov chain with discrete time steps and discrete state space.

lemma *sstart-eq'*: *sstart* Ω $(x \# xs) = \{\omega.\ shd\ \omega = x \wedge stl\ \omega \in sstart\ \Omega\ xs\}$
⟨*proof*⟩

lemma *measure-eq-stream-space-coinduct*[*consumes 1*, *case-names left right cont*]:
assumes *R N M*
assumes *R-1*: $\bigwedge N\ M.\ R\ N\ M \Longrightarrow N \in space\ (prob\text{-}algebra\ (stream\text{-}space\ (count\text{-}space\ UNIV)))$
and *R-2*: $\bigwedge N\ M.\ R\ N\ M \Longrightarrow M \in space\ (prob\text{-}algebra\ (stream\text{-}space\ (count\text{-}space\ UNIV)))$
and *cont*: $\bigwedge N\ M.\ R\ N\ M \Longrightarrow \exists\, N'\ M'\ p.\ (\forall\, y{\in}set\text{-}pmf\ p.\ R\ (N'\ y)\ (M'\ y)) \wedge$
$(\forall\, x.\ N'\ x \in space\ (prob\text{-}algebra\ (stream\text{-}space\ (count\text{-}space\ UNIV)))) \wedge (\forall\, x.\ M'\ x \in space\ (prob\text{-}algebra\ (stream\text{-}space\ (count\text{-}space\ UNIV)))) \wedge$
$N = (measure\text{-}pmf\ p \gg\!= (\lambda y.\ distr\ (N'\ y)\ (stream\text{-}space\ (count\text{-}space\ UNIV))\ ((\#\#)\ y))) \wedge$
$M = (measure\text{-}pmf\ p \gg\!= (\lambda y.\ distr\ (M'\ y)\ (stream\text{-}space\ (count\text{-}space\ UNIV))\ ((\#\#)\ y)))$
shows $N = M$
⟨*proof*⟩

3.1 Discrete Markov Kernel

locale *MC-syntax* =
fixes $K :: 's \Rightarrow 's\ pmf$
begin

abbreviation *acc* :: $('s \times 's)\ set$ **where**
$acc \equiv (SIGMA\ s{:}UNIV.\ K\ s)^*$

abbreviation *acc-on* :: $'s\ set \Rightarrow ('s \times 's)\ set$ **where**
$acc\text{-}on\ S \equiv (SIGMA\ s{:}UNIV.\ K\ s \cap S)^*$

lemma *countable-reachable*: *countable* $(acc\ ``\ \{s\})$
⟨*proof*⟩

lemma *countable-acc*: *countable* $X \Longrightarrow$ *countable* $(acc\ ``\ X)$
⟨*proof*⟩

context
notes [[*inductive-internals*]]
begin

coinductive *enabled* **where**
$enabled\ (shd\ \omega)\ (stl\ \omega) \Longrightarrow shd\ \omega \in K\ s \Longrightarrow enabled\ s\ \omega$

end

lemma *alw-enabled*: *enabled* (*shd* ω) (*stl* ω) $\Longrightarrow$ *alw* ($\lambda\omega$. *enabled* (*shd* ω) (*stl* ω)) ω
⟨*proof*⟩

abbreviation $S \equiv$ *stream-space* (*count-space UNIV*)

lemma *in-S* [*measurable* (*raw*)]: $x \in$ *space* S
⟨*proof*⟩

inductive-simps *enabled-iff*: *enabled* s ω

lemma *enabled-Stream*: *enabled* x (y ## ω) $\longleftrightarrow$ $y \in K\ x \wedge$ *enabled* y ω
⟨*proof*⟩

lemma *measurable-enabled*[*measurable*]:
Measurable.pred (*stream-space* (*count-space UNIV*)) (*enabled* s) (**is** *Measurable.pred ?S* -)
⟨*proof*⟩

lemma *enabled-iff-snth*: *enabled* s ω $\longleftrightarrow$ ($\forall\, i.\ \omega$!! $i \in K$ ((s ## ω) !! i))
⟨*proof*⟩

primcorec *force-enabled* **where**
force-enabled x ω =
(*let* y = *if shd* $\omega \in K\ x$ *then shd* ω *else* (*SOME* y. $y \in K\ x$) *in* y ## *force-enabled* y (*stl* ω))

lemma *force-enabled-in-set-pmf*[*simp*, *intro*]: *shd* (*force-enabled* x ω) $\in K\ x$
⟨*proof*⟩

lemma *enabled-force-enabled*: *enabled* x (*force-enabled* x ω)
⟨*proof*⟩

lemma *force-enabled*: *enabled* x $\omega \Longrightarrow$ *force-enabled* x $\omega = \omega$
⟨*proof*⟩

lemma *Ex-enabled*: $\exists\omega$. *enabled* x ω
⟨*proof*⟩

lemma *measurable-force-enabled*: *force-enabled* $x \in$ *measurable* S S
⟨*proof*⟩

abbreviation $D \equiv$ *stream-space* (Π_M $s \in$ *UNIV*. $K\ s$)

lemma *sets-D*: *sets* D = *sets* (*stream-space* (Π_M $s \in$ *UNIV*. *count-space UNIV*))
⟨*proof*⟩

lemma *space-D*: *space D* = *space* (*stream-space* (Π_M *s*∈*UNIV*. *count-space UNIV*))
⟨*proof*⟩

lemma *measurable-D-D*: *measurable D D* =
measurable (*stream-space* (Π_M *s*∈*UNIV*. *count-space UNIV*)) (*stream-space* (Π_M *s*∈*UNIV*. *count-space UNIV*))
⟨*proof*⟩

primcorec *walk* :: *'s* ⇒ (*'s* ⇒ *'s*) *stream* ⇒ *'s stream* **where**
shd (*walk s ω*) = (*if shd ω s* ∈ *K s then shd ω s else* (*SOME t*. *t* ∈ *K s*))
| *stl* (*walk s ω*) = *walk* (*if shd ω s* ∈ *K s then shd ω s else* (*SOME t*. *t* ∈ *K s*)) (*stl ω*)

lemma *enabled-walk*: *enabled s* (*walk s ω*)
⟨*proof*⟩

lemma *measurable-walk*[*measurable*]: *walk s* ∈ *measurable D S*
⟨*proof*⟩

3.2 Trace Space for Discrete-Time Markov Chains

definition *T* :: *'s* ⇒ *'s stream measure* **where**
T s = *distr* (*stream-space* (Π_M *s*∈*UNIV*. *K s*)) *S* (*walk s*)

lemma *space-T*[*simp*]: *space* (*T s*) = *space S*
⟨*proof*⟩

lemma *sets-T*[*simp*, *measurable-cong*]: *sets* (*T s*) = *sets S*
⟨*proof*⟩

lemma *measurable-T1*[*simp*]: *measurable* (*T s*) *M* = *measurable S M*
⟨*proof*⟩

lemma *measurable-T2*[*simp*]: *measurable M* (*T s*) = *measurable M S*
⟨*proof*⟩

lemma *in-measurable-T1*[*measurable* (*raw*)]: *f* ∈ *measurable S M* ⟹ *f* ∈ *measurable* (*T s*) *M*
⟨*proof*⟩

lemma *in-measurable-T2*[*measurable* (*raw*)]: *f* ∈ *measurable M S* ⟹ *f* ∈ *measurable M* (*T s*)
⟨*proof*⟩

lemma *AE-T-enabled*: *AE ω in T s*. *enabled s ω*
⟨*proof*⟩

sublocale *T*: *prob-space T s* **for** *s*

⟨*proof*⟩

lemma *emeasure-T-const*[*simp*]: *emeasure* (*T s*) (*space S*) = *1*
⟨*proof*⟩

lemma *nn-integral-T*:
assumes *f*[*measurable*]: *f* ∈ *borel-measurable S*
shows (∫⁺*X*. *f X* ∂*T s*) = (∫⁺*t*. (∫⁺ω. *f* (*t* ## ω) ∂*T t*) ∂*K s*)
⟨*proof*⟩

lemma *nn-integral-T-gfp*:
fixes *g*
defines *l* ≡ λ*f* ω. *g* (*shd* ω) (*f* (*stl* ω))
assumes [*measurable*]: *case-prod g* ∈ *borel-measurable* (*count-space UNIV* ⨂$_M$ *borel*)
assumes *cont-g*[*THEN inf-continuous-compose*, *order-continuous-intros*]: ⋀*s*. *inf-continuous* (*g s*)
assumes *int-g*: ⋀*f s*. *f* ∈ *borel-measurable S* ⟹ (∫⁺ω. *g s* (*f* ω) ∂*T s*) = *g s* (∫⁺ω. *f* ω ∂*T s*)
assumes *bnd-g*: ⋀*f s*. *g s f* ≤ *b* 0 ≤ *b b* < ∞
shows (∫⁺ω. *gfp l* ω ∂*T s*) = *gfp* (λ*f s*. ∫⁺*t*. *g t* (*f t*) ∂*K s*) *s*
⟨*proof*⟩

lemma *nn-integral-T-lfp*:
fixes *g*
defines *l* ≡ λ*f* ω. *g* (*shd* ω) (*f* (*stl* ω))
assumes [*measurable*]: *case-prod g* ∈ *borel-measurable* (*count-space UNIV* ⨂$_M$ *borel*)
assumes *cont-g*[*THEN sup-continuous-compose*, *order-continuous-intros*]: ⋀*s*. *sup-continuous* (*g s*)
assumes *int-g*: ⋀*f s*. *f* ∈ *borel-measurable S* ⟹ (∫⁺ω. *g s* (*f* ω) ∂*T s*) = *g s* (∫⁺ω. *f* ω ∂*T s*)
shows (∫⁺ω. *lfp l* ω ∂*T s*) = *lfp* (λ*f s*. ∫⁺*t*. *g t* (*f t*) ∂*K s*) *s*
⟨*proof*⟩

lemma *emeasure-Collect-T*:
assumes *f*[*measurable*]: *Measurable.pred S P*
shows *emeasure* (*T s*) {*x*∈*space* (*T s*). *P x*} = (∫⁺*t*. *emeasure* (*T t*) {*x*∈*space* (*T t*). *P* (*t* ## *x*)} ∂*K s*)
⟨*proof*⟩

lemma *AE-T-iff*:
assumes [*measurable*]: *Measurable.pred S P*
shows (*AE* ω *in T x*. *P* ω) ⟷ (∀ *y*∈*K x*. *AE* ω *in T y*. *P* (*y* ## ω))
⟨*proof*⟩

lemma *AE-T-alw*:
assumes [*measurable*]: *Measurable.pred S P*
assumes *P*: ⋀*s*. (*x*, *s*) ∈ *acc* ⟹ *AE* ω *in T s*. *P* ω

shows *AE ω in T x. alw P ω*
⟨*proof*⟩

lemma *emeasure-suntil-disj*:
assumes [*measurable*]: *Measurable.pred S P*
assumes ∗: ⋀*t. AE ω in T t. ¬ (P ⊓ (HLD X ⊓ nxt (HLD X suntil P))) ω*
shows *emeasure (T s) {ω∈space (T s). (HLD X suntil P) ω} =*
lfp (λF s. emeasure (T s) {ω∈space (T s). P ω} + (∫⁺t. F t ∗ indicator X t ∂K s)) s
⟨*proof*⟩

lemma *emeasure-HLD-nxt*:
assumes [*measurable*]: *Measurable.pred S P*
shows *emeasure (T s) {ω∈space (T s). (X · P) ω} =*
(∫⁺x. emeasure (T x) {ω∈space (T x). P ω} ∗ indicator X x ∂K s)
⟨*proof*⟩

lemma *emeasure-HLD*:
emeasure (T s) {ω∈space (T s). HLD X ω} = emeasure (K s) X
⟨*proof*⟩

lemma *emeasure-suntil-HLD*:
assumes [*measurable*]: *Measurable.pred S P*
shows *emeasure (T s) {x∈space (T s). (not (HLD {t}) suntil (HLD {t} aand nxt P)) x} =*
emeasure (T s) {x∈space (T s). ev (HLD {t}) x} ∗ emeasure (T t) {x∈space (T t). P x}
⟨*proof*⟩

lemma *AE-suntil*:
assumes [*measurable*]: *Measurable.pred S P*
shows *(AE x in T s. (not (HLD {t}) suntil (HLD {t} aand nxt P)) x) ⟷*
(AE x in T s. ev (HLD {t}) x) ∧ (AE x in T t. P x)
⟨*proof*⟩

3.3 Fairness

definition *fair* :: *'s ⇒ 's ⇒ 's stream ⇒ bool* **where**
fair s t = alw (ev (HLD {s})) impl alw (ev (HLD {s} aand nxt (HLD {t})))

lemma *AE-T-fair*:
assumes *t' ∈ K t*
shows *AE ω in T s. fair t t' ω*
⟨*proof*⟩

lemma *enabled-imp-trancl*:
assumes *alw (HLD B) ω enabled s ω*
shows *alw (HLD (acc-on B `` {s})) ω*
⟨*proof*⟩

lemma *AE-T-reachable*: $AE\ \omega\ in\ T\ s.\ alw\ (HLD\ (acc\ ``\ \{s\}))\ \omega$
⟨*proof*⟩

lemma *AE-T-all-fair*: $AE\ \omega\ in\ T\ s.\ \forall (t,t')\in SIGMA\ t{:}UNIV.\ K\ t.\ fair\ t\ t'\ \omega$
⟨*proof*⟩

lemma *fair-imp*: **assumes** $fair\ t\ t'\ \omega\ alw\ (ev\ (HLD\ \{t\}))\ \omega$ **shows** $alw\ (ev\ (HLD\ \{t'\}))\ \omega$
⟨*proof*⟩

lemma *AE-T-ev-HLD*:
assumes *exiting*: $\bigwedge t.\ (s,\ t) \in acc\text{-}on\ (-B) \Longrightarrow \exists\, t'\in B.\ (t,\ t') \in acc$
assumes *fin*: *finite* $(acc\text{-}on\ (-B)\ ``\ \{s\})$
shows $AE\ \omega\ in\ T\ s.\ ev\ (HLD\ B)\ \omega$
⟨*proof*⟩

lemma *AE-T-ev-HLD′*:
assumes *exiting*: $\bigwedge s.\ s \notin X \Longrightarrow \exists\, t\in X.\ (s,\ t) \in acc$
assumes *fin*: *finite* $(-X)$
shows $AE\ \omega\ in\ T\ s.\ ev\ (HLD\ X)\ \omega$
⟨*proof*⟩

lemma *AE-T-max-sfirst*:
assumes [*measurable*]: *Measurable.pred S X*
assumes *AE*: $AE\ \omega\ in\ T\ c.\ sfirst\ X\ (c\ \#\#\ \omega) < \infty$ **and** $0 < e$
shows $\exists\, N{::}nat.\ \mathcal{P}(\omega\ in\ T\ c.\ N < sfirst\ X\ (c\ \#\#\ \omega)) < e$ (**is** $\exists N.\ ?P\ N < e$)
⟨*proof*⟩

3.4 First Hitting Time

lemma *nn-integral-sfirst-finite′*:
assumes $s \notin H$
assumes [*simp*]: *finite* $(acc\text{-}on\ (-H)\ ``\ \{s\})$
assumes *until*: $AE\ \omega\ in\ T\ s.\ ev\ (HLD\ H)\ \omega$
shows $(\int^+ \omega.\ sfirst\ (HLD\ H)\ \omega\ \partial T\ s) \neq \infty$
⟨*proof*⟩

lemma *nn-integral-sfirst-finite*:
assumes [*simp*]: *finite* $(acc\text{-}on\ (-H)\ ``\ \{s\})$
assumes *until*: $AE\ \omega\ in\ T\ s.\ ev\ (HLD\ H)\ \omega$
shows $(\int^+ \omega.\ sfirst\ (HLD\ H)\ (s\ \#\#\ \omega)\ \partial T\ s) \neq \infty$
⟨*proof*⟩

lemma *prob-T*:
assumes *P*: *Measurable.pred S P*
shows $\mathcal{P}(\omega\ in\ T\ s.\ P\ \omega) = (\int t.\ \mathcal{P}(\omega\ in\ T\ t.\ P\ (t\ \#\#\ \omega))\ \partial K\ s)$
⟨*proof*⟩

lemma *T-subprob*[*measurable*]: $T \in$ *measurable* (*measure-pmf I*) (*subprob-algebra S*)
⟨*proof*⟩

3.5 Markov chain with Initial Distribution

definition $T' :: 's\ pmf \Rightarrow 's\ stream\ measure$ **where**
$T'\ I = bind\ I\ (\lambda s.\ distr\ (T\ s)\ S\ ((\#\#)\ s))$

lemma *distr-Stream-subprob*:
$(\lambda s.\ distr\ (T\ s)\ S\ ((\#\#)\ s)) \in measurable\ (measure\text{-}pmf\ I)\ (subprob\text{-}algebra\ S)$
⟨*proof*⟩

lemma *sets-T'*: $sets\ (T'\ I) = sets\ S$
⟨*proof*⟩

lemma *prob-space-T'*: *prob-space* $(T'\ I)$
⟨*proof*⟩

lemma *AE-T'*:
assumes [*measurable*]: *Measurable.pred S P*
shows $(AE\ x\ in\ T'\ I.\ P\ x) \longleftrightarrow (\forall s \in I.\ AE\ x\ in\ T\ s.\ P\ (s\ \#\#\ x))$
⟨*proof*⟩

lemma *emeasure-T'*:
assumes [*measurable*]: $X \in sets\ S$
shows $emeasure\ (T'\ I)\ X = (\int^+ s.\ emeasure\ (T\ s)\ \{\omega \in space\ S.\ s\ \#\#\ \omega \in X\}\ \partial I)$
⟨*proof*⟩

lemma *prob-T'*:
assumes [*measurable*]: *Measurable.pred S P*
shows $\mathcal{P}(x\ in\ T'\ I.\ P\ x) = (\int s.\ \mathcal{P}(x\ in\ T\ s.\ P\ (s\ \#\#\ x))\ \partial I)$
⟨*proof*⟩

lemma *T-eq-T'*: $T\ s = T'\ (K\ s)$
⟨*proof*⟩

lemma *T-eq-bind*: $T\ s = (measure\text{-}pmf\ (K\ s) \gg\!= (\lambda t.\ distr\ (T\ t)\ S\ ((\#\#)\ t)))$
⟨*proof*⟩

lemma *T-split*:
$T\ s = (T\ s \gg\!= (\lambda \omega.\ distr\ (T\ ((s\ \#\#\ \omega)\ !!\ n))\ S\ (\lambda \omega'.\ stake\ n\ \omega\ @-\ \omega')))$
⟨*proof*⟩

lemma *nn-integral-T-split*:
assumes *f*[*measurable*]: $f \in borel\text{-}measurable\ S$
shows $(\int^+ \omega.\ f\ \omega\ \partial T\ s) = (\int^+ \omega.\ (\int^+ \omega'.\ f\ (stake\ n\ \omega\ @-\ \omega')\ \partial T\ ((s\ \#\#\ \omega)\ !!\ n))\ \partial T\ s)$

⟨*proof*⟩

lemma *emeasure-T-split*:
assumes *P*[*measurable*]: *Measurable.pred S P*
shows *emeasure* (*T s*) {*ω*∈*space* (*T s*). *P ω*} =
(∫⁺*ω*. *emeasure* (*T* ((*s* ## *ω*) !! *n*)) {*ω*′∈*space* (*T* ((*s* ## *ω*) !! *n*)). *P* (*stake n ω* @− *ω*′)} *∂T s*)
⟨*proof*⟩

lemma *prob-T-split*:
assumes *P*[*measurable*]: *Measurable.pred S P*
shows *P*(*ω in T s*. *P ω*) = (∫ *ω*. *P*(*ω*′ *in T* ((*s* ## *ω*) !! *n*). *P* (*stake n ω* @− *ω*′)) *∂T s*)
⟨*proof*⟩

lemma *enabled-imp-alw*:
(⋃*s*∈*X*. *set-pmf* (*K s*)) ⊆ *X* ⟹ *x* ∈ *X* ⟹ *enabled x ω* ⟹ *alw* (*HLD X*) *ω*
⟨*proof*⟩

lemma *alw-HLD-iff-sconst*:
alw (*HLD* {*x*}) *ω* ⟷ *ω* = *sconst x*
⟨*proof*⟩

lemma *enabled-iff-sconst*:
assumes [*simp*]: *set-pmf* (*K x*) = {*x*} **shows** *enabled x ω* ⟷ *ω* = *sconst x*
⟨*proof*⟩

lemma *AE-sconst*:
assumes [*simp*]: *set-pmf* (*K x*) = {*x*}
shows (*AE ω in T x*. *P ω*) ⟷ *P* (*sconst x*)
⟨*proof*⟩

lemma *ev-eq-lfp*: *ev P* = *lfp* (*λF ω*. *P ω* ∨ (¬ *P ω* ∧ *F* (*stl ω*)))
⟨*proof*⟩

lemma *INF-eq-zero-iff-ennreal*: ((∏ *i*∈*A*. *f i*) = (*0*::*ennreal*)) = (∀ *x*>*0*. ∃ *i*∈*A*. *f i* < *x*)
⟨*proof*⟩

lemma *inf-continuous-cmul*:
fixes *c* :: *ennreal*
assumes *f*: *inf-continuous f* **and** *c*: *c* < ⊤
shows *inf-continuous* (*λx*. *c* ∗ *f x*)
⟨*proof*⟩

lemma *AE-T-ev-HLD-infinite*:
fixes *X* :: *'s set* **and** *r* :: *real*
assumes *r* < *1*
assumes *r*: ⋀*x*. *x* ∈ *X* ⟹ *measure* (*K x*) *X* ≤ *r*

shows *AE* ω *in* *T x*. *ev* (*HLD* (− *X*)) ω
⟨*proof*⟩

3.6 Trace space with Restriction

definition *rT x* = *restrict-space* (*T x*) {ω. *enabled x* ω}

lemma *space-rT*: $\omega \in$ *space* (*rT x*) ⟷ *enabled x* ω
⟨*proof*⟩

lemma *Collect-enabled-S*[*measurable*]: *Collect* (*enabled x*) ∈ *sets S*
⟨*proof*⟩

lemma *space-rT-in-S*: *space* (*rT x*) ∈ *sets S*
⟨*proof*⟩

lemma *sets-rT*: *A* ∈ *sets* (*rT x*) ⟷ *A* ∈ *sets S* ∧ *A* ⊆ {ω. *enabled x* ω}
⟨*proof*⟩

lemma *prob-space-rT*: *prob-space* (*rT x*)
⟨*proof*⟩

lemma *measurable-force-enabled2*[*measurable*]: *force-enabled x* ∈ *measurable S* (*rT x*)
⟨*proof*⟩

lemma *space-rT-not-empty*[*simp*]: *space* (*rT x*) ≠ {}
⟨*proof*⟩

lemma *T-eq-bind'*: *T x* = *do* { *y* ← *measure-pmf* (*K x*) ; ω ← *T y* ; *return S* (*y* ## ω) }
⟨*proof*⟩

lemma *rT-eq-bind*: *rT x* = *do* { *y* ← *measure-pmf* (*K x*) ; ω ← *rT y* ; *return* (*rT x*) (*y* ## ω) }
⟨*proof*⟩

lemma *snth-rT*: (λ*x*. *x* !! *n*) ∈ *measurable* (*rT x*) (*count-space* (*acc* `` {*x*}))
⟨*proof*⟩

3.7 Bisimulation

lemma *T-coinduct*[*consumes 1*, *case-names prob sets cont*]:
assumes *R x M*
assumes *prob*: ⋀*x M*. *R x M* ⟹ *prob-space M*
and *sets*: ⋀*x M*. *R x M* ⟹ *sets M* = *sets S*
and *cont'*: ⋀*x M*. *R x M* ⟹ ∃ *M'*. (∀ *y*∈*K x*. *R y* (*M' y*)) ∧ (∀ *y*. *sets* (*M' y*) = *S* ∧ *prob-space* (*M' y*)) ∧
M = (*measure-pmf* (*K x*) ⤜ (λ*y*. *distr* (*M' y*) *S* ((##) *y*)))
shows *T x* = *M*

