Markov Models

Johannes Hölzl and Tobias Nipkow

February 6, 2026

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 **3**

2 Auxiliary Theory **4**

3 Discrete-Time Markov Chain **17**
3.1 Discrete Markov Kernel 21
3.2 Trace Space for Discrete-Time Markov Chains 25
3.3 Fairness 30
3.4 First Hitting Time 34
3.5 Markov chain with Initial Distribution 37
3.6 Trace space with Restriction 43
3.7 Bisimulation 45
3.8 Reward Structure on Markov Chains 47
3.9 Bisimulation on a relation 50
3.10 Product Construction 54
3.11 Trace Space equal to Markov Chains 56

4 Classifying Markov Chain States **64**
4.1 Expected number of visits 66
4.2 Reachability probability 69
4.3 Recurrent states 75
4.4 Stationary distribution 89

5 Markov Decision Processes **116**
5.1 Configurations . . . 117
5.2 Configuration with Memoryless Scheduler . . . 119
5.3 MDP Kernel and Induced Configurations . . . 120
5.4 Trace Space . . . 121
5.5 Finite MDPs . . . 133

6 Discrete-time Markov Processes **164**
6.1 Constructing Discrete-Time Markov Processes . . . 165
6.2 Strong Markov Property for Discrete-Time Markov Processes 174

7 Continuous-time Markov chains **179**
7.1 Trace Operations: relate $('a \times real)$ *stream* and $real \Rightarrow 'a$ . . . 179
7.2 Exponential Distribution . . . 180
7.3 Transition Rates . . . 187
7.4 Continuous-time Kernel . . . 188
7.5 Kernel equals Parallel Choice . . . 190
7.6 Markov Chain Property . . . 197
7.7 Explosion time . . . 200
7.8 Transition probability p_t . . . 204

8 Example A **214**
8.1 The essential classs $\{C1, C2, C3\}$ . . . 215
8.2 The stationary distribution n . . . 215

9 Example B **216**
9.1 Enabled, accessible and communicating states . . . 217
9.2 B is aperiodic . . . 218
9.3 The stationary distribution N . . . 218
9.4 Limit behavior and recurrence times . . . 218

10 Adapt Gauss-Jordan elimination to DTMCs **219**

11 pCTL model checking **222**
11.1 Syntax . . . 222
11.2 Semantics . . . 223
11.3 Implementation of *Sat* . . . 224
11.3.1 *Prob0* . . . 224
11.3.2 *Prob1* . . . 226
11.3.3 *ProbU*, *ExpCumm*, and *ExpState* . . . 227
11.3.4 *LES* . . . 228
11.3.5 *ProbUinfty*, compute unbounded until . . . 228
11.3.6 *ExpFuture*, compute unbounded reward . . . 228
11.3.7 *Sat* . . . 228
11.3.8 Finite expected reward . . . 230

11.3.9 The expected reward implies a unique LES 233
11.4 Soundness of *Sat* 235
11.5 Completeness of *Sat* 238
11.6 Completeness and Soundness *Sat* 240

12 Probabilistic Guarded Command Language (pGCL) 240
12.1 Syntax 240
12.2 Denotational Semantics 241
12.3 Operational Semantics 241
12.4 Equate Both Semantics 245

13 Formalization of the Crowds-Protocol 246
13.1 Definition of the Crowds-Protocol 247
13.2 Server gets no information 254
13.3 Probability that collaborators gain information 254
13.4 The probability that the sender hits a collaborator 256
13.5 Probability space of hitting a collaborator 261
13.6 Estimate the information to the collaborators 262
13.6.1 Setup random variables for mutual information 262

14 Formalizing the IPv4-address allocation in ZeroConf 267
14.1 Definition of a ZeroConf allocation run 267
14.2 The allocation run is a rewarded DTMC 269
14.3 Probability of a erroneous allocation 269
14.4 An allocation run terminates almost surely 270
14.5 Expected runtime of an allocation run 271

15 Formalization of the Gossip-Broadcast 273
15.1 Definition of the Gossip-Broadcast 273
15.2 The Gossip-Broadcast forms a DTMC 275

16 Certification of Reachability Problems on MDPs 275
16.1 Computable representation 276

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*: $(\bigwedge y.\ x \leq f\ y) \Longrightarrow x \leq lfp\ f$
unfolding *lfp-def* **by** (*intro Inf-greatest*) (*auto intro*: *order-trans*)

lemma *lfp-arg*: $(\lambda t.\ lfp\ (F\ t)) = lfp\ (\lambda x\ t.\ F\ t\ (x\ t))$
apply (*auto simp*: *lfp-def le-fun-def fun-eq-iff intro*!: *Inf-eqI Inf-greatest*)
subgoal for *x y*
by (*rule INF-lower2*[*of top*($x := y$)]) *auto*
done

lemma *lfp-pair*: $lfp\ (\lambda f\ (a,\ b).\ F\ (\lambda a\ b.\ f\ (a,\ b))\ a\ b)\ (a,\ b) = lfp\ F\ a\ b$
unfolding *lfp-def*
by (*auto intro*!: *INF-eq simp*: *le-fun-def*)
(*auto intro*!: *exI*[*of* - $\lambda(a,\ b).\ x\ a\ b$ **for** *x*])

lemma *all-Suc-split*: $(\forall i.\ P\ i) \longleftrightarrow (P\ 0 \wedge (\forall i.\ P\ (Suc\ i)))$
using *nat-induct* **by** *auto*

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

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

unfolding *with-def* **by** (*auto intro*: *someI2*)

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*
by (*simp add*: *maximal-def*)

lemma *maximalI-trans*: *x* ∈ *maximal f S* ⟹ *f x* ≤ *f y* ⟹ *y* ∈ *S* ⟹ *y* ∈ *maximal f S*
unfolding *maximal-def* **by** (*blast intro*: *antisym order-trans*)

lemma *maximalD1*: *x* ∈ *maximal f S* ⟹ *x* ∈ *S*
by (*simp add*: *maximal-def*)

lemma *maximalD2*: *x* ∈ *maximal f S* ⟹ *y* ∈ *S* ⟹ *f y* ≤ *f x*
by (*simp add*: *maximal-def*)

lemma *maximal-inject*: *x* ∈ *maximal f S* ⟹ *y* ∈ *maximal f S* ⟹ *f x* = *f y*
by (*rule order.antisym*) (*simp-all add*: *maximal-def*)

lemma *maximal-empty*[*simp*]: *maximal f* {} = {}
by (*simp add*: *maximal-def*)

lemma *maximal-singleton*[*simp*]: *maximal f* {*x*} = {*x*}
by (*auto simp add*: *maximal-def*)

lemma *maximal-in-S*: *maximal f S* ⊆ *S*
by (*auto simp*: *maximal-def*)

end

context *linorder*
begin

lemma *maximal-ne*:
assumes *finite S S* ≠ {}
shows *maximal f S* ≠ {}
using *assms*
proof (*induct rule*: *finite-ne-induct*)
case (*insert s S*)
show *?case*
proof *cases*
assume ∀ *x*∈*S*. *f x* ≤ *f s*
with *insert* **have** *s* ∈ *maximal f* (*insert s S*)
by (*auto intro*!: *maximalI*)

```
    then show ?thesis
      by auto
  next
    assume ¬ (∀ x∈S. f x ≤ f s)
    then have maximal f (insert s S) = maximal f S
      by (auto simp: maximal-def)
    with insert show ?thesis
      by auto
  qed
qed simp

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: ⋀s. s ∈ N ⟹ l2 s ≤ l1 s
  assumes finitary: finite (Δ ` (S∪N))
  shows l2 s ≤ l1 s
proof −
  define M where M = {s∈S∪N. ∀ t∈S∪N. Δ t ≤ Δ s}

  have [simp]: ⋀s. s∈S ⟹ integrable (K s) Δ
    by (simp add: Δ-def[abs-def])

  have M-unqiue: ⋀s t. s ∈ M ⟹ t ∈ M ⟹ Δ s = Δ t
    by (auto intro!: antisym simp: M-def)
  have M1: ⋀s. s ∈ M ⟹ s ∈ S ∪ N
    by (auto simp: M-def)
  have M2: ⋀s t. s ∈ M ⟹ t ∈ S ∪ N ⟹ Δ t ≤ Δ s
    by (auto simp: M-def)
  have M3: ⋀s t. s ∈ M ⟹ t ∈ S ∪ N ⟹ t ∉ M ⟹ Δ t < Δ s
    by (auto simp: M-def less-le)

  have N: ∀ s∈N. Δ s ≤ 0
    using eq by (simp add: Δ-def)

  { fix s assume s: s ∈ M M ∩ N = {}
    then have s ∈ S − N
      by (auto dest: M1)
    with to-N[of s] obtain t where (s, t) ∈ (SIGMA s:UNIV. K s)* and t ∈ N
      by (auto simp: M-def)
    from this(1) ‹s ∈ M› have Δ s ≤ 0
```

```
  proof (induction rule: converse-rtrancl-induct)
    case (step s s')
    then have s: s ∈ M s ∈ S s ∉ N and s': s' ∈ S ∪ N s' ∈ K s
      using S ‹M ∩ N = {}› by (auto dest: M1)
    have s' ∈ M
    proof (rule ccontr)
      assume s' ∉ M
      with ‹s ∈ S› s' ‹s ∈ M›
      have 0 < pmf (K s) s' Δ s' < Δ s
        by (auto intro: M2 M3 pmf-positive)

      have Δ s ≤ ((∫ t. l2 t ∂K s) + c s) - ((∫ t. l1 t ∂K s) + c s)
        unfolding Δ-def using ‹s ∈ S› ‹s ∉ N› by (intro diff-mono l1 l2) auto
      then have Δ s ≤ (∫ s'. Δ s' ∂K s)
        using ‹s ∈ S› by (simp add: Δ-def)
      also have ... < (∫ s'. Δ s ∂K s)
        using ‹s' ∈ K s› ‹Δ s' < Δ s› ‹s∈S› S ‹s∈M›
        by (intro measure-pmf.integral-less-AE[where A={s'}])
       (auto simp: emeasure-measure-pmf-finite AE-measure-pmf-iff set-pmf-iff[symmetric]
             intro!: M2)
      finally show False
        using measure-pmf.prob-space[of K s] by simp
    qed
    with step.IH ‹t∈N› N have Δ s' ≤ 0 s' ∈ M
      by auto
    with ‹s∈S› show Δ s ≤ 0
      by (force simp: M-def)
  qed (insert N ‹t∈N›, auto) }

show ?thesis
proof cases
  assume M ∩ N = {}
  have Max (Δ`(S∪N)) ∈ Δ`(S∪N)
    using ‹s ∈ S› by (intro Max-in finitary) auto
  then obtain t where t ∈ S ∪ N Δ t = Max (Δ`(S∪N))
    unfolding image-iff by metis
  then have t ∈ M
    by (auto simp: M-def finitary intro!: Max-ge)
  have Δ s ≤ Δ t
    using ‹t∈M› ‹s∈S› by (auto dest: M2)
  also have Δ t ≤ 0
    using ‹t∈M› ‹M ∩ N = {}› by fact
  finally show ?thesis
    by (simp add: Δ-def)
next
  assume M ∩ N ≠ {}
  then obtain t where t ∈ M t ∈ N by auto
  with N ‹s∈S› have Δ s ≤ 0
    by (intro order-trans[of Δ s Δ t 0]) (auto simp: M-def)
```

```
    then show ?thesis
      by (simp add: Δ-def)
  qed
qed

lemma unique-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 ⋀s. s ∈ S ⟹ integrable (K s) l1
  assumes ⋀s. s ∈ S ⟹ integrable (K s) l2
  assumes ⋀s. s ∈ S ⟹ ∃t∈N. (s, t) ∈ (SIGMA s:UNIV. K s)*
  assumes ⋀s. s ∈ S ⟹ l1 s = (∫ t. l1 t ∂K s) + c s
  assumes ⋀s. s ∈ S ⟹ l2 s = (∫ t. l2 t ∂K s) + c s
  assumes ⋀s. s ∈ N ⟹ l2 s = l1 s
  assumes 1: finite (Δ ‘ (S∪N))
  shows l2 s = l1 s
proof −
  have finite ((λx. l2 x − l1 x) ‘ (S∪N))
    using 1 by (auto simp: Δ-def[abs-def])
  moreover then have finite (uminus ‘ (λx. l2 x − l1 x) ‘ (S∪N))
    by auto
  ultimately show ?thesis
    using assms
    by (intro antisym mono-les[of s S K N l2 l1 c] mono-les[of s S K N l1 l2 c])
       (auto simp: image-comp comp-def)
qed

lemma inf-continuous-suntil-disj[order-continuous-intros]:
  assumes Q: inf-continuous Q
  assumes disj: ⋀x ω. ¬ (P ω ∧ Q x ω)
  shows inf-continuous (λx. P suntil Q x)
  unfolding inf-continuous-def
proof (safe intro!: ext)
  fix M ω i assume (P suntil Q (∏ i. M i)) ω decseq M then show (P suntil Q
(M i)) ω
    unfolding inf-continuousD[OF Q ‹decseq M›] by induction (auto intro: sun-
til.intros)
next
  fix M ω assume ∗: (∏ i. P suntil Q (M i)) ω decseq M
  then have (P suntil Q (M 0)) ω
    by auto
  from this ∗ show (P suntil Q (∏ i. M i)) ω
    unfolding inf-continuousD[OF Q ‹decseq M›]
  proof induction
    case (base ω) with disj[of ω M -] show ?case by (auto intro: suntil.intros
elim: suntil.cases)
  next
    case (step ω) with disj[of ω M -] show ?case by (auto intro: suntil.intros
```

elim: *suntil.cases*)
qed
qed

lemma *inf-continuous-nxt*[*order-continuous-intros*]: *inf-continuous P* $\Longrightarrow$ *inf-continuous* (λx. *nxt* (*P x*) ω)
by (*auto simp*: *inf-continuous-def image-comp*)

lemma *sup-continuous-nxt*[*order-continuous-intros*]: *sup-continuous P* $\Longrightarrow$ *sup-continuous* (λx. *nxt* (*P x*) ω)
by (*auto simp*: *sup-continuous-def image-comp*)

lemma *mcont-ennreal-of-enat*: *mcont Sup* ($\leq$) *Sup* ($\leq$) *ennreal-of-enat*
by (*auto intro*!: *mcontI monotoneI contI ennreal-of-enat-Sup*)

lemma *mcont2mcont-ennreal-of-enat*[*cont-intro*]:
mcont lub ord Sup ($\leq$) *f* $\Longrightarrow$ *mcont lub ord Sup* ($\leq$) (λx. *ennreal-of-enat* (*f x*))
by (*auto intro*: *ccpo.mcont2mcont*[*OF complete-lattice-ccpo'*] *mcont-ennreal-of-enat*)

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

lemma *scount-eq-emeasure*: *scount P* ω = *emeasure* (*count-space UNIV*) {*i*. *P* (*sdrop i* ω)}
proof *cases*
assume *alw* (*ev P*) ω
moreover then have *infinite* {*i*. *P* (*sdrop i* ω)}
using *infinite-iff-alw-ev*[*of P* ω] **by** *simp*
ultimately show *?thesis*
by (*simp add*: *scount-infinite-iff*[*symmetric*])
next
assume $\neg$ *alw* (*ev P*) ω
moreover then have *finite* {*i*. *P* (*sdrop i* ω)}
using *infinite-iff-alw-ev*[*of P* ω] **by** *simp*
ultimately show *?thesis*
by (*simp add*: *not-alw-iff not-ev-iff scount-eq-card*)
qed

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

lemma *measurable-sfirst2*:
assumes [*measurable*]: *Measurable.pred* (*N* $\bigotimes_M$ *stream-space M*) (λ(*x*, ω). *P x* ω)
shows (λ(*x*, ω). *sfirst* (*P x*) ω) $\in$ *measurable* (*N* $\bigotimes_M$ *stream-space M*) (*count-space UNIV*)
apply (*coinduction rule*: *measurable-enat-coinduct*)
apply *simp*

apply (*rule exI*[*of* - $\lambda x.\ 0$])
apply (*rule exI*[*of* - $\lambda(x,\ \omega).\ (x,\ stl\ \omega)$])
apply (*rule exI*[*of* - $\lambda(x,\ \omega).\ P\ x\ \omega$])
apply (*subst sfirst.simps*[*abs-def*])
apply (*simp add*: *fun-eq-iff*)
done

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*)
using *measurable-sfirst2*[*measurable*] **by** *measurable*

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

lemma *measurable-epred*[*measurable*]: *epred* $\in$ *count-space UNIV* $\to_M$ *count-space UNIV*
by (*rule measurable-count-space*)

lemma *nn-integral-stretch*:
$f \in$ *borel* $\to_M$ *borel* $\Longrightarrow$ $c \neq 0 \Longrightarrow$ $(\int^+ x.\ f\ (c * x)\ \partial lborel) = (1\ /\ |c|::real) * (\int^+ x.\ f\ x\ \partial lborel)$
using *nn-integral-real-affine*[*of f c 0*] **by** (*simp add*: *mult.assoc*[*symmetric*] *ennreal-mult*[*symmetric*])

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* ($J\ i$)) $\Longrightarrow$ $(\prod i{\in}I.\ \sum j{\in}J\ i.\ f\ i\ j) = (\sum m{\in}Pi_E\ I\ J.\ \prod i{\in}I.\ f\ i\ (m\ i))$
using ‹*finite I*›
proof *induction*
case (*insert i I*) **then show** *?case*
by (*auto simp*: *PiE-insert-eq finite-PiE sum.reindex inj-combinator sum.swap*[*of* - $Pi_E\ I\ J$]
sum.cartesian-product' sum-distrib-left sum-distrib-right
intro!: *sum.cong prod.cong arg-cong*[**where** $f{=}(*)\ x$ **for** *x*])
qed *simp*