⟨*proof*⟩

lemma *T-bisim*:
assumes *M*: ⋀*x*. *prob-space* (*M x*) ⋀*x*. *sets* (*M x*) = *sets S*
and *M-eq*: ⋀*x*. *M x* = (*measure-pmf* (*K x*) ⤜ (λ*s*. *distr* (*M s*) *S* ((##) *s*)))
shows *T* = *M*
⟨*proof*⟩

lemma *T-subprob'*[*measurable*]: *T* ∈ *measurable* (*count-space UNIV*) (*subprob-algebra S*)
⟨*proof*⟩

lemma *T-subprob''*[*simp*]: *T a* ∈ *space* (*subprob-algebra S*)
⟨*proof*⟩

lemma *AE-not-suntil-coinduct* [*consumes 1*, *case-names ψ φ*]:
assumes *P s*
assumes *ψ*: ⋀*s*. *P s* ⟹ *s* ∉ *ψ*
assumes *φ*: ⋀*s t*. *P s* ⟹ *s* ∈ *φ* ⟹ *t* ∈ *K s* ⟹ *P t*
shows *AE ω in T s*. *not* (*HLD φ suntil HLD ψ*) (*s ## ω*)
⟨*proof*⟩

lemma *AE-not-suntil-coinduct-strong* [*consumes 1*, *case-names ψ φ*]:
assumes *P s*
assumes *P-ψ*: ⋀*s*. *P s* ⟹ *s* ∉ *ψ*
assumes *P-φ*: ⋀*s t*. *P s* ⟹ *s* ∈ *φ* ⟹ *t* ∈ *K s* ⟹ *P t* ∨
(*AE ω in T t*. *not* (*HLD φ suntil HLD ψ*) (*t ## ω*))
shows *AE ω in T s*. *not* (*HLD φ suntil HLD ψ*) (*s ## ω*) (**is** *?nuntil s*)
⟨*proof*⟩

end

3.8 Reward Structure on Markov Chains

locale *MC-with-rewards* = *MC-syntax K* **for** *K* :: *'s* ⇒ *'s pmf* +
fixes *ι* :: *'s* ⇒ *'s* ⇒ *ennreal* **and** *ϱ* :: *'s* ⇒ *ennreal*
assumes *ι-nonneg*: ⋀*s t*. *0* ≤ *ι s t* **and** *ϱ-nonneg*: ⋀*s*. *0* ≤ *ϱ s*
assumes *measurable-ι*[*measurable*]: (λ(*a*, *b*). *ι a b*) ∈ *borel-measurable* (*count-space UNIV* ⨂$_M$ *count-space UNIV*)
begin

definition *reward-until* :: *'s set* ⇒ *'s* ⇒ *'s stream* ⇒ *ennreal* **where**
reward-until X = *lfp* (λ*F s ω*. *if s* ∈ *X then 0 else ϱ s* + *ι s* (*shd ω*) + (*F* (*shd ω*) (*stl ω*)))

lemma *measurable-ϱ*[*measurable*]: *ϱ* ∈ *borel-measurable* (*count-space UNIV*)
⟨*proof*⟩

lemma *measurable-reward-until*[*measurable* (*raw*)]:

assumes [*measurable*]: $f \in$ *measurable* M (*count-space UNIV*)
assumes [*measurable*]: $g \in$ *measurable* M S
shows $(\lambda x.$ *reward-until* X $(f\ x)$ $(g\ x)) \in$ *borel-measurable* M
⟨*proof*⟩

lemma *continuous-reward-until*:
sup-continuous $(\lambda F\ s\ \omega.$ *if* $s \in X$ *then 0 else* $\varrho\ s + \iota\ s$ $(shd\ \omega) + (F$ $(shd\ \omega)$ $(stl\ \omega)))$
⟨*proof*⟩

lemma
shows *reward-until-unfold*: *reward-until* $X\ s\ \omega =$
(if $s \in X$ *then 0 else* $\varrho\ s + \iota\ s$ $(shd\ \omega) +$ *reward-until* X $(shd\ \omega)$ $(stl\ \omega))$
(**is** *?unfold*)
⟨*proof*⟩

lemma *reward-until-simps*[*simp*]:
shows $s \in X \Longrightarrow$ *reward-until* $X\ s\ \omega = 0$
and $s \notin X \Longrightarrow$ *reward-until* $X\ s\ \omega = \varrho\ s + \iota\ s$ $(shd\ \omega) +$ *reward-until* X $(shd\ \omega)$ $(stl\ \omega)$
⟨*proof*⟩

lemma *reward-until-SCons*[*simp*]:
reward-until $X\ s$ $(t\ \#\#\ \omega) =$ *(if* $s \in X$ *then 0 else* $\varrho\ s + \iota\ s\ t +$ *reward-until* X $t\ \omega)$
⟨*proof*⟩

lemma *nn-integral-reward-until-finite*:
assumes [*simp*]: *finite* $(acc\ ``\ \{s\})$ (**is** *finite* $(?R\ ``\ \{s\}))$
assumes ϱ: $\bigwedge t.\ (s,\ t) \in$ *acc-on* $(-H) \Longrightarrow \varrho\ t < \infty$
assumes ι: $\bigwedge t\ t'.\ (s,\ t) \in$ *acc-on* $(-H) \Longrightarrow t' \in K\ t \Longrightarrow \iota\ t\ t' < \infty$
assumes *ev*: *AE* ω *in* T $s.$ *ev* $(HLD\ H)\ \omega$
shows $(\int^+ \omega.$ *reward-until* $H\ s\ \omega\ \partial T\ s) \neq \infty$
⟨*proof*⟩

end

3.9 Bisimulation on a relation

definition *rel-set-strong* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$
where *rel-set-strong* $R\ A\ B \longleftrightarrow (\forall x\ y.\ R\ x\ y \longrightarrow (x \in A \longleftrightarrow y \in B))$

lemma *T-eq-rel-half*[*consumes 4*, *case-names prob sets cont*]:
fixes $R :: 's \Rightarrow 't \Rightarrow bool$ **and** $f :: 's \Rightarrow 't$ **and** $S :: 's\ set$
assumes *R-def*: $\bigwedge s\ t.\ R\ s\ t \longleftrightarrow (s \in S \wedge f\ s = t)$
assumes *A*[*measurable*]: $A \in sets$ $(stream\text{-}space$ $(count\text{-}space\ UNIV))$
and *B*[*measurable*]: $B \in sets$ $(stream\text{-}space$ $(count\text{-}space\ UNIV))$
and *AB*: *rel-set-strong* $(stream\text{-}all2\ R)$ A B **and** *KL*: *rel-fun* R $(rel\text{-}pmf\ R)$ K L **and** *xy*: $R\ x\ y$

shows *MC-syntax.T K x A = MC-syntax.T L y B*
⟨*proof*⟩

3.10 Product Construction

locale *MC-pair* =
K1: *MC-syntax K1* + *K2*: *MC-syntax K2* **for** *K1 K2*
begin

definition *Kp* ≡ λ(*a*, *b*). *pair-pmf* (*K1 a*) (*K2 b*)

sublocale *MC-syntax Kp* ⟨*proof*⟩

definition
$szip_E$ *a b* ≡ λ(*ω1*, *ω2*). *szip* (*K1.force-enabled a ω1*) (*K2.force-enabled b ω2*)

lemma *szip-rT*[*measurable*]: (λ(*ω1*, *ω2*). *szip ω1 ω2*) ∈ *measurable* (*K1.rT x1* ⨂$_M$ *K2.rT x2*) *S*
⟨*proof*⟩

lemma *measurable-szipE*[*measurable*]: $szip_E$ *a b* ∈ *measurable* (*K1.S* ⨂$_M$ *K2.S*) *S*
⟨*proof*⟩

lemma *T-eq-prod*: *T* = (λ(*x1*, *x2*). *do* { *ω1* ← *K1.T x1* ; *ω2* ← *K2.T x2* ; *return S* ($szip_E$ *x1 x2* (*ω1*, *ω2*)) })
(**is** - = *?B*)
⟨*proof*⟩

lemma *nn-integral-pT*:
fixes *f* **assumes** [*measurable*]: *f* ∈ *borel-measurable S*
shows (∫$^+$*ω*. *f ω* ∂*T* (*x*, *y*)) = (∫$^+$*ω1*. ∫$^+$*ω2*. *f* ($szip_E$ *x y* (*ω1*, *ω2*)) ∂*K2.T y* ∂*K1.T x*)
⟨*proof*⟩

lemma *prod-eq-prob-T*:
assumes [*measurable*]: *Measurable.pred K1.S P1 Measurable.pred K2.S P2*
shows $\mathcal{P}$(*ω in K1.T x1*. *P1 ω*) ∗ $\mathcal{P}$(*ω in K2.T x2*. *P2 ω*) =
$\mathcal{P}$(*ω in T* (*x1*, *x2*). *P1* (*smap fst ω*) ∧ *P2* (*smap snd ω*))
⟨*proof*⟩

end

end

3.11 Trace Space equal to Markov Chains

theory *Trace-Space-Equals-Markov-Processes*
imports *Discrete-Time-Markov-Chain*
begin

We can construct for each time-homogeneous discrete-time Markov chain a corresponding probability space using *Markov-Models.Discrete-Time-Markov-Chain*. The constructed probability space has the same probabilities.

locale *Time-Homogeneous-Discrete-Markov-Process = M?: prob-space +*
fixes $S :: 's\ set$ **and** $X :: nat \Rightarrow 'a \Rightarrow 's$
assumes X [*measurable*]: $\bigwedge t.\ X\ t \in measurable\ M\ (count\text{-}space\ UNIV)$
assumes S: $countable\ S\ \bigwedge n.\ AE\ x\ in\ M.\ X\ n\ x \in S$
assumes MC: $\bigwedge n\ s\ s'.$
$\mathcal{P}(\omega\ in\ M.\ \forall\, t{\leq}n.\ X\ t\ \omega = s\ t\ ) \neq 0 \Longrightarrow$
$\mathcal{P}(\omega\ in\ M.\ X\ (Suc\ n)\ \omega = s' \mid \forall\, t{\leq}n.\ X\ t\ \omega = s\ t\ ) =$
$\mathcal{P}(\omega\ in\ M.\ X\ (Suc\ n)\ \omega = s' \mid X\ n\ \omega = s\ n\ )$
assumes TH: $\bigwedge n\ m\ s\ t.$
$\mathcal{P}(\omega\ in\ M.\ X\ n\ \omega = t) \neq 0 \Longrightarrow \mathcal{P}(\omega\ in\ M.\ X\ m\ \omega = t) \neq 0 \Longrightarrow$
$\mathcal{P}(\omega\ in\ M.\ X\ (Suc\ n)\ \omega = s \mid X\ n\ \omega = t) = \mathcal{P}(\omega\ in\ M.\ X\ (Suc\ m)\ \omega = s \mid X\ m\ \omega = t)$
begin

context
begin

interpretation *pmf-as-measure* ⟨*proof*⟩

lift-definition $I :: 's\ pmf$ **is** $distr\ M\ (count\text{-}space\ UNIV)\ (X\ 0)$
⟨*proof*⟩

lemma *I-in-S*:
assumes $pmf\ I\ s \neq 0$ **shows** $s \in S$
⟨*proof*⟩

lift-definition $K :: 's \Rightarrow 's\ pmf$ **is**
$\lambda s.\ with\ (\lambda n.\ \mathcal{P}(\omega\ in\ M.\ X\ n\ \omega = s) \neq 0)$
$(\lambda n.\ distr\ (uniform\text{-}measure\ M\ \{\omega{\in}space\ M.\ X\ n\ \omega = s\})\ (count\text{-}space\ UNIV)\ (X\ (Suc\ n)))$
$(uniform\text{-}measure\ (count\text{-}space\ UNIV)\ \{s\})$
⟨*proof*⟩

lemma *pmf-K*:
assumes n: $0 < \mathcal{P}(\omega\ in\ M.\ X\ n\ \omega = s)$
shows $pmf\ (K\ s)\ t = \mathcal{P}(\omega\ in\ M.\ X\ (Suc\ n)\ \omega = t \mid X\ n\ \omega = s)$
⟨*proof*⟩

lemma *pmf-K2*:
$(\bigwedge n.\ \mathcal{P}(\omega\ in\ M.\ X\ n\ \omega = s) = 0) \Longrightarrow pmf\ (K\ s)\ t = indicator\ \{t\}\ s$
⟨*proof*⟩

end

sublocale K: *MC-syntax K* ⟨*proof*⟩

lemma *bind-I-K-eq-M*: $K.T'\ I = distr\ M\ K.S\ (\lambda\omega.\ to\text{-}stream\ (\lambda n.\ X\ n\ \omega))$ (**is** *- = ?D*)
⟨*proof*⟩

end

lemma (**in** *MC-syntax*) *is-THDTMC*:
fixes $I :: 's\ pmf$
defines $U \equiv (SIGMA\ s{:}UNIV.\ K\ s)^*\ ``\ I$
shows *Time-Homogeneous-Discrete-Markov-Process* $(T'\ I)\ U\ (\lambda n\ \omega.\ \omega\ !!\ n)$
⟨*proof*⟩

end

4 Classifying Markov Chain States

theory *Classifying-Markov-Chain-States*
imports
HOL−Computational-Algebra.Group-Closure
Discrete-Time-Markov-Chain
begin

lemma *eventually-mult-Gcd*:
fixes $S :: nat\ set$
assumes S: $\bigwedge s\ t.\ s \in S \Longrightarrow t \in S \Longrightarrow s + t \in S$
assumes s: $s \in S\ s > 0$
shows *eventually* $(\lambda m.\ m * Gcd\ S \in S)$ *sequentially*
⟨*proof*⟩

context *MC-syntax*
begin

4.1 Expected number of visits

definition $G\ s\ t = (\int^+\omega.\ scount\ (HLD\ \{t\})\ (s\ \#\#\ \omega)\ \partial T\ s)$

lemma *G-eq*: $G\ s\ t = (\int^+\omega.\ emeasure\ (count\text{-}space\ UNIV)\ \{i.\ (s\ \#\#\ \omega)\ !!\ i = t\}\ \partial T\ s)$
⟨*proof*⟩

definition $p\ s\ t\ n = \mathcal{P}(\omega\ in\ T\ s.\ (s\ \#\#\ \omega)\ !!\ n = t)$

definition *gf-G* $s\ t\ z = (\sum n.\ p\ s\ t\ n *_R z \hat{}\ n)$

definition *convergence-G* $s\ t\ z \longleftrightarrow summable\ (\lambda n.\ p\ s\ t\ n * norm\ z \hat{}\ n)$

lemma *p-nonneg*[*simp*]: $0 \le p\ x\ y\ n$
⟨*proof*⟩

lemma *p-le-1*: $p\ x\ y\ n \leq 1$
⟨*proof*⟩

lemma *p-x-x-0*[*simp*]: $p\ x\ x\ 0 = 1$
⟨*proof*⟩

lemma *p-0*: $p\ x\ y\ 0 = (\mathit{if}\ x = y\ \mathit{then}\ 1\ \mathit{else}\ 0)$
⟨*proof*⟩

lemma *p-in-reachable*: **assumes** $(x, y) \notin (\mathit{SIGMA}\ x{:}\mathit{UNIV}.\ K\ x)^*$ **shows** $p\ x\ y\ n = 0$
⟨*proof*⟩

lemma *p-Suc*: $\mathit{ennreal}\ (p\ x\ y\ (\mathit{Suc}\ n)) = (\int^+ w.\ p\ w\ y\ n\ \partial K\ x)$
⟨*proof*⟩

lemma *p-Suc'*:
$p\ x\ y\ (\mathit{Suc}\ n) = (\int x'.\ p\ x'\ y\ n\ \partial K\ x)$
⟨*proof*⟩

lemma *p-add*: $p\ x\ y\ (n + m) = (\int^+ w.\ p\ x\ w\ n * p\ w\ y\ m\ \partial \mathit{count\text{-}space}\ \mathit{UNIV})$
⟨*proof*⟩

lemma *prob-reachable-le*:
assumes [*simp*]: $m \leq n$
shows $p\ x\ y\ m * p\ y\ w\ (n - m) \leq p\ x\ w\ n$
⟨*proof*⟩

lemma *G-eq-suminf*: $G\ x\ y = (\sum i.\ \mathit{ennreal}\ (p\ x\ y\ i))$
⟨*proof*⟩

lemma *G-eq-real-suminf*:
$\mathit{convergence\text{-}G}\ x\ y\ (1{::}\mathit{real}) \Longrightarrow G\ x\ y = \mathit{ennreal}\ (\sum i.\ p\ x\ y\ i)$
⟨*proof*⟩

lemma *convergence-norm-G*:
$\mathit{convergence\text{-}G}\ x\ y\ z \Longrightarrow \mathit{summable}\ (\lambda n.\ p\ x\ y\ n * \mathit{norm}\ z\ \widehat{}\ n)$
⟨*proof*⟩

lemma *convergence-G*:
$\mathit{convergence\text{-}G}\ x\ y\ (z{::}'a{::}\{\mathit{banach}, \mathit{real\text{-}normed\text{-}div\text{-}algebra}\}) \Longrightarrow \mathit{summable}\ (\lambda n.\ p\ x\ y\ n *_R z\ \widehat{}\ n)$
⟨*proof*⟩

lemma *convergence-G-less-1*:
fixes $z :: - :: \{\mathit{banach}, \mathit{real\text{-}normed\text{-}field}\}$
assumes z: $\mathit{norm}\ z < 1$ **shows** $\mathit{convergence\text{-}G}\ x\ y\ z$
⟨*proof*⟩

lemma *lim-gf-G*: $((\lambda z.\ ennreal\ (gf\text{-}G\ x\ y\ z)) \longrightarrow G\ x\ y)\ (at\text{-}left\ (1::real))$
⟨*proof*⟩

4.2 Reachability probability

definition $u\ x\ y\ n = \mathcal{P}(\omega\ in\ T\ x.\ ev\text{-}at\ (HLD\ \{y\})\ n\ \omega)$

definition $U\ s\ t = \mathcal{P}(\omega\ in\ T\ s.\ ev\ (HLD\ \{t\})\ \omega)$

definition $gf\text{-}U\ x\ y\ z = (\sum n.\ u\ x\ y\ n *_R z \mathbin{\hat{}} Suc\ n)$

definition $f\ x\ y\ n = \mathcal{P}(\omega\ in\ T\ x.\ ev\text{-}at\ (HLD\ \{y\})\ n\ (x\ \#\#\ \omega))$

definition $F\ s\ t = \mathcal{P}(\omega\ in\ T\ s.\ ev\ (HLD\ \{t\})\ (s\ \#\#\ \omega))$

definition $gf\text{-}F\ x\ y\ z = (\sum n.\ f\ x\ y\ n * z \mathbin{\hat{}} n)$

lemma *f-Suc*: $x \neq y \Longrightarrow f\ x\ y\ (Suc\ n) = u\ x\ y\ n$
⟨*proof*⟩

lemma *f-Suc-eq*: $f\ x\ x\ (Suc\ n) = 0$
⟨*proof*⟩

lemma *f-0*: $f\ x\ y\ 0 = (if\ x = y\ then\ 1\ else\ 0)$
⟨*proof*⟩

lemma shows *u-nonneg*: $0 \leq u\ x\ y\ n$ **and** *u-le-1*: $u\ x\ y\ n \leq 1$
⟨*proof*⟩

lemma shows *f-nonneg*: $0 \leq f\ x\ y\ n$ **and** *f-le-1*: $f\ x\ y\ n \leq 1$
⟨*proof*⟩

lemma *U-nonneg*[*simp*]: $0 \leq U\ x\ y$
⟨*proof*⟩

lemma *U-le-1*: $U\ s\ t \leq 1$
⟨*proof*⟩

lemma *U-cases*: $U\ s\ s = 1 \vee U\ s\ s < 1$
⟨*proof*⟩

lemma *u-sums-U*: $u\ x\ y\ sums\ U\ x\ y$
⟨*proof*⟩

lemma *gf-U-eq-U*: $gf\text{-}U\ x\ y\ 1 = U\ x\ y$
⟨*proof*⟩

lemma *f-sums-F*: $f\ x\ y\ sums\ F\ x\ y$
⟨*proof*⟩

lemma *F-nonneg*[*simp*]: $0 \leq F\ x\ y$
⟨*proof*⟩

lemma *F-le-1*: $F\ x\ y \leq 1$
⟨*proof*⟩

lemma *gf-F-eq-F*: $gf\text{-}F\ x\ y\ 1 = F\ x\ y$
⟨*proof*⟩

lemma *gf-F-le-1*:
fixes $z :: real$
assumes z: $0 \leq z\ z \leq 1$
shows $gf\text{-}F\ x\ y\ z \leq 1$
⟨*proof*⟩

lemma *u-le-p*: $u\ x\ y\ n \leq p\ x\ y\ (Suc\ n)$
⟨*proof*⟩

lemma *f-le-p*: $f\ x\ y\ n \leq p\ x\ y\ n$
⟨*proof*⟩

lemma *convergence-norm-U*:
fixes $z :: - :: real\text{-}normed\text{-}div\text{-}algebra$
assumes z: $convergence\text{-}G\ x\ y\ z$
shows $summable\ (\lambda n.\ u\ x\ y\ n * norm\ z \mathbin{\hat{}} Suc\ n)$
⟨*proof*⟩

lemma *convergence-norm-F*:
fixes $z :: - :: real\text{-}normed\text{-}div\text{-}algebra$
assumes z: $convergence\text{-}G\ x\ y\ z$
shows $summable\ (\lambda n.\ f\ x\ y\ n * norm\ z \mathbin{\hat{}} n)$
⟨*proof*⟩

lemma *gf-G-nonneg*:
fixes $z :: real$
shows $0 \leq z \Longrightarrow z < 1 \Longrightarrow 0 \leq gf\text{-}G\ x\ y\ z$
⟨*proof*⟩

lemma *gf-F-nonneg*:
fixes $z :: real$
shows $0 \leq z \Longrightarrow z < 1 \Longrightarrow 0 \leq gf\text{-}F\ x\ y\ z$
⟨*proof*⟩

lemma *convergence-U*:
fixes $z :: - :: banach$
shows $convergence\text{-}G\ x\ y\ z \Longrightarrow summable\ (\lambda n.\ u\ x\ y\ n * z \mathbin{\hat{}} Suc\ n)$
⟨*proof*⟩

lemma *p-eq-sum-p-u*: $p\ x\ y\ (Suc\ n) = (\sum i{\leq}n.\ p\ y\ y\ (n - i) * u\ x\ y\ i)$
⟨*proof*⟩

lemma *p-eq-sum-p-f*: $p\ x\ y\ n = (\sum i{\leq}n.\ p\ y\ y\ (n - i) * f\ x\ y\ i)$
⟨*proof*⟩

lemma *gf-G-eq-gf-F*:
assumes *z*: $norm\ z < 1$
shows $gf\text{-}G\ x\ y\ z = gf\text{-}F\ x\ y\ z * gf\text{-}G\ y\ y\ z$
⟨*proof*⟩

lemma *gf-G-eq-gf-U*:
fixes $z :: 'z :: \{banach, real\text{-}normed\text{-}field\}$
assumes *z*: *convergence-G x x z*
shows $gf\text{-}G\ x\ x\ z = 1\ /\ (1 - gf\text{-}U\ x\ x\ z)\ gf\text{-}U\ x\ x\ z \neq 1$
⟨*proof*⟩

lemma *gf-U*: $(gf\text{-}U\ x\ y \longrightarrow U\ x\ y)\ (at\text{-}left\ 1)$
⟨*proof*⟩

lemma *gf-U-le-1*: **assumes** *z*: $0 < z\ z < 1$ **shows** $gf\text{-}U\ x\ y\ z \leq (1{::}real)$
⟨*proof*⟩

lemma *gf-F*: $(gf\text{-}F\ x\ y \longrightarrow F\ x\ y)\ (at\text{-}left\ 1)$
⟨*proof*⟩

lemma *U-bounded*: $0 \leq U\ x\ y\ U\ x\ y \leq 1$
⟨*proof*⟩

4.3 Recurrent states

definition *recurrent* :: $'s \Rightarrow bool$ **where**
$recurrent\ s \longleftrightarrow (AE\ \omega\ in\ T\ s.\ ev\ (HLD\ \{s\})\ \omega)$

lemma *recurrent-iff-U-eq-1*: $recurrent\ s \longleftrightarrow U\ s\ s = 1$
⟨*proof*⟩

definition $H\ s\ t = \mathcal{P}(\omega\ in\ T\ s.\ alw\ (ev\ (HLD\ \{t\}))\ \omega)$

lemma *H-eq*:
$recurrent\ s \longleftrightarrow H\ s\ s = 1$
$\neg\ recurrent\ s \longleftrightarrow H\ s\ s = 0$
$H\ s\ t = U\ s\ t * H\ t\ t$
⟨*proof*⟩

lemma *recurrent-iff-G-infinite*: $recurrent\ x \longleftrightarrow G\ x\ x = \infty$
⟨*proof*⟩

definition *communicating* :: $('s \times 's)\ set$ **where**

communicating = *acc* ∩ *acc*$^{-1}$

definition *essential-class* :: *'s set* ⇒ *bool* **where**
essential-class C ⟷ *C* ∈ *UNIV* // *communicating* ∧ *acc* `` *C* ⊆ *C*

lemma *accI-U*:
assumes *0* < *U x y* **shows** (*x*, *y*) ∈ *acc*
⟨*proof*⟩

lemma *accD-pos*:
assumes (*x*, *y*) ∈ *acc*
shows ∃ *n*. *0* < *p x y n*
⟨*proof*⟩

lemma *accI-pos*: *0* < *p x y n* ⟹ (*x*, *y*) ∈ *acc*
⟨*proof*⟩

lemma *recurrent-iffI-communicating*:
assumes (*x*, *y*) ∈ *communicating*
shows *recurrent x* ⟷ *recurrent y*
⟨*proof*⟩

lemma *recurrent-acc*:
assumes *recurrent x* (*x*, *y*) ∈ *acc*
shows *U y x* = *1 H y x* = *1 recurrent y* (*x*, *y*) ∈ *communicating*
⟨*proof*⟩

lemma *equiv-communicating*: *equiv UNIV communicating*
⟨*proof*⟩

lemma *recurrent-class*:
assumes *recurrent x*
shows *acc* `` {*x*} = *communicating* `` {*x*}
⟨*proof*⟩

lemma *irreduccible-recurrent-class*:
assumes *recurrent x* **shows** *acc* `` {*x*} ∈ *UNIV* // *communicating*
⟨*proof*⟩

lemma *essential-classI*:
assumes *C*: *C* ∈ *UNIV* // *communicating*
assumes *eq*: ⋀*x y*. *x* ∈ *C* ⟹ (*x*, *y*) ∈ *acc* ⟹ *y* ∈ *C*
shows *essential-class C*
⟨*proof*⟩

lemma *essential-recurrent-class*:
assumes *recurrent x* **shows** *essential-class* (*communicating* `` {*x*})
⟨*proof*⟩

lemma *essential-classD2*:
essential-class $C \Longrightarrow x \in C \Longrightarrow (x, y) \in acc \Longrightarrow y \in C$
⟨*proof*⟩

lemma *essential-classD3*:
essential-class $C \Longrightarrow x \in C \Longrightarrow y \in C \Longrightarrow (x, y) \in communicating$
⟨*proof*⟩

lemma *AE-acc*:
shows $AE\ \omega\ in\ T\ x.\ \forall m.\ (x, (x\ \#\#\ \omega)\ !!\ m) \in acc$
⟨*proof*⟩

lemma *finite-essential-class-imp-recurrent*:
assumes *C*: *essential-class C finite C* **and** *x*: $x \in C$
shows *recurrent x*
⟨*proof*⟩

lemma *irreducibleD*:
$C \in UNIV\ //\ communicating \Longrightarrow a \in C \Longrightarrow b \in C \Longrightarrow (a, b) \in communicating$
⟨*proof*⟩

lemma *irreducibleD2*:
$C \in UNIV\ //\ communicating \Longrightarrow a \in C \Longrightarrow (a, b) \in communicating \Longrightarrow b \in C$
⟨*proof*⟩

lemma *essential-class-iff-recurrent*:
$finite\ C \Longrightarrow C \in UNIV\ //\ communicating \Longrightarrow essential\text{-}class\ C \longleftrightarrow (\forall x{\in}C.\ recurrent\ x)$
⟨*proof*⟩

definition $U'\ x\ y = (\int{}^{+}\omega.\ eSuc\ (sfirst\ (HLD\ \{y\})\ \omega)\ \partial T\ x)$

lemma *U′-neq-zero*[*simp*]: $U'\ x\ y \neq 0$
⟨*proof*⟩

definition *gf-U′* $x\ y\ z = (\sum n.\ u\ x\ y\ n * Suc\ n * z\ \widehat{}\ n)$

definition *pos-recurrent* $x \longleftrightarrow recurrent\ x \wedge U'\ x\ x \neq \infty$

lemma *summable-gf-U′*:
assumes *z*: $norm\ z < 1$
shows $summable\ (\lambda n.\ u\ x\ y\ n * Suc\ n * z\ \widehat{}\ n)$
⟨*proof*⟩

lemma *gf-U′-nonneg*[*simp*]: $0 < z \Longrightarrow z < 1 \Longrightarrow 0 \leq gf\text{-}U'\ x\ y\ z$
⟨*proof*⟩

lemma *DERIV-gf-U*:

fixes z :: *real* **assumes** z: $0 < z\ z < 1$
shows *DERIV* (*gf-U x y*) z :> *gf-U′ x y z*
⟨*proof*⟩

lemma *sfirst-finiteI-recurrent*:
recurrent x ⟹ $(x, y) \in$ *acc* ⟹ *AE ω in T x. sfirst* (*HLD* {*y*}) $\omega < \infty$
⟨*proof*⟩

lemma *U′-eq-suminf*:
assumes x: *recurrent x* $(x, y) \in$ *acc*
shows *U′ x y* = $(\sum i.$ *ennreal* (*u x y i* ∗ *Suc i*))
⟨*proof*⟩

lemma *gf-U′-tendsto-U′*:
assumes x: *recurrent x* $(x, y) \in$ *acc*
shows $((\lambda z.$ *ennreal* (*gf-U′ x y z*)) ⟶ *U′ x y*) (*at-left 1*)
⟨*proof*⟩

lemma *one-le-integral-t*:
assumes x: *recurrent x* **shows** $1 \leq$ *U′ x x*
⟨*proof*⟩

lemma *gf-U′-pos*:
fixes z :: *real*
assumes z: $0 < z\ z < 1$ **and** *U x y* ≠ *0*
shows *0* < *gf-U′ x y z*
⟨*proof*⟩

lemma *inverse-gf-U′-tendsto*:
assumes *recurrent y*
shows $((\lambda x. -1 / -$ *gf-U′ y y x*) ⟶ *enn2real* (*1 / U′ y y*)) (*at-left* (*1*::*real*))
⟨*proof*⟩

lemma *gf-G-pos*:
fixes z :: *real*
assumes z: $0 < z\ z < 1$ **and** ∗: $(x, y) \in$ *acc*
shows *0* < *gf-G x y z*
⟨*proof*⟩

lemma *pos-recurrentI-communicating*:
assumes y: *pos-recurrent y* **and** x: $(y, x) \in$ *communicating*
shows *pos-recurrent x*
⟨*proof*⟩

lemma *pos-recurrent-iffI-communicating*:
$(y, x) \in$ *communicating* ⟹ *pos-recurrent y* ⟷ *pos-recurrent x*
⟨*proof*⟩

lemma *U-le-F*: *U x y* ≤ *F x y*

⟨*proof*⟩

lemma *not-empty-irreducible*: $C \in UNIV\ //\ communicating \Longrightarrow C \neq \{\}$
⟨*proof*⟩