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 −
have $(\prod i{\in}I.\ f\ i + g\ i) = (\prod i{\in}I.\ \sum b{\in}\{True,\ False\}.\ if\ b\ then\ f\ i\ else\ g\ i)$
by *simp*
also have $\ldots = (\sum m{\in}I \to_E \{True,\ False\}.\ \prod i{\in}I.\ if\ m\ i\ then\ f\ i\ else\ g\ i)$

using ‹*finite I*› **by** (*rule prod-sum-distrib*) *simp*
also have ... = $(\sum J{\in}Pow\ I.\ (\prod i{\in}J.\ f\ i) * (\prod i{\in}I - J.\ g\ i))$
by (*rule sum.reindex-bij-witness*[**where** $i{=}\lambda J.\ \lambda i{\in}I.\ i{\in}J$ **and** $j{=}\lambda m.\ \{i{\in}I.\ m\ i\}$])
(*auto simp*: *fun-eq-iff prod.If-cases* ‹*finite I*› *intro*!: *arg-cong2*[**where** $f{=}(*)$] *prod.cong*)
finally show *?thesis* **.**
qed

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

lemma (**in** *linordered-nonzero-semiring*) *prod-nonneg*: $(\forall\, a{\in}A.\ 0 \le f\ a) \Longrightarrow 0 \le prod\ f\ A$
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma (**in** *linordered-nonzero-semiring*) *prod-mono*:
$\forall\, i{\in}A.\ 0 \le f\ i \wedge f\ i \le g\ i \Longrightarrow prod\ f\ A \le prod\ g\ A$
by (*induct A rule*: *infinite-finite-induct*) (*auto intro*!: *prod-nonneg mult-mono*)

lemma (**in** *linordered-nonzero-semiring*) *prod-mono2*:
assumes *finite J* $I \subseteq J$ $\bigwedge i.\ i \in I \Longrightarrow 0 \le g\ i \wedge g\ i \le f\ i$ $(\bigwedge i.\ i \in J - I \Longrightarrow 1 \le f\ i)$
shows $prod\ g\ I \le prod\ f\ J$
proof −
have $prod\ g\ I = (\prod i{\in}J.\ if\ i \in I\ then\ g\ i\ else\ 1)$
using ‹*finite J*› ‹$I \subseteq J$› **by** (*simp add*: *prod.If-cases Int-absorb1*)
also have ... $\le prod\ f\ J$
using *assms* **by** (*intro prod-mono*) *auto*
finally show *?thesis* **.**
qed

lemma (**in** *linordered-nonzero-semiring*) *prod-mono3*:
assumes *finite J* $I \subseteq J$ $\bigwedge i.\ i \in J \Longrightarrow 0 \le g\ i$ $\bigwedge i.\ i \in I \Longrightarrow g\ i \le f\ i$ $(\bigwedge i.\ i \in J - I \Longrightarrow g\ i \le 1)$
shows $prod\ g\ J \le prod\ f\ I$
proof −
have $prod\ g\ J \le (\prod i{\in}J.\ if\ i \in I\ then\ f\ i\ else\ 1)$
using *assms* **by** (*intro prod-mono*) *auto*
also have ... = $prod\ f\ I$
using ‹*finite J*› ‹$I \subseteq J$› **by** (*simp add*: *prod.If-cases Int-absorb1*)
finally show *?thesis* **.**
qed

lemma (**in** *linordered-nonzero-semiring*) *one-le-prod*: $(\bigwedge i.\ i \in I \Longrightarrow 1 \le f\ i) \Longrightarrow 1 \le prod\ f\ I$
proof (*induction I rule*: *infinite-finite-induct*)
case (*insert i I*) **then show** *?case*
using *mult-mono*[*of 1 f i 1 prod f I*]

```
    by (auto intro: order-trans[OF zero-le-one])
qed auto

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 cases
  assume [simp]: finite I
  with assms have [simp]: J ⊆ I ⟹ 0 ≤ prod p J for J
    by (intro prod-nonneg) auto
  have 1 + (∑ i∈I. p i) = (∑ J∈insert {} ((λx. {x}) ‘I). (∏ i∈J. p i) * (∏ i∈I
− J. 1))
    by (subst sum.insert) (auto simp: sum.reindex)
  also have ... ≤ (∑ J∈Pow I. (∏ i∈J. p i) * (∏ i∈I − J. 1))
    using assms by (intro sum-mono2) auto
  finally show ?thesis
    by (subst prod-add-distrib) (auto simp: add.commute)
qed simp

lemma summable-iff-convergent-prod:
  fixes p :: nat ⇒ real assumes p: ⋀i. 0 ≤ p i
  shows summable p ⟷ convergent (λn. ∏ i<n. p i + 1)
  unfolding summable-iff-convergent
proof
  assume convergent (λn. ∏ i<n. p i + 1)
  then obtain x where x: (λn. ∏ i<n. p i + 1) ⟶ x
    by (auto simp: convergent-def)
  then have 1 ≤ x
    by (rule tendsto-lowerbound) (auto intro!: always-eventually one-le-prod p)

  have convergent (λn. 1 + (∑ i<n. p i))
  proof (intro Bseq-mono-convergent BseqI allI)
    show 0 < x using ‹1 ≤ x› by auto
  next
    fix n
    have norm ((∑ i<n. p i) + 1) ≤ (∏ i<n. p i + 1)
      using p by (simp add: sum-nonneg sum-plus-one-le-prod-plus-one p)
    also have ... ≤ x
      using assms
      by (intro tendsto-lowerbound[OF x])
         (auto simp: eventually-sequentially intro!: exI[of - n] prod-mono2)
    finally show norm (1 + sum p {..<n}) ≤ x
      by (simp add: add.commute)
  qed (insert p, auto intro!: sum-mono2)
  then show convergent (λn. ∑ i<n. p i)
    unfolding convergent-add-const-iff .
next
  assume convergent (λn. ∑ i<n. p i)
```

```
  then obtain x where x: (λn. exp (∑ i<n. p i)) ⟶ exp x
    by (force simp: convergent-def intro!: tendsto-exp)
  show convergent (λn. ∏ i<n. p i + 1)
  proof (intro Bseq-mono-convergent BseqI allI)
    show 0 < exp x by simp
  next
    fix n
    have norm (∏ i<n. p i + 1) ≤ exp (∑ i<n. p i)
      using p exp-ge-add-one-self[of p -] by (auto simp add: prod-nonneg exp-sum
add.commute intro!: prod-mono)
    also have ... ≤ exp x
      using p
     by (intro tendsto-lowerbound[OF x]) (auto simp: eventually-sequentially intro!:
sum-mono2 )
    finally show norm (∏ i<n. p i + 1) ≤ exp x .
  qed (insert p, auto intro!: prod-mono2)
qed

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
  using eexp.simps by simp-all

lemma eexp-0[simp]: eexp 0 = 1
  by (simp add: zero-ereal-def)

lemma eexp-inj[simp]: eexp x = eexp y ⟷ x = y
  by (cases x; cases y; simp)

lemma eexp-mono[simp]: eexp x ≤ eexp y ⟷ x ≤ y
  by (cases x; cases y; simp add: top-unique)

lemma eexp-strict-mono[simp]: eexp x < eexp y ⟷ x < y
  by (simp add: less-le)

lemma exp-eq-0-iff[simp]: eexp x = 0 ⟷ x = −∞
  using eexp-inj[of x −∞] unfolding eexp-minus-infty .

lemma eexp-surj: range eexp = UNIV
proof −
  have part: UNIV = {0} ∪ {0 <..< top} ∪ {top::ennreal}
    by (auto simp: less-top)
  show ?thesis
```

unfolding *part*
by (*force simp*: *image-iff less-top less-top-ennreal intro*!: *eexp.simps*[*symmetric*] *eexp.simps dest*: *exp-total*)
qed

lemma *continuous-on-eexp'*: *continuous-on UNIV eexp*
by (*rule continuous-onI-mono*) (*auto simp*: *eexp-surj*)

lemma *continuous-on-eexp*[*continuous-intros*]: *continuous-on A f* $\Longrightarrow$ *continuous-on A* (λx. *eexp* (*f x*))
by (*rule continuous-on-compose2*[*OF continuous-on-eexp'*]) *auto*

lemma *tendsto-eexp*[*tendsto-intros*]: (*f* $\longrightarrow$ *x*) *F* $\Longrightarrow$ ((λx. *eexp* (*f x*)) $\longrightarrow$ *eexp x*) *F*
by (*rule continuous-on-tendsto-compose*[*OF continuous-on-eexp'*]) *auto*

lemma *measurable-eexp*[*measurable*]: *eexp* $\in$ *borel* $\rightarrow_M$ *borel*
using *continuous-on-eexp'* **by** (*rule borel-measurable-continuous-onI*)

lemma *eexp-add*: $\neg$ (($x = \infty \wedge y = -\infty$) $\vee$ ($x = -\infty \wedge y = \infty$)) $\Longrightarrow$ *eexp* ($x + y$) = *eexp x* $*$ *eexp y*
by (*cases x*; *cases y*; *simp add*: *exp-add ennreal-mult ennreal-top-mult ennreal-mult-top*)

lemma *sum-Pinfty*:
fixes *f* :: *'a* $\Rightarrow$ *ereal*
shows *sum f I* = ∞ $\longleftrightarrow$ (*finite I* $\wedge$ ($\exists$ *i*$\in$*I*. *f i* = ∞))
by (*induction I rule*: *infinite-finite-induct*) *auto*

lemma *sum-Minfty*:
fixes *f* :: *'a* $\Rightarrow$ *ereal*
shows *sum f I* = $-\infty$ $\longleftrightarrow$ (*finite I* $\wedge$ $\neg$ ($\exists$ *i*$\in$*I*. *f i* = ∞) $\wedge$ ($\exists$ *i*$\in$*I*. *f i* = $-\infty$))
by (*induction I rule*: *infinite-finite-induct*)
(*auto simp*: *sum-Pinfty*)

lemma *eexp-sum*: $\neg$ ($\exists$ *i*$\in$*I*. $\exists$ *j*$\in$*I*. *f i* = $-\infty \wedge$ *f j* = ∞) $\Longrightarrow$ *eexp* ($\sum$ *i*$\in$*I*. *f i*) = ($\prod$ *i*$\in$*I*. *eexp* (*f i*))
proof (*induction I rule*: *infinite-finite-induct*)
case (*insert i I*)
have *eexp* (*sum f* (*insert i I*)) = *eexp* (*f i*) $*$ *eexp* (*sum f I*)
using *insert.prems insert.hyps* **by** (*auto simp*: *sum-Pinfty sum-Minfty intro*!: *eexp-add*)
then show *?case*
using *insert* **by** *auto*
qed *simp-all*

lemma *eexp-suminf*:
assumes *wf-f*: $\neg$ {$-\infty$, ∞} $\subseteq$ *range f* **and** *f*: *summable f*
shows (λn. $\prod$ *i*$<$*n*. *eexp* (*f i*)) $\longrightarrow$ *eexp* ($\sum$ *i*. *f i*)
proof $-$

```
  have (λn. eexp (∑ i<n. f i)) ⟶ eexp (∑ i. f i)
    by (intro tendsto-eexp summable-LIMSEQ f)
  also have (λn. eexp (∑ i<n. f i)) = (λn. ∏ i<n. eexp (f i))
    using wf-f by (auto simp: fun-eq-iff image-iff eq-commute intro!: eexp-sum)
  finally show ?thesis .
qed

lemma continuous-onI-antimono:
  fixes f :: 'a::linorder-topology ⇒ 'b::{dense-order,linorder-topology}
  assumes open (f‘A)
    and mono: ⋀x y. x ∈ A ⟹ y ∈ A ⟹ x ≤ y ⟹ f y ≤ f x
  shows continuous-on A f
proof (rule continuous-on-generate-topology[OF open-generated-order], safe)
  have monoD: ⋀x y. x ∈ A ⟹ y ∈ A ⟹ f y < f x ⟹ x < y
    by (auto simp: not-le[symmetric] mono)
  have ∃x. x ∈ A ∧ f x < b ∧ x < a if a: a ∈ A and fa: f a < b for a b
  proof −
    obtain y where f a < y {f a ..< y} ⊆ f‘A
      using open-right[OF ‹open (f‘A)›, of f a b] a fa
      by auto
    obtain z where z: f a < z z < min b y
      using dense[of f a min b y] ‹f a < y› ‹f a < b› by auto
    then obtain c where z = f c c ∈ A
      using ‹{f a ..< y} ⊆ f‘A›[THEN subsetD, of z] by (auto simp: less-imp-le)
    with a z show ?thesis
      by (auto intro!: exI[of - c] simp: monoD)
  qed
  then show ∃C. open C ∧ C ∩ A = f −‘ {..<b} ∩ A for b
    by (intro exI[of - (⋃x∈{x∈A. f x < b}. {x <..})])
       (auto intro: le-less-trans[OF mono] less-imp-le)

  have ∃x. x ∈ A ∧ b < f x ∧ x > a if a: a ∈ A and fa: b < f a for a b
  proof −
    note a fa
    moreover
    obtain y where y < f a {y <.. f a} ⊆ f‘A
      using open-left[OF ‹open (f‘A)›, of f a b]  a fa
      by auto
    then obtain z where z: max b y < z z < f a
      using dense[of max b y f a] ‹y < f a› ‹b < f a› by auto
    then obtain c where z = f c c ∈ A
      using ‹{y <.. f a} ⊆ f‘A›[THEN subsetD, of z] by (auto simp: less-imp-le)
    with a z show ?thesis
      by (auto intro!: exI[of - c] simp: monoD)
  qed
  then show ∃C. open C ∧ C ∩ A = f −‘ {b <..} ∩ A for b
    by (intro exI[of - (⋃x∈{x∈A. b < f x}. {..< x})])
       (auto intro: less-le-trans[OF - mono] less-imp-le)
qed
```

lemma *minus-add-eq-ereal*: ¬ ((*a* = ∞ ∧ *b* = −∞) ∨ (*a* = −∞ ∧ *b* = ∞)) ⟹ − (*a* + *b*::*ereal*) = −*a* − *b*
by (*cases a*; *cases b*; *simp*)

lemma *setsum-negf-ereal*: ¬ {−∞, ∞} ⊆ *f*‘*I* ⟹ (∑ *i*∈*I*. − *f i*) = − (∑ *i*∈*I*. *f i*::*ereal*)
by (*induction I rule*: *infinite-finite-induct*)
(*auto simp*: *minus-add-eq-ereal sum-Minfty sum-Pinfty*,
(*subst minus-add-eq-ereal*; *auto simp*: *sum-Pinfty sum-Minfty image-iff minus-ereal-def*)+)

lemma *convergent-minus-iff-ereal*: *convergent* (λ*x*. − *f x*::*ereal*) ⟷ *convergent f*
unfolding *convergent-def* **by** (*metis ereal-uminus-uminus ereal-Lim-uminus*)

lemma *summable-minus-ereal*: ¬ {−∞, ∞} ⊆ *range f* ⟹ *summable* (λ*n*. *f n*) ⟹ *summable* (λ*n*. − *f n*::*ereal*)
unfolding *summable-iff-convergent*
by (*subst setsum-negf-ereal*) (*auto simp*: *convergent-minus-iff-ereal*)

lemma (**in** *product-prob-space*) *product-nn-integral-component*:
assumes *f* ∈ *borel-measurable* (*M i*)*i* ∈ *I*
shows *integral*^{N} (*Pi*$_M$ *I M*) (λ*x*. *f* (*x i*)) = *integral*^{N} (*M i*) *f*
proof −
from *assms* **show** *?thesis*
apply (*subst PiM-component*[*symmetric*, *OF* ‹*i* ∈ *I*›])
apply (*subst nn-integral-distr*[*OF measurable-component-singleton*])
apply *simp-all*
done
qed

lemma *ennreal-inverse-le*[*simp*]: *inverse x* ≤ *inverse y* ⟷ *y* ≤ (*x*::*ennreal*)
by (*cases 0* < *x*; *cases x*; *cases 0* < *y*; *cases y*; *auto simp*: *top-unique inverse-ennreal*)

lemma *inverse-inverse-ennreal*[*simp*]: *inverse* (*inverse x*::*ennreal*) = *x*
by (*cases 0* < *x*; *cases x*; *auto simp*: *inverse-ennreal*)

lemma *range-inverse-ennreal*: *range inverse* = (*UNIV*::*ennreal set*)
proof −
have ∃ *x*. *y* = *inverse x* **for** *y* :: *ennreal*
by (*intro exI*[*of* - *inverse y*]) *simp*
then show *?thesis*
unfolding *surj-def* **by** *auto*
qed

lemma *continuous-on-inverse-ennreal*′: *continuous-on* (*UNIV* :: *ennreal set*) *inverse*
by (*rule continuous-onI-antimono*) (*auto simp*: *range-inverse-ennreal*)

lemma *sums-minus-ereal*: $\neg \{-\infty, \infty\} \subseteq f \text{ ` } UNIV \Longrightarrow (\lambda n.\ -\ f\ n :: ereal)\ sums\ x \Longrightarrow f\ sums\ -\ x$
unfolding *sums-def*
apply (*subst ereal-Lim-uminus*)
apply (*subst* (*asm*) *setsum-negf-ereal*)
apply *auto*
done

lemma *suminf-minus-ereal*: $\neg \{-\infty, \infty\} \subseteq f \text{ ` } UNIV \Longrightarrow summable\ f \Longrightarrow (\sum n.\ -\ f\ n :: ereal) = -\ suminf\ f$
apply (*rule sums-unique*[*symmetric*])
apply (*rule sums-minus-ereal*)
apply (*auto simp*: *ereal-uminus-eq-reorder*)
done

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'*: $sstart\ \Omega\ (x\ \#\ xs) = \{\omega.\ shd\ \omega = x \wedge stl\ \omega \in sstart\ \Omega\ xs\}$
by (*auto simp*: *sstart-eq*)

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 −
let *?S* = $stream\text{-}space\ (count\text{-}space\ UNIV)$
have $\forall\, N\ M.\ R\ N\ M \longrightarrow (\exists\, N'\ M'\ p.\ (\forall\, y{\in}set\text{-}pmf\ p.\ R\ (N'\ y)\ (M'\ y)) \wedge$
$(\forall\, x.\ N'\ x \in space\ (prob\text{-}algebra\ ?S)) \wedge (\forall\, x.\ M'\ x \in space\ (prob\text{-}algebra\ ?S)) \wedge$
$N = (measure\text{-}pmf\ p \gg\!= (\lambda y.\ distr\ (N'\ y)\ ?S\ ((\#\#)\ y))) \wedge$
$M = (measure\text{-}pmf\ p \gg\!= (\lambda y.\ distr\ (M'\ y)\ ?S\ ((\#\#)\ y))))$

using *cont* **by** *auto*
then obtain *n m p* **where**
p: $\bigwedge N\ M\ y.\ R\ N\ M \Longrightarrow y \in$ *set-pmf* $(p\ N\ M) \Longrightarrow R\ (n\ N\ M\ y)\ (m\ N\ M\ y)$ **and**
n: $\bigwedge N\ M\ x.\ R\ N\ M \Longrightarrow n\ N\ M\ x \in$ *space* $($*prob-algebra ?S*$)$ **and**
n-eq: $\bigwedge N\ M\ y.\ R\ N\ M \Longrightarrow N = ($*measure-pmf* $(p\ N\ M) \gg\!= (\lambda y.$ *distr* $(n\ N\ M\ y)$ *?S* $((\#\#)\ y)))$ **and**
m: $\bigwedge N\ M\ x.\ R\ N\ M \Longrightarrow m\ N\ M\ x \in$ *space* $($*prob-algebra ?S*$)$ **and**
m-eq: $\bigwedge N\ M\ y.\ R\ N\ M \Longrightarrow M = ($*measure-pmf* $(p\ N\ M) \gg\!= (\lambda y.$ *distr* $(m\ N\ M\ y)$ *?S* $((\#\#)\ y)))$
unfolding *choice-iff' choice-iff* **by** *blast*

define *A* **where** $A = ($*SIGMA nm:UNIV*$.\ (\lambda x.\ (n\ ($*fst nm*$)\ ($*snd nm*$)\ x,\ m\ ($*fst nm*$)\ ($*snd nm*$)\ x))\ `\ p\ ($*fst nm*$)\ ($*snd nm*$))$
have *A-singleton*: $A\ ``\ \{nm\} = (\lambda x.\ (n\ ($*fst nm*$)\ ($*snd nm*$)\ x,\ m\ ($*fst nm*$)\ ($*snd nm*$)\ x))\ `\ p\ ($*fst nm*$)\ ($*snd nm*$)$ **for** *nm*
by (*auto simp*: *A-def*)

have *sets-n*[*measurable-cong*, *simp*]: *sets* $(n\ N\ M\ y) =$ *sets ?S* **if** $R\ N\ M$ **for** $N\ M\ y$
using *n*[*OF that*, *of y*] **by** (*auto simp*: *space-prob-algebra*)
have *sets-m*[*measurable-cong*, *simp*]: *sets* $(m\ N\ M\ y) =$ *sets ?S* **if** $R\ N\ M$ **for** $N\ M\ y$
using *m*[*OF that*, *of y*] **by** (*auto simp*: *space-prob-algebra*)
have [*simp*]: $R\ N\ M \Longrightarrow$ *prob-space* $(n\ N\ M\ y)$ **for** $N\ M\ y$
using *n*[*of N M y*] **by** (*auto simp*: *space-prob-algebra*)
have [*simp*]: $R\ N\ M \Longrightarrow$ *prob-space* $(m\ N\ M\ y)$ **for** $N\ M\ y$
using *m*[*of N M y*] **by** (*auto simp*: *space-prob-algebra*)
have [*measurable*]: $R\ N\ M \Longrightarrow n\ N\ M \in$ *count-space UNIV* $\rightarrow_M$ *subprob-algebra ?S* **for** $N\ M$
by (*rule measurable-prob-algebraD*) (*auto intro*: *n*)
have [*measurable*]: $R\ N\ M \Longrightarrow m\ N\ M \in$ *count-space UNIV* $\rightarrow_M$ *subprob-algebra ?S* **for** $N\ M$
by (*rule measurable-prob-algebraD*) (*auto intro*: *m*)

define n' **where** $n'\ N\ M\ y =$ *distr* $(n\ N\ M\ y)$ *?S* $((\#\#)\ y)$ **for** $N\ M\ y$
define m' **where** $m'\ N\ M\ y =$ *distr* $(m\ N\ M\ y)$ *?S* $((\#\#)\ y)$ **for** $N\ M\ y$
have *n'-eq*: $R\ N\ M \Longrightarrow N = ($*measure-pmf* $(p\ N\ M) \gg\!= n'\ N\ M)$ **for** $N\ M$ **unfolding** *n'-def* **by** (*rule n-eq*)
have *m'-eq*: $R\ N\ M \Longrightarrow M = ($*measure-pmf* $(p\ N\ M) \gg\!= m'\ N\ M)$ **for** $N\ M$ **unfolding** *m'-def* **by** (*rule m-eq*)
have [*measurable*]: $R\ N\ M \Longrightarrow n'\ N\ M \in$ *count-space UNIV* $\rightarrow_M$ *subprob-algebra ?S* **for** $N\ M$
unfolding *n'-def* **by** (*rule measurable-distr2*[**where** *M*=*?S*]) *measurable*
have [*measurable*]: $R\ N\ M \Longrightarrow m'\ N\ M \in$ *count-space UNIV* $\rightarrow_M$ *subprob-algebra ?S* **for** $N\ M$
unfolding *m'-def* **by** (*rule measurable-distr2*[**where** *M*=*?S*]) *measurable*

have *n'-shd*: $R\ N\ M \Longrightarrow$ *distr* $(n'\ N\ M\ y)$ $($*count-space UNIV*$)$ *shd* = *measure-pmf*

(*return-pmf y*) **for** *N M y*
unfolding *n'-def* **by** (*subst distr-distr*) (*auto simp*: *comp-def prob-space.distr-const return-pmf.rep-eq*)
have *m'-shd*: *R N M* ⟹ *distr* (*m' N M y*) (*count-space UNIV*) *shd* = *measure-pmf* (*return-pmf y*) **for** *N M y*
unfolding *m'-def* **by** (*subst distr-distr*) (*auto simp*: *comp-def prob-space.distr-const return-pmf.rep-eq*)
have *n'-stl*: *R N M* ⟹ *distr* (*n' N M y*) *?S stl* = *n N M y* **for** *N M y*
unfolding *n'-def* **by** (*subst distr-distr*) (*auto simp*: *comp-def distr-id2*)
have *m'-stl*: *R N M* ⟹ *distr* (*m' N M y*) *?S stl* = *m N M y* **for** *N M y*
unfolding *m'-def* **by** (*subst distr-distr*) (*auto simp*: *comp-def distr-id2*)

define *F* **where** *F* = (*A** `` {(*N*, *M*)})
have *countable F*
unfolding *F-def*
apply (*intro countable-rtrancl countable-insert*[*of* - (*N*, *M*)] *countable-empty*)
apply (*rule countable-Image*)
apply (*auto simp*: *A-singleton*)
done
have *F-NM*[*simp*]: (*N*, *M*) ∈ *F* **unfolding** *F-def* **by** *auto*
have *R-F*[*simp*]: *R N' M'* **if** (*N'*, *M'*) ∈ *F* **for** *N' M'*
proof −
have ((*N*, *M*), (*N'*, *M'*)) ∈ *A** **using** *that* **by** (*auto simp*: *F-def*)
then show *R N' M'*
by (*induction p*==(*N'*, *M'*) *arbitrary*: *N' M' rule*: *rtrancl-induct*) (*auto simp*: ‹*R N M*› *A-def p*)
qed
have *nm-F*: (*n N' M' y*, *m N' M' y*) ∈ *F* **if** *y* ∈ *p N' M'* (*N'*, *M'*) ∈ *F* **for** *N' M' y*
proof −
have ∗: ((*N*, *M*), (*N'*, *M'*)) ∈ *A** **using** *that* **by** (*auto simp*: *F-def*)
with *that* **show** *?thesis*
apply (*simp add*: *F-def*)
apply (*intro rtrancl.rtrancl-into-rtrancl*[*OF* ∗])
apply (*auto simp*: *A-def*)
done
qed

define Ω **where** Ω = (⋃(*n*, *m*)∈*F*. *set-pmf* (*p n m*))
have [*measurable*]: Ω ∈ *sets* (*count-space UNIV*) **by** *auto*
have *in-Ω*: (*N*, *M*) ∈ *F* ⟹ *y* ∈ *p N M* ⟹ *y* ∈ Ω **for** *N M y*
by (*auto simp*: Ω-*def Bex-def*)

show *?thesis*
proof (*intro stream-space-eq-sstart*)
from ‹*countable F*› **show** *countable* Ω
by (*auto simp add*: Ω-*def*)
show *prob-space N prob-space M sets N* = *sets ?S sets M* = *sets ?S*
using *R-1*[*OF* ‹*R N M*›] *R-2*[*OF* ‹*R N M*›] **by** (*auto simp add*: *space-prob-algebra*)

```
  have ⋀N M. (N, M) ∈ F ⟹ AE x in N. x !! i ∈ Ω for i
  proof (induction i)
    case 0 note NM = 0[THEN R-F, simp] show ?case
      apply (subst n'-eq[OF NM])
      apply (subst AE-bind[where B=?S])
        apply measurable
      apply (auto intro!: AE-distrD[where f=shd and M'=count-space UNIV]
                    simp: AE-measure-pmf-iff n[OF NM] n'-shd in-Ω[OF 0] cong:
AE-cong-simp)
      done
  next
    case (Suc i) note NM = Suc(2)[THEN R-F, simp]
    show ?case
      apply (subst n'-eq[OF NM])
      apply (subst AE-bind[where B=?S])
        apply measurable
      apply (auto intro!: AE-distrD[where f=stl and M'=?S] Suc(1)[OF nm-F]
Suc(2)
        simp: AE-measure-pmf-iff n'-stl cong: AE-cong-simp)
      done
  qed
  then have AE-N: ⋀N M. (N, M) ∈ F ⟹ AE x in N. x ∈ streams Ω
    unfolding streams-iff-snth AE-all-countable by auto
  then show AE x in N. x ∈ streams Ω by (blast intro: F-NM)

  have ⋀N M. (N, M) ∈ F ⟹ AE x in M. x !! i ∈ Ω for i
  proof (induction i arbitrary: N M)
    case 0 note NM = 0[THEN R-F, simp] show ?case
      apply (subst m'-eq[OF NM])
      apply (subst AE-bind[where B=?S])
        apply measurable
      apply (auto intro!: AE-distrD[where f=shd and M'=count-space UNIV]
                    simp: AE-measure-pmf-iff m[OF NM] m'-shd in-Ω[OF 0] cong:
AE-cong-simp)
      done
  next
    case (Suc i) note NM = Suc(2)[THEN R-F, simp]
    show ?case
      apply (subst m'-eq[OF NM])
      apply (subst AE-bind[where B=?S])
        apply measurable
      apply (auto intro!: AE-distrD[where f=stl and M'=?S] Suc(1)[OF nm-F]
Suc(2)
        simp: AE-measure-pmf-iff m'-stl cong: AE-cong-simp)
      done
  qed
  then have AE-M: ⋀N M. (N, M) ∈ F ⟹ AE x in M. x ∈ streams Ω
    unfolding streams-iff-snth AE-all-countable by auto
  then show AE x in M. x ∈ streams Ω by (blast intro: F-NM)
```

```
    fix xs assume xs ∈ lists Ω
    with ‹(N, M) ∈ F› show emeasure N (sstart Ω xs) = emeasure M (sstart Ω
xs)
    proof (induction xs arbitrary: N M)
      case Nil
      have prob-space N prob-space M sets N = sets ?S sets M = sets ?S
        using R-1[OF R-F[OF Nil(1)]] R-2[OF R-F[OF Nil(1)]] by (auto simp
add: space-prob-algebra)
      have emeasure N (streams Ω) = 1
          by (rule prob-space.emeasure-eq-1-AE[OF ‹prob-space N› - AE-N[OF
Nil(1)]])
           (auto simp add: ‹sets N = sets ?S› intro!: streams-sets)
      moreover have emeasure M (streams Ω) = 1
          by (rule prob-space.emeasure-eq-1-AE[OF ‹prob-space M› - AE-M[OF
Nil(1)]])
           (auto simp add: ‹sets M = sets ?S› intro!: streams-sets)
      ultimately show ?case by simp
    next
      case (Cons x xs)
      note NM = Cons(2)[THEN R-F, simp]
      have ∗: (##) y −‘ sstart Ω (x # xs) = (if x = y then sstart Ω xs else {})
for y
        by auto
      show ?case
        apply (subst n′-eq[OF NM])
        apply (subst (3) m′-eq[OF NM])
        apply (subst emeasure-bind[OF - - sstart-sets])
          apply simp []
         apply measurable []
        apply (subst emeasure-bind[OF - - sstart-sets])
          apply simp []
         apply measurable []
        apply (intro nn-integral-cong-AE AE-pmfI)
        apply (subst n′-def)
        apply (subst m′-def)
        using Cons(3)
        apply (auto intro!: Cons nm-F
       simp add: emeasure-distr sets-eq-imp-space-eq[OF sets-n] sets-eq-imp-space-eq[OF
sets-m]
                    space-stream-space ∗)
        done
    qed
  qed
qed
```

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* $(acc\ ``\ \{s\})$
by (*auto intro*!: *countable-rtrancl countable-set-pmf simp*: *Sigma-Image*)

lemma *countable-acc*: *countable* $X \Longrightarrow$ *countable* $(acc\ ``\ X)$
apply (*rule countable-Image*)
apply (*rule countable-reachable*)
apply *assumption*
done

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* $(shd\ \omega)\ (stl\ \omega) \Longrightarrow alw\ (\lambda\omega.\ enabled\ (shd\ \omega)\ (stl\ \omega))\ \omega$
by (*coinduction arbitrary*: ω *rule*: *alw-coinduct*) (*auto elim*: *enabled.cases*)

abbreviation $S \equiv stream\text{-}space\ (count\text{-}space\ UNIV)$

lemma *in-S* [*measurable* (*raw*)]: $x \in space\ S$
by (*simp add*: *space-stream-space*)

inductive-simps *enabled-iff*: *enabled* $s\ \omega$

lemma *enabled-Stream*: *enabled* $x\ (y\ \#\#\ \omega) \longleftrightarrow y \in K\ x \wedge enabled\ y\ \omega$
by (*subst enabled-iff*) *auto*

lemma *measurable-enabled*[*measurable*]:
Measurable.pred (*stream-space* (*count-space UNIV*)) (*enabled s*) (**is** *Measurable.pred ?S* -)
unfolding *enabled-def*
proof (*coinduction arbitrary*: *s rule*: *measurable-gfp2-coinduct*)
case (*step A s*)
then have [*measurable*]: $\bigwedge t.$ *Measurable.pred ?S* $(A\ t)$ **by** *auto*
have $*$: $\bigwedge x.\ (\exists \omega\ t.\ s = t \wedge x = \omega \wedge A\ (shd\ \omega)\ (stl\ \omega) \wedge shd\ \omega \in set\text{-}pmf\ (K$

```
t)) ⟷
    (∃ t∈K s. A t (stl x) ∧ t = shd x)
    by auto
  note countable-set-pmf[simp]
  show ?case
    unfolding ∗ by measurable
qed (auto simp: inf-continuous-def)

lemma enabled-iff-snth: enabled s ω ⟷ (∀ i. ω !! i ∈ K ((s ## ω) !! i))
proof safe
  fix i assume enabled s ω then show ω !! i ∈ K ((s ## ω) !! i)
    by (induct i arbitrary: s ω)
       (force elim: enabled.cases)+
next
  assume ∀ i. ω !! i ∈ set-pmf (K ((s ## ω) !! i)) then show enabled s ω
    by (coinduction arbitrary: s ω)
       (auto elim: allE[of - Suc i for i] allE[of - 0])
qed

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

lemma force-enabled-in-set-pmf[simp, intro]: shd (force-enabled x ω) ∈ K x
  by (auto simp: some-in-eq set-pmf-not-empty)

lemma enabled-force-enabled: enabled x (force-enabled x ω)
  by (coinduction arbitrary: x ω) (auto simp: some-in-eq set-pmf-not-empty)

lemma force-enabled: enabled x ω ⟹ force-enabled x ω = ω
  by (coinduction arbitrary: x ω) (auto elim: enabled.cases)

lemma Ex-enabled: ∃ω. enabled x ω
  by (rule exI[of - force-enabled x undefined] enabled-force-enabled)+

lemma measurable-force-enabled: force-enabled x ∈ measurable S S
proof (rule measurable-stream-space2)
  fix n show (λω. force-enabled x ω !! n) ∈ measurable S (count-space UNIV)
  proof (induction n arbitrary: x)
    case (Suc n) show ?case
      apply simp
      apply (rule measurable-compose-countable'[OF measurable-compose[OF mea-
surable-stl Suc], where I=set-pmf (K x)])
      apply (rule measurable-compose[OF measurable-shd])
      apply (auto simp: countable-set-pmf some-in-eq set-pmf-not-empty)
      done
  qed (auto intro!: measurable-compose[OF measurable-shd])
qed
```