4.4 Stationary distribution

definition *stat* :: *'s set* ⇒ *'s measure* **where**
stat C = *point-measure UNIV* (λ*x*. *indicator C x* / *U' x x*)

lemma *sets-stat*[*simp*]: *sets* (*stat C*) = *sets* (*count-space UNIV*)
⟨*proof*⟩

lemma *space-stat*[*simp*]: *space* (*stat C*) = *UNIV*
⟨*proof*⟩

lemma *stat-subprob*:
assumes *C*: *essential-class C* **and** *countable C* **and** *pos*: ∀ *c*∈*C*. *pos-recurrent c*
shows *emeasure* (*stat C*) *C* ≤ *1*
⟨*proof*⟩

lemma *emeasure-stat-not-C*:
assumes *y* ∉ *C*
shows *emeasure* (*stat C*) {*y*} = *0*
⟨*proof*⟩

definition *stationary-distribution* :: *'s pmf* ⇒ *bool* **where**
stationary-distribution N ⟷ *N* = *bind-pmf N K*

lemma *stationary-distributionI*:
assumes *le*: ⋀*y*. (∫ *x*. *pmf* (*K x*) *y* ∂*measure-pmf N*) ≤ *pmf N y*
shows *stationary-distribution N*
⟨*proof*⟩

lemma *stationary-distribution-iterate*:
assumes *N*: *stationary-distribution N*
shows *ennreal* (*pmf N y*) = (∫ $^+$*x*. *p x y n* ∂*N*)
⟨*proof*⟩

lemma *stationary-distribution-iterate'*:
assumes *stationary-distribution N*
shows *measure N* {*y*} = (∫ *x*. *p x y n* ∂*N*)
⟨*proof*⟩

lemma *stationary-distributionD*:
assumes *C*: *essential-class C countable C*
assumes *N*: *stationary-distribution N N* ⊆ *C*
shows ∀ *x*∈*C*. *pos-recurrent x measure-pmf N* = *stat C*
⟨*proof*⟩

lemma *measure-point-measure-singleton*:
$x \in A \Longrightarrow$ *measure* (*point-measure A X*) $\{x\}$ = *enn2real* (*X x*)
⟨*proof*⟩

lemma *stationary-distribution-imp-int-t*:
assumes *C*: *essential-class C countable C stationary-distribution N N* $\subseteq$ *C*
assumes *x*: $x \in C$ **shows** *U' x x* = *1 / ennreal* (*pmf N x*)
⟨*proof*⟩

definition *period-set x* = $\{i.\ 0 < i \wedge 0 < p\ x\ x\ i\ \}$
definition *period C* = (*SOME d*. $\forall x{\in}C.\ d$ = *Gcd* (*period-set x*))

lemma *Gcd-period-set-invariant*:
assumes *c*: $(x, y) \in$ *communicating*
shows *Gcd* (*period-set x*) = *Gcd* (*period-set y*)
⟨*proof*⟩

lemma *period-eq*:
assumes *C* $\in$ *UNIV // communicating x* $\in$ *C*
shows *period C* = *Gcd* (*period-set x*)
⟨*proof*⟩

definition *aperiodic C* $\longleftrightarrow$ *C* $\in$ *UNIV // communicating* $\wedge$ *period C* = *1*

definition *not-ephemeral C* $\longleftrightarrow$ *C* $\in$ *UNIV // communicating* $\wedge$ $\neg$ ($\exists x.$ *C* = $\{x\}$ $\wedge$ *p x x 1* = *0*)

lemma *not-ephemeralD*:
assumes *C*: *not-ephemeral C x* $\in$ *C*
shows $\exists\, n{>}0.\ 0 < p\ x\ x\ n$
⟨*proof*⟩

lemma *not-ephemeralD-pos-period*:
assumes *C*: *not-ephemeral C*
shows *0* < *period C*
⟨*proof*⟩

lemma *period-posD*:
assumes *C*: *C* $\in$ *UNIV // communicating* **and** *0* < *period C x* $\in$ *C*
shows $\exists\, n{>}0.\ 0 < p\ x\ x\ n$
⟨*proof*⟩

lemma *not-ephemeralD-pos-period'*:
assumes *C*: *C* $\in$ *UNIV // communicating*
shows *not-ephemeral C* $\longleftrightarrow$ *0* < *period C*
⟨*proof*⟩

lemma *eventually-periodic*:
assumes *C*: $C \in UNIV$ // *communicating* $0 < period\ C\ x \in C$
shows *eventually* $(\lambda m.\ 0 < p\ x\ x\ (m * period\ C))$ *sequentially*
⟨*proof*⟩

lemma *aperiodic-eventually-recurrent*:
$aperiodic\ C \longleftrightarrow C \in UNIV$ // *communicating* $\wedge$ $(\forall x{\in}C.\ eventually\ (\lambda m.\ 0 < p\ x\ x\ m)\ sequentially)$
⟨*proof*⟩

lemma *stationary-distributionD-emeasure*:
assumes *N*: *stationary-distribution N*
shows $emeasure\ N\ A = (\int^+ s.\ emeasure\ (K\ s)\ A\ \partial N)$
⟨*proof*⟩

lemma *communicatingD1*:
$C \in UNIV$ // $communicating \Longrightarrow (a,\ b) \in communicating \Longrightarrow a \in C \Longrightarrow b \in C$
⟨*proof*⟩

lemma *communicatingD2*:
$C \in UNIV$ // $communicating \Longrightarrow (a,\ b) \in communicating \Longrightarrow b \in C \Longrightarrow a \in C$
⟨*proof*⟩

lemma *acc-iff*: $(x,\ y) \in acc \longleftrightarrow (\exists n.\ 0 < p\ x\ y\ n)$
⟨*proof*⟩

lemma *communicating-iff*: $(x,\ y) \in communicating \longleftrightarrow (\exists n.\ 0 < p\ x\ y\ n) \wedge (\exists n.\ 0 < p\ y\ x\ n)$
⟨*proof*⟩

end

context *MC-pair*
begin

lemma *p-eq-p1-p2*:
$p\ (x1,\ x2)\ (y1,\ y2)\ n = K1.p\ x1\ y1\ n * K2.p\ x2\ y2\ n$
⟨*proof*⟩

lemma *P-accD*:
assumes $((x1,\ x2),\ (y1,\ y2)) \in acc$**shows** $(x1,\ y1) \in K1.acc\ (x2,\ y2) \in K2.acc$
⟨*proof*⟩

lemma *aperiodicI-pair*:
assumes *C1*: *K1.aperiodic C1* **and** *C2*: *K2.aperiodic C2*

shows *aperiodic* (*C1* × *C2*)
⟨*proof*⟩

lemma *stationary-distributionI-pair*:
assumes *N1*: *K1.stationary-distribution N1*
assumes *N2*: *K2.stationary-distribution N2*
shows *stationary-distribution* (*pair-pmf N1 N2*)
⟨*proof*⟩

end

context *MC-syntax*
begin

lemma *stationary-distribution-imp-limit*:
assumes *C*: *aperiodic C essential-class C countable C* **and** *N*: *stationary-distribution N N* ⊆ *C*
assumes [*simp*]: *y* ∈ *C*
shows (λ*n*. ∫ *x*. |*p y x n* − *pmf N x*| ∂*count-space C*) ⟶ *0*
(**is** *?L* ⟶ *0*)
⟨*proof*⟩

lemma *stationary-distribution-imp-p-limit*:
assumes *aperiodic C essential-class C* **and** [*simp*]: *countable C*
assumes *N*: *stationary-distribution N N* ⊆ *C*
assumes [*simp*]: *x* ∈ *C y* ∈ *C*
shows *p x y* ⟶ *pmf N y*
⟨*proof*⟩

end

lemma (**in** *MC-syntax*) *essential-classI2*:
assumes *X* ≠ {}
assumes *accI*: ⋀*x y*. *x* ∈ *X* ⟹ *y* ∈ *X* ⟹ (*x*, *y*) ∈ *acc*
assumes *ED*: ⋀*x y*. *x* ∈ *X* ⟹ *y* ∈ *set-pmf* (*K x*) ⟹ *y* ∈ *X*
shows *essential-class X*
⟨*proof*⟩

end

5 Markov Decision Processes

theory *Markov-Decision-Process*
imports *Discrete-Time-Markov-Chain*
begin

definition *some-elem s* = (*SOME x*. *x* ∈ *s*)

lemma *some-elem-ne*: *s* ≠ {} ⟹ *some-elem s* ∈ *s*

⟨*proof*⟩

5.1 Configurations

We want to construct a *non-free* codatatype $'s\ cfg = Cfg$ (*state*: $'s$) (*action*: $'s\ pmf$) (*cont*: $'s \Rightarrow 's\ cfg$). with the restriction $state\ (cont\ cfg\ s) = s$

hide-const *cont*

codatatype $'s\ scheduler = Scheduler$ (*action-sch*: $'s\ pmf$) (*cont-sch*: $'s \Rightarrow 's$ *scheduler*)

lemma *equivp-rel-prod*: *equivp R* $\Longrightarrow$ *equivp Q* $\Longrightarrow$ *equivp* (*rel-prod R Q*)
⟨*proof*⟩

coinductive *eq-scheduler* :: $'s$ *scheduler* $\Rightarrow$ $'s$ *scheduler* $\Rightarrow$ *bool*
where
$\bigwedge D$. *action-sch sc1* $= D \Longrightarrow$ *action-sch sc2* $= D \Longrightarrow$
$(\forall\, s{\in}D.$ *eq-scheduler* (*cont-sch sc1 s*) (*cont-sch sc2 s*)) $\Longrightarrow$ *eq-scheduler sc1 sc2*

lemma *eq-scheduler-refl*[*intro*]: *eq-scheduler sc sc*
⟨*proof*⟩

quotient-type $'s\ cfg = 's \times 's$ *scheduler* / *rel-prod* (=) *eq-scheduler*
⟨*proof*⟩

lift-definition *state* :: $'s\ cfg \Rightarrow 's$ **is** *fst*
⟨*proof*⟩

lift-definition *action* :: $'s\ cfg \Rightarrow 's\ pmf$ **is** $\lambda(s,\ sc).$ *action-sch sc*
⟨*proof*⟩

lift-definition *cont* :: $'s\ cfg \Rightarrow 's \Rightarrow 's\ cfg$ **is**
$\lambda(s,\ sc)\ t.$ *if* $t \in$ *action-sch sc then* $(t,$ *cont-sch sc* $t)$ *else*
$(t,$ *cont-sch sc* (*some-elem* (*action-sch sc*)))
⟨*proof*⟩

lift-definition *Cfg* :: $'s \Rightarrow 's\ pmf \Rightarrow ('s \Rightarrow 's\ cfg) \Rightarrow 's\ cfg$ **is**
$\lambda s\ D\ c.\ (s,$ *Scheduler D* $(\lambda t.$ *snd* $(c\ t)))$
⟨*proof*⟩

lift-definition *cfg-corec* :: $'s \Rightarrow ('a \Rightarrow 's\ pmf) \Rightarrow ('a \Rightarrow 's \Rightarrow 'a) \Rightarrow 'a \Rightarrow 's\ cfg$
is
$\lambda s\ D\ C\ x.\ (s,$ *corec-scheduler D* $(\lambda x\ s.$ *Inr* $(C\ x\ s))\ x)$ ⟨*proof*⟩

lemma *state-cont*[*simp*]: *state* (*cont cfg s*) $= s$
⟨*proof*⟩

lemma *state-Cfg*[*simp*]: *state* $(Cfg\ s\ d'\ c') = s$
⟨*proof*⟩

lemma *action-Cfg*[*simp*]: *action* (*Cfg s d′ c′*) = *d′*
⟨*proof*⟩

lemma *cont-Cfg*[*simp*]: *t* ∈ *set-pmf d′* ⟹ *state* (*c′ t*) = *t* ⟹ *cont* (*Cfg s d′ c′*) *t* = *c′ t*
⟨*proof*⟩

lemma *state-cfg-corec*[*simp*]: *state* (*cfg-corec s d c x*) = *s*
⟨*proof*⟩

lemma *action-cfg-corec*[*simp*]: *action* (*cfg-corec s d c x*) = *d x*
⟨*proof*⟩

lemma *cont-cfg-corec*[*simp*]: *t* ∈ *set-pmf* (*d x*) ⟹ *cont* (*cfg-corec s d c x*) *t* = *cfg-corec t d c* (*c x t*)
⟨*proof*⟩

lemma *cfg-coinduct*[*consumes 1*, *case-names state action cont*, *coinduct pred*]:
X c d ⟹ (⋀*c d*. *X c d* ⟹ *state c* = *state d*) ⟹ (⋀*c d*. *X c d* ⟹ *action c* = *action d*) ⟹
(⋀*c d t*. *X c d* ⟹ *t* ∈ *set-pmf* (*action c*) ⟹ *X* (*cont c t*) (*cont d t*)) ⟹ *c* = *d*
⟨*proof*⟩

coinductive *rel-cfg* :: (*′a* ⇒ *′b* ⇒ *bool*) ⇒ *′a cfg* ⇒ *′b cfg* ⇒ *bool* **for** *P* :: *′a* ⇒ *′b* ⇒ *bool*
where
P (*state cfg1*) (*state cfg2*) ⟹
rel-pmf (λ*s t*. *rel-cfg P* (*cont cfg1 s*) (*cont cfg2 t*)) (*action cfg1*) (*action cfg2*) ⟹
rel-cfg P cfg1 cfg2

lemma *rel-cfg-state*: *rel-cfg P cfg1 cfg2* ⟹ *P* (*state cfg1*) (*state cfg2*)
⟨*proof*⟩

lemma *rel-cfg-cont*:
rel-cfg P cfg1 cfg2 ⟹
rel-pmf (λ*s t*. *rel-cfg P* (*cont cfg1 s*) (*cont cfg2 t*)) (*action cfg1*) (*action cfg2*)
⟨*proof*⟩

lemma *rel-cfg-action*:
assumes *P*: *rel-cfg P cfg1 cfg2* **shows** *rel-pmf P* (*action cfg1*) (*action cfg2*)
⟨*proof*⟩

lemma *rel-cfg-eq*: *rel-cfg* (=) *cfg1 cfg2* ⟷ *cfg1* = *cfg2*
⟨*proof*⟩

5.2 Configuration with Memoryless Scheduler

definition *memoryless-on f s = cfg-corec s f* (λ- *t. t*) *s*

lemma
shows *state-memoryless-on*[*simp*]: *state* (*memoryless-on f s*) = *s*
and *action-memoryless-on*[*simp*]: *action* (*memoryless-on f s*) = *f s*
and *cont-memoryless-on*[*simp*]: *t* ∈ (*f s*) ⟹ *cont* (*memoryless-on f s*) *t* = *memoryless-on f t*
⟨*proof*⟩

definition *K-cfg* :: *'s cfg* ⇒ *'s cfg pmf* **where**
K-cfg cfg = *map-pmf* (*cont cfg*) (*action cfg*)

lemma *set-K-cfg*: *set-pmf* (*K-cfg cfg*) = *cont cfg* ‘ *set-pmf* (*action cfg*)
⟨*proof*⟩

lemma *nn-integral-K-cfg*: (∫⁺ *cfg. f cfg* ∂*K-cfg cfg*) = (∫⁺ *s. f* (*cont cfg s*) ∂*action cfg*)
⟨*proof*⟩

5.3 MDP Kernel and Induced Configurations

locale *Markov-Decision-Process* =
fixes *K* :: *'s* ⇒ *'s pmf set*
assumes *K-wf*: ⋀*s. K s* ≠ {}
begin

definition *E* = (*SIGMA s*:*UNIV*. ⋃ *D*∈*K s. set-pmf D*)

coinductive *cfg-onp* :: *'s* ⇒ *'s cfg* ⇒ *bool* **where**
⋀*s. state cfg* = *s* ⟹ *action cfg* ∈ *K s* ⟹ (⋀*t. t* ∈ *action cfg* ⟹ *cfg-onp t* (*cont cfg t*)) ⟹
cfg-onp s cfg

definition *cfg-on s* = {*cfg. cfg-onp s cfg*}

lemma
shows *cfg-onD-action*[*intro*, *simp*]: *cfg* ∈ *cfg-on s* ⟹ *action cfg* ∈ *K s*
and *cfg-onD-cont*[*intro*, *simp*]: *cfg* ∈ *cfg-on s* ⟹ *t* ∈ *action cfg* ⟹ *cont cfg t* ∈ *cfg-on t*
and *cfg-onD-state*[*simp*]: *cfg* ∈ *cfg-on s* ⟹ *state cfg* = *s*
and *cfg-onI*: *state cfg* = *s* ⟹ *action cfg* ∈ *K s* ⟹ (⋀*t. t* ∈ *action cfg* ⟹ *cont cfg t* ∈ *cfg-on t*) ⟹ *cfg* ∈ *cfg-on s*
⟨*proof*⟩

lemma *cfg-on-coinduct*[*coinduct set*: *cfg-on*]:
assumes *P s cfg*
assumes ⋀*cfg s. P s cfg* ⟹ *state cfg* = *s*
assumes ⋀*cfg s. P s cfg* ⟹ *action cfg* ∈ *K s*

assumes $\bigwedge cfg\ s\ t.\ P\ s\ cfg \Longrightarrow t \in action\ cfg \Longrightarrow P\ t\ (cont\ cfg\ t)$
shows $cfg \in cfg\text{-}on\ s$
⟨*proof*⟩

lemma *memoryless-on-cfg-onI*:
assumes $\bigwedge s.\ f\ s \in K\ s$
shows $memoryless\text{-}on\ f\ s \in cfg\text{-}on\ s$
⟨*proof*⟩

lemma *cfg-of-cfg-onI*:
$D \in K\ s \Longrightarrow (\bigwedge t.\ t \in D \Longrightarrow c\ t \in cfg\text{-}on\ t) \Longrightarrow Cfg\ s\ D\ c \in cfg\text{-}on\ s$
⟨*proof*⟩

definition $arb\text{-}act\ s = (SOME\ D.\ D \in K\ s)$

lemma *arb-actI*[*simp*]: $arb\text{-}act\ s \in K\ s$
⟨*proof*⟩

lemma *cfg-on-not-empty*[*intro*, *simp*]: $cfg\text{-}on\ s \neq \{\}$
⟨*proof*⟩

sublocale *MC*: *MC-syntax K-cfg* ⟨*proof*⟩

abbreviation *St* :: *'s stream measure* **where**
$St \equiv stream\text{-}space\ (count\text{-}space\ UNIV)$

5.4 Trace Space

definition $T\ cfg = distr\ (MC.T\ cfg)\ St\ (smap\ state)$

sublocale *T*: *prob-space T cfg* **for** *cfg*
⟨*proof*⟩

lemma *space-T*[*simp*]: $space\ (T\ cfg) = space\ St$
⟨*proof*⟩

lemma *sets-T*[*simp*]: $sets\ (T\ cfg) = sets\ St$
⟨*proof*⟩

lemma *measurable-T1*[*simp*]: $measurable\ (T\ cfg)\ N = measurable\ St\ N$
⟨*proof*⟩

lemma *measurable-T2*[*simp*]: $measurable\ N\ (T\ cfg) = measurable\ N\ St$
⟨*proof*⟩

lemma *nn-integral-T*:
assumes [*measurable*]: $f \in borel\text{-}measurable\ St$
shows $(\int^+ X.\ f\ X\ \partial T\ cfg) = (\int^+ cfg'.\ (\int^+ x.\ f\ (state\ cfg'\ \#\#\ x)\ \partial T\ cfg')\ \partial K\text{-}cfg\ cfg)$

⟨*proof*⟩

lemma *T-eq*:
T cfg = (*measure-pmf* (*K-cfg cfg*) ⤜ (λ*cfg'*. *distr* (*T cfg'*) *St* (λω. *state cfg'* ## ω)))
⟨*proof*⟩

lemma *T-memoryless-on*: *T* (*memoryless-on ct s*) = *MC-syntax.T ct s*
⟨*proof*⟩

lemma *nn-integral-T-lfp*:
assumes [*measurable*]: *case-prod g* ∈ *borel-measurable* (*count-space UNIV* ⨂$_M$ *borel*)
assumes *cont-g*: ⋀*s*. *sup-continuous* (*g s*)
assumes *int-g*: ⋀*f cfg*. *f* ∈ *borel-measurable* (*stream-space* (*count-space UNIV*)) ⟹
(∫$^+$ω. *g* (*state cfg*) (*f* ω) ∂*T cfg*) = *g* (*state cfg*) (∫$^+$ω. *f* ω ∂*T cfg*)
shows (∫$^+$ω. *lfp* (λ*f* ω. *g* (*shd* ω) (*f* (*stl* ω))) ω ∂*T cfg*) =
lfp (λ*f cfg*. ∫$^+$*t*. *g* (*state t*) (*f t*) ∂*K-cfg cfg*) *cfg*
⟨*proof*⟩

lemma *emeasure-Collect-T*:
assumes [*measurable*]: *Measurable.pred St P*
shows *emeasure* (*T cfg*) {*x*∈*space St*. *P x*} =
(∫$^+$*cfg'*. *emeasure* (*T cfg'*) {*x*∈*space St*. *P* (*state cfg'* ## *x*)} ∂*K-cfg cfg*)
⟨*proof*⟩

definition *E-sup* :: *'s* ⇒ (*'s stream* ⇒ *ennreal*) ⇒ *ennreal*
where
E-sup s f = (⨆ *cfg*∈*cfg-on s*. ∫$^+$*x*. *f x* ∂*T cfg*)

lemma *E-sup-const*: *0* ≤ *c* ⟹ *E-sup s* (λ-. *c*) = *c*
⟨*proof*⟩

lemma *E-sup-mult-right*:
assumes [*measurable*]: *f* ∈ *borel-measurable St* **and** [*simp*]: *0* ≤ *c*
shows *E-sup s* (λ*x*. *c* ∗ *f x*) = *c* ∗ *E-sup s f*
⟨*proof*⟩

lemma *E-sup-mono*:
(⋀ω. *f* ω ≤ *g* ω) ⟹ *E-sup s f* ≤ *E-sup s g*
⟨*proof*⟩

lemma *E-sup-add*:
assumes [*measurable*]: *f* ∈ *borel-measurable St g* ∈ *borel-measurable St*
shows *E-sup s* (λ*x*. *f x* + *g x*) ≤ *E-sup s f* + *E-sup s g*
⟨*proof*⟩

lemma *E-sup-add-left*:

assumes [*measurable*]: $f \in$ *borel-measurable St*
shows *E-sup s* $(\lambda x.\ f\ x + c) =$ *E-sup s f* $+ c$
⟨*proof*⟩

lemma *E-sup-add-right*:
$f \in$ *borel-measurable St* $\Longrightarrow$ *E-sup s* $(\lambda x.\ c + f\ x) = c +$ *E-sup s f*
⟨*proof*⟩

lemma *E-sup-SUP*:
assumes [*measurable*]: $\bigwedge i.\ f\ i \in$ *borel-measurable St* **and** [*simp*]: *incseq f*
shows *E-sup s* $(\lambda x.\ \bigsqcup i.\ f\ i\ x) = (\bigsqcup i.$ *E-sup s* $(f\ i))$
⟨*proof*⟩

lemma *E-sup-iterate*:
assumes [*measurable*]: $f \in$ *borel-measurable St*
shows *E-sup s f* $= (\bigsqcup D{\in}K\ s.\ \int^+ t.$ *E-sup t* $(\lambda\omega.\ f\ (t\ \#\#\ \omega))\ \partial$*measure-pmf D*)
⟨*proof*⟩

lemma *E-sup-bot*: *E-sup s* $\bot = 0$
⟨*proof*⟩

lemma *E-sup-lfp*:
fixes *g*
defines $l \equiv \lambda f\ \omega.\ g$ (*shd* ω) (f (*stl* ω))
assumes *measurable-g*[*measurable*]: *case-prod g* $\in$ *borel-measurable* (*count-space UNIV* $\bigotimes_M$ *borel*)
assumes *cont-g*: $\bigwedge s.$ *sup-continuous* ($g\ s$)
assumes *int-g*: $\bigwedge f$ *cfg*. $f \in$ *borel-measurable St* $\Longrightarrow$
($\int^+ \omega.\ g$ (*state cfg*) ($f\ \omega$) ∂T *cfg*) $= g$ (*state cfg*) (*integral*^{N} (T *cfg*) f)
shows $(\lambda s.$ *E-sup s* (*lfp l*)$) =$ *lfp* $(\lambda f\ s.\ \bigsqcup D{\in}K\ s.\ \int^+ t.\ g\ t\ (f\ t)\ \partial$*measure-pmf D*)
⟨*proof*⟩

definition *P-sup s P* $= (\bigsqcup$ *cfg*$\in$*cfg-on s. emeasure* (T *cfg*) $\{x{\in}$*space St*. *P x*$\})$

lemma *P-sup-eq-E-sup*:
assumes [*measurable*]: *Measurable.pred St P*
shows *P-sup s P* $=$ *E-sup s* (*indicator* $\{x{\in}$*space St*. *P x*$\}$)
⟨*proof*⟩

lemma *P-sup-True*[*simp*]: *P-sup t* $(\lambda\omega.$ *True*$) = 1$
⟨*proof*⟩

lemma *P-sup-False*[*simp*]: *P-sup t* $(\lambda\omega.$ *False*$) = 0$
⟨*proof*⟩

lemma *P-sup-SUP*:
fixes *P* :: *nat* $\Rightarrow$ *'s stream* $\Rightarrow$ *bool*

assumes *mono P* **and** *P*[*measurable*]: $\bigwedge i$. *Measurable.pred St* (*P i*)
shows *P-sup s* (λx. $\exists\, i$. *P i x*) = ($\bigsqcup i$. *P-sup s* (*P i*))
⟨*proof*⟩

lemma *P-sup-lfp*:
assumes *Q*: *sup-continuous Q*
assumes *f*: *f* ∈ *measurable St M*
assumes *Q-m*: $\bigwedge P$. *Measurable.pred M P* ⟹ *Measurable.pred M* (*Q P*)
shows *P-sup s* (λx. *lfp Q* (*f x*)) = ($\bigsqcup i$. *P-sup s* (λx. (*Q* ^^ *i*) ⊥ (*f x*)))
⟨*proof*⟩

lemma *P-sup-iterate*:
assumes [*measurable*]: *Measurable.pred St P*
shows *P-sup s P* = ($\bigsqcup D{\in}K$ *s*. $\int^+$ *t*. *P-sup t* ($\lambda\omega$. *P* (*t* ## *ω*)) ∂*measure-pmf D*)
⟨*proof*⟩

definition *E-inf s f* = ($\bigsqcap$ *cfg*∈*cfg-on s*. $\int^+ x$. *f x* ∂*T cfg*)

lemma *E-inf-const*: *0* ≤ *c* ⟹ *E-inf s* (λ-. *c*) = *c*
⟨*proof*⟩

lemma *E-inf-mono*:
($\bigwedge\omega$. *f ω* ≤ *g ω*) ⟹ *E-inf s f* ≤ *E-inf s g*
⟨*proof*⟩

lemma *E-inf-iterate*:
assumes [*measurable*]: *f* ∈ *borel-measurable St*
shows *E-inf s f* = ($\bigsqcap D{\in}K$ *s*. $\int^+$ *t*. *E-inf t* ($\lambda\omega$. *f* (*t* ## *ω*)) ∂*measure-pmf D*)
⟨*proof*⟩

lemma *emeasure-T-const*[*simp*]: *emeasure* (*T s*) (*space St*) = *1*
⟨*proof*⟩

lemma *E-inf-greatest*:
($\bigwedge$*cfg*. *cfg* ∈ *cfg-on s* ⟹ *x* ≤ ($\int^+ x$. *f x* ∂*T cfg*)) ⟹ *x* ≤ *E-inf s f*
⟨*proof*⟩

lemma *E-inf-lower2*:
cfg ∈ *cfg-on s* ⟹ ($\int^+ x$. *f x* ∂*T cfg*) ≤ *x* ⟹ *E-inf s f* ≤ *x*
⟨*proof*⟩