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

lemma *sets-D*: *sets D* = *sets* (*stream-space* $(\Pi_M\ s{\in}UNIV.$ *count-space UNIV*))
  by (*intro sets-stream-space-cong sets-PiM-cong*) *simp-all*

lemma *space-D*: *space D* = *space* (*stream-space* $(\Pi_M\ s{\in}UNIV.$ *count-space UNIV*))
  using *sets-eq-imp-space-eq*[*OF sets-D*] **.**

lemma *measurable-D-D*: *measurable D D* =
    measurable (*stream-space* $(\Pi_M\ s{\in}UNIV.$ *count-space UNIV*)) (*stream-space* $(\Pi_M\ s{\in}UNIV.$ *count-space UNIV*))
  by (*simp add*: *measurable-def space-D sets-D*)

primcorec *walk* :: $'s \Rightarrow ('s \Rightarrow 's)$ *stream* $\Rightarrow$ $'s$ *stream* **where**
  shd (*walk s* ω) = (*if shd* ω *s* $\in$ *K s then shd* ω *s else* (*SOME t*. *t* $\in$ *K s*))
| *stl* (*walk s* ω) = *walk* (*if shd* ω *s* $\in$ *K s then shd* ω *s else* (*SOME t*. *t* $\in$ *K s*)) (*stl* ω)

lemma *enabled-walk*: *enabled s* (*walk s* ω)
  by (*coinduction arbitrary*: *s* ω) (*auto simp*: *some-in-eq set-pmf-not-empty*)

lemma *measurable-walk*[*measurable*]: *walk s* $\in$ *measurable D S*
proof −
  note *measurable-compose*[*OF measurable-snth*, *intro*!]
  note *measurable-compose*[*OF measurable-component-singleton*, *intro*!]
  note *if-weak-cong*[*cong del*]
  note *measurable-g* = *measurable-compose-countable'*[*OF - - countable-reachable*]

  define *n* :: *nat* **where** *n* = *0*
  define *g* **where** *g* = $(\lambda$-::$('s \Rightarrow 's)$ *stream*. *s*)
  then have *g* $\in$ *measurable D* (*count-space* (*acc* `` {*s*}))
    by *auto*
  then have $(\lambda x.$ *walk* (*g x*) (*sdrop n x*)) $\in$ *measurable D S*
  proof (*coinduction arbitrary*: *g n rule*: *measurable-stream-coinduct*)
    case (*shd g*) **show** *?case*
      by (*fastforce intro*: *measurable-g*[*OF - shd*])
  next
    case (*stl g*) **show** *?case*
      by (*fastforce simp add*: *sdrop.simps*[*symmetric*] *some-in-eq set-pmf-not-empty*
                simp del: *sdrop.simps intro*: *rtrancl-into-rtrancl measurable-g*[*OF - stl*])
  qed
  then show *?thesis*
    by (*simp add*: *g-def n-def*)
qed

3.2 Trace Space for Discrete-Time Markov Chains

definition $T :: 's \Rightarrow 's\ stream\ measure$ **where**
$T\ s = distr\ (stream\text{-}space\ (\Pi_M\ s{\in}UNIV.\ K\ s))\ S\ (walk\ s)$

lemma *space-T*[*simp*]: $space\ (T\ s) = space\ S$
by (*simp add*: *T-def*)

lemma *sets-T*[*simp*, *measurable-cong*]: $sets\ (T\ s) = sets\ S$
by (*simp add*: *T-def*)

lemma *measurable-T1*[*simp*]: $measurable\ (T\ s)\ M = measurable\ S\ M$
by (*intro measurable-cong-sets*) *simp-all*

lemma *measurable-T2*[*simp*]: $measurable\ M\ (T\ s) = measurable\ M\ S$
by (*intro measurable-cong-sets*) *simp-all*

lemma *in-measurable-T1*[*measurable* (*raw*)]: $f \in measurable\ S\ M \Longrightarrow f \in measurable\ (T\ s)\ M$
by *simp*

lemma *in-measurable-T2*[*measurable* (*raw*)]: $f \in measurable\ M\ S \Longrightarrow f \in measurable\ M\ (T\ s)$
by *simp*

lemma *AE-T-enabled*: $AE\ \omega\ in\ T\ s.\ enabled\ s\ \omega$
unfolding *T-def* **by** (*simp add*: *AE-distr-iff enabled-walk*)

sublocale *T*: *prob-space T s* **for** *s*
proof −
interpret *P*: *product-prob-space K UNIV* **..**
interpret *prob-space stream-space* $(\Pi_M\ s{\in}UNIV.\ K\ s)$
by (*rule P.prob-space-stream-space*)
fix *s* **show** *prob-space* $(T\ s)$
by (*simp add*: *T-def prob-space-distr*)
qed

lemma *emeasure-T-const*[*simp*]: $emeasure\ (T\ s)\ (space\ S) = 1$
using *T.emeasure-space-1*[*of s*] **by** *simp*

lemma *nn-integral-T*:
assumes *f*[*measurable*]: $f \in borel\text{-}measurable\ S$
shows $(\int^+ X.\ f\ X\ \partial T\ s) = (\int^+ t.\ (\int^+ \omega.\ f\ (t\ \#\#\ \omega)\ \partial T\ t)\ \partial K\ s)$
proof −
interpret *product-prob-space K UNIV* **..**
interpret *D*: *prob-space stream-space* $(\Pi_M\ s{\in}UNIV.\ K\ s)$
by (*rule prob-space-stream-space*)

have *T*: $\bigwedge f\ s.\ f \in borel\text{-}measurable\ S \Longrightarrow (\int^+ X.\ f\ X\ \partial T\ s) = (\int^+ \omega.\ f\ (walk\ s\ \omega)\ \partial D)$

by (*simp add*: *T-def nn-integral-distr*)

have $(\int^+ X.\ f\ X\ \partial T\ s) = (\int^+ \omega.\ f\ (walk\ s\ \omega)\ \partial D)$
by (*rule T*) *measurable*
also have $\ldots = (\int^+ d.\ \int^+ \omega.\ f\ (walk\ s\ (d\ \#\#\ \omega))\ \partial D\ \partial\Pi_M\ i{\in}UNIV.\ K\ i)$
by (*simp add*: *P.nn-integral-stream-space*)
also have $\ldots = (\int^+ d.\ (\int^+ \omega.\ f\ (d\ s\ \#\#\ walk\ (d\ s)\ \omega) * indicator\ \{t.\ t \in K\ s\}\ (d\ s)\ \partial D)\ \partial\Pi_M\ i{\in}UNIV.\ K\ i)$
apply (*rule nn-integral-cong-AE*)
apply (*subst walk.ctr*)
apply (*simp add*: *frequently-def cong del*: *if-weak-cong*)
apply (*auto simp*: *AE-measure-pmf-iff intro*: *AE-component*)
done
also have $\ldots = (\int^+ d.\ \int^+ \omega.\ f\ (d\ s\ \#\#\ \omega) * indicator\ (K\ s)\ (d\ s)\ \partial T\ (d\ s)\ \partial\Pi_M\ i{\in}UNIV.\ K\ i)$
by (*subst T*) (*simp-all split*: *split-indicator*)
also have $\ldots = (\int^+ t.\ \int^+ \omega.\ f\ (t\ \#\#\ \omega) * indicator\ (K\ s)\ t\ \partial T\ t\ \partial K\ s)$
by (*subst* (*2*) *PiM-component*[*symmetric*]) (*simp-all add*: *nn-integral-distr*)
also have $\ldots = (\int^+ t.\ \int^+ \omega.\ f\ (t\ \#\#\ \omega)\ \partial T\ t\ \partial K\ s)$
by (*rule nn-integral-cong-AE*) (*simp add*: *AE-measure-pmf-iff*)
finally show *?thesis* **.**
qed

lemma *nn-integral-T-gfp*:
fixes *g*
defines $l \equiv \lambda f\ \omega.\ g\ (shd\ \omega)\ (f\ (stl\ \omega))$
assumes [*measurable*]: *case-prod g* $\in$ *borel-measurable* (*count-space UNIV* $\bigotimes_M$ *borel*)
assumes *cont-g*[*THEN inf-continuous-compose*, *order-continuous-intros*]: $\bigwedge s.$ *inf-continuous* $(g\ s)$
assumes *int-g*: $\bigwedge f\ s.\ f \in$ *borel-measurable S* $\Longrightarrow (\int^+ \omega.\ g\ s\ (f\ \omega)\ \partial T\ s) = g\ s\ (\int^+ \omega.\ f\ \omega\ \partial T\ s)$
assumes *bnd-g*: $\bigwedge f\ s.\ g\ s\ f \le b\ 0 \le b\ b < \infty$
shows $(\int^+ \omega.\ gfp\ l\ \omega\ \partial T\ s) = gfp\ (\lambda f\ s.\ \int^+ t.\ g\ t\ (f\ t)\ \partial K\ s)\ s$
proof (*rule nn-integral-gfp*)
show $\bigwedge s.$ *sets* $(T\ s) =$ *sets S* $\bigwedge F.\ F \in$ *borel-measurable S* $\Longrightarrow l\ F \in$ *borel-measurable S*
by (*auto simp*: *l-def*)
show $\bigwedge s.$ *emeasure* $(T\ s)$ $(space\ (T\ s)) \neq 0$
by (*rewrite T.emeasure-space-1*) *simp*
{ fix *s F*
have $integral^N\ (T\ s)\ (l\ F) \le (\int^+ x.\ b\ \partial T\ s)$
by (*intro nn-integral-mono*) (*simp add*: *l-def bnd-g*)
also have $\ldots < \infty$
using *bnd-g* **by** *simp*
finally show $integral^N\ (T\ s)\ (l\ F) < \infty$ **. }**
show *inf-continuous* $(\lambda f\ s.\ \int^+ t.\ g\ t\ (f\ t)\ \partial K\ s)$
proof (*intro order-continuous-intros*)
fix *f s*

have $(\int^+ t.\ g\ t\ (f\ t)\ \partial K\ s) \leq (\int^+ t.\ b\ \partial K\ s)$
by (*intro nn-integral-mono bnd-g*)
also have $\ldots < \infty$
using *bnd-g* **by** *simp*
finally show $(\int^+ t.\ g\ t\ (f\ t)\ \partial K\ s) \neq \infty$
by *simp*
qed *simp*
next
fix *s* **and** *F* :: *'s stream* ⇒ *ennreal* **assume** *F* ∈ *borel-measurable S*
then show $integral^N\ (T\ s)\ (l\ F) = (\int^+ t.\ g\ t\ (integral^N\ (T\ t)\ F)\ \partial K\ s)$
by (*rewrite nn-integral-T*) (*simp-all add*: *l-def int-g*)
qed (*auto intro*!: *order-continuous-intros simp*: *l-def*)

lemma *nn-integral-T-lfp*:
fixes *g*
defines $l \equiv \lambda f\ \omega.\ g\ (shd\ \omega)\ (f\ (stl\ \omega))$
assumes [*measurable*]: *case-prod g* ∈ *borel-measurable* (*count-space UNIV* $\bigotimes_M$ *borel*)
assumes *cont-g*[*THEN sup-continuous-compose*, *order-continuous-intros*]: $\bigwedge s.$ *sup-continuous* (*g s*)
assumes *int-g*: $\bigwedge f\ s.\ f \in$ *borel-measurable S* ⟹ $(\int^+\omega.\ g\ s\ (f\ \omega)\ \partial T\ s) = g\ s\ (\int^+\omega.\ f\ \omega\ \partial T\ s)$
shows $(\int^+\omega.\ lfp\ l\ \omega\ \partial T\ s) = lfp\ (\lambda f\ s.\ \int^+ t.\ g\ t\ (f\ t)\ \partial K\ s)\ s$
proof (*rule nn-integral-lfp*)
show $\bigwedge s.$ *sets* (*T s*) = *sets S* $\bigwedge F.$ *F* ∈ *borel-measurable S* ⟹ *l F* ∈ *borel-measurable S*
by (*auto simp*: *l-def*)
next
fix *s* **and** *F* :: *'s stream* ⇒ *ennreal* **assume** *F* ∈ *borel-measurable S*
then show $integral^N\ (T\ s)\ (l\ F) = (\int^+ t.\ g\ t\ (integral^N\ (T\ t)\ F)\ \partial K\ s)$
by (*rewrite nn-integral-T*) (*simp-all add*: *l-def int-g*)
qed (*auto simp*: *l-def intro*!: *order-continuous-intros*)

lemma *emeasure-Collect-T*:
assumes *f*[*measurable*]: *Measurable.pred S P*
shows *emeasure* (*T s*) {*x*∈*space* (*T s*). *P x*} = $(\int^+ t.$ *emeasure* (*T t*) {*x*∈*space* (*T t*). *P* (*t ##* *x*)} *∂K s*)
apply (*subst* (*1 2*) *nn-integral-indicator*[*symmetric*])
apply *simp*
apply *simp*
apply (*subst nn-integral-T*)
apply (*auto intro*!: *nn-integral-cong simp add*: *space-stream-space indicator-def*)
done

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*))
by (*simp add*: *AE-iff-nn-integral nn-integral-T*[**where** *s*=*x*])
(*auto simp add*: *nn-integral-0-iff-AE AE-measure-pmf-iff split*: *split-indicator*)

lemma *AE-T-alw*:
assumes [*measurable*]: *Measurable.pred S P*
assumes *P*: $\bigwedge s.\ (x,\ s) \in acc \Longrightarrow AE\ \omega\ in\ T\ s.\ P\ \omega$
shows *AE ω in T x. alw P ω*
proof −
define *F* **where** $F = (\lambda p\ x.\ P\ x \wedge p\ (stl\ x))$
have [*measurable*]: $\bigwedge p.\ Measurable.pred\ S\ p \Longrightarrow Measurable.pred\ S\ (F\ p)$
by (*auto simp*: *F-def*)
have *almost-everywhere* (*T s*) ((*F* ^^ *i*) *top*)
if (*x*, *s*) ∈ *acc* **for** *i s*
using *that*
proof (*induction i arbitrary*: *s*)
case (*Suc i*) **then show** *?case*
apply *simp*
apply (*subst F-def*)
apply (*simp add*: *P*)
apply (*subst AE-T-iff*)
apply (*measurable*; *simp*)
apply (*auto dest*: *rtrancl-into-rtrancl*)
done
qed *simp*
then have *almost-everywhere* (*T x*) (*gfp F*)
by (*subst inf-continuous-gfp*) (*auto simp*: *inf-continuous-def AE-all-countable F-def*)
then show *?thesis*
by (*simp add*: *alw-def F-def*)
qed

lemma *emeasure-suntil-disj*:
assumes [*measurable*]: *Measurable.pred S P*
assumes ∗: $\bigwedge t.\ AE\ \omega\ in\ T\ t.\ \neg\ (P \sqcap (HLD\ X \sqcap nxt\ (HLD\ X\ suntil\ P)))\ \omega$
shows *emeasure* (*T s*) {ω∈*space* (*T s*). (*HLD X suntil P*) ω} =
lfp (λ*F s*. *emeasure* (*T s*) {ω∈*space* (*T s*). *P* ω} + (∫ $^+$*t*. *F t* ∗ *indicator X t* ∂*K s*)) *s*
unfolding *suntil-lfp*
proof (*rule emeasure-lfp*[**where** *s*=*s*])
fix *F t* **assume** [*measurable*]: *Measurable.pred* (*T s*) *F* **and**
F: $F \leq lfp\ (\lambda a\ b.\ P\ b \vee HLD\ X\ b \wedge a\ (stl\ b))$
have *emeasure* (*T t*) {ω ∈ *space* (*T s*). *P* ω ∨ *HLD X* ω ∧ *F* (*stl* ω)} =
emeasure (*T t*) {ω ∈ *space* (*T t*). *P* ω} + *emeasure* (*T t*) {ω∈*space* (*T t*). *HLD X* ω ∧ *F* (*stl* ω)}
proof (*rule emeasure-add-AE*)
show *AE x in T t*. ¬ (*x* ∈ {ω ∈ *space* (*T t*). *P* ω} ∧ *x* ∈ {ω ∈ *space* (*T t*). *HLD X* ω ∧ *F* (*stl* ω)})
using ∗ **by** *eventually-elim* (*insert F*, *auto simp*: *suntil-lfp*[*symmetric*])
qed *auto*
also have *emeasure* (*T t*) {ω∈*space* (*T t*). *HLD X* ω ∧ *F* (*stl* ω)} =
(∫ $^+$*t*. *emeasure* (*T t*) {ω ∈ *space* (*T s*). *F* ω} ∗ *indicator X t* ∂*K t*)

by (*subst emeasure-Collect-T*) (*auto intro*!: *nn-integral-cong split*: *split-indicator*)
finally show *emeasure* (*T t*) {ω ∈ *space* (*T s*). *P* ω ∨ *HLD X* ω ∧ *F* (*stl* ω)} =
emeasure (*T t*) {ω ∈ *space* (*T t*). *P* ω} + (∫⁺ *t*. *emeasure* (*T t*) {ω ∈ *space* (*T s*). *F* ω} ∗ *indicator X t* ∂*K t*) **.**
qed (*auto intro*!: *order-continuous-intros split*: *split-indicator*)

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*)
by (*subst emeasure-Collect-T*)
(*auto intro*!: *nn-integral-cong-AE simp*: *AE-measure-pmf-iff split*: *split-indicator*)

lemma *emeasure-HLD*:
emeasure (*T s*) {ω∈*space* (*T s*). *HLD X* ω} = *emeasure* (*K s*) *X*
using *emeasure-HLD-nxt*[*of* λω. *True s X*] *T.emeasure-space-1* **by** *simp*

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 −
let *?P* = *emeasure* (*T t*) {ω∈*space* (*T t*). *P* ω}
let *?F* = λ*Q F s*. *emeasure* (*T s*) {ω∈*space* (*T s*). *Q* ω} + (∫⁺*t*′. *F t*′ ∗ *indicator* (− {*t*}) *t*′ ∂*K s*)
have *emeasure* (*T s*) {*x*∈*space* (*T s*). (*HLD* (−{*t*}) *suntil* ({*t*} · *P*)) *x*} = *lfp* (*?F* ({*t*} · *P*)) *s*
by (*rule emeasure-suntil-disj*) (*auto simp*: *HLD-iff*)
also have *lfp* (*?F* ({*t*} · *P*)) = (λ*s*. *lfp* (*?F* (*HLD* {*t*})) *s* ∗ *?P*)
proof (*rule lfp-transfer*[*symmetric*, **where** α=λ*x s*. *x s* ∗ *emeasure* (*T t*) {ω∈*space* (*T t*). *P* ω}])
fix *F* **show** (λ*s*. *?F* (*HLD* {*t*}) *F s* ∗ *?P*) = *?F* ({*t*} · *P*) (λ*s*. *F s* ∗ *?P*)
unfolding *emeasure-HLD emeasure-HLD-nxt*[*OF assms*] *distrib-right*
by (*auto simp*: *fun-eq-iff nn-integral-multc*[*symmetric*]
intro!: *arg-cong2*[**where** *f*=(+)] *nn-integral-cong ac-simps*
split: *split-indicator*)
qed (*auto intro*!: *order-continuous-intros sup-continuous-mono lfp-upperbound*
intro: *le-funI add-nonneg-nonneg*
simp: *bot-ennreal split*: *split-indicator*)
also have *lfp* (*?F* (*HLD* {*t*})) *s* = *emeasure* (*T s*) {*x*∈*space* (*T s*). (*HLD* (−{*t*}) *suntil HLD* {*t*}) *x*}
by (*rule emeasure-suntil-disj*[*symmetric*]) (*auto simp*: *HLD-iff*)
finally show *?thesis*
by (*simp add*: *HLD-iff*[*abs-def*] *ev-eq-suntil*)
qed

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*)
apply (*subst* (*1 2 3*) *T.prob-Collect-eq-1*[*symmetric*])
apply *simp*
apply *simp*
apply *simp*
apply (*simp-all add*: *measure-def emeasure-suntil-HLD del*: *space-T nxt.simps*)
apply (*auto simp*: *T.emeasure-eq-measure mult-eq-1*)
done

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 −
let *?M* = λ*P s. emeasure* (*T s*) {*ω*∈*space* (*T s*). *P ω*}
let *?t* = *HLD* {*t*} **and** *?t'* = *HLD* {*t'*}
define *N* **where** *N* = *alw* (*ev ?t*) *aand alw* (*not* (*?t aand nxt ?t'*))
let *?until* = *not ?t suntil* (*?t aand nxt* (*not ?t' aand nxt N*))
have *N-stl*: ⋀*ω. N ω* ⟹ *N* (*stl ω*)
by (*auto simp*: *N-def*)
have [*measurable*]: *Measurable.pred S N*
unfolding *N-def* **by** *measurable*

let *?c* = *pmf* (*K t*) *t'*
let *?R* = λ*x. 1* ⊓ *x* ∗ (*1* − *ennreal ?c*)
have *mono ?R*
by (*intro monoI mult-right-mono inf-mono*) (*auto simp*: *mono-def field-simps*)
have ⋀*s. ?M N s* ≤ *gfp ?R*
proof (*induction rule*: *gfp-ordinal-induct*[*OF* ‹*mono ?R*›])
fix *x s* **assume** *x*: ⋀*s. ?M N s* ≤ *x*
{ fix *ω* **assume** *N ω*
then have *ev* (*HLD* {*t*}) *ω N ω*
by (*auto simp*: *N-def*)
then have *?until ω*
by (*induct rule*: *ev-induct-strong*) (*auto simp*: *N-def intro*: *suntil.intros dest*: *N-stl*) **}**
then have *?M N s* ≤ *?M ?until s*
by (*intro emeasure-mono-AE*) *auto*
also have ... = *?M* (*ev ?t*) *s* ∗ *?M* (*not ?t' aand nxt N*) *t*
by (*simp-all add*: *emeasure-suntil-HLD del*: *nxt.simps space-T*)
also have ... ≤ *?M* (*ev ?t*) *s* ∗ (∫⁺*s'. 1* ⊓ *x* ∗ *indicator* (*UNIV* − {*t'*}) *s'* ∂*K t*)
by (*auto intro*!: *mult-left-mono nn-integral-mono T.measure-le-1 emeasure-mono*

```
            split: split-indicator simp add: x emeasure-Collect-T[of - t] simp del:
space-T)
    also have ... ≤ 1 * (∫⁺s'. 1 ⊓ x * indicator (UNIV - {t'}) s' ∂K t)
      by (intro mult-right-mono T.measure-le-1) simp
    finally show ?M N s ≤ 1 ⊓ x * (1 - ennreal ?c)
      by (subst (asm) nn-integral-cmult-indicator) (auto simp: emeasure-Diff emea-
sure-pmf-single)
  qed (auto intro: Inf-greatest)
  also
  from ‹mono ?R› have gfp ?R = ?R (gfp ?R) by (rule gfp-unfold)
  then have gfp ?R ≤ ?R (gfp ?R) by simp
  with assms[THEN pmf-positive] have gfp ?R ≤ 0
    by (cases gfp ?R)
     (auto simp: top-unique inf-ennreal.rep-eq field-simps mult-le-0-iff ennreal-1[symmetric]
                 pmf-le-1 ennreal-minus ennreal-mult[symmetric] ennreal-le-iff2
inf-min min-def
           simp del: ennreal-1
           split: if-split-asm)
  finally have ⋀s. AE ω in T s. ¬ N ω
    by (subst AE-iff-measurable[OF - refl]) (auto intro: antisym simp: le-fun-def)
  then have AE ω in T s. alw (not N) ω
    by (intro AE-T-alw) auto
  moreover
  { fix ω assume alw (ev (HLD {t})) ω
    then have alw (alw (ev (HLD {t}))) ω
      unfolding alw-alw .
    moreover assume alw (not N) ω
    then have alw (alw (ev (HLD {t})) impl ev (HLD {t} aand nxt (HLD {t'})))
ω
      unfolding N-def not-alw-iff not-ev-iff de-Morgan-disj de-Morgan-conj not-not
imp-conv-disj .
    ultimately have alw (ev (HLD {t} aand nxt (HLD {t'}))) ω
      by (rule alw-mp) }
  then have ∀ω. alw (not N) ω ⟶ fair t t' ω
    by (auto simp: fair-def)
  ultimately show ?thesis
    by (simp add: eventually-mono)
qed

lemma enabled-imp-trancl:
  assumes alw (HLD B) ω enabled s ω
  shows alw (HLD (acc-on B `` {s})) ω
proof −
  define t where t = s
  then have (s, t) ∈ acc-on B
    by auto
  moreover note ‹alw (HLD B) ω›
  moreover note ‹enabled s ω›[unfolded ‹t == s›[symmetric]]
  ultimately show ?thesis
```

```
  proof (coinduction arbitrary: t ω rule: alw-coinduct)
    case stl from this(1,2,3) show ?case
      by (auto simp: enabled.simps[of - ω] alw.simps[of - ω] HLD-iff
               intro!: exI[of - shd ω] rtrancl-trans[of s t])
  next
    case (alw t ω) then show ?case
    by (auto simp: HLD-iff enabled.simps[of - ω] alw.simps[of - ω] intro!: rtrancl-trans[of
s t])
  qed
qed

lemma AE-T-reachable: AE ω in T s. alw (HLD (acc `` {s})) ω
  using AE-T-enabled
proof eventually-elim
  fix ω assume enabled s ω
  from enabled-imp-trancl[of UNIV, OF - this]
  show alw (HLD (acc `` {s})) ω
    by (auto simp: HLD-iff[abs-def] all-imp-alw)
qed

lemma AE-T-all-fair: AE ω in T s. ∀(t,t')∈SIGMA t:UNIV. K t. fair t t' ω
proof −
  let ?Rn = SIGMA s:(acc `` {s}). K s
  have AE ω in T s. ∀(t,t')∈?Rn. fair t t' ω
  proof (subst AE-ball-countable)
    show countable ?Rn
      by (intro countable-SIGMA countable-rtrancl[OF countable-Image]) (auto
simp: Image-def)
  qed (auto intro!: AE-T-fair)
  then show ?thesis
    using AE-T-reachable
  proof (eventually-elim, safe)
    fix ω t t' assume ∀(t,t')∈?Rn. fair t t' ω t' ∈ K t and alw: alw (HLD (acc ``
{s})) ω
    moreover
    { assume t ∉ acc `` {s}
      then have alw (not (HLD {t})) ω
        by (intro alw-mono[OF alw]) (auto simp: HLD-iff)
      then have not (alw (ev (HLD {t}))) ω
        unfolding not-alw-iff not-ev-iff by auto
      then have fair t t' ω
        unfolding fair-def by auto }
    ultimately show fair t t' ω
      by auto
  qed
qed

lemma fair-imp: assumes fair t t' ω alw (ev (HLD {t})) ω shows alw (ev (HLD
{t'})) ω
```

proof −
{ fix ω **assume** *ev* (*HLD* $\{t\}$ *aand nxt* (*HLD* $\{t'\}$)) ω **then have** *ev* (*HLD* $\{t'\}$) ω
by *induction auto* **}**
with *assms* **show** *?thesis*
by (*auto simp*: *fair-def elim*!: *alw-mp intro*: *all-imp-alw*)
qed

lemma *AE-T-ev-HLD*:
assumes *exiting*: $\bigwedge t.\ (s,\ t) \in$ *acc-on* $(-B) \Longrightarrow \exists\, t' \in B.\ (t,\ t') \in$ *acc*
assumes *fin*: *finite* (*acc-on* $(-B)$ `` $\{s\}$)
shows *AE* ω *in T s. ev* (*HLD B*) ω
using *AE-T-all-fair AE-T-enabled*
proof *eventually-elim*
fix ω **assume** *fair*: $\forall\, (t,\ t') \in$ (*SIGMA s*:*UNIV*. *K s*). *fair t t'* ω **and** *enabled s* ω

show *ev* (*HLD B*) ω
proof (*rule ccontr*)
assume ¬ *ev* (*HLD B*) ω
then have *alw* (*HLD* (− *B*)) ω
by (*simp add*: *not-ev-iff HLD-iff*[*abs-def*])
from *enabled-imp-trancl*[*OF this* ‹*enabled s* ω›]
have *alw* (*HLD* (*acc-on* $(-B)$ `` $\{s\}$)) ω
by (*simp add*: *Diff-eq*)
from *pigeonhole-stream*[*OF this fin*]
obtain *t* **where** $(s,\ t) \in$ *acc-on* $(-B)$ *alw* (*ev* (*HLD* $\{t\}$)) ω
by *auto*
from *exiting*[*OF this*(*1*)] **obtain** t' **where** $(t,\ t') \in$ *acc* $t' \in B$
by *auto*
from *this*(*1*) **have** *alw* (*ev* (*HLD* $\{t'\}$)) ω
proof *induction*
case (*step u w*) **then show** *?case*
using *fair fair-imp*[*of u w* ω] **by** *auto*
qed *fact*
{ assume *ev* (*HLD* $\{t'\}$) ω **then have** *ev* (*HLD B*) ω
by (*rule ev-mono*) (*auto simp*: *HLD-iff* ‹$t' \in B$›) **}**
then show *False*
using ‹*alw* (*ev* (*HLD* $\{t'\}$)) ω› ‹¬ *ev* (*HLD B*) ω› **by** *auto*
qed
qed

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* ω *in T s. ev* (*HLD X*) ω
proof (*rule AE-T-ev-HLD*)
show $\bigwedge t.\ (s,\ t) \in$ *acc-on* $(-\ X) \Longrightarrow \exists\, t' \in X.\ (t,\ t') \in$ *acc*
using *exiting* **by** (*auto elim*: *rtrancl.cases*)
have *acc-on* $(-\ X)$ `` $\{s\} \subseteq -X \cup \{s\}$

by (*auto elim*: *rtrancl.cases*)
with *fin* **show** *finite* (*acc-on* (− *X*) `` {*s*})
by (*auto dest*: *finite-subset*)
qed

lemma *AE-T-max-sfirst*:
assumes [*measurable*]: *Measurable.pred S X*
assumes *AE*: *AE ω in T c. sfirst X* (*c ## ω*) < ∞ **and** *0* < *e*
shows ∃ *N*::*nat*. *P*(*ω in T c. N* < *sfirst X* (*c ## ω*)) < *e* (**is** ∃ *N*. *?P N* < *e*)
proof −
have *?P* ⟶ *measure* (*T c*) (⋂ *N*::*nat*. {*bT* ∈ *space* (*T c*). *N* < *sfirst X* (*c ## bT*)})
using *dual-order.strict-trans enat-ord-simps*(*2*)
by (*intro T.finite-Lim-measure-decseq*) (*force simp*: *decseq-Suc-iff simp del*: *enat-ord-simps*)+
also have *measure* (*T c*) (⋂ *N*::*nat*. {*bT* ∈ *space* (*T c*). *N* < *sfirst X* (*c ## bT*)}) =
P(*bT in T c. sfirst X* (*c ## bT*) = ∞)
by (*auto simp del*: *not-infinity-eq intro*!: *arg-cong*[**where** *f*=*measure* (*T c*)])
(*metis less-irrefl not-infinity-eq*)
also have *P*(*bT in T c. sfirst X* (*c ## bT*) = ∞) = *0*
using *AE* **by** (*intro T.prob-eq-0-AE*) *auto*
finally have ∃ *N*. ∀ *n*≥*N*. *norm* (*?P n* − *0*) < *e*
using ‹*0* < *e*› **by** (*rule LIMSEQ-D*)
then show *?thesis*
by (*auto simp*: *measure-nonneg*)
qed

3.4 First Hitting Time

lemma *nn-integral-sfirst-finite'*:
assumes *s* ∉ *H*
assumes [*simp*]: *finite* (*acc-on* (−*H*) `` {*s*})
assumes *until*: *AE ω in T s. ev* (*HLD H*) *ω*
shows (∫ + *ω*. *sfirst* (*HLD H*) *ω* ∂*T s*) ≠ ∞
proof −
have *R-ne*[*simp*]: *acc-on* (−*H*) `` {*s*} ≠ {}
by *auto*
have [*measurable*]: *H* ∈ *sets* (*count-space UNIV*)
by *simp*

let *?Pf* = λ*n t*. *P*(*ω in T t. enat n* < *sfirst* (*HLD H*) (*t ## ω*))
have *Pf-mono*: ⋀ *N n t*. *N* ≤ *n* ⟹ *?Pf n t* ≤ *?Pf N t*
by (*auto intro*!: *T.finite-measure-mono simp del*: *enat-ord-code*(*1*) *simp*: *enat-ord-code*(*1*)[*symmetric*])

have *not-H*: ⋀ *t*. (*s*, *t*) ∈ *acc-on* (−*H*) ⟹ *t* ∉ *H*
using ‹*s* ∉ *H*› **by** (*auto elim*: *rtrancl.cases*)

have ∀ *F n in sequentially*. ∀ *t*∈*acc-on* (−*H*) ``{*s*}. *?Pf n t* < *1*

proof (*safe intro*!: *eventually-ball-finite*)
fix *t* **assume** (*s*, *t*) ∈ *acc-on* (−*H*)
then have *AE ω in T t. sfirst* (*HLD H*) (*t* ## *ω*) < ∞
unfolding *sfirst-finite*
proof *induction*
case (*step t u*) **with** *step.IH* **show** *?case*
by (*subst* (*asm*) *AE-T-iff*) (*auto simp*: *ev-Stream not-H*)
qed (*simp add*: *ev-Stream eventually-frequently-simps until*)
from *AE-T-max-sfirst*[*OF - this*, *of 1*]
obtain *N* **where** *?Pf N t* < *1* **by** *auto*
with *Pf-mono*[*of N*] **show** ∀ *F n in sequentially. ?Pf n t* < *1*
by (*auto simp*: *eventually-sequentially intro*: *le-less-trans*)
qed *simp*
then obtain *n* **where** ⋀*t*. (*s*, *t*) ∈ *acc-on* (−*H*) ⟹ *?Pf n t* < *1*
by (*auto simp*: *eventually-sequentially*)
moreover define *d* **where** *d* = *Max* (*?Pf n* ‘ *acc-on* (−*H*) ‘‘ {*s*})
ultimately have *d*: *0* ≤ *d d* < *1* ⋀*t*. (*s*, *t*) ∈ *acc-on* (−*H*) ⟹ *?Pf* (*Suc n*) *t* ≤ *d*
using *Pf-mono*[*of n Suc n*] **by** (*auto simp*: *Max-ge-iff measure-nonneg*)