Maybe the following statement can be generalized to infinite *K s*.

lemma *E-inf-lfp*:
fixes *g*
defines *l* ≡ $\lambda f\ \omega$. *g* (*shd ω*) (*f* (*stl ω*))
assumes *measurable-g*[*measurable*]: *case-prod g* ∈ *borel-measurable* (*count-space UNIV* ⨂$_M$ *borel*)
assumes *cont-g*: $\bigwedge s$. *sup-continuous* (*g s*)

assumes *int-g*: $\bigwedge f\ cfg.\ f \in$ *borel-measurable St* $\Longrightarrow$
$(\int^+ \omega.\ g\ (state\ cfg)\ (f\ \omega)\ \partial T\ cfg) = g\ (state\ cfg)\ (integral^N\ (T\ cfg)\ f)$
assumes *K-finite*: $\bigwedge s.$ *finite* $(K\ s)$
shows $(\lambda s.$ *E-inf s* $(lfp\ l)) = lfp\ (\lambda f\ s. \prod D{\in}K\ s. \int^+ t.\ g\ t\ (f\ t)\ \partial measure\text{-}pmf\ D)$
⟨*proof*⟩

definition *P-inf s P* $= (\prod cfg{\in}cfg\text{-}on\ s.\ emeasure\ (T\ cfg)\ \{x{\in}space\ St.\ P\ x\})$

lemma *P-inf-eq-E-inf*:
assumes [*measurable*]: *Measurable.pred St P*
shows *P-inf s P* $=$ *E-inf s* $(indicator\ \{x{\in}space\ St.\ P\ x\})$
⟨*proof*⟩

lemma *P-inf-True*[*simp*]: *P-inf t* $(\lambda\omega.\ True) = 1$
⟨*proof*⟩

lemma *P-inf-False*[*simp*]: *P-inf t* $(\lambda\omega.\ False) = 0$
⟨*proof*⟩

lemma *P-inf-INF*:
fixes *P* :: *nat* $\Rightarrow$ *'s stream* $\Rightarrow$ *bool*
assumes *decseq P* **and** *P*[*measurable*]: $\bigwedge i.$ *Measurable.pred St* $(P\ i)$
shows *P-inf s* $(\lambda x. \forall i.\ P\ i\ x) = (\prod i.$ *P-inf s* $(P\ i))$
⟨*proof*⟩

lemma *P-inf-gfp*:
assumes *Q*: *inf-continuous Q*
assumes *f*: $f \in$ *measurable St M*
assumes *Q-m*: $\bigwedge P.$ *Measurable.pred M P* $\Longrightarrow$ *Measurable.pred M* $(Q\ P)$
shows *P-inf s* $(\lambda x.\ gfp\ Q\ (f\ x)) = (\prod i.$ *P-inf s* $(\lambda x.\ (Q\ \widehat{}\ i)\ \top\ (f\ x)))$
⟨*proof*⟩

lemma *P-inf-iterate*:
assumes [*measurable*]: *Measurable.pred St P*
shows *P-inf s P* $= (\prod D{\in}K\ s.\ \int^+ t.$ *P-inf t* $(\lambda\omega.\ P\ (t\ \#\#\ \omega))\ \partial measure\text{-}pmf\ D)$
⟨*proof*⟩

end

5.5 Finite MDPs

locale *Finite-Markov-Decision-Process* = *Markov-Decision-Process K* **for** *K* :: *'s* $\Rightarrow$ *'s pmf set* +
fixes *S* :: *'s set*
assumes *S-not-empty*: $S \neq \{\}$
assumes *S-finite*: *finite S*
assumes *K-closed*: $\bigwedge s.\ s \in S \Longrightarrow (\bigcup D{\in}K\ s.$ *set-pmf D*$) \subseteq S$
assumes *K-finite*: $\bigwedge s.\ s \in S \Longrightarrow$ *finite* $(K\ s)$

begin

lemma *action-closed*: $s \in S \Longrightarrow cfg \in cfg\text{-}on\ s \Longrightarrow t \in action\ cfg \Longrightarrow t \in S$
⟨*proof*⟩

lemma *set-pmf-closed*: $s \in S \Longrightarrow D \in K\ s \Longrightarrow t \in D \Longrightarrow t \in S$
⟨*proof*⟩

lemma *Pi-closed*: $ct \in Pi\ S\ K \Longrightarrow s \in S \Longrightarrow t \in ct\ s \Longrightarrow t \in S$
⟨*proof*⟩

lemma *E-closed*: $s \in S \Longrightarrow (s, t) \in E \Longrightarrow t \in S$
⟨*proof*⟩

lemma *set-pmf-finite*: $s \in S \Longrightarrow D \in K\ s \Longrightarrow finite\ D$
⟨*proof*⟩

definition *valid-cfg* = $(\bigcup s{\in}S.\ cfg\text{-}on\ s)$

lemma *valid-cfgI*: $s \in S \Longrightarrow cfg \in cfg\text{-}on\ s \Longrightarrow cfg \in valid\text{-}cfg$
⟨*proof*⟩

lemma *valid-cfgD*: $cfg \in valid\text{-}cfg \Longrightarrow cfg \in cfg\text{-}on\ (state\ cfg)$
⟨*proof*⟩

lemma
shows *valid-cfg-state-in-S*: $cfg \in valid\text{-}cfg \Longrightarrow state\ cfg \in S$
and *valid-cfg-action*: $cfg \in valid\text{-}cfg \Longrightarrow s \in action\ cfg \Longrightarrow s \in S$
and *valid-cfg-cont*: $cfg \in valid\text{-}cfg \Longrightarrow s \in action\ cfg \Longrightarrow cont\ cfg\ s \in valid\text{-}cfg$
⟨*proof*⟩

lemma *valid-K-cfg*[*intro*]: $cfg \in valid\text{-}cfg \Longrightarrow cfg' \in K\text{-}cfg\ cfg \Longrightarrow cfg' \in valid\text{-}cfg$
⟨*proof*⟩

definition *simple ct* = *memoryless-on* $(\lambda s.\ if\ s \in S\ then\ ct\ s\ else\ arb\text{-}act\ s)$

lemma *simple-cfg-on*[*simp*]: $ct \in Pi\ S\ K \Longrightarrow simple\ ct\ s \in cfg\text{-}on\ s$
⟨*proof*⟩

lemma *simple-valid-cfg*[*simp*]: $ct \in Pi\ S\ K \Longrightarrow s \in S \Longrightarrow simple\ ct\ s \in valid\text{-}cfg$
⟨*proof*⟩

lemma *cont-simple*[*simp*]: $s \in S \Longrightarrow t \in set\text{-}pmf\ (ct\ s) \Longrightarrow cont\ (simple\ ct\ s)\ t = simple\ ct\ t$
⟨*proof*⟩

lemma *state-simple*[*simp*]: $state\ (simple\ ct\ s) = s$
⟨*proof*⟩

lemma *action-simple*[*simp*]: $s \in S \Longrightarrow action\ (simple\ ct\ s) = ct\ s$
⟨*proof*⟩

lemma *simple-valid-cfg-iff*: $ct \in Pi\ S\ K \Longrightarrow simple\ ct\ s \in valid\text{-}cfg \longleftrightarrow s \in S$
⟨*proof*⟩

end

end
theory *MDP-Reachability-Problem*
imports *Markov-Decision-Process*
begin

inductive-set *directed-towards* :: $'a\ set \Rightarrow ('a \times 'a)\ set \Rightarrow 'a\ set$ **for** $A\ r$ **where**
start: $\bigwedge x.\ x \in A \Longrightarrow x \in directed\text{-}towards\ A\ r$
| *step*: $\bigwedge x\ y.\ y \in directed\text{-}towards\ A\ r \Longrightarrow (x, y) \in r \Longrightarrow x \in directed\text{-}towards\ A\ r$

hide-fact (**open**) *start step*

lemma *directed-towards-mono*:
assumes $s \in directed\text{-}towards\ A\ F\ F \subseteq G$ **shows** $s \in directed\text{-}towards\ A\ G$
⟨*proof*⟩

lemma *directed-eq-rtrancl*: $x \in directed\text{-}towards\ A\ r \longleftrightarrow (\exists\, a{\in}A.\ (x, a) \in r^*)$
⟨*proof*⟩

lemma *directed-eq-rtrancl-Image*: $directed\text{-}towards\ A\ r = (r^*)^{-1}\ ``\ A$
⟨*proof*⟩

locale *Reachability-Problem* = *Finite-Markov-Decision-Process K S* **for** $K :: 's \Rightarrow 's\ pmf\ set$ **and** $S +$
fixes *S1 S2* :: $'s\ set$
assumes *S1*: $S1 \subseteq S$ **and** *S2*: $S2 \subseteq S$ **and** *S1-S2*: $S1 \cap S2 = \{\}$
begin

lemma [*measurable*]:
$S \in sets\ (count\text{-}space\ UNIV)\ S1 \in sets\ (count\text{-}space\ UNIV)\ S2 \in sets\ (count\text{-}space\ UNIV)$
⟨*proof*⟩

definition
$v = (\lambda cfg{\in}valid\text{-}cfg.\ emeasure\ (T\ cfg)\ \{x{\in}space\ St.\ (HLD\ S1\ suntil\ HLD\ S2)\ (state\ cfg\ \#\#\ x)\})$

lemma *v-eq*: $cfg \in valid\text{-}cfg \Longrightarrow$
$v\ cfg = emeasure\ (T\ cfg)\ \{x{\in}space\ St.\ (HLD\ S1\ suntil\ HLD\ S2)\ (state\ cfg\ \#\#\ x)\}$
⟨*proof*⟩

lemma *real-v*: *cfg* ∈ *valid-cfg* ⟹ *enn2real* (*v cfg*) = $\mathcal{P}$(ω *in T cfg*. (*HLD S1 suntil HLD S2*) (*state cfg ## ω*))
⟨*proof*⟩

lemma *v-le-1*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* ≤ *1*
⟨*proof*⟩

lemma *v-neq-Pinf*[*simp*]: *cfg* ∈ *valid-cfg* ⟹ *v cfg* ≠ *top*
⟨*proof*⟩

lemma *v-1-AE*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* = *1* ⟷ (*AE ω in T cfg*. (*HLD S1 suntil HLD S2*) (*state cfg ## ω*))
⟨*proof*⟩

lemma *v-0-AE*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* = *0* ⟷ (*AE x in T cfg*. *not* (*HLD S1 suntil HLD S2*) (*state cfg ## x*))
⟨*proof*⟩

lemma *v-S2*[*simp*]: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∈ *S2* ⟹ *v cfg* = *1*
⟨*proof*⟩

lemma *v-nS12*[*simp*]: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∉ *S1* ⟹ *state cfg* ∉ *S2* ⟹ *v cfg* = *0*
⟨*proof*⟩

lemma *v-nS*[*simp*]: *cfg* ∉ *valid-cfg* ⟹ *v cfg* = *undefined*
⟨*proof*⟩

lemma *v-S1*:
assumes *cfg*[*simp*, *intro*]: *cfg* ∈ *valid-cfg* **and** *cfg-S1*[*simp*]: *state cfg* ∈ *S1*
shows *v cfg* = (∫⁺*s*. *v* (*cont cfg s*) ∂*action cfg*)
⟨*proof*⟩

lemma *real-v-integrable*:
integrable (*action cfg*) (λ*s*. *enn2real* (*v* (*cont cfg s*)))
⟨*proof*⟩

lemma *real-v-integral-eq*:
assumes *cfg*[*simp*]: *cfg* ∈ *valid-cfg*
shows *enn2real* (∫⁺ *s*. *v* (*cont cfg s*) ∂*action cfg*) = ∫ *s*. *enn2real* (*v* (*cont cfg s*)) ∂*action cfg*
⟨*proof*⟩

lemma *v-eq-0-coinduct*[*consumes 3*, *case-names valid nS2 cont*]:
assumes ∗: *P cfg*
assumes *valid*: ⋀*cfg*. *P cfg* ⟹ *cfg* ∈ *valid-cfg*
assumes *nS2*: ⋀*cfg*. *P cfg* ⟹ *state cfg* ∉ *S2*
assumes *cont*: ⋀*cfg cfg'*. *P cfg* ⟹ *state cfg* ∈ *S1* ⟹ *cfg'* ∈ *K-cfg cfg* ⟹ *P cfg'* ∨ *v cfg'* = *0*

shows *v cfg = 0*
⟨*proof*⟩

definition $p = (\lambda s{\in}S.\ P\text{-}sup\ s\ (\lambda\omega.\ (HLD\ S1\ suntil\ HLD\ S2)\ (s\ \#\#\ \omega)))$

lemma *p-eq-SUP-v*: $s \in S \Longrightarrow p\ s = \bigsqcup\ (v\ `\ cfg\text{-}on\ s)$
⟨*proof*⟩

lemma *v-le-p*: $cfg \in valid\text{-}cfg \Longrightarrow v\ cfg \leq p\ (state\ cfg)$
⟨*proof*⟩

lemma *p-eq-0-imp*: $cfg \in valid\text{-}cfg \Longrightarrow p\ (state\ cfg) = 0 \Longrightarrow v\ cfg = 0$
⟨*proof*⟩

lemma *p-eq-0-iff*: $s \in S \Longrightarrow p\ s = 0 \longleftrightarrow (\forall\, cfg{\in}cfg\text{-}on\ s.\ v\ cfg = 0)$
⟨*proof*⟩

lemma *p-le-1*: $s \in S \Longrightarrow p\ s \leq 1$
⟨*proof*⟩

lemma *p-undefined*[*simp*]: $s \notin S \Longrightarrow p\ s = undefined$
⟨*proof*⟩

lemma *p-not-inf*[*simp*]: $s \in S \Longrightarrow p\ s \neq top$
⟨*proof*⟩

lemma *p-S1*: $s \in S1 \Longrightarrow p\ s = (\bigsqcup D{\in}K\ s.\ \int^+ t.\ p\ t\ \partial measure\text{-}pmf\ D)$
⟨*proof*⟩

lemma *p-S2*[*simp*]: $s \in S2 \Longrightarrow p\ s = 1$
⟨*proof*⟩

lemma *p-nS12*: $s \in S \Longrightarrow s \notin S1 \Longrightarrow s \notin S2 \Longrightarrow p\ s = 0$
⟨*proof*⟩

lemma *p-pos*:
assumes $(s,\ t) \in (SIGMA\ s{:}S1.\ \bigcup D{\in}K\ s.\ set\text{-}pmf\ D)^*\ t \in S2$ **shows** $0 < p\ s$
⟨*proof*⟩

definition *F-sup* :: $('s \Rightarrow ennreal) \Rightarrow 's \Rightarrow ennreal$ **where**
$F\text{-}sup\ f = (\lambda s{\in}S.\ if\ s \in S2\ then\ 1\ else\ if\ s \in S1\ then\ SUP\ D{\in}K\ s.\ \int^+ t.\ f\ t\ \partial measure\text{-}pmf\ D\ else\ 0)$

lemma *F-sup-cong*: $(\bigwedge s.\ s \in S \Longrightarrow f\ s = g\ s) \Longrightarrow F\text{-}sup\ f\ s = F\text{-}sup\ g\ s$
⟨*proof*⟩

lemma *continuous-F-sup*: *sup-continuous F-sup*
⟨*proof*⟩

lemma *mono-F-sup*: *mono F-sup*
⟨*proof*⟩

lemma *lfp-F-sup-iterate*: *lfp F-sup* = (*SUP i*. (*F-sup* ^^ *i*) (λ*x*∈*S*. *0*))
⟨*proof*⟩

lemma *p-eq-lfp-F-sup*: *p* = *lfp F-sup*
⟨*proof*⟩

definition S_e = {*s*∈*S*. *p s* = *0*}

lemma S_e: $S_e \subseteq S$
⟨*proof*⟩

lemma *v-*S_e: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∈ S_e ⟹ *v cfg* = *0*
⟨*proof*⟩

lemma S_e*-nS2*: $S_e \cap$ *S2* = {}
⟨*proof*⟩

lemma S_e*-E1*: *s* ∈ $S_e \cap$ *S1* ⟹ (*s*, *t*) ∈ *E* ⟹ *t* ∈ S_e
⟨*proof*⟩

lemma S_e*-E2*: *s* ∈ *S1* ⟹ (⋀*t*. (*s*, *t*) ∈ *E* ⟹ *t* ∈ S_e) ⟹ *s* ∈ S_e
⟨*proof*⟩

lemma S_e*-E-iff*: *s* ∈ *S1* ⟹ *s* ∈ S_e ⟷ (∀ *t*. (*s*, *t*) ∈ *E* ⟶ *t* ∈ S_e)
⟨*proof*⟩

definition S_r = *S* − (S_e ∪ *S2*)

lemma S_r: $S_r \subseteq S$
⟨*proof*⟩

lemma S_r*-S1*: $S_r \subseteq$ *S1*
⟨*proof*⟩

lemma S_r*-eq*: S_r = *S1* − S_e
⟨*proof*⟩

lemma *v-neq-0-imp*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* ≠ *0* ⟹ *state cfg* ∈ S_r ∪ *S2*
⟨*proof*⟩

lemma *valid-cfg-action-in-K*: *cfg* ∈ *valid-cfg* ⟹ *action cfg* ∈ *K* (*state cfg*)
⟨*proof*⟩

lemma *K-cfg-E*: *cfg* ∈ *valid-cfg* ⟹ *cfg*′ ∈ *K-cfg cfg* ⟹ (*state cfg*, *state cfg*′) ∈ *E*

⟨*proof*⟩

lemma S_r*-directed-towards-S2*:
assumes *s*: $s \in S_r$
shows $s \in$ *directed-towards S2* $\{(s,\ t) \mid s\ t.\ s \in S_r \wedge (s,\ t) \in E\}$ (**is** $s \in ?D$)
⟨*proof*⟩

definition *proper ct* $\longleftrightarrow$ $ct \in Pi_E\ S\ K \wedge (\forall\, s{\in}S_r.\ v\ (simple\ ct\ s) > 0)$

lemma S_r*-nS2*: $s \in S_r \Longrightarrow s \notin S2$
⟨*proof*⟩

lemma *properD1*: *proper ct* $\Longrightarrow$ $ct \in Pi_E\ S\ K$
⟨*proof*⟩

lemma *proper-eq*:
assumes *ct*[*simp*, *intro*]: $ct \in Pi_E\ S\ K$
shows *proper ct* $\longleftrightarrow$ $S_r \subseteq$ *directed-towards S2* (*SIGMA s*:S_r. *ct s*)
(**is** - $\longleftrightarrow$ - $\subseteq$ *?D*)
⟨*proof*⟩

lemma *exists-proper*:
obtains *ct* **where** *proper ct*
⟨*proof*⟩

definition *l-desc X ct l s* $\longleftrightarrow$
$s \in$ *directed-towards S2* (*SIGMA s* : *X*. $\{l\ s\}$) $\wedge$
v (*simple ct s*) $\leq$ *v* (*simple ct* (*l s*)) $\wedge$
l s $\in$ *maximal* ($\lambda s.$ *v* (*simple ct s*)) (*ct s*)

lemma *exists-l-desc*:
assumes *ct*: *proper ct*
shows $\exists\, l{\in}S_r \to S_r \cup S2.\ \forall\, s{\in}S_r.$ *l-desc* S_r *ct l s*
⟨*proof*⟩

lemma *F-v-memoryless*:
obtains *ct* **where** $ct \in Pi_E\ S\ K$ *v*∘*simple ct* = *F-sup* (*v*∘*simple ct*)
⟨*proof*⟩

lemma *p-v-memoryless*:
obtains *ct* **where** $ct \in Pi_E\ S\ K\ p =$ *v*∘*simple ct*
⟨*proof*⟩

definition $n = (\lambda s{\in}S.$ *P-inf s* ($\lambda\omega.$ (*HLD S1 suntil HLD S2*) (*s ## ω*)))

lemma *n-eq-INF-v*: $s \in S \Longrightarrow n\ s = (\bigsqcap cfg{\in}cfg\text{-}on\ s.\ v\ cfg)$
⟨*proof*⟩

lemma *n-le-v*: $s \in S \Longrightarrow cfg \in cfg\text{-}on\ s \Longrightarrow n\ s \leq v\ cfg$

⟨*proof*⟩

lemma *n-eq-1-imp*: $s \in S \Longrightarrow cfg \in cfg\text{-}on\ s \Longrightarrow n\ s = 1 \Longrightarrow v\ cfg = 1$
⟨*proof*⟩

lemma *n-eq-1-iff*: $s \in S \Longrightarrow n\ s = 1 \longleftrightarrow (\forall\, cfg{\in}cfg\text{-}on\ s.\ v\ cfg = 1)$
⟨*proof*⟩

lemma *n-le-1*: $s \in S \Longrightarrow n\ s \leq 1$
⟨*proof*⟩

lemma *n-undefined*[*simp*]: $s \notin S \Longrightarrow n\ s = undefined$
⟨*proof*⟩

lemma *n-eq-0*: $s \in S \Longrightarrow cfg \in cfg\text{-}on\ s \Longrightarrow v\ cfg = 0 \Longrightarrow n\ s = 0$
⟨*proof*⟩

lemma *n-not-inf*[*simp*]: $s \in S \Longrightarrow n\ s \neq top$
⟨*proof*⟩

lemma *n-S1*: $s \in S1 \Longrightarrow n\ s = (\prod D{\in}K\ s.\ \int^+ t.\ n\ t\ \partial measure\text{-}pmf\ D)$
⟨*proof*⟩

lemma *n-S2*[*simp*]: $s \in S2 \Longrightarrow n\ s = 1$
⟨*proof*⟩

lemma *n-nS12*: $s \in S \Longrightarrow s \notin S1 \Longrightarrow s \notin S2 \Longrightarrow n\ s = 0$
⟨*proof*⟩

lemma *n-pos*:
assumes $P\ s\ s \in S1\ wf\ R$
assumes *cont*: $\bigwedge s\ D.\ P\ s \Longrightarrow s \in S1 \Longrightarrow D \in K\ s \Longrightarrow \exists\, w{\in}D.\ ((w,\ s) \in R \wedge w \in S1 \wedge P\ w) \vee 0 < n\ w$
shows $0 < n\ s$
⟨*proof*⟩

definition *F-inf* :: $('s \Rightarrow ennreal) \Rightarrow ('s \Rightarrow ennreal)$ **where**
$F\text{-}inf\ f = (\lambda s{\in}S.\ if\ s \in S2\ then\ 1\ else\ if\ s \in S1\ then\ (\prod D{\in}K\ s.\ \int^+ t.\ f\ t\ \partial measure\text{-}pmf\ D)\ else\ 0)$

lemma *F-inf-n*: $F\text{-}inf\ n = n$
⟨*proof*⟩

lemma *F-inf-nS*[*simp*]: $s \notin S \Longrightarrow F\text{-}inf\ f\ s = undefined$
⟨*proof*⟩

lemma *mono-F-inf*: *mono F-inf*
⟨*proof*⟩

lemma *S1-nS2*: $s \in S1 \Longrightarrow s \notin S2$
⟨*proof*⟩

lemma *n-eq-lfp-F-inf*: $n = lfp\ F\text{-}inf$
⟨*proof*⟩

lemma *real-n*: $s \in S \Longrightarrow ennreal\ (enn2real\ (n\ s)) = n\ s$
⟨*proof*⟩

lemma *real-p*: $s \in S \Longrightarrow ennreal\ (enn2real\ (p\ s)) = p\ s$
⟨*proof*⟩

lemma *p-ub*:
fixes x
assumes $s \in S$
assumes *solution*: $\bigwedge s\ D.\ s \in S1 \Longrightarrow D \in K\ s \Longrightarrow (\sum t \in S.\ pmf\ D\ t * x\ t) \leq x\ s$
assumes *solution-0*: $\bigwedge s.\ s \in S \Longrightarrow p\ s = 0 \Longrightarrow x\ s = 0$
assumes *solution-S2*: $\bigwedge s.\ s \in S2 \Longrightarrow x\ s = 1$
shows $enn2real\ (p\ s) \leq x\ s$ (**is** *?y s* ≤ -)
⟨*proof*⟩

lemma *n-lb*:
fixes x
assumes $s \in S$
assumes *solution*: $\bigwedge s\ D.\ s \in S1 \Longrightarrow D \in K\ s \Longrightarrow x\ s \leq (\sum t \in S.\ pmf\ D\ t * x\ t)$
assumes *solution-n0*: $\bigwedge s.\ s \in S \Longrightarrow n\ s = 0 \Longrightarrow x\ s = 0$
assumes *solution-S2*: $\bigwedge s.\ s \in S2 \Longrightarrow x\ s = 1$
shows $x\ s \leq enn2real\ (n\ s)$ (**is** - ≤ *?y s*)
⟨*proof*⟩

end

end

6 Discrete-time Markov Processes

In this file we construct discrete-time Markov processes, e.g. with arbitrary state spaces.

theory *Discrete-Time-Markov-Process*
imports *Markov-Models-Auxiliary*
begin

lemma *measure-eqI-PiM-sequence*:
fixes $M :: nat \Rightarrow 'a\ measure$
assumes ∗[*simp*]: $sets\ P = PiM\ UNIV\ M\ sets\ Q = PiM\ UNIV\ M$
assumes *eq*: $\bigwedge A\ n.\ (\bigwedge i.\ A\ i \in sets\ (M\ i)) \Longrightarrow$
$P\ (prod\text{-}emb\ UNIV\ M\ \{..n\}\ (Pi_E\ \{..n\}\ A)) = Q\ (prod\text{-}emb\ UNIV\ M\ \{..n\}$

$(Pi_E\ \{..n\}\ A))$
assumes A: *finite-measure* P
shows $P = Q$
⟨*proof*⟩

lemma *distr-cong-simp*:
$M = K \Longrightarrow$ *sets* $N =$ *sets* $L \Longrightarrow (\bigwedge x.\ x \in$ *space* M *=simp=>* $f\ x = g\ x) \Longrightarrow$ *distr* $M\ N\ f =$ *distr* $K\ L\ g$
⟨*proof*⟩

6.1 Constructing Discrete-Time Markov Processes

locale *discrete-Markov-process* =
fixes $M :: 'a$ *measure* **and** $K :: 'a \Rightarrow 'a$ *measure*
assumes K[*measurable*]: $K \in M \rightarrow_M$ *prob-algebra* M
begin

lemma *space-K*: $x \in$ *space* $M \Longrightarrow$ *space* $(K\ x) =$ *space* M
⟨*proof*⟩

lemma *sets-K*[*measurable-cong*]: $x \in$ *space* $M \Longrightarrow$ *sets* $(K\ x) =$ *sets* M
⟨*proof*⟩

lemma *prob-space-K*: $x \in$ *space* $M \Longrightarrow$ *prob-space* $(K\ x)$
⟨*proof*⟩

definition $K' :: 'a \Rightarrow$ *nat* $\Rightarrow$ (*nat* $\Rightarrow 'a) \Rightarrow 'a$ *measure*
where
$K'\ x\ n'\ \omega' = K\ (case\text{-}nat\ x\ \omega'\ n')$

lemma *IT-K'*:
assumes x: $x \in$ *space* M **shows** *Ionescu-Tulcea* $(K'\ x)$ $(\lambda\text{-}.\ M)$
⟨*proof*⟩

definition *lim-sequence* $:: 'a \Rightarrow (nat \Rightarrow 'a)$ *measure*
where
lim-sequence $x =$ *projective-family.lim UNIV* $(Ionescu\text{-}Tulcea.CI\ (K'\ x)\ (\lambda\text{-}.\ M))$ $(\lambda\text{-}.\ M)$

lemma
assumes x: $x \in$ *space* M
shows *space-lim-sequence*: *space* $(lim\text{-}sequence\ x) =$ *space* $(\Pi_M\ i \in UNIV.\ M)$
and *sets-lim-sequence*[*measurable-cong*]: *sets* $(lim\text{-}sequence\ x) =$ *sets* $(\Pi_M\ i \in UNIV.\ M)$
and *emeasure-lim-sequence-emb*: $\bigwedge J\ X.$ *finite* $J \Longrightarrow X \in$ *sets* $(\Pi_M\ j \in J.\ M) \Longrightarrow$
emeasure $(lim\text{-}sequence\ x)$ $(prod\text{-}emb\ UNIV\ (\lambda\text{-}.\ M)\ J\ X) =$
emeasure $(Ionescu\text{-}Tulcea.CI\ (K'\ x)\ (\lambda\text{-}.\ M)\ J)\ X$
and *emeasure-lim-sequence-emb-I0o*: $\bigwedge n\ X.\ X \in$ *sets* $(\Pi_M\ i \in \{0..<n\}.\ M)$