let *?F* = *λF ω. if shd ω* ∈ *H then 0 else F* (*stl ω*) + *1* :: *ennreal*
have *sup-continuous ?F*
by (*intro order-continuous-intros*)
then have *mono ?F*
by (*rule sup-continuous-mono*)
have *lfp-nonneg*[*simp*]: ⋀*ω*. *0* ≤ *lfp ?F ω*
by (*subst lfp-unfold*[*OF* ‹*mono ?F*›]) *auto*

let *?I* = *λF s*. ∫ +*t*. (*if t* ∈ *H then 0 else F t* + *1*) ∂*K s*
have *sup-continuous ?I*
by (*intro order-continuous-intros*) *auto*
then have *mono ?I*
by (*rule sup-continuous-mono*)

define *p* **where** *p* = *Suc n* / (*1* − *d*)
have *p*: *p* = *Suc n* + *d* ∗ *p*
unfolding *p-def* **using** *d*(*1*,*2*) **by** (*auto simp*: *field-simps*)
have [*simp*]: *0* ≤ *p*
using *d*(*1*,*2*) **by** (*auto simp*: *p-def*)

have (∫ + *ω*. *sfirst* (*HLD H*) *ω* ∂*T s*) = (∫ + *ω*. *lfp ?F ω* ∂*T s*)
proof (*intro nn-integral-cong-AE*)
show *AE x in T s. sfirst* (*HLD H*) *x* = *lfp ?F x*
using *until*
proof *eventually-elim*
fix *ω* **assume** *ev* (*HLD H*) *ω* **then show** *sfirst* (*HLD H*) *ω* = *lfp ?F ω*
by (*induction rule*: *ev-induct-strong*;
subst lfp-unfold[*OF* ‹*mono ?F*›], *simp add*: *HLD-iff*[*abs-def*] *ac-simps max-absorb2*)

```
    qed
  qed
  also have ... = lfp (?I^^Suc n) s
    unfolding lfp-funpow[OF ‹mono ?I›]
    by (subst nn-integral-T-lfp)
       (auto simp: nn-integral-add max-absorb2 intro!: order-continuous-intros)
  also have lfp (?I^^Suc n) t ≤ p if (s, t) ∈ acc-on (−H) for t
    using that
  proof (induction arbitrary: t rule: lfp-ordinal-induct[of ?I^^Suc n])
    case (step S)
    have (?I^^i) S t ≤ i + ?Pf i t * ennreal p for i
      using step(3)
    proof (induction i arbitrary: t)
      case 0 then show ?case
        using T.prob-space step(1)
      by (auto simp add: zero-ennreal-def[symmetric] not-H zero-enat-def[symmetric]
one-ennreal-def[symmetric])
    next
      case (Suc i)
      then have t ∉ H
        by (auto simp: not-H)
      from Suc.prems have ⋀t'. t' ∈ K t ⟹ t' ∉ H ⟹ (s, t') ∈ acc-on (−H)
        by (rule rtrancl-into-rtrancl) (insert Suc.prems, auto dest: not-H)
      then have (?I ^^ Suc i) S t ≤ ?I (λt. i + ennreal (?Pf i t) * p) t
        by (auto simp: AE-measure-pmf-iff simp del: sfirst-eSuc space-T
              intro!: nn-integral-mono-AE add-mono max.mono Suc)
      also have ... ≤ (∫⁺ t. ennreal (Suc i) + ennreal P(ω in T t. enat i < sfirst
(HLD H) (t ## ω)) * p ∂K t)
        by (intro nn-integral-mono) auto
      also have ... ≤ Suc i + ennreal (?Pf (Suc i) t) * p
        unfolding T.emeasure-eq-measure[symmetric]
        by (subst (2) emeasure-Collect-T)
         (auto simp: ‹t ∉ H› eSuc-enat[symmetric] nn-integral-add nn-integral-multc
ennreal-of-nat-eq-real-of-nat)
      finally show ?case
        by (simp add: ennreal-of-nat-eq-real-of-nat)
    qed
    then have (?I^^Suc n) S t ≤ Suc n + ?Pf (Suc n) t * ennreal p .
    also have ... ≤ p
        using d step by (subst (2) p) (auto intro!: mult-right-mono simp: en-
nreal-of-nat-eq-real-of-nat ennreal-mult)
    finally show ?case .
  qed (auto simp: SUP-least intro!: mono-pow ‹mono ?I› simp del: funpow.simps)
  finally show ?thesis
    unfolding p-def by (auto simp: top-unique)
qed

lemma nn-integral-sfirst-finite:
  assumes [simp]: finite (acc-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 *cases*
assume $s \notin H$ **then show** *?thesis*
using *nn-integral-sfirst-finite'*[*of s H*] *until* **by** (*simp add*: *nn-integral-add*)
qed (*simp add*: *sfirst.simps*)

lemma *prob-T*:
assumes *P*: *Measurable.pred S P*
shows $\mathcal{P}(\omega\ in\ T\ s.\ P\ \omega) = (\int t.\ \mathcal{P}(\omega\ in\ T\ t.\ P\ (t\ \#\#\ \omega))\ \partial K\ s)$
using *emeasure-Collect-T*[*OF P*, *of s*] **unfolding** *T.emeasure-eq-measure*
by (*subst* (*asm*) *nn-integral-eq-integral*)
(*auto intro*!: *measure-pmf.integrable-const-bound*[**where** *B=1*])

lemma *T-subprob*[*measurable*]: $T \in measurable\ (measure\text{-}pmf\ I)\ (subprob\text{-}algebra\ S)$
by (*auto intro*!: *space-bind simp*: *space-subprob-algebra*) *unfold-locales*

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*:
$(\lambda s.\ distr\ (T\ s)\ S\ ((\#\#)\ s)) \in measurable\ (measure\text{-}pmf\ I)\ (subprob\text{-}algebra\ S)$
apply (*intro measurable-distr2*[*OF - T-subprob*])
apply (*subst measurable-cong-sets*[**where** M'=*count-space UNIV* $\bigotimes_M$ *S* **and** N'=*S*])
apply (*rule sets-pair-measure-cong*)
apply *auto*
done

lemma *sets-T'*: $sets\ (T'\ I) = sets\ S$
by (*simp add*: *T'-def*)

lemma *prob-space-T'*: *prob-space* $(T'\ I)$
unfolding *T'-def*
proof (*rule measure-pmf.prob-space-bind*)
show $AE\ s\ in\ I.\ prob\text{-}space\ (distr\ (T\ s)\ S\ ((\#\#)\ s))$
by (*intro AE-measure-pmf-iff*[*THEN iffD2*] *ballI T.prob-space-distr*) *simp*
qed (*rule distr-Stream-subprob*)

lemma *AE-T'*:
assumes [*measurable*]: *Measurable.pred S P*
shows $(AE\ x\ in\ T'\ I.\ P\ x) \longleftrightarrow (\forall s \in I.\ AE\ x\ in\ T\ s.\ P\ (s\ \#\#\ x))$
unfolding *T'-def* **by** (*simp add*: *AE-bind*[*OF distr-Stream-subprob*] *AE-measure-pmf-iff AE-distr-iff*)

lemma *emeasure-T'*:

assumes [*measurable*]: $X \in sets\ S$
shows *emeasure* $(T'\ I)\ X = (\int^+ s.\ emeasure\ (T\ s)\ \{\omega \in space\ S.\ s\ \#\#\ \omega \in X\}\ \partial I)$
unfolding *T'-def*
by (*simp add*: *emeasure-bind*[*OF - distr-Stream-subprob*] *emeasure-distr vimage-def Int-def conj-ac*)

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 −
interpret *T'*: *prob-space T' I* **by** (*rule prob-space-T'*)
show *?thesis*
using *emeasure-T'*[*of* $\{x \in space\ (T'\ I).\ P\ x\}$ *I*]
unfolding *T'.emeasure-eq-measure T.emeasure-eq-measure sets-eq-imp-space-eq*[*OF sets-T'*]
apply *simp*
apply (*subst* (*asm*) *nn-integral-eq-integral*)
apply (*auto intro*!: *measure-pmf.integrable-const-bound*[**where** *B=1*] *integral-cong arg-cong2*[**where** *f=measure*]
simp: *AE-measure-pmf measure-nonneg space-stream-space*)
done
qed

lemma *T-eq-T'*: $T\ s = T'\ (K\ s)$
proof (*rule measure-eqI*)
fix *X* **assume** *X*: $X \in sets\ (T\ s)$
then have [*measurable*]: $X \in sets\ S$
by *simp*
have *X-eq*: $X = \{x \in space\ (T\ s).\ x \in X\}$
using *sets.sets-into-space*[*OF X*] **by** *auto*
show $emeasure\ (T\ s)\ X = emeasure\ (T'\ (K\ s))\ X$
apply (*subst X-eq*)
apply (*subst emeasure-Collect-T*, *simp*)
apply (*subst emeasure-T'*, *simp*)
apply *simp*
done
qed (*simp add*: *sets-T'*)

lemma *T-eq-bind*: $T\ s = (measure\text{-}pmf\ (K\ s) \gg\!= (\lambda t.\ distr\ (T\ t)\ S\ ((\#\#)\ t)))$
by (*subst T-eq-T'*) (*simp add*: *T'-def*)

lemma *T-split*:
$T\ s = (T\ s \gg\!= (\lambda \omega.\ distr\ (T\ ((s\ \#\#\ \omega)\ !!\ n))\ S\ (\lambda \omega'.\ stake\ n\ \omega\ @-\ \omega')))$
proof (*induction n arbitrary*: *s*)
case *0* **then show** *?case*
apply (*simp add*: *distr-cong*[*OF refl sets-T*[*symmetric*, *of s*] *refl*])
apply (*subst bind-const'*)
apply *unfold-locales*

```
    ..
next
  case (Suc n)
  let ?K = measure-pmf (K s) and ?m = λn ω ω'. stake n ω @− ω'
  note sets-stream-space-cong[simp, measurable-cong]

  have T s = (?K ⤜ (λt. distr (T t) S ((##) t)))
    by (rule T-eq-bind)
  also have ... = (?K ⤜ (λt. distr (T t ⤜ (λω. distr (T ((t ## ω) !! n)) S
(?m n ω))) S ((##) t)))
    unfolding Suc[symmetric] ..
  also have ... = (?K ⤜ (λt. T t ⤜ (λω. distr (distr (T ((t ## ω) !! n)) S
(?m n ω)) S ((##) t))))
     by (simp add: distr-bind[where K=S, OF measurable-distr2[where M=S]]
space-stream-space)
  also have ... = (?K ⤜ (λt. T t ⤜ (λω. distr (T ((t ## ω) !! n)) S (?m
(Suc n) (t ## ω)))))
    by (simp add: distr-distr space-stream-space comp-def)
  also have ... = (?K ⤜ (λt. distr (T t) S ((##) t) ⤜ (λω. distr (T (ω !! n))
S (?m (Suc n) ω))))
     by (simp add: space-stream-space bind-distr[OF - measurable-distr2[where
M=S]] del: stake.simps)
  also have ... = (T s ⤜ (λω. distr (T (ω !! n)) S (?m (Suc n) ω)))
    unfolding T-eq-bind[of s]
   by (subst bind-assoc[OF measurable-distr2[where M=S] measurable-distr2[where
M=S], OF - T-subprob])
       (simp-all add: space-stream-space del: stake.simps)
  finally show ?case
    by simp
qed

lemma nn-integral-T-split:
  assumes f[measurable]: f ∈ borel-measurable S
  shows (∫⁺ω. f ω ∂T s) = (∫⁺ω. (∫⁺ω'. f (stake n ω @− ω') ∂T ((s ## ω) !!
n)) ∂T s)
  apply (subst T-split[of s n])
  apply (simp add: nn-integral-bind[OF f measurable-distr2[where M=S]])
  apply (subst nn-integral-distr)
  apply (simp-all add: space-stream-space)
  done

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)
  apply (subst T-split[of s n])
  apply (subst emeasure-bind[OF - measurable-distr2[where M=S]])
  apply (simp-all add: )
```

apply (*simp add*: *space-stream-space*)
apply (*subst emeasure-distr*)
apply *simp-all*
apply (*simp-all add*: *space-stream-space*)
done

lemma *prob-T-split*:
assumes *P*[*measurable*]: *Measurable.pred S P*
shows $\mathcal{P}(\omega \text{ in } T\ s.\ P\ \omega) = (\int \omega.\ \mathcal{P}(\omega' \text{ in } T\ ((s\ \#\#\ \omega)\ !!\ n).\ P\ (stake\ n\ \omega\ @-\ \omega'))\ \partial T\ s)$
using *emeasure-T-split*[*OF P*, *of s n*]
unfolding *T.emeasure-eq-measure*
by (*subst* (*asm*) *nn-integral-eq-integral*)
(*auto intro*!: *T.integrable-const-bound*[**where** *B=1*] *measure-measurable-subprob-algebra2*[**where** *N=S*]
simp: *T.emeasure-eq-measure SIGMA-Collect-eq*)

lemma *enabled-imp-alw*:
$(\bigcup s{\in}X.\ set\text{-}pmf\ (K\ s)) \subseteq X \Longrightarrow x \in X \Longrightarrow enabled\ x\ \omega \Longrightarrow alw\ (HLD\ X)\ \omega$
proof (*coinduction arbitrary*: ω *x*)
case *alw* **then show** *?case*
unfolding *enabled.simps*[*of* - ω]
by (*auto simp*: *HLD-iff*)
qed

lemma *alw-HLD-iff-sconst*:
$alw\ (HLD\ \{x\})\ \omega \longleftrightarrow \omega = sconst\ x$
proof
assume $alw\ (HLD\ \{x\})\ \omega$ **then show** $\omega = sconst\ x$
by (*coinduction arbitrary*: ω) (*auto simp*: *HLD-iff*)
qed (*auto simp*: *alw-sconst HLD-iff*)

lemma *enabled-iff-sconst*:
assumes [*simp*]: $set\text{-}pmf\ (K\ x) = \{x\}$ **shows** $enabled\ x\ \omega \longleftrightarrow \omega = sconst\ x$
proof
assume $enabled\ x\ \omega$ **then show** $\omega = sconst\ x$
by (*coinduction arbitrary*: ω) (*auto elim*: *enabled.cases*)
next
assume $\omega = sconst\ x$ **then show** $enabled\ x\ \omega$
by (*coinduction arbitrary*: ω) *auto*
qed

lemma *AE-sconst*:
assumes [*simp*]: $set\text{-}pmf\ (K\ x) = \{x\}$
shows $(AE\ \omega\ in\ T\ x.\ P\ \omega) \longleftrightarrow P\ (sconst\ x)$
proof −
have $(AE\ \omega\ in\ T\ x.\ P\ \omega) \longleftrightarrow (AE\ \omega\ in\ T\ x.\ P\ \omega \wedge \omega = sconst\ x)$
using *AE-T-enabled*[*of x*] **by** (*simp add*: *enabled-iff-sconst*)
also have $\ldots = (AE\ \omega\ in\ T\ x.\ P\ (sconst\ x) \wedge \omega = sconst\ x)$

by (*simp del*: *AE-conj-iff cong*: *rev-conj-cong*)
also have ... = (*AE* ω *in T x*. *P* (*sconst x*))
using *AE-T-enabled*[*of x*] **by** (*simp add*: *enabled-iff-sconst*)
finally show *?thesis*
by *simp*
qed

lemma *ev-eq-lfp*: *ev P* = *lfp* (λF ω. *P* ω $\vee$ ($\neg$ *P* ω $\wedge$ *F* (*stl* ω)))
unfolding *ev-def* **by** (*intro antisym lfp-mono*) *blast+*

lemma *INF-eq-zero-iff-ennreal*: (($\prod$ *i*$\in$*A*. *f i*) = (*0*::*ennreal*)) = ($\forall$ *x*>*0*. $\exists$ *i*$\in$*A*. *f i* < *x*)
using *INF-eq-bot-iff*[**where** *'a*=*ennreal*] **unfolding** *bot-ennreal-def zero-ennreal-def*
by *auto*

lemma *inf-continuous-cmul*:
fixes *c* :: *ennreal*
assumes *f*: *inf-continuous f* **and** *c*: *c* < $\top$
shows *inf-continuous* (λx. *c* * *f x*)
proof (*rule inf-continuous-compose*[*OF - f*], *clarsimp simp add*: *inf-continuous-def*)
fix *M* :: *nat* $\Rightarrow$ *ennreal* **assume** *M*: *decseq M*
show *c* * ($\prod$ *i*. *M i*) = ($\prod$ *i*. *c* * *M i*)
using *M*
by (*intro LIMSEQ-unique*[*OF ennreal-tendsto-cmult*[*OF c*] *LIMSEQ-INF*] *LIMSEQ-INF*)
(*auto simp*: *decseq-def mult-left-mono*)
qed

lemma *AE-T-ev-HLD-infinite*:
fixes *X* :: *'s set* **and** *r* :: *real*
assumes *r* < *1*
assumes *r*: $\bigwedge$*x*. *x* $\in$ *X* $\Longrightarrow$ *measure* (*K x*) *X* $\leq$ *r*
shows *AE* ω *in T x*. *ev* (*HLD* (- *X*)) ω
proof -
{ fix *x* **assume** *x* $\in$ *X*
have *0* $\leq$ *r* **using** *r*[*OF* ‹*x* $\in$ *X*›] *measure-nonneg*[*of K x X*] **by** (*blast intro*: *order.trans*)
define *P* **where** *P F x* = $\int^+$*y*. *indicator X y* * (*F y* $\sqcap$ *1*) ∂*K x* **for** *F x*
have [*measurable*]: *X* $\in$ *sets* (*count-space UNIV*) **by** *auto*
have *bnd*: ($\int^+$ *y*. *indicator X y* * (*f y* $\sqcap$ *1*) ∂*K x*) $\leq$ *1* **for** *x f*
by (*intro measure-pmf.nn-integral-le-const AE-pmfI*) (*auto split*: *split-indicator*)
have *emeasure* (*T x*) {$\omega$$\in$*space* (*T x*). *alw* (*HLD X*) ω} =
emeasure (*T x*) {$\omega$$\in$*space* (*T x*). *gfp* ($\lambda F$ ω. *shd* ω $\in$ *X* $\wedge$ *F* (*stl* ω)) ω}
by (*simp add*: *alw-def HLD-def*)
also have ... = *gfp P x*
apply (*rule emeasure-gfp*)
apply (*auto intro*!: *order-continuous-intros inf-continuous-cmul split*: *split-indicator simp*: *P-def*)
subgoal for *x f* **using** *bnd*[*of x f*] **by** (*auto simp*: *top-unique*)