$\Longrightarrow$
emeasure (*lim-sequence x*) (*prod-emb UNIV* (λ-. *M*) {*0*..<*n*} *X*) =
emeasure (*Ionescu-Tulcea.C* (*K' x*) (λ-. *M*) *0 n* (λ*x*. *undefined*)) *X*
⟨*proof*⟩

lemma *lim-sequence*[*measurable*]: *lim-sequence* ∈ *M* →$_M$ *prob-algebra* (Π$_M$ *i*∈*UNIV*. *M*)
⟨*proof*⟩

lemma *step-C*:
assumes *x*: *x* ∈ *space M*
shows *Ionescu-Tulcea.C* (*K' x*) (λ-. *M*) *0 1* (λ-. *undefined*) ⤜ *Ionescu-Tulcea.C* (*K' x*) (λ-. *M*) *1 n* =
K x ⤜ (λ*y*. *Ionescu-Tulcea.C* (*K' x*) (λ-. *M*) *1 n* (*case-nat y* (λ-. *undefined*)))
⟨*proof*⟩

lemma *lim-sequence-eq*:
assumes *x*: *x* ∈ *space M*
shows *lim-sequence x* = *bind* (*K x*) (λ*y*. *distr* (*lim-sequence y*) (Π$_M$ *j*∈*UNIV*. *M*) (*case-nat y*))
(**is** - = *?B x*)
⟨*proof*⟩

lemma *AE-lim-sequence*:
assumes *x*[*simp*]: *x* ∈ *space M* **and** *P*[*measurable*]: *Measurable.pred* (Π$_M$ *i*∈*UNIV*. *M*) *P*
shows (*AE ω in lim-sequence x*. *P ω*) ⟷ (*AE y in K x*. *AE ω in lim-sequence y*. *P* (*case-nat y ω*))
⟨*proof*⟩

definition *lim-stream* :: *'a* ⇒ *'a stream measure*
where
lim-stream x = *distr* (*lim-sequence x*) (*stream-space M*) *to-stream*

lemma *space-lim-stream*: *space* (*lim-stream x*) = *streams* (*space M*)
⟨*proof*⟩

lemma *sets-lim-stream*[*measurable-cong*]: *sets* (*lim-stream x*) = *sets* (*stream-space M*)
⟨*proof*⟩

lemma *lim-stream*[*measurable*]: *lim-stream* ∈ *M* →$_M$ *prob-algebra* (*stream-space M*)
⟨*proof*⟩

lemma *space-stream-space-M-ne*: *x* ∈ *space M* ⟹ *space* (*stream-space M*) ≠ {}
⟨*proof*⟩

lemma *prob-space-lim-stream*: *x* ∈ *space M* ⟹ *prob-space* (*lim-stream x*)

⟨*proof*⟩

lemma *lim-stream-eq*:
assumes *x*: $x \in$ *space M*
shows *lim-stream x* = *do* { $y \leftarrow K\ x$; $\omega \leftarrow$ *lim-stream y*; *return* (*stream-space M*) (y ## ω) }
⟨*proof*⟩

lemma *AE-lim-stream*:
assumes *x*[*simp*]: $x \in$ *space M* **and** *P*[*measurable*]: *Measurable.pred* (*stream-space M*) *P*
shows (*AE* ω *in lim-stream x*. *P* ω) $\longleftrightarrow$ (*AE y in K x*. *AE* ω *in lim-stream y*. *P* (y ## ω))
⟨*proof*⟩

lemma *emeasure-lim-stream*:
assumes *x*[*measurable*, *simp*]: $x \in$ *space M* **and** *A*[*measurable*, *simp*]: *A* $\in$ *sets* (*stream-space M*)
shows *lim-stream x A* = ($\int^+ y$. *emeasure* (*lim-stream y*) (((##) *y*) −‘ *A* ∩ *space* (*stream-space M*)) *∂K x*)
⟨*proof*⟩

lemma *lim-stream-eq-coinduct*[*case-names in-space step*]:
fixes *R* :: *'a* ⇒ *'a stream measure* ⇒ *bool*
assumes *x*: *R x B x* $\in$ *space M*
assumes *R*: $\bigwedge x\ B.\ R\ x\ B \Longrightarrow \exists B' \in M \to_M$ *prob-algebra* (*stream-space M*).
(*AE y in K x*. *R y* (*B' y*) ∨ *lim-stream y* = *B' y*) ∧
B = *do* { $y \leftarrow K\ x$; $\omega \leftarrow B'\ y$; *return* (*stream-space M*) (y ## ω) }
shows *lim-stream x* = *B*
⟨*proof*⟩

lemma *prob-space-lim-sequence*: $x \in$ *space M* $\Longrightarrow$ *prob-space* (*lim-sequence x*)
⟨*proof*⟩

end

6.2 Strong Markov Property for Discrete-Time Markov Processes

The filtration adopted to streams, i.e. to the n-th projection.

definition *stream-filtration* :: *'a measure* ⇒ *enat* ⇒ *'a stream measure*
where *stream-filtration M n* = (*SUP* $i \in \{i::nat.\ i \le n\}$. *vimage-algebra* (*streams* (*space M*)) ($\lambda\omega$. ω !! *i*) *M*)

lemma *measurable-stream-filtration1*: *enat i* ≤ *n* $\Longrightarrow$ ($\lambda\omega$. ω !! *i*) $\in$ *stream-filtration M n* $\to_M$ *M*
⟨*proof*⟩

lemma *measurable-stream-filtration2*:

$f \in$ *space* $N \to$ *streams* (*space* M) $\Longrightarrow$ ($\bigwedge i$. *enat* $i \leq n \Longrightarrow$ (λx. $f\ x$!! i) $\in N \to_M M$) $\Longrightarrow f \in N \to_M$ *stream-filtration* $M\ n$
⟨*proof*⟩

lemma *space-stream-filtration*: *space* (*stream-filtration* $M\ n$) = *space* (*stream-space* M)
⟨*proof*⟩

lemma *sets-stream-filteration-le-stream-space*: *sets* (*stream-filtration* $M\ n$) $\subseteq$ *sets* (*stream-space* M)
⟨*proof*⟩

interpretation *stream-filtration*: *filtration space* (*stream-space* M) *stream-filtration* M
⟨*proof*⟩

lemma *measurable-stopping-time-stream*:
stopping-time (*stream-filtration* M) $T \Longrightarrow T \in$ *stream-space* $M \to_M$ *count-space UNIV*
⟨*proof*⟩

lemma *measurable-stopping-time-All-eq-0*:
assumes T: *stopping-time* (*stream-filtration* M) T
shows {x∈*space* M. ∀ω∈*streams* (*space* M). T (x ## ω) = 0} ∈ *sets* M
⟨*proof*⟩

lemma *stopping-time-0*:
assumes T: *stopping-time* (*stream-filtration* M) T
and x: $x \in$ *space* M **and** ω: $\omega \in$ *streams* (*space* M) T (x ## ω) > 0
and ω': $\omega' \in$ *streams* (*space* M)
shows T (x ## ω') > 0
⟨*proof*⟩

lemma *stopping-time-epred-SCons*:
assumes T: *stopping-time* (*stream-filtration* M) T
and x: $x \in$ *space* M **and** ω: $\omega \in$ *streams* (*space* M) T (x ## ω) > 0
shows *stopping-time* (*stream-filtration* M) ($\lambda\omega$. *epred* (T (x ## ω)))
⟨*proof*⟩

context *discrete-Markov-process*
begin

lemma *lim-stream-strong-Markov*:
assumes x: $x \in$ *space* M **and** T: *stopping-time* (*stream-filtration* M) T
shows *lim-stream* x =
lim-stream x ⤜ ($\lambda\omega$. *case* $T\ \omega$ *of*
enat $i \Rightarrow$ *distr* (*lim-stream* (ω !! i)) (*stream-space* M) ($\lambda\omega'$. *stake* (*Suc* i) ω @− ω')
| ∞ ⇒ *return* (*stream-space* M) ω)

(**is** - = *?L T x*)
⟨*proof*⟩

end

end

7 Continuous-time Markov chains

theory *Continuous-Time-Markov-Chain*
imports *Discrete-Time-Markov-Process Discrete-Time-Markov-Chain*
begin

7.1 Trace Operations: relate ('*a* × *real*) *stream* and *real* ⇒ '*a*

partial-function (*tailrec*) *trace-at* :: '*a* ⇒ (*real* × '*a*) *stream* ⇒ *real* ⇒ '*a*
where
trace-at s ω j = (*case ω of* (*t'*, *s'*)##*ω* ⇒ *if t'* ≤ *j then trace-at s' ω j else s*)

lemma *trace-at-simp*[*simp*]: *trace-at s* ((*t'*, *s'*)##*ω*) *j* = (*if t'* ≤ *j then trace-at s' ω j else s*)
⟨*proof*⟩

lemma *trace-at-eq*:
trace-at s ω j = (*case sfirst* (λ*x*. *j* < *fst* (*shd x*)) *ω of* ∞ ⇒ *undefined* | *enat i* ⇒ (*s* ## *smap snd ω*) !! *i*)
⟨*proof*⟩

lemma *trace-at-shift*: *trace-at s* (*smap* (λ(*t*, *s'*). (*t* + *t'*, *s'*)) *ω*) *t* = *trace-at s ω* (*t* − *t'*)
⟨*proof*⟩

primcorec *merge-at* :: (*real* × '*a*) *stream* ⇒ *real* ⇒ (*real* × '*a*) *stream* ⇒ (*real* × '*a*) *stream*
where
merge-at ω j ω' = (*case ω of* (*t*, *s*) ## *ω* ⇒ *if t* ≤ *j then* (*t*, *s*)##*merge-at ω j ω' else ω'*)

lemma *merge-at-simp*[*simp*]: *merge-at* (*x*##*ω*) *j ω'* = (*if fst x* ≤ *j then x*##*merge-at ω j ω' else ω'*)
⟨*proof*⟩

7.2 Exponential Distribution

definition *exponential* :: *real* ⇒ *real measure*
where
exponential l = *density lborel* (*exponential-density l*)

lemma *space-exponential*: *space* (*exponential l*) = *UNIV*

⟨*proof*⟩

lemma *sets-exponential*[*measurable-cong*]: *sets* (*exponential l*) = *sets borel*
⟨*proof*⟩

lemma *prob-space-exponential*: *0* < *l* ⟹ *prob-space* (*exponential l*)
⟨*proof*⟩

lemma *AE-exponential*: *0* < *l* ⟹ *AE x in exponential l*. *0* < *x*
⟨*proof*⟩

lemma *emeasure-exponential-Ioi-cutoff*:
assumes *0* < *l*
shows *emeasure* (*exponential l*) {*x* <..} = *exp* (− (*max 0 x*) ∗ *l*)
⟨*proof*⟩

lemma *emeasure-exponential-Ioi*:
0 < *l* ⟹ *0* ≤ *x* ⟹ *emeasure* (*exponential l*) {*x* <..} = *exp* (− *x* ∗ *l*)
⟨*proof*⟩

lemma *exponential-eq-stretch*:
assumes *0* < *l*
shows *exponential l* = *distr* (*exponential 1*) *borel* (λ*x*. (*1*/*l*) ∗ *x*)
⟨*proof*⟩

lemma *uniform-measure-exponential*:
assumes *0* < *l 0* ≤ *t*
shows *uniform-measure* (*exponential l*) {*t* <..} = *distr* (*exponential l*) *borel* ((+) *t*) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *emeasure-PiM-exponential-Ioi-finite*:
assumes *J* ⊆ *I finite J* ⋀*i*. *i* ∈ *I* ⟹ *0* < *R i 0* ≤ *x*
shows *emeasure* (Π$_M$ *i*∈*I*. *exponential* (*R i*)) (*prod-emb I* (λ*i*. *exponential* (*R i*)) *J* (Π$_E$ *j*∈*J*. {*x*<..})) = *exp* (− *x* ∗ (∑ *i*∈*J*. *R i*))
⟨*proof*⟩

lemma *emeasure-PiM-exponential-Ioi-sequence*:
assumes *R*: *summable R* ⋀*i*. *0* < *R i 0* ≤ *x*
shows *emeasure* (Π$_M$ *i*∈*UNIV*. *exponential* (*R i*)) (Π *i*∈*UNIV*. {*x*<..}) = *exp* (− *x* ∗ *suminf R*)
⟨*proof*⟩

lemma *emeasure-PiM-exponential-Ioi-countable*:
assumes *R*: *J* ⊆ *I countable J* ⋀*i*. *i* ∈ *I* ⟹ *0* < *R i 0* ≤ *x* **and** *finite*: *integrable* (*count-space J*) *R*
shows *emeasure* (Π$_M$ *i*∈*I*. *exponential* (*R i*)) (*prod-emb I* (λ*i*. *exponential* (*R i*)) *J* (Π$_E$ *j*∈*J*. {*x*<..})) =
exp (− *x* ∗ (*LINT i*|*count-space J*. *R i*))

⟨*proof*⟩

lemma *AE-PiM-exponential-suminf-infty*:
fixes $R :: nat \Rightarrow real$
assumes *R*: $\bigwedge n.\ 0 < R\ n$ **and** *finite*: $(\sum n.\ ennreal\ (1\ /\ R\ n)) = top$
shows $AE\ \omega\ in\ \Pi_M\ n \in UNIV.\ exponential\ (R\ n).\ (\sum n.\ ereal\ (\omega\ n)) = \infty$
⟨*proof*⟩

7.3 Transition Rates

locale *transition-rates* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow real$
assumes *R-nonneg*[*simp*]: $\bigwedge x\ y.\ 0 \le R\ x\ y$
assumes *R-diagonal-0*[*simp*]: $\bigwedge x.\ R\ x\ x = 0$
assumes *finite-weight*: $\bigwedge x.\ (\int^+ y.\ R\ x\ y\ \partial count\text{-}space\ UNIV) < \infty$
assumes *positive-weight*: $\bigwedge x.\ 0 < (\int^+ y.\ R\ x\ y\ \partial count\text{-}space\ UNIV)$
begin

abbreviation $S :: (real \times 'a)\ measure$
where $S \equiv (borel \bigotimes_M count\text{-}space\ UNIV)$

abbreviation $T :: (real \times 'a)\ stream\ measure$
where $T \equiv stream\text{-}space\ S$

abbreviation $I :: 'a \Rightarrow 'a\ set$
where $I\ x \equiv \{y.\ 0 < R\ x\ y\}$

lemma *I-countable*: $countable\ (I\ x)$
⟨*proof*⟩

definition *escape-rate* :: $'a \Rightarrow real$ **where**
$escape\text{-}rate\ x = \int y.\ R\ x\ y\ \partial count\text{-}space\ UNIV$

lemma *ennreal-escape-rate*: $ennreal\ (escape\text{-}rate\ x) = (\int^+ y.\ R\ x\ y\ \partial count\text{-}space\ UNIV)$
⟨*proof*⟩

lemma *escape-rate-pos*: $0 < escape\text{-}rate\ x$
⟨*proof*⟩

lemma *nonneg-escape-rate*[*simp*]: $0 \le escape\text{-}rate\ x$
⟨*proof*⟩

lemma *prob-space-exponential-escape-rate*: $prob\text{-}space\ (exponential\ (escape\text{-}rate\ x))$
⟨*proof*⟩

lemma *measurable-escape-rate*[*measurable*]: $escape\text{-}rate \in count\text{-}space\ UNIV \to_M borel$
⟨*proof*⟩

lemma *measurable-exponential-escape-rate*[*measurable*]: $(\lambda x.\ exponential\ (escape\text{-}rate\ x)) \in count\text{-}space\ UNIV \rightarrow_M prob\text{-}algebra\ borel$
⟨*proof*⟩

interpretation *pmf-as-function* ⟨*proof*⟩

lift-definition $J :: 'a \Rightarrow 'a\ pmf$ **is** $\lambda x\ y.\ \ R\ x\ y\ /\ escape\text{-}rate\ x$
⟨*proof*⟩

lemma *set-pmf-J*: $set\text{-}pmf\ (J\ x) = I\ x$
⟨*proof*⟩

interpretation *exp-esc*: *pair-prob-space distr* $(exponential\ (escape\text{-}rate\ x))$ *borel* $((+)\ t)\ J\ x$ **for** x
⟨*proof*⟩

7.4 Continuous-time Kernel

definition $K :: (real \times 'a) \Rightarrow (real \times 'a)\ measure$ **where**
$K = (\lambda(t,\ x).\ (distr\ (exponential\ (escape\text{-}rate\ x))\ borel\ ((+)\ t)) \bigotimes_M J\ x)$

interpretation *K*: *discrete-Markov-process borel* $\bigotimes_M$ *count-space UNIV K*
⟨*proof*⟩

interpretation *DTMC*: *MC-syntax J* ⟨*proof*⟩

lemma *in-space-S*[*simp*]: $x \in space\ S$
⟨*proof*⟩

lemma *in-space-T*[*simp*]: $x \in space\ T$
⟨*proof*⟩

lemma *in-space-lim-stream*: $\omega \in space\ (K.lim\text{-}stream\ x)$
⟨*proof*⟩

lemma *prob-space-K-lim*: $prob\text{-}space\ (K.lim\text{-}stream\ x)$
⟨*proof*⟩

definition *select-first* $:: 'a \Rightarrow ('a \Rightarrow real) \Rightarrow 'a \Rightarrow bool$
where $select\text{-}first\ x\ p\ y = (y \in I\ x \wedge (\forall y' \in I\ x - \{y\}.\ p\ y < p\ y'))$

lemma *select-firstD1*: $select\text{-}first\ x\ p\ y \Longrightarrow y \in I\ x$
⟨*proof*⟩

lemma *select-first-unique*:
assumes *y*: *select-first x p y1 select-first x p y2* **shows** *y1* = *y2*
⟨*proof*⟩

lemma *The-select-first*[*simp*]: *select-first x p y* $\Longrightarrow$ *The* (*select-first x p*) = *y*
⟨*proof*⟩

lemma *select-first-INF*:
select-first x p y $\Longrightarrow$ (*INF* $x{\in}I$ *x*. *p x*) = *p y*
⟨*proof*⟩

lemma *measurable-select-first*[*measurable*]:
(λ*p*. *select-first x p y*) ∈ (Π$_M$ $y{\in}I$ *x*. *borel*) →$_M$ *count-space UNIV*
⟨*proof*⟩

lemma *measurable-THE-select-first*[*measurable*]:
(λ*p*. *The* (*select-first x p*)) ∈ (Π$_M$ $y{\in}I$ *x*. *borel*) →$_M$ *count-space UNIV*
⟨*proof*⟩

lemma *sets-S-eq*: *sets S* = *sigma-sets UNIV* { {*t* ..} × *A* | *t A*. *A* ⊆ − *I x* ∨ (∃ $s{\in}I$ *x*. *A* = {*s*}) }
⟨*proof*⟩

7.5 Kernel equals Parallel Choice

abbreviation *PAR* :: *'a* ⇒ (*'a* ⇒ *real*) *measure*
where
PAR x ≡ (Π$_M$ $y{\in}I$ *x*. *exponential* (*R x y*))

lemma *PAR-least*:
assumes *y*: *y* ∈ *I x*
shows *PAR x* {*p*∈*space* (*PAR x*). *t* ≤ *p y* ∧ *select-first x p y*} =
emeasure (*exponential* (*escape-rate x*)) {*t* ..} ∗ *ennreal* (*pmf* (*J x*) *y*)
⟨*proof*⟩

lemma *AE-PAR-least*: *AE p in PAR x*. ∃ $y{\in}I$ *x*. *select-first x p y*
⟨*proof*⟩

lemma *K-alt*: *K* (*t*, *x*) = *distr* (Π$_M$ $y{\in}I$ *x*. *exponential* (*R x y*)) *S* (λ*p*. (*t* + (*INF* $y{\in}I$ *x*. *p y*), *The* (*select-first x p*))) (**is** - = *?R*)
⟨*proof*⟩

lemma *AE-K*: *AE y in K x*. *fst x* < *fst y* ∧ *snd y* ∈ *J* (*snd x*)
⟨*proof*⟩

lemma *AE-lim-stream*:
AE ω *in K.lim-stream x*. ∀ *i*. *snd* ((*x* ## ω) !! *i*) ∈ *DTMC.acc*‘‘{*snd x*} ∧ *snd* (ω !! *i*) ∈ *J* (*snd* ((*x* ## ω) !! *i*)) ∧ *fst* ((*x* ## ω) !! *i*) < *fst* (ω !! *i*)
(**is** *AE* ω *in K.lim-stream x*. ∀ *i*. *?P* ω *i*)
⟨*proof*⟩

lemma *measurable-merge-at*[*measurable*]: (λ(ω, ω′). *merge-at* ω *j* ω′) ∈ (*T* ⨂$_M$ *T*) →$_M$ *T*

⟨*proof*⟩

lemma *measurable-trace-at*[*measurable*]: (λ(*s*, *ω*). *trace-at s ω j*) ∈ (*count-space UNIV* ⨂$_M$ *T*) →$_M$ *count-space UNIV*
⟨*proof*⟩

lemma *measurable-trace-at'*: (λ((*s*, *j*), *ω*). *trace-at s ω j*) ∈ ((*count-space UNIV* ⨂$_M$ *borel*) ⨂$_M$ *T*) →$_M$ *count-space UNIV*
⟨*proof*⟩

lemma *K-time-split*:
assumes $t \leq j$ **and** [*measurable*]: $f \in S \to_M$ *borel*
shows $(\int^+ x.\ f\ x * \textit{indicator}\ \{j <..\}\ (\textit{fst}\ x)\ \partial K\ (t, s)) = (\int^+ x.\ f\ x\ \partial K\ (j, s)) * \textit{exponential}\ (\textit{escape-rate}\ s)\ \{j - t <..\}$
⟨*proof*⟩

lemma *K-in-space*[*simp*]: *K x* ∈ *space* (*prob-algebra S*)
⟨*proof*⟩

lemma *L-in-space*[*simp*]: *K.lim-stream x* ∈ *space* (*prob-algebra T*)
⟨*proof*⟩

7.6 Markov Chain Property

lemma *lim-time-split*:
$t \leq j \Longrightarrow$ *K.lim-stream* (*t*, *s*) = *do* { *ω* ← *K.lim-stream* (*t*, *s*) ; *ω'* ← *K.lim-stream* (*j*, *trace-at s ω j*) ; *return T* (*merge-at ω j ω'*)}
(**is** - ⟹ - = *?DO t s*)
⟨*proof*⟩

lemma *K-eq*: *K* (*t*, *s*) = *distr* (*exponential* (*escape-rate s*) ⨂$_M$ *J s*) *S* (λ(*t'*, *s*). (*t* + *t'*, *s*))
⟨*proof*⟩

lemma *K-shift*: *K* (*t* + *t'*, *s*) = *distr* (*K* (*t*, *s*)) *S* (λ(*t*, *s*). (*t* + *t'*, *s*))
⟨*proof*⟩

lemma *K-not-empty*: *space* (*K x*) ≠ {}
⟨*proof*⟩

lemma *lim-stream-not-empty*: *space* (*K.lim-stream x*) ≠ {}
⟨*proof*⟩

lemma *lim-shift*: — Generalize to bijective function on *K.lim-stream* invariant on *K*
K.lim-stream (*t* + *t'*, *s*) = *distr* (*K.lim-stream* (*t*, *s*)) *T* (*smap* (λ(*t*, *s*). (*t* + *t'*, *s*)))
(**is** - = *?D t s*)
⟨*proof*⟩

lemma *lim-0*: *K.lim-stream* (*t*, *s*) = *distr* (*K.lim-stream* (*0*, *s*)) *T* (*smap* (λ(*t'*, *s*). (*t'* + *t*, *s*)))
⟨*proof*⟩

7.7 Explosion time

definition *explosion* :: (*real* × *'a*) *stream* ⇒ *ereal*
where *explosion* ω = (*SUP i*. *ereal* (*fst* (ω !! *i*)))

lemma *ball-less-Suc-eq*: (∀ *i*<*Suc n*. *P i*) ⟷ (*P 0* ∧ (∀ *i*<*n*. *P* (*Suc i*)))
⟨*proof*⟩

lemma *lim-stream-timediff-eq-exponential-1*:
distr (*K.lim-stream ts*) (*PiM UNIV* (λ-. *borel*))
(λω *i*. *escape-rate* (*snd* ((*ts*##ω) !! *i*)) ∗ (*fst* (ω !! *i*) − *fst* ((*ts*##ω) !! *i*))) =
PiM UNIV (λ-. *exponential 1*)
(**is** *?D* = *?P*)
⟨*proof*⟩

lemma *AE-explosion-infty*:
assumes *bdd*: *bdd-above* (*range escape-rate*)
shows *AE* ω *in K.lim-stream x*. *explosion* ω = ∞
⟨*proof*⟩

7.8 Transition probability p_t

context
begin

declare [[*inductive-internals* = *true*]]

inductive *trace-in* :: *'a set* ⇒ *real* ⇒ *'a* ⇒ (*real* × *'a*) *stream* ⇒ *bool* **for** *S t*
where
t < *t'* ⟹ *s* ∈ *S* ⟹ *trace-in S t s* ((*t'*, *s'*)##ω)
| *t* ≥ *t'* ⟹ *trace-in S t s'* ω ⟹ *trace-in S t s* ((*t'*, *s'*)##ω)

end

lemma *trace-in-simps*[*simp*]:
trace-in ss t s (*x*##ω) = (*if t* < *fst x then s* ∈ *ss else trace-in ss t* (*snd x*) ω)
⟨*proof*⟩

lemma *trace-in-eq-lfp*:
trace-in ss t = *lfp* (λ*F s*. λ(*t'*, *s'*)##ω ⇒ *if t* < *t' then s* ∈ *ss else F s'* ω)
⟨*proof*⟩

lemma *trace-in-shiftD*: *trace-in ss t s* ω ⟹ *trace-in ss* (*t* + *t'*) *s* (*smap* (λ(*t*, *s'*). (*t* + *t'*, *s'*)) ω)
⟨*proof*⟩

lemma *trace-in-shift*[*simp*]: *trace-in ss t s* (*smap* (λ(*t*, *s'*). (*t* + *t'*, *s'*)) *ω*) ⟷ *trace-in ss* (*t* − *t'*) *s ω*
⟨*proof*⟩

lemma *measurable-trace-in'*:
Measurable.pred (*borel* ⨂$_M$ *count-space UNIV* ⨂$_M$ *T*) (λ(*t*, *s*, *ω*). *trace-in ss t s ω*)
(**is** *?M* (λ(*t*, *s*, *ω*). *trace-in ss t s ω*))
⟨*proof*⟩

lemma *measurable-trace-in*[*measurable* (*raw*)]:
assumes [*measurable*]: *f* ∈ *M* →$_M$ *borel g* ∈ *M* →$_M$ *count-space UNIV h* ∈ *M* →$_M$ *T*
shows *Measurable.pred M* (λ*x*. *trace-in ss* (*f x*) (*g x*) (*h x*))
⟨*proof*⟩

definition *p* :: *'a* ⇒ *'a* ⇒ *real* ⇒ *real*
where *p s s' t* = *P*(*ω in K.lim-stream* (*0*, *s*). *trace-in* {*s'*} *t s ω*)

lemma *p*[*measurable*]: (λ(*s*, *t*). *p s s' t*) ∈ (*count-space UNIV* ⨂$_M$ *borel*) →$_M$ *borel*
⟨*proof*⟩

lemma *p-nonpos*: **assumes** *t* ≤ *0* **shows** *p s s' t* = *of-bool* (*s* = *s'*)
⟨*proof*⟩

lemma *p-0*: *p s s' 0* = *of-bool* (*s* = *s'*)
⟨*proof*⟩

lemma *in-sets-T*[*measurable* (*raw*)]: *Measurable.pred T P* ⟹ {*ω*. *P ω*} ∈ *sets T*
⟨*proof*⟩

lemma *distr-id'*: *sets M* = *sets N* ⟹ *distr M N* (λ*x*. *x*) = *M*
⟨*proof*⟩

lemma *p-nonneg*[*simp*]: *0* ≤ *p s s' t*
⟨*proof*⟩

lemma *p-le-1*[*simp*]: *p s s' t* ≤ *1*
⟨*proof*⟩

lemma *p-eq*:
assumes *0* ≤ *t*
shows *p s s'' t* = (*of-bool* (*s* = *s''*) + (*LINT u*:{*0*..*t*}|*lborel*. *escape-rate s* ∗ *exp* (*escape-rate s* ∗ *u*) ∗ (*LINT s'*|*J s*. *p s' s'' u*))) / *exp* (*t* ∗ *escape-rate s*)
⟨*proof*⟩

lemma *continuous-on-p*: *continuous-on A* (*p s s'*)