```
    subgoal for P x
      apply (subst T-eq-bind)
      apply (subst emeasure-bind[where N=S])
      apply simp
      apply (rule measurable-distr2[where M=S])
    apply (auto intro: T-subprob[THEN measurable-space] intro!: nn-integral-cong-AE
AE-pmfI
          simp: emeasure-distr split: split-indicator)
      apply (simp-all add: space-stream-space T.emeasure-le-1 inf.absorb1)
      done
    apply (intro le-funI)
    apply (subst nn-integral-indicator[symmetric])
    apply simp
    apply (intro nn-integral-mono)
    apply (auto split: split-indicator)
    done
  also have ... ≤ (INF n. ennreal r ^ n)
  proof (intro INF-greatest)
    have mono-P: mono P
      by (force simp: le-fun-def mono-def P-def intro!: nn-integral-mono intro:
le-infI1 split: split-indicator)
    fix n show gfp P x ≤ ennreal r ^ n
      using ‹x ∈ X›
    proof (induction n arbitrary: x)
      case 0 then show ?case
      by (subst gfp-unfold[OF mono-P]) (auto intro!: measure-pmf.nn-integral-le-const
AE-pmfI split: split-indicator simp: P-def)
    next
      case (Suc n x)
      have gfp P x = P (gfp P) x by (subst gfp-unfold[OF mono-P]) rule
      also have ... ≤ P (λx. ennreal r ^ n) x
        unfolding P-def[of - x] by (auto intro!: nn-integral-mono le-infI1 Suc
split: split-indicator)
      also have ... ≤ ennreal r ^ (Suc n)
      using Suc by (auto simp: P-def nn-integral-multc measure-pmf.emeasure-eq-measure
intro!: mult-mono ennreal-leI r)
      finally show ?case .
    qed
  qed
  also have ... = 0
    unfolding ennreal-power[OF ‹0 ≤ r›]
  proof (intro LIMSEQ-unique[OF LIMSEQ-INF])
    show decseq (λi. ennreal (r ^ i))
      using ‹0 ≤ r› ‹r < 1› by (auto intro!: ennreal-leI power-decreasing simp:
decseq-def)
    have (λi. ennreal (r ^ i)) ⟶ ennreal 0
      using ‹0 ≤ r› ‹r < 1› by (intro tendsto-ennrealI LIMSEQ-power-zero) auto
    then show (λi. ennreal (r ^ i)) ⟶ 0 by simp
  qed
```

finally have $*$: *emeasure* (*T x*) {ω∈*space* (*T x*). *alw* (*HLD X*) ω} = *0* **by** *auto*
have *AE* ω *in T x. ev* (*HLD* (− *X*)) ω
by (*rule AE-I*[*OF* - $*$]) (*auto simp*: *not-ev-iff not-HLD*[*symmetric*]) **}**
note $* = this$
show *?thesis*
apply (*clarsimp simp add*: *AE-T-iff*[*of* - *x*])
subgoal for x'
by (*cases* $x' \in X$) (*auto simp add*: *ev-Stream* $*$)
done
qed

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
by (*auto simp*: *rT-def space-restrict-space space-stream-space*)

lemma *Collect-enabled-S*[*measurable*]: *Collect* (*enabled x*) ∈ *sets S*
proof −
have *Collect* (*enabled x*) = {ω∈*space S*. *enabled x* ω}
by (*auto simp*: *space-stream-space*)
then show *?thesis*
by *simp*
qed

lemma *space-rT-in-S*: *space* (*rT x*) ∈ *sets S*
by (*simp add*: *rT-def space-restrict-space*)

lemma *sets-rT*: *A* ∈ *sets* (*rT x*) ⟷ *A* ∈ *sets S* ∧ *A* ⊆ {ω. *enabled x* ω}
by (*auto simp*: *rT-def sets-restrict-space space-stream-space*)

lemma *prob-space-rT*: *prob-space* (*rT x*)
unfolding *rT-def* **by** (*auto intro*!: *prob-space-restrict-space T.emeasure-eq-1-AE AE-T-enabled*)

lemma *measurable-force-enabled2*[*measurable*]: *force-enabled x* ∈ *measurable S* (*rT x*)
unfolding *rT-def*
by (*rule measurable-restrict-space2*)
(*auto intro*: *measurable-force-enabled enabled-force-enabled*)

lemma *space-rT-not-empty*[*simp*]: *space* (*rT x*) ≠ {}
by (*simp add*: *rT-def space-restrict-space Ex-enabled*)

lemma *T-eq-bind'*: *T x* = *do* { *y* ← *measure-pmf* (*K x*) ; ω ← *T y* ; *return S* (*y* ## ω) }
apply (*subst T-eq-bind*)
apply (*subst bind-return-distr*[*symmetric*])

```
  apply (simp-all add: space-stream-space comp-def)
  done

lemma rT-eq-bind: rT x = do { y ← measure-pmf (K x) ; ω ← rT y ; return (rT
x) (y ## ω) }
  unfolding rT-def
  apply (subst T-eq-bind)
  apply (subst restrict-space-bind[where K=S])
  apply (rule measurable-distr2[where M=S])
  apply (auto simp del: measurable-pmf-measure1
            simp add: Ex-enabled return-restrict-space intro!: bind-cong )
  apply (subst restrict-space-bind[symmetric, where K=S])
  apply (auto simp add: Ex-enabled space-restrict-space return-cong[OF sets-T]
                    intro!:   measurable-restrict-space1 measurable-compose[OF - re-
turn-measurable]
            arg-cong2[where f=restrict-space])
  apply (subst bind-return-distr[unfolded comp-def])
  apply (simp add: space-restrict-space Ex-enabled)
  apply (simp add: measurable-restrict-space1)
  apply (rule measure-eqI)
  apply simp
  apply (subst (1 2) emeasure-distr)
  apply (auto simp: measurable-restrict-space1)
  apply (subst emeasure-restrict-space)
  apply (auto simp: space-restrict-space intro!: emeasure-eq-AE)
  using AE-T-enabled
  apply eventually-elim
  apply (simp add: space-stream-space)
  apply (rule sets-Int-pred)
  apply auto
  apply (simp add: space-stream-space)
  done

lemma snth-rT: (λx. x !! n) ∈ measurable (rT x) (count-space (acc `` {x}))
proof −
  have ⋀ω. enabled x ω ⟹ (x, ω !! n) ∈ acc
  proof (induction n arbitrary: x)
    case (Suc n) from Suc.prems Suc.IH[of shd ω stl ω] show ?case
      by (auto simp: enabled.simps[of x ω] intro: rtrancl-trans)
  qed (auto elim: enabled.cases)
  moreover
  { fix X :: 's set
    have [measurable]: X ∈ count-space UNIV by simp
    have ∗: (λx. x !! n) −` X ∩ space (rT x) = {ω∈space S. ω !! n ∈ X ∧ enabled
x ω}
      by (auto simp: space-stream-space space-rT)
    have (λx. x !! n) −` X ∩ space (rT x) ∈ sets S
      unfolding ∗ by measurable }
  ultimately show ?thesis
```

by (*auto simp*: *measurable-def space-rT sets-rT*)
qed

3.7 Bisimulation

lemma *T-coinduct*[*consumes 1*, *case-names prob sets cont*]:
assumes *R x M*
assumes *prob*: $\bigwedge x\ M.\ R\ x\ M \Longrightarrow$ *prob-space M*
and *sets*: $\bigwedge x\ M.\ R\ x\ M \Longrightarrow$ *sets M* = *sets S*
and *cont'*: $\bigwedge x\ M.\ R\ x\ M \Longrightarrow \exists\, M'.\ (\forall\, y{\in}K\ x.\ R\ y\ (M'\ y)) \wedge (\forall\, y.\ sets\ (M'\ y) = S \wedge prob\text{-}space\ (M'\ y)) \wedge$
M = (*measure-pmf* (*K x*) ⤜ (λ*y*. *distr* (*M' y*) *S* ((##) *y*)))
shows *T x* = *M*
using ‹*R x M*›
proof (*coinduction arbitrary*: *x M rule*: *measure-eq-stream-space-coinduct*)
case *left* **then show** *?case* **using** *T.prob-space-axioms*[*of x*] *sets-T*[*of x*] **by** (*auto simp*: *space-prob-algebra*)
next
case (*right M*) **with** *prob*[*of M*] *sets*[*of M*] **show** *?case* **by** (*auto simp*: *space-prob-algebra*)
next
case (*cont x M*) **with** *cont'*[*OF cont*] **obtain** *M'* **where** ∗:
(∀ *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*)))
by *auto*
show *?case*
apply (*rule exI*[*of* - *T*])
apply (*rule exI*[*of* - *M'*])
apply (*rule exI*[*of* - *K x*])
using ∗ *T.prob-space-axioms sets-T*[*of x*]
apply (*auto simp*: *space-prob-algebra intro*: *T-eq-bind*)
done
qed

lemma *T-bisim*:
assumes *M*: $\bigwedge x.$ *prob-space* (*M x*) $\bigwedge x.$ *sets* (*M x*) = *sets S*
and *M-eq*: $\bigwedge x.$ *M x* = (*measure-pmf* (*K x*) ⤜ (λ*s*. *distr* (*M s*) *S* ((##) *s*)))
shows *T* = *M*
proof
fix *x* **show** *T x* = *M x*
proof (*coinduction arbitrary*: *x rule*: *T-coinduct*)
case (*cont x*) **then show** *?case*
apply (*intro exI*[*of* - *M*])
apply (*subst M-eq*[*of x*])
apply (*simp add*: *M*)
done
qed *fact*+
qed

lemma *T-subprob'*[*measurable*]: $T \in$ *measurable* (*count-space UNIV*) (*subprob-algebra S*)
by (*auto intro*!: *space-bind simp*: *space-subprob-algebra*) *unfold-locales*

lemma *T-subprob''*[*simp*]: *T a* ∈ *space* (*subprob-algebra S*)
using *measurable-space*[*OF T-subprob'*, *of a*] **by** *simp*

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 −
{ fix *ω* **have** ¬ (*HLD φ suntil HLD ψ*) (*s ##* *ω*) ⟷
(∀ *n*. ¬ ((λ*R*. *HLD ψ or* (*HLD φ aand nxt R*)) ^^ *n*) ⊥ (*s ##* *ω*))
unfolding *suntil-def*
by (*subst sup-continuous-lfp*)
(*auto simp add*: *sup-continuous-def*) **}**
moreover
{ fix *n* **from** ‹*P s*› **have** *AE ω in T s*. ¬ ((λ*R*. *HLD ψ or* (*HLD φ aand nxt R*)) ^^ *n*) ⊥ (*s ##* *ω*)
proof (*induction n arbitrary*: *s*)
case (*Suc n*) **then show** *?case*
apply (*subst AE-T-iff*)
apply (*rule measurable-compose*[*OF measurable-Stream*, **where** *M1*=*count-space UNIV*])
apply *measurable*
apply *simp*
apply (*auto simp*: *bot-fun-def intro*!: *AE-impI dest*: *φ ψ*)
done
qed *simp* **}**
ultimately show *?thesis*
by (*simp add*: *AE-all-countable*)
qed

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 −
have *P s* ∨ *?nuntil s*
using ‹*P s*› **by** *auto*
then show *?thesis*
proof (*coinduction arbitrary*: *s rule*: *AE-not-suntil-coinduct*)
case (*φ t s*) **then show** *?case*
by (*auto simp*: *AE-T-iff*[*of - s*] *suntil-Stream*[*of - - s*] *dest*: *P-φ*)
qed (*auto simp*: *suntil-Stream dest*: *P-ψ*)

qed

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* (*λF 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*)
  by *simp*

lemma *measurable-reward-until*[*measurable* (*raw*)]:
  assumes [*measurable*]: *f* ∈ *measurable M* (*count-space UNIV*)
  assumes [*measurable*]: *g* ∈ *measurable M S*
  shows (*λx*. *reward-until X* (*f x*) (*g x*)) ∈ *borel-measurable M*
proof −
  let *?F* = *λF* (*s*, *ω*). *if s* ∈ *X then 0 else ϱ s* + *ι s* (*shd ω*) + (*F* (*shd ω*, *stl ω*))
  { fix *s ω*
    have *reward-until X s ω* = *lfp ?F* (*s*, *ω*)
      unfolding *reward-until-def lfp-pair*[*symmetric*] **.. }**
  note ∗ = *this*

  have [*measurable*]: *lfp ?F* ∈ *borel-measurable* (*count-space UNIV* ⨂*M* *S*)
  proof (*rule borel-measurable-lfp*)
    fix *f* :: (*'s* × *'s stream*) ⇒ *ennreal*
    assume [*measurable*]: *f* ∈ *borel-measurable* (*count-space UNIV* ⨂*M* *S*)
    show *?F f* ∈ *borel-measurable* (*count-space UNIV* ⨂*M* *S*)
      unfolding *split-beta'*
      apply (*intro measurable-If*)
      apply *measurable* []
      apply *measurable* []
      apply (*rule predE*)
      apply (*rule measurable-compose*[*OF measurable-fst*])
      apply *measurable* []
      done
  qed (*auto intro*!: *ι-nonneg ϱ-nonneg order-continuous-intros*)
  show *?thesis*
    unfolding ∗ **by** *measurable*
qed

lemma *continuous-reward-until*:
sup-continuous ($\lambda F\ s\ \omega$. *if* $s \in X$ *then 0 else* $\varrho\ s + \iota\ s\ (shd\ \omega) + (F\ (shd\ \omega)\ (stl\ \omega))$)
by (*intro* ι*-nonneg* ϱ*-nonneg order-continuous-intros*) (*auto simp*: *sup-continuous-def image-comp*)

lemma
shows *reward-until-unfold*: *reward-until X s* ω =
(*if* $s \in X$ *then 0 else* $\varrho\ s + \iota\ s\ (shd\ \omega) +$ *reward-until X* $(shd\ \omega)\ (stl\ \omega)$)
(**is** *?unfold*)
proof −
let *?F* = $\lambda F\ s\ \omega$. *if* $s \in X$ *then 0 else* $\varrho\ s + \iota\ s\ (shd\ \omega) + (F\ (shd\ \omega)\ (stl\ \omega))$
{ fix $s\ \omega$ **have** *reward-until X s* ω = *?F* (*reward-until X*) $s\ \omega$
unfolding *reward-until-def*
apply (*subst lfp-unfold*)
apply (*rule continuous-reward-until*[*THEN sup-continuous-mono*, *of X*])
apply *rule*
done }
note *step* = *this*
show *?unfold*
by (*subst step*) (*auto intro*!: *arg-cong2*[**where** *f*=(+)])
qed

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* $(shd\ \omega)\ (stl\ \omega)$
unfolding *reward-until-unfold*[*of X s* ω] **by** *simp-all*

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* $t\ \omega$)
by *simp*

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*) ω
shows $(\int^+ \omega.$ *reward-until H s* $\omega\ \partial T\ s) \neq \infty$
proof *cases*
assume $s \in H$ **then show** *?thesis*
by *simp*
next
assume $s \notin H$
let *?L* = *acc-on* $(-H)$
define *M* **where** *M* = *Max* $((\lambda(s, t).\ \varrho\ s + \iota\ s\ t)$ ‘ (*SIGMA t*:*?L*‘‘$\{s\}$. *K t*))
have *?L* $\subseteq$ *?R*
by (*intro rtrancl-mono*) *auto*