⟨*proof*⟩

lemma *p-vector-derivative*: — Backward equation
assumes $0 \leq t$
shows (*p s s' has-vector-derivative* (*LINT s''*|*count-space UNIV*. *R s s''* ∗ *p s'' s' t*) − *escape-rate s* ∗ *p s s' t*)
(*at t within* {*0*..})
(**is** (- *has-vector-derivative ?A*) -)
⟨*proof*⟩

coinductive *wf-times* :: *real* ⇒ (*real* × *'a*) *stream* ⇒ *bool*
where
$t < t' \Longrightarrow$ *wf-times t' ω* ⟹ *wf-times t* ((*t'*, *s'*) *##* *ω*)

lemma *wf-times-simp*[*simp*]: *wf-times t* (*x ##* *ω*) ⟷ *t* < *fst x* ∧ *wf-times* (*fst x*) *ω*
⟨*proof*⟩

lemma *trace-in-merge-at*:
assumes *ω'*: *wf-times t' ω'*
shows *trace-in ss t x* (*merge-at ω t' ω'*) ⟷
(*if t* < *t' then trace-in ss t x ω else* ∃ *y*. *trace-in* {*y*} *t' x ω* ∧ *trace-in ss t y ω'*)
(**is** *?merge* ⟷ *?cases*)
⟨*proof*⟩

lemma *AE-lim-wf-times*: *AE ω in K.lim-stream* (*t*, *s*). *wf-times t ω*
⟨*proof*⟩

lemma *wf-times-shiftD*: *wf-times t'* (*smap* (λ(*t'*, *y*). (*t'* + *t*, *y*)) *ω*) ⟹ *wf-times* (*t'* − *t*) *ω*
⟨*proof*⟩

lemma *wf-times-shift*[*simp*]: *wf-times t'* (*smap* (λ(*t'*, *y*). (*t'* + *t*, *y*)) *ω*) = *wf-times* (*t'* − *t*) *ω*
⟨*proof*⟩

lemma *trace-in-unique*: *trace-in* {*y1*} *t x ω* ⟹ *trace-in* {*y2*} *t x ω* ⟹ *y1* = *y2*
⟨*proof*⟩

lemma *trace-at-eq*: *trace-in* {*z*} *t x ω* ⟹ *trace-at x ω t* = *z*
⟨*proof*⟩

lemma *AE-lim-acc*: *AE ω in K.lim-stream* (*t*, *x*). ∀ *t z*. *trace-in* {*z*} *t x ω* ⟶ (*x*, *z*) ∈ *DTMC.acc*
⟨*proof*⟩

lemma *p-add*:
assumes $0 \leq t$ $0 \leq t'$
shows *p x y* (*t* + *t'*) = (*LINT z*|*count-space* (*DTMC.acc*‘‘{*x*}). *p x z t* ∗ *p z y*

t')
⟨*proof*⟩

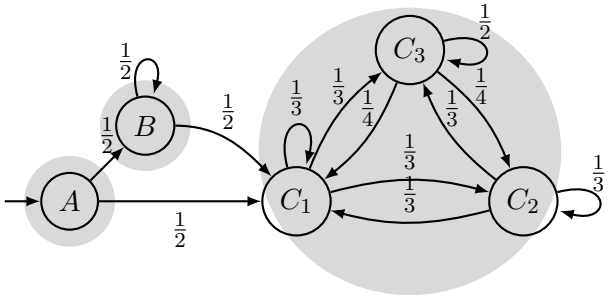
end

end
theory *Markov-Models*
imports
Markov-Models-Auxiliary
Discrete-Time-Markov-Chain
Trace-Space-Equals-Markov-Processes
Classifying-Markov-Chain-States
Markov-Decision-Process
MDP-Reachability-Problem
Discrete-Time-Markov-Process
Continuous-Time-Markov-Chain
begin

end
theory *Example-A*
imports *../Classifying-Markov-Chain-States*
begin

8 Example A

We formalize the following Markov chain:



First we define the state space as its own type:

datatype *state* = *A* | *B* | *C1* | *C2* | *C3*

Now the state space is *UNIV* :: *state set*

lemma *UNIV-state*: *UNIV* = {*A*, *B*, *C1*, *C2*, *C3*}
⟨*proof*⟩

instance *state* :: *finite*
⟨*proof*⟩

The transition function *tau* is easily defined using the case statement, this allows us to give a sparse specification as all *0* cases are collected at the end.

definition *tau* :: *state* ⇒ *state* ⇒ *real* **where**
tau s t = (*case* (*s*, *t*) *of*
(*A*, *B*) ⇒ *1 / 2* | (*A*, *C1*) ⇒ *1 / 2*
| (*B*, *B*) ⇒ *1 / 2* | (*B*, *C1*) ⇒ *1 / 2*
| (*C1*, *C1*) ⇒ *1 / 3* | (*C1*, *C2*) ⇒ *1 / 3* | (*C1*, *C3*) ⇒ *1 / 3*
| (*C2*, *C1*) ⇒ *1 / 3* | (*C2*, *C2*) ⇒ *1 / 3* | (*C2*, *C3*) ⇒ *1 / 3*
| (*C3*, *C1*) ⇒ *1 / 4* | (*C3*, *C2*) ⇒ *1 / 4* | (*C3*, *C3*) ⇒ *1 / 2*
| - ⇒ *0*)

lift-definition *K* :: *state* ⇒ *state pmf* **is** *tau*
⟨*proof*⟩

We use the *finite-pmf*-locale which introduces the point measure *tau.M*, and provides us with the necessary simplifier setup.

interpretation *A*: *MC-syntax K* ⟨*proof*⟩

8.1 The essential classs {*C1*, *C2*, *C3*}

context
begin

interpretation *pmf-as-function* ⟨*proof*⟩

lemma *A-E-eq*:
set-pmf (*K x*) = (*case x of A* ⇒ {*B*, *C1*} | *B* ⇒ {*B*, *C1*} | - ⇒ {*C1*, *C2*, *C3*})
⟨*proof*⟩

lemma *A-essential*: *A.essential-class* {*C1*, *C2*, *C3*}
⟨*proof*⟩

lemma *A-aperiodic*: *A.aperiodic* {*C1*, *C2*, *C3*}
⟨*proof*⟩

8.2 The stationary distribution *n*

Similar to *tau* we introduce *n* using the *finite-pmf*-locale.

lift-definition *n* :: *state pmf* **is** λ*C1* ⇒ *0.3* | *C2* ⇒ *0.3* | *C3* ⇒ *0.4* | - ⇒ *0*
⟨*proof*⟩

lemma *stationary-distribution-N*: *A.stationary-distribution n*
⟨*proof*⟩

lemma *exclusive-N*[*simp*]: *set-pmf n* = {*C1*, *C2*, *C3*}
⟨*proof*⟩

end

lemma *n-is-limit*:
assumes *x*: $x \in \{C1,\ C2,\ C3\}$ **and** *y*: $y \in \{C1,\ C2,\ C3\}$
shows $(A.p\ x\ y) \longrightarrow pmf\ n\ y$
⟨*proof*⟩

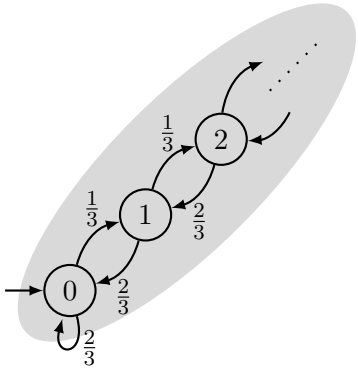
lemma *C-is-pos-recurrent*: $x \in \{C1,\ C2,\ C3\} \Longrightarrow A.pos\text{-}recurrent\ x$
⟨*proof*⟩

lemma *C-recurrence-time*:
assumes *x*: $x \in \{C1,\ C2,\ C3\}$
shows $A.U'\ x\ x = 1\ /\ pmf\ n\ x$
⟨*proof*⟩

end
theory *Example-B*
imports *../Classifying-Markov-Chain-States*
begin

9 Example B

We now formalize the following Markov chain:



As state space we have the set of natural numbers, the transition function *tau* has three cases:

definition $K :: nat \Rightarrow nat\ pmf$ **where**
$K\ x = map\text{-}pmf\ (\lambda True \Rightarrow x + 1 \mid False \Rightarrow x - 1)\ (bernoulli\text{-}pmf\ (1/3))$

For the special case when $x = 0$ we have $x - 1 = 0$ and hence $tau\ (0::'b)\ (0::'c) = (2::'a)\ /\ (3::'a)$.

We pack this transition function into a discrete Markov kernel.

We call the locale of the Markov chain *B*, hence all constants and theorems from this Markov chain get a *B* prefix.

interpretation *B*: *MC-syntax K* ⟨*proof*⟩

9.1 Enabled, accessible and communicating states

For each step the predecessor and the successor are enabled (in the $0::'a$ case, the predecessor is again $0::'a$. Hence every state is accessible from everywhere and every states is communicating with each other state. Finally we know that the state space is an essential class.

lemma *B-E-eq*: *set-pmf* $(K\ x) = \{x - 1, x + 1\}$
⟨*proof*⟩

lemma *B-E-Suc*: *Suc* $x \in$ *set-pmf* $(K\ x)$ $x \in$ *set-pmf* $(K\ (Suc\ x))$
⟨*proof*⟩

lemma *B-accessible*[*intro*]: $(i, j) \in B.acc$
⟨*proof*⟩

lemma *B-communicating*[*intro*]: $(i, j) \in B.communicating$
⟨*proof*⟩

lemma *B-essential*: *B.essential-class UNIV*
⟨*proof*⟩

9.2 B is aperiodic

lemma *B-aperiodic*: *B.aperiodic UNIV*
⟨*proof*⟩

9.3 The stationary distribution N

abbreviation $N ::$ *nat pmf* **where**
$N \equiv$ *geometric-pmf* $(1\ /\ 2)$

lemma *stationary-distribution-N*: *B.stationary-distribution N*
⟨*proof*⟩

9.4 Limit behavior and recurrence times

lemma *limit*: $(B.p\ i\ j) \longrightarrow (1/2)\hat{}Suc\ j$
⟨*proof*⟩

lemma *pos-recurrent*: *B.pos-recurrent i*
⟨*proof*⟩

lemma *recurrence-time*: $B.U'\ i\ i = 2\hat{}Suc\ i$
⟨*proof*⟩

end

theory *PCTL*
imports

../Discrete-Time-Markov-Chain
Gauss−Jordan−Elim−Fun.Gauss-Jordan-Elim-Fun
HOL−Library.While-Combinator
HOL−Library.Monad-Syntax
begin

10 Adapt Gauss-Jordan elimination to DTMCs

locale *Finite-DTMC* =
fixes $K :: 's \Rightarrow 's\ pmf$ **and** $S :: 's\ set$ **and** $\varrho :: 's \Rightarrow real$ **and** $\iota :: 's \Rightarrow 's \Rightarrow real$
assumes *ι-nonneg*[*simp*]: $\bigwedge s\ t.\ 0 \leq \iota\ s\ t$ **and** *ϱ-nonneg*[*simp*]: $\bigwedge s.\ 0 \leq \varrho\ s$
assumes *measurable-ι*: $(\lambda(a,\ b).\ \iota\ a\ b) \in$ *borel-measurable* (*count-space UNIV* $\bigotimes_M$ *count-space UNIV*)
assumes *finite-S*[*simp*]: *finite S* **and** *S-not-empty*: $S \neq \{\}$
assumes *E-closed*: $(\bigcup s{\in}S.\ set\text{-}pmf\ (K\ s)) \subseteq S$
begin

lemma *measurable-ι'*[*measurable* (*raw*)]:
$f \in$ *measurable M* (*count-space UNIV*) $\Longrightarrow$ $g \in$ *measurable M* (*count-space UNIV*) $\Longrightarrow$
$(\lambda x.\ \iota\ (f\ x)\ (g\ x)) \in$ *borel-measurable M*
⟨*proof*⟩

lemma *measurable-ϱ*[*measurable*]: $\varrho \in$ *borel-measurable* (*count-space UNIV*)
⟨*proof*⟩

sublocale *R?*: *MC-with-rewards K ι ϱ*
⟨*proof*⟩

lemma *single-l*:
fixes *s* **and** *x* :: *real* **assumes** $s \in S$
shows $(\sum s'{\in}S.\ (if\ s' = s\ then\ 1\ else\ 0) * l\ s') = x \longleftrightarrow l\ s = x$
⟨*proof*⟩

definition *order* = (*SOME f*. *bij-betw f* {..< *card S*} *S*)

lemma
shows *bij-order*[*simp*]: *bij-betw order* {..< *card S*} *S*
and *inj-order*[*simp*]: *inj-on order* {..<*card S*}
and *image-order*[*simp*]: *order* ' {..<*card S*} = *S*
and *order-S*[*simp*, *intro*]: $\bigwedge i.\ i < card\ S \Longrightarrow order\ i \in S$
⟨*proof*⟩

lemma *order-Ex*:
assumes $s \in S$ **obtains** *i* **where** $i < card\ S\ s = order\ i$
⟨*proof*⟩

definition *iorder* = *the-inv-into* {..<*card S*} *order*

lemma *bij-iorder*: *bij-betw iorder S* {..<*card S*}
⟨*proof*⟩

lemma *iorder-image-eq*: *iorder* ‘ *S* = {..<*card S*}
and *inj-iorder*: *inj-on iorder S*
⟨*proof*⟩

lemma *order-iorder*: ⋀*s*. *s* ∈ *S* ⟹ *order* (*iorder s*) = *s*
⟨*proof*⟩

definition *gauss-jordan'* :: ('*s* ⇒ '*s* ⇒ *real*) ⇒ ('*s* ⇒ *real*) ⇒ ('*s* ⇒ *real*) *option*
where
gauss-jordan' M a = *do* {
let M' = (λ*i j*. *if j* = *card S then a* (*order i*) *else M* (*order i*) (*order j*)) ;
sol ← *gauss-jordan M'* (*card S*) ;
Some (λ*i*. *sol* (*iorder i*) (*card S*))
}

lemma *gauss-jordan'-correct*:
assumes *gauss-jordan' M a* = *Some f*
shows ∀ *s*∈*S*. (∑ *s'*∈*S*. *M s s'* ∗ *f s'*) = *a s*
⟨*proof*⟩

lemma *gauss-jordan'-complete*:
assumes *exists*: ∀ *s*∈*S*. (∑ *s'*∈*S*. *M s s'* ∗ *x s'*) = *a s*
assumes *unique*: ⋀*y*. ∀ *s*∈*S*. (∑ *s'*∈*S*. *M s s'* ∗ *y s'*) = *a s* ⟹ ∀ *s*∈*S*. *y s* = *x s*
shows ∃ *y*. *gauss-jordan' M a* = *Some y*
⟨*proof*⟩

end

11 pCTL model checking

11.1 Syntax

datatype *realrel* = *LessEqual* | *Less* | *Greater* | *GreaterEqual* | *Equal*

datatype '*s sform* = *true*
| *Label* '*s set*
| *Neg* '*s sform*
| *And* '*s sform* '*s sform*
| *Prob realrel real* '*s pform*
| *Exp realrel real* '*s eform*
and '*s pform* = *X* '*s sform*
| *U nat* '*s sform* '*s sform*
| *UInfinity* '*s sform* '*s sform* (U^{∞})
and '*s eform* = *Cumm nat* ($C^{\leq}$)
| *State nat* ($I^{=}$)
| *Future* '*s sform*

primrec *bound-until* **where**
bound-until 0 φ ψ = ψ
| *bound-until* (*Suc n*) φ ψ = ψ *or* (φ *aand nxt* (*bound-until n* φ ψ))

lemma *measurable-bound-until*[*measurable*]:
assumes [*measurable*]: *Measurable.pred* (*stream-space M*) φ *Measurable.pred* (*stream-space M*) ψ
shows *Measurable.pred* (*stream-space M*) (*bound-until n* φ ψ)
⟨*proof*⟩

11.2 Semantics

primrec *inrealrel* :: *realrel* ⇒ 'a ⇒ ('a::*linorder*) ⇒ *bool* **where**
inrealrel LessEqual r q ⟷ $q \leq r$ |
inrealrel Less r q ⟷ $q < r$ |
inrealrel Greater r q ⟷ $q > r$ |
inrealrel GreaterEqual r q ⟷ $q \geq r$ |
inrealrel Equal r q ⟷ $q = r$

context *Finite-DTMC*
begin

abbreviation *prob s P* ≡ *measure* (*T s*) {x∈*space* (*T s*). *P x*}
abbreviation *E s* ≡ *set-pmf* (*K s*)

primrec *svalid* :: 's *sform* ⇒ 's *set*
and *pvalid* :: 's *pform* ⇒ 's *stream* ⇒ *bool*
and *reward* :: 's *eform* ⇒ 's *stream* ⇒ *ennreal* **where**
svalid true = *S* |
svalid (*Label L*) = {$s \in S.\ s \in L$} |
svalid (*Neg F*) = *S* − *svalid F* |
svalid (*And F1 F2*) = *svalid F1* ∩ *svalid F2* |
svalid (*Prob rel r F*) = {$s \in S.$ *inrealrel rel r* $\mathcal{P}(\omega$ *in T s. pvalid F* (*s* ## $\omega)$) } |
svalid (*Exp rel r F*) = {$s \in S.$ *inrealrel rel* (*ennreal r*) ($\int^+ \omega.$ *reward F* (*s* ## ω) ∂*T s*) } |

pvalid (*X F*) = *nxt* (*HLD* (*svalid F*)) |
pvalid (*U k F1 F2*) = *bound-until k* (*HLD* (*svalid F1*)) (*HLD* (*svalid F2*)) |
pvalid (U^∞ *F1 F2*) = *HLD* (*svalid F1*) *suntil HLD* (*svalid F2*) |

reward ($C^{\leq}$ *k*) = ($\lambda\omega.$ ($\sum i{<}k.$ ϱ (ω !! *i*) + ι (ω !! *i*) (ω !! (*Suc i*)))) |
reward ($I^{=}$ *k*) = ($\lambda\omega.$ ϱ (ω !! *k*)) |
reward (*Future F*) = ($\lambda\omega.$ *if ev* (*HLD* (*svalid F*)) ω *then reward-until* (*svalid F*) (*shd* ω) (*stl* ω) *else* ∞)

lemma *svalid-subset-S*: *svalid F* ⊆ *S*
⟨*proof*⟩

lemma *finite-svalid*[*simp*, *intro*]: *finite* (*svalid F*)
⟨*proof*⟩

lemma *svalid-sets*[*measurable*]: *svalid F* ∈ *sets* (*count-space S*)
⟨*proof*⟩

lemma *pvalid-sets*[*measurable*]: *Measurable.pred R.S* (*pvalid F*)
⟨*proof*⟩

lemma *reward-measurable*[*measurable*]: *reward F* ∈ *borel-measurable R.S*
⟨*proof*⟩

11.3 Implementation of *Sat*

11.3.1 *Prob0*

definition *Prob0* **where**
Prob0 Φ Ψ = *S* − *while* (λ*R*. ∃ *s*∈Φ. *R* ∩ *E s* ≠ {} ∧ *s* ∉ *R*) (λ*R*. *R* ∪ {*s*∈Φ. *R* ∩ *E s* ≠ {}}) Ψ

lemma *Prob0-subset-S*: *Prob0* Φ Ψ ⊆ *S*
⟨*proof*⟩

lemma *Prob0-iff-reachable*:
assumes Φ ⊆ *S* Ψ ⊆ *S*
shows *Prob0* Φ Ψ = {*s* ∈ *S*. ((*SIGMA x*:Φ. *E x*)* `` {*s*}) ∩ Ψ = {}} (**is** - = *?U*)
⟨*proof*⟩

lemma *Prob0-iff*:
assumes Φ ⊆ *S* Ψ ⊆ *S*
shows *Prob0* Φ Ψ = {*s*∈*S*. *AE* ω *in T s*. ¬ (*HLD* Φ *suntil HLD* Ψ) (*s* ## ω)} (**is** - = *?U*)
⟨*proof*⟩

lemma *E-rtrancl-closed*:
assumes *s* ∈ *S* (*s*, *t*) ∈ (*SIGMA x*:*A*. *B x*)* ⋀*x*. *x* ∈ *A* ⟹ *B x* ⊆ *E x* **shows** *t* ∈ *S*
⟨*proof*⟩

11.3.2 *Prob1*

definition *Prob1* **where**
Prob1 Y Φ Ψ = *Prob0* (Φ − Ψ) *Y*

lemma *Prob1-iff*:
assumes Φ ⊆ *S* Ψ ⊆ *S*
shows *Prob1* (*Prob0* Φ Ψ) Φ Ψ = {*s*∈*S*. *AE* ω *in T s*. (*HLD* Φ *suntil HLD* Ψ) (*s* ## ω)}
(**is** *Prob1 ?P0* - - = {*s*∈*S*. *?pU s*})
⟨*proof*⟩

11.3.3 *ProbU*, *ExpCumm*, and *ExpState*

abbreviation $\tau\ s\ t \equiv pmf\ (K\ s)\ t$

fun $ProbU :: 's \Rightarrow nat \Rightarrow 's\ set \Rightarrow 's\ set \Rightarrow real$ **where**
$ProbU\ q\ 0\ S1\ S2 = (if\ q \in S2\ then\ 1\ else\ 0)$ |
$ProbU\ q\ (Suc\ k)\ S1\ S2 =$
$(if\ q \in S1 - S2\ then\ (\sum q' \in S.\ \tau\ q\ q' * ProbU\ q'\ k\ S1\ S2)$
$else\ if\ q \in S2\ then\ 1\ else\ 0)$

fun $ExpCumm :: 's \Rightarrow nat \Rightarrow ennreal$ **where**
$ExpCumm\ s\ 0 = 0$ |
$ExpCumm\ s\ (Suc\ k) = \varrho\ s + (\sum s' \in S.\ \tau\ s\ s' * (\iota\ s\ s' + ExpCumm\ s'\ k))$

fun $ExpState :: 's \Rightarrow nat \Rightarrow ennreal$ **where**
$ExpState\ s\ 0 = \varrho\ s$ |
$ExpState\ s\ (Suc\ k) = (\sum s' \in S.\ \tau\ s\ s' * ExpState\ s'\ k)$

11.3.4 *LES*

definition $LES :: 's\ set \Rightarrow 's \Rightarrow 's \Rightarrow real$ **where**
$LES\ F\ r\ c =$
$(if\ r \in F\ then\ (if\ c = r\ then\ 1\ else\ 0)$
$else\ (if\ c = r\ then\ \tau\ r\ c - 1\ else\ \tau\ r\ c\ ))$

11.3.5 *ProbUinfty*, compute unbounded until

definition $ProbUinfty :: 's\ set \Rightarrow 's\ set \Rightarrow ('s \Rightarrow real)\ option$ **where**
$ProbUinfty\ S1\ S2 = gauss\text{-}jordan'\ (LES\ (Prob0\ S1\ S2 \cup S2))$
$(\lambda i.\ if\ i \in S2\ then\ 1\ else\ 0)$

11.3.6 *ExpFuture*, compute unbounded reward

definition $ExpFuture :: 's\ set \Rightarrow ('s \Rightarrow ennreal)\ option$ **where**
$ExpFuture\ F = do\ \{$
$let\ N = Prob0\ S\ F\ ;$
$let\ Y = Prob1\ N\ S\ F\ ;$
$sol \leftarrow gauss\text{-}jordan'\ (LES\ (S - Y \cup F))$
$(\lambda i.\ if\ i \in Y \wedge i \notin F\ then - \varrho\ i - (\sum s' \in S.\ \tau\ i\ s' * \iota\ i\ s')\ else\ 0)\ ;$
$Some\ (\lambda s.\ if\ s \in Y\ then\ ennreal\ (sol\ s)\ else\ \infty)$
$\}$

11.3.7 *Sat*

fun $Sat :: 's\ sform \Rightarrow 's\ set\ option$ **where**
$Sat\ true = Some\ S$ |
$Sat\ (Label\ L) = Some\ \{s \in S.\ s \in L\}$ |
$Sat\ (Neg\ F) = do\ \{\ F \leftarrow Sat\ F\ ;\ Some\ (S - F)\ \}$ |
$Sat\ (And\ F1\ F2) = do\ \{\ F1 \leftarrow Sat\ F1\ ;\ F2 \leftarrow Sat\ F2\ ;\ Some\ (F1 \cap F2)$
$\}$ |

Sat (Prob rel r (X F)) = *do* { *F* ← *Sat F* ; *Some* {*q* ∈ *S*. *inrealrel rel r* (∑ *q*′∈*F*. *τ q q*′)} } |
Sat (Prob rel r (U k F1 F2)) = *do* { *F1* ← *Sat F1* ; *F2* ← *Sat F2* ; *Some* {*q* ∈ *S*. *inrealrel rel r* (*ProbU q k F1 F2*) } } |
Sat (Prob rel r (U$^{\infty}$ *F1 F2))* = *do* { *F1* ← *Sat F1* ; *F2* ← *Sat F2* ; *P* ← *ProbUinfty F1 F2* ; *Some* {*q* ∈ *S*. *inrealrel rel r* (*P q*) } } |

Sat (Exp rel r (Cumm k)) = *Some* {*s* ∈ *S*. *inrealrel rel r* (*ExpCumm s k*) } |
Sat (Exp rel r (State k)) = *Some* {*s* ∈ *S*. *inrealrel rel r* (*ExpState s k*) } |
Sat (Exp rel r (Future F)) = *do* { *F* ← *Sat F* ; *E* ← *ExpFuture F* ; *Some* {*q* ∈ *S*. *inrealrel rel* (*ennreal r*) (*E q*) } }

lemma *prob-sum*:
s ∈ *S* ⟹ *Measurable.pred R.S P* ⟹ $\mathcal{P}$(*ω in T s*. *P ω*) = (∑ *t*∈*S*. *τ s t* ∗ $\mathcal{P}$(*ω in T t*. *P* (*t ## ω*)))
⟨*proof*⟩

lemma *nn-integral-eq-sum*:
s ∈ *S* ⟹ *f* ∈ *borel-measurable R.S* ⟹ (∫$^{+}$*x*. *f x* ∂*T s*) = (∑ *t*∈*S*. *τ s t* ∗ (∫$^{+}$*x*. *f* (*t ## x*) ∂*T t*))
⟨*proof*⟩

lemma *T-space*[*simp*]: *measure* (*T s*) (*space R.S*) = *1*
⟨*proof*⟩

lemma *emeasure-T-space*[*simp*]: *emeasure* (*T s*) (*space R.S*) = *1*
⟨*proof*⟩

lemma *τ-distr*[*simp*]: *s* ∈ *S* ⟹ (∑ *t*∈*S*. *τ s t*) = *1*
⟨*proof*⟩

lemma *ProbU*:
q ∈ *S* ⟹ *ProbU q k* (*svalid F1*) (*svalid F2*) = $\mathcal{P}$(*ω in T q*. *pvalid* (*U k F1 F2*) (*q ## ω*))
⟨*proof*⟩

lemma *Prob0-imp-not-Psi*:
assumes Φ ⊆ *S* Ψ ⊆ *S s* ∈ *Prob0* Φ Ψ **shows** *s* ∉ Ψ
⟨*proof*⟩

lemma *Psi-imp-not-Prob0*:
assumes Φ ⊆ *S* Ψ ⊆ *S* **shows** *s* ∈ Ψ ⟹ *s* ∉ *Prob0* Φ Ψ
⟨*proof*⟩

11.3.8 Finite expected reward

abbreviation *s0* ≡ *SOME s*. *s* ∈ *S*

lemma *s0-in-S*: $s0 \in S$
⟨*proof*⟩

lemma *nn-integral-reward-finite*:
assumes $s \in S$
assumes *until*: $AE\ \omega\ in\ T\ s.\ (HLD\ S\ suntil\ HLD\ (svalid\ F))\ (s\ \#\#\ \omega)$
shows $(\int^+ \omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) \neq \infty$
⟨*proof*⟩

lemma *unique*:
assumes *in-S*: $\Phi \subseteq S\ \Psi \subseteq S\ N \subseteq S\ Prob0\ \Phi\ \Psi \subseteq N\ \Psi \subseteq N$
assumes *l1*: $\bigwedge s.\ s \in S \Longrightarrow s \notin N \Longrightarrow l1\ s - c\ s = (\sum s' \in S.\ \tau\ s\ s' * l1\ s')$
assumes *l2*: $\bigwedge s.\ s \in S \Longrightarrow s \notin N \Longrightarrow l2\ s - c\ s = (\sum s' \in S.\ \tau\ s\ s' * l2\ s')$
assumes *eq*: $\bigwedge s.\ s \in N \Longrightarrow l1\ s = l2\ s$
shows $\forall s \in S.\ l1\ s = l2\ s$
⟨*proof*⟩

lemma *uniqueness-of-ProbU*:
assumes *sol*:
$\forall s \in S.\ (\sum s' \in S.\ LES\ (Prob0\ (svalid\ F1)\ (svalid\ F2) \cup svalid\ F2)\ s\ s' * l\ s') =$
$(if\ s \in svalid\ F2\ then\ 1\ else\ 0)$
shows $\forall s \in S.\ l\ s = \mathcal{P}(\omega\ in\ T\ s.\ pvalid\ (U^\infty\ F1\ F2)\ (s\ \#\#\ \omega))$
⟨*proof*⟩

lemma *infinite-reward*:
fixes *s F*
defines $N \equiv Prob0\ S\ (svalid\ F)$ (**is** - ≡ *Prob0 S ?F*)
defines $Y \equiv Prob1\ N\ S\ (svalid\ F)$
assumes *s*: $s \in S\ s \notin Y$
shows $(\int^+ \omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) = \infty$
⟨*proof*⟩

11.3.9 The expected reward implies a unique LES

lemma *existence-of-ExpFuture*:
fixes *s F*
assumes *N-def*: $N \equiv Prob0\ S\ (svalid\ F)$ (**is** - ≡ *Prob0 S ?F*)
assumes *Y-def*: $Y \equiv Prob1\ N\ S\ (svalid\ F)$
assumes *s*: $s \in S\ s \notin S - (Y - ?F)$
shows $enn2real\ (\int^+ \omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) - (\varrho\ s + (\sum s' \in S.\ \tau\ s\ s' * \iota\ s\ s')) =$
$(\sum s' \in S.\ \tau\ s\ s' * enn2real\ (\int^+ \omega.\ reward\ (Future\ F)\ (s'\ \#\#\ \omega)\ \partial T\ s'))$
⟨*proof*⟩

lemma *uniqueness-of-ExpFuture*:
fixes *F*
assumes *N-def*: $N \equiv Prob0\ S\ (svalid\ F)$ (**is** - ≡ *Prob0 S ?F*)
assumes *Y-def*: $Y \equiv Prob1\ N\ S\ (svalid\ F)$

assumes *const-def*: $const \equiv \lambda s.\ if\ s \in Y \wedge s \notin svalid\ F\ then - \varrho\ s - (\sum s' \in S.\ \tau\ s\ s' * \iota\ s\ s')\ else\ 0$
assumes *sol*: $\bigwedge s.\ s \in S \Longrightarrow (\sum s' \in S.\ LES\ (S - Y \cup ?F)\ s\ s' * l\ s') = const\ s$
shows $\forall s \in S.\ l\ s = enn2real\ (\int^+ \omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s)$
(**is** $\forall s \in S.\ l\ s = enn2real\ (\int^+ \omega.\ ?R\ (s\ \#\#\ \omega)\ \partial T\ s)$)
⟨*proof*⟩

11.4 Soundness of *Sat*

theorem *Sat-sound*:
$Sat\ F \neq None \Longrightarrow Sat\ F = Some\ (svalid\ F)$
⟨*proof*⟩

11.5 Completeness of *Sat*

theorem *Sat-complete*:
$Sat\ F \neq None$
⟨*proof*⟩

11.6 Completeness and Soundness *Sat*

corollary *Sat*: $Sat\ \Phi = Some\ (svalid\ \Phi)$
⟨*proof*⟩

end

end

12 Probabilistic Guarded Command Language (pGCL)

theory *PGCL*
imports *../Markov-Decision-Process*
begin

12.1 Syntax

datatype *'s pgcl* =
Skip
| *Abort*
| *Assign 's ⇒ 's*
| *Seq 's pgcl 's pgcl*
| *Par 's pgcl 's pgcl*
| *If 's ⇒ bool 's pgcl 's pgcl*
| *Prob bool pmf 's pgcl 's pgcl*
| *While 's ⇒ bool 's pgcl*

12.2 Denotational Semantics

primrec *wp* :: *'s pgcl ⇒ ('s ⇒ ennreal) ⇒ ('s ⇒ ennreal)* **where**

wp Skip f = *f*
| *wp Abort f* = (λ-. *0*)
| *wp* (*Assign u*) *f* = *f* ∘ *u*
| *wp* (*Seq* c_1 c_2) *f* = *wp* c_1 (*wp* c_2 *f*)
| *wp* (*If b* c_1 c_2) *f* = (λ*s*. *if b s then wp* c_1 *f s else wp* c_2 *f s*)
| *wp* (*Par* c_1 c_2) *f* = *wp* c_1 *f* ⊓ *wp* c_2 *f*
| *wp* (*Prob p* c_1 c_2) *f* = (λ*s*. *pmf p True* ∗ *wp* c_1 *f s* + *pmf p False* ∗ *wp* c_2 *f s*)
| *wp* (*While b c*) *f* = *lfp* (λ*X s*. *if b s then wp c X s else f s*)

lemma *wp-mono*: *mono* (*wp c*)
⟨*proof*⟩

abbreviation *det* :: *'s pgcl* ⇒ *'s* ⇒ (*'s pgcl* × *'s*) *pmf set* (≪ -, - ≫) **where**
det c s ≡ {*return-pmf* (*c*, *s*)}

12.3 Operational Semantics

fun *step* :: (*'s pgcl* × *'s*) ⇒ (*'s pgcl* × *'s*) *pmf set* **where**
step (*Skip*, *s*) = ≪*Skip*, *s*≫
| *step* (*Abort*, *s*) = ≪*Abort*, *s*≫
| *step* (*Assign u*, *s*) = ≪*Skip*, *u s*≫
| *step* (*Seq* c_1 c_2, *s*) = (*map-pmf* (λ(*p1'*, *s'*). (*if p1' = Skip then* c_2 *else Seq p1'* c_2, *s'*))) ‘ *step* (c_1, *s*)
| *step* (*If b* c_1 c_2, *s*) = (*if b s then* ≪c_1, *s*≫ *else* ≪c_2, *s*≫)
| *step* (*Par* c_1 c_2, *s*) = ≪c_1, *s*≫ ∪ ≪c_2, *s*≫
| *step* (*Prob p* c_1 c_2, *s*) = {*map-pmf* (λ*b*. *if b then* (c_1, *s*) *else* (c_2, *s*)) *p*}
| *step* (*While b c*, *s*) = (*if b s then* ≪*Seq c* (*While b c*), *s*≫ *else* ≪*Skip*, *s*≫)

lemma *step-finite*: *finite* (*step x*)
⟨*proof*⟩

lemma *step-non-empty*: *step x* ≠ {}
⟨*proof*⟩

interpretation *step*: *Markov-Decision-Process step*
⟨*proof*⟩

definition *rF* :: (*'s* ⇒ *ennreal*) ⇒ ((*'s pgcl* × *'s*) *stream* ⇒ *ennreal*) ⇒ (*'s pgcl* × *'s*) *stream* ⇒ *ennreal* **where**
rF f F ω = (*if fst* (*shd ω*) = *Skip then f* (*snd* (*shd ω*)) *else F* (*stl ω*))

abbreviation *r* :: (*'s* ⇒ *ennreal*) ⇒ (*'s pgcl* × *'s*) *stream* ⇒ *ennreal* **where**
r f ≡ *lfp* (*rF f*)

lemma *continuous-rF*: *sup-continuous* (*rF f*)
⟨*proof*⟩

lemma *mono-rF*: *mono* (*rF f*)
⟨*proof*⟩

lemma *r-unfold*: $r\ f\ \omega = (\mathit{if}\ \mathit{fst}\ (\mathit{shd}\ \omega) = \mathit{Skip}\ \mathit{then}\ f\ (\mathit{snd}\ (\mathit{shd}\ \omega))\ \mathit{else}\ r\ f\ (\mathit{stl}\ \omega))$
⟨*proof*⟩

lemma *mono-r*: $F \leq G \Longrightarrow r\ F\ \omega \leq r\ G\ \omega$
⟨*proof*⟩

lemma *measurable-rF*:
assumes *F*[*measurable*]: $F \in \mathit{borel\text{-}measurable}\ \mathit{step.St}$
shows $\mathit{rF}\ f\ F \in \mathit{borel\text{-}measurable}\ \mathit{step.St}$
⟨*proof*⟩

lemma *measurable-r*[*measurable*]: $r\ f \in \mathit{borel\text{-}measurable}\ \mathit{step.St}$
⟨*proof*⟩

lemma *mono-r'*: $\mathit{mono}\ (\lambda F\ s.\ \prod D{\in}\mathit{step}\ s.\ \int^+ t.\ (\mathit{if}\ \mathit{fst}\ t = \mathit{Skip}\ \mathit{then}\ f\ (\mathit{snd}\ t)\ \mathit{else}\ F\ t)\ \partial\mathit{measure\text{-}pmf}\ D)$
⟨*proof*⟩

lemma *E-inf-r*:
$\mathit{step.E\text{-}inf}\ s\ (r\ f) =$
$\mathit{lfp}\ (\lambda F\ s.\ \prod D{\in}\mathit{step}\ s.\ \int^+ t.\ (\mathit{if}\ \mathit{fst}\ t = \mathit{Skip}\ \mathit{then}\ f\ (\mathit{snd}\ t)\ \mathit{else}\ F\ t)\ \partial\mathit{measure\text{-}pmf}\ D)\ s$
⟨*proof*⟩

lemma *E-inf-r-unfold*:
$\mathit{step.E\text{-}inf}\ s\ (r\ f) = (\prod D{\in}\mathit{step}\ s.\ \int^+ t.\ (\mathit{if}\ \mathit{fst}\ t = \mathit{Skip}\ \mathit{then}\ f\ (\mathit{snd}\ t)\ \mathit{else}\ \mathit{step.E\text{-}inf}\ t\ (r\ f))\ \partial\mathit{measure\text{-}pmf}\ D)$
⟨*proof*⟩

lemma *E-inf-r-induct*[*consumes 1*, *case-names step*]:
assumes $P\ s\ y$
assumes $*$: $\bigwedge F\ s\ y.\ P\ s\ y \Longrightarrow$
$(\bigwedge s\ y.\ P\ s\ y \Longrightarrow F\ s \leq y) \Longrightarrow (\bigwedge s.\ F\ s \leq \mathit{step.E\text{-}inf}\ s\ (r\ f)) \Longrightarrow$
$(\prod D{\in}\mathit{step}\ s.\ \int^+ t.\ (\mathit{if}\ \mathit{fst}\ t = \mathit{Skip}\ \mathit{then}\ f\ (\mathit{snd}\ t)\ \mathit{else}\ F\ t)\ \partial\mathit{measure\text{-}pmf}\ D) \leq y$
shows $\mathit{step.E\text{-}inf}\ s\ (r\ f) \leq y$
⟨*proof*⟩

lemma *E-inf-Skip*: $\mathit{step.E\text{-}inf}\ (\mathit{Skip}, s)\ (r\ f) = f\ s$
⟨*proof*⟩

lemma *E-inf-Seq*:
assumes [*simp*]: $\bigwedge x.\ 0 \leq f\ x$
shows $\mathit{step.E\text{-}inf}\ (\mathit{Seq}\ a\ b, s)\ (r\ f) = \mathit{step.E\text{-}inf}\ (a, s)\ (r\ (\lambda s.\ \mathit{step.E\text{-}inf}\ (b, s)\ (r\ f)))$
⟨*proof*⟩

lemma *E-inf-While*:
step.E-inf (*While g c*, *s*) (*r f*) =
lfp (λ*F s*. *if g s then step.E-inf* (*c*, *s*) (*r F*) *else f s*) *s*
⟨*proof*⟩

12.4 Equate Both Semantics

lemma *E-inf-r-eq-wp*: *step.E-inf* (*c*, *s*) (*r f*) = *wp c f s*
⟨*proof*⟩

end

13 Formalization of the Crowds-Protocol

theory *Crowds-Protocol*
imports *../Discrete-Time-Markov-Chain*
begin

lemma *cond-prob-nonneg*[*simp*]: *0* ≤ *cond-prob M A B*
⟨*proof*⟩

lemma (**in** *MC-syntax*) *emeasure-suntil-geometric*:
assumes [*measurable*]: *Measurable.pred S P*
assumes *s* ∈ *X* **and** ∗[*simp*]: *0* ≤ *p 0* ≤ *r*
assumes *r*: ⋀*s*. *s* ∈ *X* ⟹ *emeasure* (*T s*) {ω∈*space* (*T s*). *P* ω} = *ennreal r*
assumes *p*: ⋀*s*. *s* ∈ *X* ⟹ *emeasure* (*K s*) *X* = *ennreal p p* < *1*
assumes ⋀*t*. *AE* ω *in T t*. ¬ (*P* ⊓ (*HLD X* ⊓ *nxt* (*HLD X suntil P*))) ω
shows *emeasure* (*T s*) {ω∈*space* (*T s*). (*HLD X suntil P*) ω} = *r* / (*1* − *p*)
⟨*proof*⟩

13.1 Definition of the Crowds-Protocol

datatype *'a state* = *Start* | *Init 'a* | *Mix 'a* | *End*

lemma *inj-Mix*[*simp*]: *inj-on Mix A*
⟨*proof*⟩

lemma *inj-Init*[*simp*]: *inj-on Init A*
⟨*proof*⟩

lemma *distinct-state-image*[*simp*]:
Start ∉ *Mix* ‘ *A Init j* ∉ *Mix* ‘ *A End* ∉ *Mix* ‘ *A Mix j* ∈ *Mix* ‘ *A* ⟷ *j* ∈ *A*
Start ∉ *Init* ‘ *A Mix j* ∉ *Init* ‘ *A End* ∉ *Init* ‘ *A Init j* ∈ *Init* ‘ *A* ⟷ *j* ∈ *A*
⟨*proof*⟩

lemma *Init-cut-Mix*[*simp*]:
Init ‘ *H* ∩ *Mix* ‘ *J* = {}
⟨*proof*⟩

abbreviation *Jondo B* ≡ *Init‘B* ∪ *Mix‘B*

locale *Crowds-Protocol* =
  fixes *J* :: *'a set* **and** *C* :: *'a set* **and** *p-f* :: *real* **and** *p-i* :: *'a* ⇒ *real*
  assumes *J-not-empty*: *J* ≠ {} **and** *finite-J*[*simp*]: *finite J*
  assumes *C-smaller*: *C* ⊂ *J* **and** *C-non-empty*: *C* ≠ {}
  assumes *p-f*: *0* < *p-f p-f* < *1*
  assumes *p-i-nonneg*[*simp*]: ⋀*j*. *j* ∈ *J* ⟹ *0* ≤ *p-i j*
  assumes *p-i-distr*: (∑ *j*∈*J*. *p-i j*) = *1*
  assumes *p-i-C*: ⋀*j*. *j* ∈ *C* ⟹ *p-i j* = *0*
begin

abbreviation *H* :: *'a set* **where**
  H ≡ *J* − *C*

definition *p-j* = *1* / *card J*

lemma *p-f-nonneg*[*simp*]: *0* ≤ *p-f p-f* ≤ *1*
  ⟨*proof*⟩

lemma *p-j-nonneg*[*simp*]: *0* ≤ *p-j*
  ⟨*proof*⟩

definition *p-H* = *card H* / *card J*

lemma *p-H-nonneg*[*simp*]: *0* ≤ *p-H p-H* ≤ *1*
  ⟨*proof*⟩

definition *next-prob* :: *'a state* ⇒ *'a state* ⇒ *real* **where**
  next-prob s t = (*case* (*s*, *t*) *of* (*Start*, *Init j*) ⇒ *if j* ∈ *H then p-i j else 0*
  | (*Init j*, *Mix j'*) ⇒ *if j'* ∈ *J then p-j else 0*
  | (*Mix j*, *Mix j'*) ⇒ *if j'* ∈ *J then p-f* ∗ *p-j else 0*
  | (*Mix j*, *End*) ⇒ *1* − *p-f*
  | (*End*, *End*) ⇒ *1*
  | - ⇒ *0*)

definition *N s* = *embed-pmf* (*next-prob s*)

interpretation *MC-syntax N* ⟨*proof*⟩

abbreviation 𝔓 ≡ *T Start*

abbreviation *E s* ≡ *set-pmf* (*N s*)

lemma *finite-C*[*simp*]: *finite C*
  ⟨*proof*⟩

lemma *sum-p-i-C*[*simp*]: *sum p-i C* = *0*
  ⟨*proof*⟩

lemma *sum-p-i-H*[*simp*]: *sum p-i H = 1*
⟨*proof*⟩

lemma *possible-jondo*:
obtains *j* **where** *j ∈ J j ∉ C p-i j ≠ 0*
⟨*proof*⟩

lemma *C-le-J*[*simp*]: *card C < card J*
⟨*proof*⟩

lemma *p-H*: *0 < p-H p-H < 1*
⟨*proof*⟩

lemma *p-H-p-f-pos*: *0 < p-H * p-f*
⟨*proof*⟩

lemma *p-H-p-f-less-1*: *p-H * p-f < 1*
⟨*proof*⟩

lemma *p-j-pos*: *0 < p-j*
⟨*proof*⟩

lemma *H-compl*: *1 − p-H = real (card C) / real (card J)*
⟨*proof*⟩

lemma *H-compl2*: *1 − p-H = card C * p-j*
⟨*proof*⟩

lemma *H-eq2*: *card H * p-j = p-H*
⟨*proof*⟩

lemma *pmf-next-pmf*[*simp*]: *pmf (N s) t = next-prob s t*
⟨*proof*⟩

lemma *next-prob-Start*[*simp*]: *next-prob Start (Init j) = (if j ∈ H then p-i j else 0)*
⟨*proof*⟩

lemma *next-prob-to-Init*[*simp*]: *j ∈ H ⟹ next-prob s (Init j) =*
(case s of Start ⇒ p-i j | - ⇒ 0)
⟨*proof*⟩

lemma *next-prob-to-Mix*[*simp*]: *j ∈ J ⟹ next-prob s (Mix j) =*
*(case s of Init j ⇒ p-j | Mix j ⇒ p-f * p-j | - ⇒ 0)*
⟨*proof*⟩

lemma *next-prob-to-End*[*simp*]: *next-prob s End =*
(case s of Mix j ⇒ 1 − p-f | End ⇒ 1 | - ⇒ 0)

⟨*proof*⟩

lemma *next-prob-from-End*[*simp*]: *next-prob End s* = *0* ⟷ *s* ≠ *End*
⟨*proof*⟩

lemma *next-prob-Mix-MixI*: ∃ *j*. *s* = *Mix j* ⟹ ∃*j*∈*J*. *s'* = *Mix j* ⟹ *next-prob s s'* = *p-f* ∗ *p-j*
⟨*proof*⟩

lemma *E-Start*: *E Start* = {*Init j* | *j*. *j* ∈ *H* ∧ *p-i j* ≠ *0* }
⟨*proof*⟩

lemma *E-Init*: *E* (*Init j*) = {*Mix j* | *j*. *j* ∈ *J* }
⟨*proof*⟩

lemma *E-Mix*: *E* (*Mix j*) = {*Mix j* | *j*. *j* ∈ *J* } ∪ {*End*}
⟨*proof*⟩

lemma *E-End*: *E End* = {*End*}
⟨*proof*⟩

lemma *enabled-End*:
enabled End ω ⟷ *ω* = *sconst End*
⟨*proof*⟩

lemma *AE-End*: (*AE ω in T End*. *P ω*) ⟷ *P* (*sconst End*)
⟨*proof*⟩

lemma *emeasure-Init-eq-Mix*:
assumes [*measurable*]: *Measurable.pred S P*
assumes *AE-End*: *AE x in T End*. ¬ *P* (*End* ## *x*)
shows *emeasure* (*T* (*Init j*)) {*x*∈*space* (*T* (*Init j*)). *P x*} =
emeasure (*T* (*Mix j*)) {*x*∈*space* (*T* (*Mix j*)). *P x*} / *p-f*
⟨*proof*⟩

What is the probability that the server sees a specific jondo (including the initiator) as sender.

definition *visit* :: *'a set* ⇒ *'a set* ⇒ *'a state stream* ⇒ *bool* **where**
visit I L = *Init*‘(*I* ∩ *H*) · (*HLD* (*Mix*‘*J*) *suntil* (*Mix*‘(*L* ∩ *J*) · *HLD* {*End*}))

lemma *visit-unique1*:
visit I1 L1 ω ⟹ *visit I2 L2 ω* ⟹ *I1* ∩ *I2* ≠ {}
⟨*proof*⟩

lemma *visit-unique2*:
assumes *visit I1 L1 ω visit I2 L2 ω*
shows *L1* ∩ *L2* ≠ {}
⟨*proof*⟩

lemma *visit-imp-in-H*: *visit* $\{i\}$ J $\omega \Longrightarrow i \in H$
⟨*proof*⟩

lemma *emeasure-visit*:
assumes *I*: $I \subseteq H$ **and** *L*: $L \subseteq J$
shows *emeasure* $\mathfrak{P}$ $\{\omega \in space\ \mathfrak{P}.\ visit\ I\ L\ \omega\} = (\sum i \in I.\ p\text{-}i\ i) * (card\ L * p\text{-}j)$
⟨*proof*⟩

lemma *measurable-visit*[*measurable*]: *Measurable.pred S* (*visit I L*)
⟨*proof*⟩

lemma *AE-visit*: *AE* ω *in* $\mathfrak{P}$. *visit H J* ω
⟨*proof*⟩

13.2 Server gets no information

lemma *server-view1*: $j \in J \Longrightarrow \mathcal{P}(\omega\ in\ \mathfrak{P}.\ visit\ H\ \{j\}\ \omega) = p\text{-}j$
⟨*proof*⟩

lemma *server-view-indep*:
$L \subseteq J \Longrightarrow I \subseteq H \Longrightarrow \mathcal{P}(\omega\ in\ \mathfrak{P}.\ visit\ I\ L\ \omega) = \mathcal{P}(\omega\ in\ \mathfrak{P}.\ visit\ H\ L\ \omega) * \mathcal{P}(\omega\ in\ \mathfrak{P}.\ visit\ I\ J\ \omega)$
⟨*proof*⟩

lemma *server-view*: $\mathcal{P}(\omega\ in\ \mathfrak{P}.\ \exists j \in H.\ visit\ \{j\}\ \{j\}\ \omega) = p\text{-}j$
⟨*proof*⟩

13.3 Probability that collaborators gain information

definition *hit-C* = *Init*‘*H* · *ev* (*HLD* (*Mix*‘*C*))
definition *before-C B* = (*HLD* (*Jondo H*)) *suntil* ((*Jondo* (*B* ∩ *H*)) · *HLD* (*Mix* ‘ *C*))

lemma *measurable-hit-C*[*measurable*]: *Measurable.pred S hit-C*
⟨*proof*⟩

lemma *measurable-before-C*[*measurable*]: *Measurable.pred S* (*before-C B*)
⟨*proof*⟩

lemma *before-C*:
assumes ω: *enabled Start* ω
shows *before-C B* $\omega \longleftrightarrow$
((*Init*‘*H* · (*HLD* (*Mix*‘*H*) *suntil* (*Mix*‘(*B* ∩ *H*) · *HLD* (*Mix*‘*C*)))) *or* (*Init*‘(*B* ∩ *H*) · *HLD* (*Mix*‘*C*))) ω
⟨*proof*⟩

lemma *before-C-unique*:
assumes ω: *before-C I1* ω *before-C I2* ω **shows** *I1* ∩ *I2* ≠ {}
⟨*proof*⟩

lemma *hit-C-imp-before-C*:
assumes *enabled Start* ω *hit-C* ω **shows** *before-C H* ω
⟨*proof*⟩

lemma *before-C-single*:
assumes *before-C I* ω **shows** $\exists\, i{\in}I \cap H.$ *before-C* $\{i\}$ ω
⟨*proof*⟩

lemma *before-C-imp-in-H*: *before-C* $\{i\}$ $\omega \Longrightarrow i \in H$
⟨*proof*⟩

13.4 The probability that the sender hits a collaborator

lemma *Pr-hit-C*: $\mathcal{P}(\omega$ *in* $\mathfrak{P}.$ *hit-C* $\omega) = (1 - p\text{-}H) \,/\, (1 - p\text{-}H * p\text{-}f)$
⟨*proof*⟩

lemma *before-C-imp-hit-C*:
assumes *enabled Start* ω *before-C B* ω
shows *hit-C* ω
⟨*proof*⟩

lemma *negE*: $\neg\ P \Longrightarrow P \Longrightarrow$ *False*
⟨*proof*⟩

lemma *Pr-visit-before-C*:
assumes *L*: $L \subseteq H$ **and** *I*: $I \subseteq H$
shows $\mathcal{P}(\omega$ *in* $\mathfrak{P}.$ *visit I J* $\omega \wedge$ *before-C L* $\omega \mid$ *hit-C* $\omega\ ) =$
$(\sum i{\in}I.\ p\text{-}i\ i) * card\ L * p\text{-}j * p\text{-}f + (\sum i{\in}I \cap L.\ p\text{-}i\ i) * (1 - p\text{-}H * p\text{-}f)$
⟨*proof*⟩

lemma *Pr-visit-eq-before-C*:
$\mathcal{P}(\omega$ *in* $\mathfrak{P}.\ \exists j{\in}H.$ *visit* $\{j\}$ *J* $\omega \wedge$ *before-C* $\{j\}$ $\omega \mid$ *hit-C* $\omega\ ) = 1 - (p\text{-}H - p\text{-}j)$
$* p\text{-}f$
⟨*proof*⟩

lemma *probably-innocent*:
assumes *approx*: $1 \,/\, (2 * (p\text{-}H - p\text{-}j)) \leq p\text{-}f$ **and** $p\text{-}H \neq p\text{-}j$
shows $\mathcal{P}(\omega$ *in* $\mathfrak{P}.\ \exists j{\in}H.$ *visit* $\{j\}$ *J* $\omega \wedge$ *before-C* $\{j\}$ $\omega \mid$ *hit-C* $\omega\ ) \leq 1 \,/\, 2$
⟨*proof*⟩

lemma *Pr-before-C*:
assumes *L*: $L \subseteq H$
shows $\mathcal{P}(\omega$ *in* $\mathfrak{P}.$ *before-C L* $\omega \mid$ *hit-C* $\omega\ ) =$
$card\ L * p\text{-}j * p\text{-}f + (\sum l{\in}L.\ p\text{-}i\ l) * (1 - p\text{-}H * p\text{-}f)$
⟨*proof*⟩

lemma *P-visit*:
assumes *I*: $I \subseteq H$

shows $\mathcal{P}(\omega \text{ in } \mathfrak{P}.\ \textit{visit } I\ J\ \omega \mid \textit{hit-C } \omega\ ) = (\sum i{\in}I.\ p\text{-}i\ i)$
⟨*proof*⟩

13.5 Probability space of hitting a collaborator

definition $hC = \textit{uniform-measure } \mathfrak{P}\ \{\omega{\in}\textit{space } \mathfrak{P}.\ \textit{hit-C } \omega\}$

lemma *emeasure-hit-C-not-0*: $\textit{emeasure } \mathfrak{P}\ \{\omega \in \textit{space } \mathfrak{P}.\ \textit{hit-C } \omega\} \neq 0$
⟨*proof*⟩

lemma *measurable-hC*[*measurable* (*raw*)]:
$A \in \textit{sets } S \Longrightarrow A \in \textit{sets } hC$
$f \in \textit{measurable } M\ S \Longrightarrow f \in \textit{measurable } M\ hC$
$g \in \textit{measurable } S\ M \Longrightarrow g \in \textit{measurable } hC\ M$
$A \cap \textit{space } S \in \textit{sets } S \Longrightarrow A \cap \textit{space } hC \in \textit{sets } S$
⟨*proof*⟩

lemma *vimage-Int-space-C*[*simp*]:
$f -` \{x\} \cap \textit{space } hC = \{\omega{\in}\textit{space } S.\ f\ \omega = x\}$
⟨*proof*⟩

sublocale *hC*: *information-space hC 2*
⟨*proof*⟩

abbreviation
mutual-information-Pow-CP ($\mathcal{I}'(\text{-}\ ;\ \text{-}')$) **where**
$\mathcal{I}(X\ ;\ Y) \equiv hC.\textit{mutual-information } 2\ (\textit{count-space } (X`\textit{space } hC))\ (\textit{count-space } (Y`\textit{space } hC))\ X\ Y$

lemma *simple-functionI*:
assumes *finite* (*range f*)
assumes [*measurable*]: $\bigwedge x.\ \{\omega{\in}\textit{space } S.\ f\ \omega = x\} \in \textit{sets } S$
shows *simple-function hC f*
⟨*proof*⟩

13.6 Estimate the information to the collaborators

lemma *measure-hC*[*simp*]:
assumes A[*measurable*]: $A \in \textit{sets } S$
shows $\textit{measure } hC\ A = \mathcal{P}(\omega \text{ in } \mathfrak{P}.\ \omega \in A \mid \textit{hit-C } \omega\ )$
⟨*proof*⟩

13.6.1 Setup random variables for mutual information

definition $\textit{first-J } \omega = (\textit{THE } i.\ \textit{visit } \{i\}\ J\ \omega)$

lemma *first-J-eq*:
$\textit{visit } \{i\}\ J\ \omega \Longrightarrow \textit{first-J } \omega = i$
⟨*proof*⟩

lemma *AE-first-J*:
AE ω in ℙ. visit {i} J ω ⟷ first-J ω = i
⟨*proof*⟩

lemma *measurbale-first-J*[*measurable*]: *first-J ∈ measurable S* (*count-space UNIV*)
⟨*proof*⟩

definition *last-H ω* = (*THE i. before-C {i} ω*)

lemma *measurbale-last-H*[*measurable*]: *last-H ∈ measurable S* (*count-space UNIV*)
⟨*proof*⟩

lemma *last-H-eq*:
before-C {i} ω ⟹ last-H ω = i
⟨*proof*⟩

lemma *last-H*:
assumes *enabled Start ω hit-C ω*
shows *before-C {last-H ω} ω last-H ω ∈ H*
⟨*proof*⟩

lemma *AE-last-H*:
AE ω in ℙ. hit-C ω ⟶ before-C {i} ω ⟷ last-H ω = i
⟨*proof*⟩

lemma *information-flow*:
defines *h ≡ real* (*card H*)
assumes *init-uniform*: ⋀*i. i ∈ H ⟹ p-i i = 1 / h*
shows *ℐ*(*first-J ; last-H*) ≤ (*1 − (h − 1) ∗ p-j ∗ p-f*) ∗ *log 2 h*
⟨*proof*⟩

end

end

14 Formalizing the IPv4-address allocation in ZeroConf

theory *Zeroconf-Analysis*
imports *../Discrete-Time-Markov-Chain*
begin

declare *UNIV-bool*[*simp*]

14.1 Definition of a ZeroConf allocation run

datatype *zc-state* = *start*
| *probe nat*

| *ok*
| *error*

lemma *inj-probe*: *inj-on probe X*
⟨*proof*⟩

Countability of *zc-state* simplifies measurability of functions on *zc-state*.

instance *zc-state* :: *countable*
⟨*proof*⟩

locale *Zeroconf-Analysis* =
fixes *N* :: *nat* **and** *p q r e* :: *real*
assumes *p*: *0* < *p p* < *1* **and** *q*: *0* < *q q* < *1*
assumes *r*[*simp*]: *0* ≤ *r* **and** *e*[*simp*]: *0* ≤ *e*
begin

lemma *p-bounds*[*simp*]: *0* ≤ *p p* ≤ *1*
⟨*proof*⟩

lemma *q-bounds*[*simp*]: *0* ≤ *q q* ≤ *1*
⟨*proof*⟩

abbreviation *states* **where**
states ≡ *probe* ' {.. *N*} ∪ {*start*, *ok*, *error*}

primrec *τ* :: *zc-state* ⇒ *zc-state pmf* **where**
τ start = *map-pmf* (λ*True* ⇒ *probe 0* | *False* ⇒ *ok*) (*bernoulli-pmf q*)
| *τ* (*probe n*) = *map-pmf* (λ*True* ⇒ (*if n* < *N then probe* (*Suc n*) *else error*) | *False* ⇒ *start*) (*bernoulli-pmf p*)
| *τ ok* = *return-pmf ok*
| *τ error* = *return-pmf error*

primrec *ϱ* :: *zc-state* ⇒ *zc-state* ⇒ *real* **where**
ϱ start = (λ-. *0*) (*probe 0* := *r*, *ok* := *r* ∗ (*N* + *1*))
| *ϱ* (*probe n*) = (*if n* < *N then* (λ-. *0*) (*probe* (*Suc n*) := *r*) *else* (λ-. *0*) (*error* := *e*))
| *ϱ ok* = (λ-. *0*) (*ok* := *0*)
| *ϱ error* = (λ-. *0*) (*error* := *0*)

lemma *ϱ-nonneg'*[*simp*]: *0* ≤ *ϱ s t*
⟨*proof*⟩

sublocale *MC-with-rewards τ ϱ* λ*s*. *0*
⟨*proof*⟩

14.2 The allocation run is a rewarded DTMC

abbreviation *E s* ≡ *set-pmf* (*τ s*)

lemma *enabled-ok*: *enabled ok* $\omega \longleftrightarrow \omega = sconst\ ok$
⟨*proof*⟩

lemma *finite-E*[*intro*, *simp*]: *finite* (*E s*)
⟨*proof*⟩

lemma *E-closed*: $s \in states \Longrightarrow E\ s \subseteq states$
⟨*proof*⟩

lemma *enabled-error*: *enabled error* $\omega \longleftrightarrow \omega = sconst\ error$
⟨*proof*⟩

lemma *pos-neg-q-pn*: $0 < 1 - q * (1 - p \hat{} Suc\ N)$
⟨*proof*⟩

lemma *to-error*: **assumes** $n \leq N$ **shows** $(probe\ n, error) \in acc$
⟨*proof*⟩

14.3 Probability of a erroneous allocation

definition *P-err* $s = \mathcal{P}(\omega\ in\ T\ s.\ ev\ (HLD\ \{error\})\ (s\ \#\#\ \omega))$

lemma *P-err*:
defines *p-start* $== (q * p \hat{} Suc\ N) / (1 - q * (1 - p \hat{} Suc\ N))$
defines *p-probe* $== (\lambda n.\ p \hat{} Suc\ (N - n) + (1 - p \hat{} Suc\ (N - n)) * p\text{-}start)$
assumes *s*: $s \in states - \{ok, error\}$
shows *P-err s* = (*case s of ok* ⇒ *0* | *error* ⇒ *1* | *probe n* ⇒ *p-probe n* | *start* ⇒ *p-start*)
(**is** ... = *?E s*)
⟨*proof*⟩

lemma *P-err-start*: *P-err start* $= (q * p \hat{} Suc\ N) / (1 - q * (1 - p \hat{} Suc\ N))$
⟨*proof*⟩

14.4 An allocation run terminates almost surely

lemma *states-closed*:
assumes $s \in states$
assumes $(s, t) \in acc\text{-}on\ (- \{error, ok\})$
shows $t \in states$
⟨*proof*⟩

lemma *finite-reached*:
assumes *s*: $s \in states$ **shows** *finite* (*acc-on* (− {*error*, *ok*}) `` {*s*})
⟨*proof*⟩

lemma *AE-reaches-error-or-ok*:
assumes *s*: $s \in states$
shows *AE* ω *in T s. ev* (*HLD* {*error*, *ok*}) ω
⟨*proof*⟩

14.5 Expected runtime of an allocation run

definition $R\ s = (\int^+ \omega.\ \textit{reward-until}\ \{\textit{error},\ \textit{ok}\}\ s\ \omega\ \partial T\ s)$

definition $R'\ s = \textit{enn2real}\ (R\ s)$

lemma *R-iter*: $s \neq \textit{error} \Longrightarrow s \neq \textit{ok} \Longrightarrow R\ s = (\int^+ t.\ \textit{ennreal}\ (\varrho\ s\ t) + R\ t\ \partial\tau\ s)$
⟨*proof*⟩

lemma *R-finite*:
assumes *s*: $s \in \textit{states}$
shows $R\ s \neq \infty$
⟨*proof*⟩

lemma *R-less-top*: $s \in \textit{states} \Longrightarrow R\ s < \textit{top}$
⟨*proof*⟩

lemma *R′-iter*: **assumes** *s*: $s \in \textit{states}\ s \neq \textit{error}\ s \neq \textit{ok}$ **shows** $R'\ s = (\int t.\ \varrho\ s\ t + R'\ t\ \partial\tau\ s)$
⟨*proof*⟩

lemma *cost-from-start*:
$R'\ \textit{start} =$
$(q * (r + p\hat{}\textit{Suc}\ N * e + r * p * (1 - p\hat{}N) / (1 - p)) + (1 - q) * (r * \textit{Suc}\ N)) /$
$(1 - q + q * p\hat{}\textit{Suc}\ N)$
⟨*proof*⟩

end

interpretation *ZC*: *Zeroconf-Analysis 2 16 / 65024* :: *real 0.01 0.002 3600*
⟨*proof*⟩

lemma $\textit{ZC.P-err start} \leq 1 / 10\hat{}12$
⟨*proof*⟩

lemma $\textit{ZC.R' start} \leq 0.007$
⟨*proof*⟩

end

15 Formalization of the Gossip-Broadcast

theory *Gossip-Broadcast*
imports *../Discrete-Time-Markov-Chain*
begin

lemma *inj-on-upd-PiE*:

assumes $i \notin I$ **shows** *inj-on* $(\lambda(x,f).\ f(i := x))\ (M \times (\Pi_E\ i{\in}I.\ A\ i))$
⟨*proof*⟩

lemma *sum-folded-product*:
fixes $I :: 'i\ set$ **and** $f :: 's \Rightarrow 'i \Rightarrow 'a{::}\{semiring\text{-}0,\ comm\text{-}monoid\text{-}mult\}$
assumes *finite* I $\bigwedge i.\ i \in I \Longrightarrow$ *finite* $(S\ i)$
shows $(\sum x{\in}Pi_E\ I\ S.\ \prod i{\in}I.\ f\ (x\ i)\ i) = (\prod i{\in}I.\ \sum s{\in}S\ i.\ f\ s\ i)$
⟨*proof*⟩

15.1 Definition of the Gossip-Broadcast

datatype *state* = *listening* | *sending* | *sleeping*

type-synonym *sys-state* = $(nat \times nat) \Rightarrow state$

lemma *state-UNIV*: $UNIV = \{listening,\ sending,\ sleeping\}$
⟨*proof*⟩

locale *gossip-broadcast* =
fixes *size* :: *nat* **and** *p* :: *real*
assumes *size*: $0 < size$
assumes *p*: $0 < p\ p < 1$
begin

interpretation *pmf-as-function* ⟨*proof*⟩

definition *states* :: *sys-state set* **where**
$states = (\{..< size\} \times \{..< size\}) \rightarrow_E \{listening,\ sending,\ sleeping\}$

definition *start* :: *sys-state* **where**
$start = (\lambda x{\in}\{..< size\}{\times}\{..< size\}.\ listening)((0,\ 0) := sending)$

definition *neighbour-sending* **where**
$neighbour\text{-}sending\ s = (\lambda(x,y).$
$(x > 0 \wedge s\ (x - 1,\ y) = sending) \vee$
$(x < size \wedge s\ (x + 1,\ y) = sending) \vee$
$(y > 0 \wedge s\ (x,\ y - 1) = sending) \vee$
$(y < size \wedge s\ (x,\ y + 1) = sending))$

definition *node-trans* :: *sys-state* ⇒ $(nat \times nat)$ ⇒ *state* ⇒ *state* ⇒ *real* **where**
node-trans g x s = (*case s of*
listening ⇒ (*if neighbour-sending g x*
then (λ-.*0*) (*sending* := *p*, *sleeping* := *1* − *p*)
else (λ-.*0*) (*listening* := *1*))
| *sending* ⇒ (λ-.*0*) (*sleeping* := *1*)
| *sleeping* ⇒ (λ-.*0*) (*sleeping* := *1*))

lemma *node-trans-sum-eq-1*[*simp*]:
node-trans g x s′ listening + (*node-trans g x s′ sending* + *node-trans g x s′*

sleeping) = *1*
⟨*proof*⟩

lemma *node-trans-nonneg*[*simp*]: *0* ≤ *node-trans s x i j*
⟨*proof*⟩

lift-definition *proto-trans* :: *sys-state* ⇒ *sys-state pmf* **is**
λ*s s'*. *if s'* ∈ *states then* (∏ *x*∈{..< *size*}×{..< *size*}. *node-trans s x* (*s x*) (*s' x*))
else 0
⟨*proof*⟩

end

15.2 The Gossip-Broadcast forms a DTMC

sublocale *gossip-broadcast* ⊆ *MC-syntax proto-trans* ⟨*proof*⟩

end

16 Certification of Reachability Problems on MDPs

theory *MDP-RP-Certification*
imports
../MDP-Reachability-Problem
HOL−Library.IArray
HOL−Library.Code-Target-Numeral
begin

context *Reachability-Problem*
begin

lemma *p-ub'*:
fixes *x*
assumes *1*: *s* ∈ *S* ⋀*s D*. *s* ∈ *S1* ⟹ *D* ∈ *K s* ⟹ (∑ *t*∈*S*. *pmf D t* ∗ *x t*) ≤ *x s*
assumes *2*: ⋀*s*. *s* ∈ *S1* ⟹ *x s* ≠ *0* ⟹ (∃ *t*∈*S2*. (*s*, *t*) ∈ (*SIGMA s*:*S1*. ⋃ *D*∈*K s*. *set-pmf D*)*)
assumes *3*: ⋀*s*. *s* ∈ *S* − *S1* − *S2* ⟹ *x s* = *0*
assumes *4*: ⋀*s*. *s* ∈ *S2* ⟹ *x s* = *1*
shows *enn2real* (*p s*) ≤ *x s*
⟨*proof*⟩

lemma *n-lb'*:
fixes *x*
assumes *wf R*
assumes *1*: *s* ∈ *S* ⋀*s D*. *s* ∈ *S1* ⟹ *D* ∈ *K s* ⟹ *x s* ≤ (∑ *t*∈*S*. *pmf D t* ∗ *x t*)
assumes *2*: ⋀*s D*. *s* ∈ *S1* ⟹ *D* ∈ *K s* ⟹ *x s* ≠ *0* ⟹ ∃ *t*∈*D*. ((*t*, *s*) ∈ *R* ∧ *t* ∈ *S1* ∧ *x t* ≠ *0*) ∨ *t* ∈ *S2*
assumes *3*: ⋀*s*. *s* ∈ *S* − *S1* − *S2* ⟹ *x s* = *0*
assumes *4*: ⋀*s*. *s* ∈ *S2* ⟹ *x s* = *1*

shows $x\ s \leq enn2real\ (n\ s)$
⟨*proof*⟩

end

no-notation *Stream.snth* (**infixl** !! *100*) — we use !! for IArray

16.1 Computable representation

record *mdp-reachability-problem* =
state-count :: *nat*
distrs :: (*nat* × *rat*) *list list iarray*
states1 :: *bool iarray*
states2 :: *bool iarray*

record *'a RP-sub-cert* =
solution :: *rat iarray*
witness :: (*'a* × *nat*) *iarray*

record *RP-cert* =
pos-cert :: (*nat* × *nat*) *RP-sub-cert*
neg-cert :: *nat list RP-sub-cert*

definition *sparse-mult sx y* = *sum-list* (*map* (λ(*n*, *x*). *x* ∗ *y* !! *n*) *sx*)

primrec *lookup* **where**
lookup d [] *x* = *d*
| *lookup d* (*y*#*ys*) *x* = (*if fst y* = *x then snd y else lookup d ys x*)

lemma *lookup-eq-map-of*: *lookup d xs x* = (*case map-of xs x of Some x* ⇒ *x* | *None* ⇒ *d*)
⟨*proof*⟩

lemma *lookup-in-set*:
distinct (*map fst xs*) ⟹ *x* ∈ *set xs* ⟹ *lookup d xs* (*fst x*) = *snd x*
⟨*proof*⟩

lemma *lookup-not-in-set*:
x ∉ *fst* ‘ *set xs* ⟹ *lookup d xs x* = *d*
⟨*proof*⟩

lemma *lookup-nonneg*:
(⋀*x v*. (*x*, *v*) ∈ *set xs* ⟹ *0* ≤ *v*) ⟹ (*0*::*'a*::*ordered-comm-monoid-add*) ≤ *lookup 0 xs x*
⟨*proof*⟩

lemma *sparse-mult-eq-sum-lookup*:
fixes *xs* :: (*nat* × *'a*::*comm-semiring-1*) *list*
assumes *list-all* (λ(*n*, *x*). *n* < *M*) *xs distinct* (*map fst xs*)

shows *sparse-mult xs y* = ($\sum$ *i*<*M*. *lookup 0 xs i* * *y* !! *i*)
⟨*proof*⟩

lemma *sum-list-eq-sum-lookup*:
fixes *xs* :: (*nat* × *'a*::*comm-semiring-1*) *list*
assumes *list-all* (λ(*n*, *x*). *n* < *M*) *xs distinct* (*map fst xs*)
shows *sum-list* (*map snd xs*) = ($\sum$ *i*<*M*. *lookup 0 xs i*)
⟨*proof*⟩

definition
valid-mdp-rp mdp ⟷
0 < *state-count mdp* ∧
IArray.length (*distrs mdp*) = *state-count mdp* ∧
IArray.length (*states1 mdp*) = *state-count mdp* ∧
IArray.length (*states2 mdp*) = *state-count mdp* ∧
(∀ *i*<*state-count mdp*. ¬ (*states1 mdp* !! *i* ∧ *states2 mdp* !! *i*) ∧
list-all (λ*ds*. *distinct* (*map fst ds*) ∧ *list-all* (λ(*n*, *x*). *0* ≤ *x* ∧ *n* < *state-count mdp*) *ds* ∧
sum-list (*map snd ds*) = *1*) (*distrs mdp* !! *i*) ∧
¬ *List.null* (*distrs mdp* !! *i*))

definition
valid-sub-cert mdp c ord check ⟷
IArray.length (*witness c*) = *state-count mdp* ∧
IArray.length (*solution c*) = *state-count mdp* ∧
(∀ *i*<*state-count mdp*.
if states2 mdp !! *i then solution c* !! *i* = *1*
else if states1 mdp !! *i then 0* ≤ *solution c* !! *i* ∧
(*list-all* (λ*ds*. *ord* (*sparse-mult ds* (*solution c*)) (*solution c* !! *i*)) (*distrs mdp* !! *i*)) ∧
(*0* < *solution c* !! *i* ⟶ *check* (*distrs mdp* !! *i*) (*witness c* !! *i*))
else solution c !! *i* = *0*)

definition
valid-pos-cert mdp c ⟷
valid-sub-cert mdp c (≤)
(λ*D* ((*j*, *a*), *n*). *j* < *state-count mdp* ∧ *snd* (*witness c* !! *j*) < *n* ∧ *0* < *solution c* !! *j* ∧
a < *length D* ∧ *lookup 0* (*D* ! *a*) *j* ≠ *0*)

definition
valid-neg-cert mdp c ⟷
valid-sub-cert mdp c (≥)
(λ*D* (*J*, *n*). *list-all2* (λ*j d*. *j* < *state-count mdp* ∧ *snd* (*witness c* !! *j*) < *n* ∧
lookup 0 d j ≠ *0* ∧ *0* < *solution c* !! *j*) *J D*)

definition
valid-cert mdp c ⟷ *valid-pos-cert mdp* (*pos-cert c*) ∧ *valid-neg-cert mdp* (*neg-cert c*)

lemma *valid-mdp-rpD-length*:
assumes *valid-mdp-rp mdp*
shows *0 < state-count mdp IArray.length (distrs mdp) = state-count mdp*
IArray.length (states1 mdp) = state-count mdp IArray.length (states2 mdp) = state-count mdp
⟨*proof*⟩

lemma *valid-mdp-rpD*:
assumes *valid-mdp-rp mdp i < state-count mdp*
shows ¬ (*states1 mdp !! i ∧ states2 mdp !! i*)
and ⋀*ds n x. ds ∈ set (distrs mdp !! i) ⟹ (n, x) ∈ set ds ⟹ n < state-count mdp*
and ⋀*ds n x. ds ∈ set (distrs mdp !! i) ⟹ (n, x) ∈ set ds ⟹ 0 ≤ x*
and ⋀*ds. ds ∈ set (distrs mdp !! i) ⟹ sum-list (map snd ds) = 1*
and ⋀*ds. ds ∈ set (distrs mdp !! i) ⟹ distinct (map fst ds)*
and *distrs mdp !! i ≠ []*
⟨*proof*⟩

lemma *valid-mdp-rp-sparse-mult*:
assumes *valid-mdp-rp mdp i < state-count mdp ds ∈ set (distrs mdp !! i)*
shows *sparse-mult ds y = (∑ i<state-count mdp. lookup 0 ds i * y !! i)*
⟨*proof*⟩

lemma *valid-sub-certD*:
assumes *valid-mdp-rp mdp valid-sub-cert mdp c ord check i < state-count mdp*
shows ¬ *states1 mdp !! i ⟹* ¬ *states2 mdp !! i ⟹ solution c !! i = 0*
and *states2 mdp !! i ⟹ solution c !! i = 1*
and *states1 mdp !! i ⟹ 0 ≤ solution c !! i*
and ⋀*ds. states1 mdp !! i ⟹ ds ∈ set (distrs mdp !! i) ⟹ ord (sparse-mult ds (solution c)) (solution c !! i)*
and ⋀*ds. states1 mdp !! i ⟹ 0 < solution c !! i ⟶ check (distrs mdp !! i) (witness c !! i)*
⟨*proof*⟩

lemma *valid-pos-certD*:
assumes *valid-mdp-rp mdp valid-pos-cert mdp c i < state-count mdp states1 mdp !! i*
0 < solution c !! i witness c !! i = ((j, a), n)
shows *snd (witness c !! j) < n ∧ j < state-count mdp ∧ a < length (distrs mdp !! i) ∧*
lookup 0 ((distrs mdp !! i) ! a) j ≠ 0 ∧ 0 < solution c !! j
⟨*proof*⟩

lemma *valid-neg-certD*:
assumes *valid-mdp-rp mdp valid-neg-cert mdp c i < state-count mdp states1 mdp !! i*
0 < solution c !! i witness c !! i = (js, n)
shows *list-all2 (λj ds. j < state-count mdp ∧ snd (witness c !! j) < n ∧ lookup*

0 ds j ≠ 0 ∧ 0 < solution c !! j) js (distrs mdp !! i)
⟨*proof*⟩

context
fixes *mdp c*
assumes *rp*: *valid-mdp-rp mdp*
assumes *cert*: *valid-cert mdp c*
begin

interpretation *pmf-as-function* ⟨*proof*⟩

abbreviation *S* ≡ {..< *state-count mdp*}
abbreviation *S1* ≡ {*i*. *i* < *state-count mdp* ∧ (*states1 mdp*) !! *i*}
abbreviation *S2* ≡ {*i*. *i* < *state-count mdp* ∧ (*states2 mdp*) !! *i*}

lift-definition *K* :: *nat* ⇒ *nat pmf set* **is**
λi. if i < state-count mdp then
{ (*λj. of-rat* (*lookup 0 D j*) :: *real*) | *D*. *D* ∈ *set* (*distrs mdp* !! *i*) }
else { *indicator* {*0*} }
⟨*proof*⟩

interpretation *MDP*: *Reachability-Problem K S S1 S2*
⟨*proof*⟩

definition *P-max s* = *enn2real* (*MDP.p s*)
definition *P-min s* = *enn2real* (*MDP.n s*)

lemma
assumes *i* < *state-count mdp*
shows *P-max*: *P-max i* ≤ *real-of-rat* (*solution* (*pos-cert c*) !! *i*) (**is** *?max*)
and *P-min*: *P-min i* ≥ *real-of-rat* (*solution* (*neg-cert c*) !! *i*) (**is** *?min*)
⟨*proof*⟩

end

end

References

[1] J. Hölzl. Formalising semantics for expected running time of probabilistic programs. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, pages 475–482. Springer, 2016.

[2] J. Hölzl. Markov processes in isabelle/hol. In Y. Bertot and V. Vafeiadis, editors, *Certified Programs and Proofs (CPP 2017)*. ACM, 2017.

[3] J. Hölzl and T. Nipkow. Interactive verification of Markov chains: Two distributed protocol case studies. In U. Fahrenberg, A. Legay, and

C. Thrane, editors, *Quantities in Formal Methods (QFM 2012)*, volume 103 of *EPTCS*. arXiv, 2012.

[4] J. Hölzl and T. Nipkow. Verifying pCTL model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, LNCS, 2012.

[5] H. Johannes. Markov chains and markov decision processes in isabelle/hol. *Journal of Automated Reasoning*, 2017.