```
  with ‹s ∉ H› have subset: (SIGMA t:?L``{s}. K t) ⊆ (?R``{s} × ?R``{s})
    by (auto intro: rtrancl-into-rtrancl elim: rtrancl.cases)
  then have [simp, intro!]: finite ((λ(s, t). ϱ s + ι s t) ` (SIGMA t:?L``{s}. K t))
    by (intro finite-imageI) (auto dest: finite-subset)
  { fix t t' assume (s, t) ∈ ?L t ∉ H t' ∈ K t
    then have (t, t') ∈ (SIGMA t:?L``{s}. K t)
      by (auto intro: rtrancl-into-rtrancl)
    then have ϱ t + ι t t' ≤ M
      unfolding M-def by (intro Max-ge) auto }
  note le-M = this

  have fin-L: finite (?L `` {s})
    by (intro finite-subset[OF - assms(1)] Image-mono ‹?L ⊆ ?R› order-refl)

  have M < ∞
    unfolding M-def
  proof (subst Max-less-iff, safe)
    show (SIGMA x:?L `` {s}. set-pmf (K x)) = {} ⟹ False
      using ‹s ∉ H› by (auto simp add: Sigma-empty-iff set-pmf-not-empty)
    fix t t' assume (s, t) ∈ ?L t' ∈ K t then show ϱ t + ι t t' < ∞
      using ϱ[of t] ι[of t t'] by simp
  qed

  from set-pmf-not-empty[of K s] obtain t where t ∈ K s
    by auto
  with le-M[of s t] have 0 ≤ M
    using set-pmf-not-empty[of K s] ‹s ∉ H› le-M[of s] ι-nonneg[of s] ϱ-nonneg[of
s]
    by (intro order-trans[OF - le-M]) auto

  have AE ω in T s. reward-until H s ω ≤ M * sfirst (HLD H) (s ## ω)
    using ev AE-T-enabled
  proof eventually-elim
    fix ω assume ev (HLD H) ω enabled s ω
    moreover define t where t = s
    ultimately have ev (HLD H) ω enabled t ω t ∈ ?L``{s}
      by auto
    then show reward-until H t ω ≤ M * sfirst (HLD H) (t ## ω)
    proof (induction arbitrary: t rule: ev-induct-strong)
      case (base ω t) then show ?case
        by (auto simp: HLD-iff sfirst-Stream elim: enabled.cases intro: le-M)
    next
      case (step ω t) from step.IH[of shd ω] step.prems step.hyps show ?case
        by (auto simp add: HLD-iff enabled.simps[of t] distrib-left sfirst-Stream
                           reward-until-simps[of t]
                 simp del: reward-until-simps
                 intro!: add-mono le-M intro: rtrancl-into-rtrancl)
    qed
  qed
```

then have $(\int^+\omega.\ \textit{reward-until}\ H\ s\ \omega\ \partial T\ s) \leq (\int^+\omega.\ M * \textit{sfirst}\ (\textit{HLD}\ H)\ (s\ \#\#\ \omega)\ \partial T\ s)$
by (*rule nn-integral-mono-AE*)
also have $\ldots < \infty$
using ‹$0 \leq M$› ‹$M < \infty$› *nn-integral-sfirst-finite*[*OF fin-L ev*]
by (*simp add*: *nn-integral-cmult less-top*[*symmetric*] *ennreal-mult-eq-top-iff*)
finally show *?thesis*
by *simp*
qed

end

3.9 Bisimulation on a relation

definition *rel-set-strong* :: $('a \Rightarrow 'b \Rightarrow \textit{bool}) \Rightarrow 'a\ \textit{set} \Rightarrow 'b\ \textit{set} \Rightarrow \textit{bool}$
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 $R :: 's \Rightarrow 't \Rightarrow \textit{bool}$ **and** $f :: 's \Rightarrow 't$ **and** $S :: 's\ \textit{set}$
assumes *R-def*: $\bigwedge s\ t.\ R\ s\ t \longleftrightarrow (s \in S \wedge f\ s = t)$
assumes *A*[*measurable*]: $A \in \textit{sets}\ (\textit{stream-space}\ (\textit{count-space}\ \textit{UNIV}))$
and *B*[*measurable*]: $B \in \textit{sets}\ (\textit{stream-space}\ (\textit{count-space}\ \textit{UNIV}))$
and *AB*: *rel-set-strong* (*stream-all2 R*) *A B* **and** *KL*: *rel-fun R* (*rel-pmf R*) *K L* **and** *xy*: *R x y*
shows *MC-syntax.T K x A* = *MC-syntax.T L y B*
proof −
interpret *K*: *MC-syntax K* **by** *unfold-locales*
interpret *L*: *MC-syntax L* **by** *unfold-locales*

have $x \in S$ **using** ‹*R x y*› **by** (*auto simp*: *R-def*)

define *g* **where** *g t* = (*SOME s. R s t*) **for** *t*
have *measurable-g*: $g \in \textit{count-space}\ \textit{UNIV} \rightarrow_M \textit{count-space}\ \textit{UNIV}$ **by** *auto*
have *g*: *R i j* $\Longrightarrow$ *R* (*g j*) *j* **for** *i j*
unfolding *g-def* **by** (*rule someI*)

have *K-subset*: $x \in S \Longrightarrow K\ x \subseteq S$ **for** *x*
using *KL*[*THEN rel-funD*, *of x f x*, *THEN rel-pmf-imp-rel-set*] **by** (*auto simp*: *rel-set-def R-def*)

have *in-S*: *AE ω in K.T x. ω ∈ streams S*
using *K.AE-T-enabled*
proof *eventually-elim*
case (*elim ω*) **with** ‹$x \in S$› **show** *?case*
apply (*coinduction arbitrary*: *x ω*)
subgoal for *x ω* **using** *K-subset* **by** (*cases ω*) (*auto simp*: *K.enabled-Stream*)
done
qed

```
  have L-eq: L y = map-pmf f (K x) if xy: R x y for x y
  proof −
    have rel-pmf (λx y. x = y) (map-pmf f (K x)) (L y)
      using KL[THEN rel-funD, OF xy] by (auto intro: pmf.rel-mono-strong simp:
R-def pmf.rel-map)
    then show ?thesis unfolding pmf.rel-eq by simp
  qed

  let ?D = λx. distr (K.T x) K.S (smap f)
  have prob-space-D: ?D x ∈ space (prob-algebra K.S) for x
    by (auto simp: space-prob-algebra K.T.prob-space-distr)

  have D-eq-D: ?D x = ?D x' if R x y R x' y for x x' y
  proof (rule stream-space-eq-sstart)
    define A where A = K.acc `` {x, x'}
    have x-A: x ∈ A x' ∈ A by (auto simp: A-def)
    let ?Ω = f ` A
    show countable ?Ω
      unfolding A-def by (intro countable-image K.countable-acc) auto
   show prob-space (?D x) prob-space (?D x') by (auto intro!: K.T.prob-space-distr)
    show sets (?D x) = sets L.S sets (?D x') = sets L.S by auto
    have AE-streams: AE x in ?D x''. x ∈ streams ?Ω if x'' ∈ A for x''
      apply (simp add: space-stream-space streams-sets AE-distr-iff)
      using K.AE-T-reachable[of x''] unfolding alw-HLD-iff-streams
    proof eventually-elim
      fix s assume s ∈ streams (K.acc `` {x''})
      moreover have K.acc `` {x''} ⊆ A
        using ‹x'' ∈ A› by (auto simp: A-def Image-def intro: rtrancl-trans)
      ultimately show smap f s ∈ streams (f ` A)
        by (auto intro: smap-streams)
    qed
    with x-A show AE x in ?D x'. x ∈ streams ?Ω AE x in ?D x. x ∈ streams ?Ω
      by auto
    from ‹x ∈ A› ‹x' ∈ A› that show ?D x (sstart (f ` A) xs) = ?D x' (sstart (f `
A) xs) for xs
    proof (induction xs arbitrary: x x' y)
      case Nil
      moreover have ?D x (streams (f ` A)) = 1 if x ∈ A for x
        using AE-streams[of x] that
          by (intro prob-space.emeasure-eq-1-AE[OF K.T.prob-space-distr]) (auto
simp: streams-sets)
      ultimately show ?case by simp
    next
      case (Cons z zs x x' y)
      have rel-pmf (R OO R^-1^-1) (K x) (K x')
        using KL[THEN rel-funD, OF Cons(4)] KL[THEN rel-funD, OF Cons(5)]
        unfolding pmf.rel-compp pmf.rel-flip by auto
     then obtain p :: ('s × 's) pmf where p: ⋀a b. (a, b) ∈ p ⟹ (R OO R^-1^-1)
a b and
```

```
      eq: map-pmf fst p = K x map-pmf snd p = K x'
      by (auto simp: pmf.in-rel)
    let ?S = stream-space (count-space UNIV)
    have *: (##) y -‘ smap f -‘ sstart (f ‘ A) (z # zs) = (if f y = z then smap
f -‘ sstart (f ‘ A) zs else {}) for y z zs
      by auto
     have **: ?D x (sstart (f ‘ A) (z # zs)) = (∫+ y'. (if f y' = z then ?D y'
(sstart (f ‘ A) zs) else 0) ∂K x) for x
      apply (simp add: emeasure-distr)
      apply (subst K.T-eq-bind)
      apply (subst emeasure-bind[where N=?S])
        apply simp
       apply (rule measurable-distr2[where M=?S])
        apply measurable
      apply (intro nn-integral-cong-AE AE-pmfI)
      apply (auto simp add: emeasure-distr)
      apply (simp-all add: * space-stream-space)
      done
    have fst-A: fst ab ∈ A if ab ∈ p for ab
    proof −
     have fst ab ∈ K x using ‹ab ∈ p› set-map-pmf [of fst p] by (auto simp: eq)
      with ‹x ∈ A› show fst ab ∈ A
       by (auto simp: A-def intro: rtrancl.rtrancl-into-rtrancl)
    qed
    have snd-A: snd ab ∈ A if ab ∈ p for ab
    proof −
      have snd ab ∈ K x' using ‹ab ∈ p› set-map-pmf [of snd p] by (auto simp:
eq)
      with ‹x' ∈ A› show snd ab ∈ A
       by (auto simp: A-def intro: rtrancl.rtrancl-into-rtrancl)
    qed
    show ?case
      unfolding ** eq[symmetric] nn-integral-map-pmf
      apply (intro nn-integral-cong-AE AE-pmfI)
     subgoal for ab using p[of fst ab snd ab] by (auto simp: R-def intro!: Cons(1)
fst-A snd-A)
      done
  qed
 qed

 have L-eq-D: L.T y = ?D x
  using ‹R x y›
 proof (coinduction arbitrary: x y rule: L.T-coinduct)
  case (cont x y)
  then have Kx-Ly: rel-pmf R (K x) (L y)
   by (rule KL[THEN rel-funD])
  then have *: y' ∈ L y ⟹ ∃x'∈K x. R x' y' for y'
   by (auto dest!: rel-pmf-imp-rel-set simp: rel-set-def)
  have **: y' ∈ L y ⟹ R (g y') y' for y'
```

```
    using ∗[of y'] unfolding g-def by (auto intro: someI)

  have D-SCons-eq-D-D: distr (K.T i) K.S (λx. z ## smap f x) = distr (?D i)
K.S (λx. z ## x) for i z
    by (subst distr-distr) (auto simp: comp-def)
  have D-eq-D-gi: ?D i = ?D (g (f i)) if i: i ∈ K x for i
  proof −
    obtain j where j ∈ L y R i j f i = j
      using Kx-Ly i by (force dest!: rel-pmf-imp-rel-set simp: rel-set-def R-def)
    then show ?thesis
      by (auto intro!: D-eq-D[OF ‹R i j›] g)
  qed

  have ∗∗∗: ?D x = measure-pmf (L y) ⤜ (λy. distr (?D (g y)) K.S ((##) y))
    apply (subst K.T-eq-bind)
    apply (subst distr-bind[of - - K.S])
     apply (rule measurable-distr2[of - - K.S])
      apply (simp-all add: Pi-iff)
    apply (simp add: distr-distr comp-def L-eq[OF cont] map-pmf-rep-eq)
    apply (subst bind-distr[where K=K.S])
     apply measurable []
    apply (rule measurable-distr2[of - - K.S])
    apply measurable []
    apply (rule measurable-compose[OF measurable-g])
    apply measurable []
   apply simp
    apply (rule bind-measure-pmf-cong[where N=K.S])
   apply (auto simp: space-subprob-algebra space-stream-space intro!: K.T.subprob-space-distr)
     unfolding D-SCons-eq-D-D D-eq-D-gi ..
  show ?case
    by (intro exI[of - λt. distr (K.T (g t)) (stream-space (count-space UNIV))
(smap f)])
       (auto simp add: K.T.prob-space-distr ∗∗∗ dest: ∗∗)
 qed (auto intro: K.T.prob-space-distr)

 have stream-all2 R s t ⟷ (s ∈ streams S ∧ smap f s = t) for s t
 proof safe
   show stream-all2 R s t ⟹ s ∈ streams S
    apply (coinduction arbitrary: s t)
    subgoal for s t by (cases s; cases t) (auto simp: R-def)
    done
   show stream-all2 R s t ⟹ smap f s = t
    apply (coinduction arbitrary: s t)
    subgoal for s t by (cases s; cases t) (auto simp: R-def)
    done
 qed (auto intro!: stream.rel-refl-strong simp: stream.rel-map R-def streams-iff-sset)
 then have ω ∈ streams S ⟹ ω ∈ A ⟷ smap f ω ∈ B for ω
   using AB by (auto simp: rel-set-strong-def)
 with in-S have K.T x A = K.T x (smap f −‘ B ∩ space (K.T x))
```

by (*auto intro*!: *emeasure-eq-AE streams-sets*)
also have ... = (*distr* (*K.T x*) *K.S* (*smap f*)) *B*
by (*intro emeasure-distr*[*symmetric*]) *auto*
also have ... = (*L.T y*) *B* **unfolding** *L-eq-D* **..**
finally show *?thesis* **.**
qed

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* **.**

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 (*rule measurable-stream-space2*)
fix *n*
have (λ*x*. (*case x of* (*ω1*, *ω2*) ⇒ *szip ω1 ω2*) !! *n*) = (λ*ω*. (*fst ω* !! *n*, *snd ω* !! *n*))
by *auto*
also have ... ∈ *measurable* (*K1.rT x1* ⨂$_M$ *K2.rT x2*) (*count-space UNIV*)
apply (*rule measurable-compose-countable'*[*OF - measurable-compose*[*OF measurable-fst K1.snth-rT*, *of n*]])
apply (*rule measurable-compose-countable'*[*OF - measurable-compose*[*OF measurable-snd K2.snth-rT*, *of n*]])
apply (*auto intro*!: *K1.countable-acc K2.countable-acc*)
done
finally show (λ*x*. (*case x of* (*ω1*, *ω2*) ⇒ *szip ω1 ω2*) !! *n*) ∈ *measurable* (*K1.rT x1* ⨂$_M$ *K2.rT x2*) (*count-space UNIV*)
.
qed

lemma *measurable-szipE*[*measurable*]: $szip_E$ *a b* ∈ *measurable* (*K1.S* ⨂$_M$ *K2.S*) *S*
unfolding $szip_E$*-def* **by** *measurable*

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 (*rule T-bisim*)
have *T1x*: ⋀*x*. *subprob-space* (*K1.T x*)
by (*rule prob-space-imp-subprob-space*) *unfold-locales*

interpret *T12*: *pair-prob-space K1.T x K2.T y* **for** *x y*
by *unfold-locales*
interpret *T1K2*: *pair-prob-space K1.T x K2 y* **for** *x y*
by *unfold-locales*

let *?P* = λ*x1 x2*. *K1.T x1* $\bigotimes_M$ *K2.T x2*

fix *x* **show** *prob-space* (*?B x*)
by (*auto simp*: *space-stream-space split*: *prod.splits*
intro!: *prob-space.prob-space-bind prob-space-return*
measurable-bind[**where** *N*=*S*] *measurable-compose*[*OF* - *return-measurable*] *AE-I2*)
unfold-locales

show *sets* (*?B x*) = *sets S*
by (*simp split*: *prod.splits add*: *measurable-bind*[**where** *N*=*S*] *sets-bind*[**where** *N*=*S*] *space-stream-space*)

obtain *a b* **where** *x-eq*: *x* = (*a*, *b*)
by (*cases x*) *auto*
show *?B x* = (*measure-pmf* (*Kp x*) $\gg\!=$ (λ*s*. *distr* (*?B s*) *S* ((##) *s*)))
unfolding *x-eq*
apply (*subst K1.T-eq-bind*′)
apply (*subst K2.T-eq-bind*′)
apply (*auto*
simp add: *space-stream-space bind-assoc*[**where** *R*=*S* **and** *N*=*S*] *bind-return-distr*[*symmetric*]
Kp-def T1K2.bind-rotate[**where** *N*=*S*] *split-beta*′ *set-pair-pmf space-subprob-algebra*
bind-pair-pmf[*of case-prod M* **for** *M*, *unfolded split*, *symmetric*, **where** *N*=*S*] *szip*$_E$*-def*
stream-eq-Stream-iff bind-return[**where** *N*=*S*] *space-bind*[**where** *N*=*S*]
simp del: *measurable-pmf-measure1*
intro!: *bind-measure-pmf-cong*[**where** *N*=*S*] *subprob-space-bind*[**where** *N*=*S*] *subprob-space-measure-pmf*
T1x bind-cong[**where** *M*=*MC-syntax.T K x* **for** *K x*] *arg-cong2*[**where** *f*=*return*])
done
qed

lemma *nn-integral-pT*:
fixes *f* **assumes** [*measurable*]: *f* $\in$ *borel-measurable S*
shows ($\int^+\omega$. *f* ω ∂*T* (*x*, *y*)) = ($\int^+\omega 1$. $\int^+\omega 2$. *f* (*szip*$_E$ *x y* ($\omega 1$, $\omega 2$)) ∂*K2.T y* ∂*K1.T x*)
by (*simp add*: *nn-integral-bind*[**where** *B*=*S*] *nn-integral-return in-S T-eq-prod*)

lemma *prod-eq-prob-T*:
assumes [*measurable*]: *Measurable.pred K1.S P1 Measurable.pred K2.S P2*

shows $\mathcal{P}(\omega \ in \ K1.T \ x1. \ P1 \ \omega) * \mathcal{P}(\omega \ in \ K2.T \ x2. \ P2 \ \omega) =$
$\mathcal{P}(\omega \ in \ T \ (x1, \ x2). \ P1 \ (smap \ fst \ \omega) \wedge P2 \ (smap \ snd \ \omega))$
proof −
have $\mathcal{P}(\omega \ in \ T \ (x1, \ x2). \ P1 \ (smap \ fst \ \omega) \wedge P2 \ (smap \ snd \ \omega)) =$
$(\int \ x. \int \ xa. \ indicator \ \{\omega \in space \ S. \ P1 \ (smap \ fst \ \omega) \wedge P2 \ (smap \ snd \ \omega)\}$
$(szip_E \ x1 \ x2 \ (x, \ xa)) \ \partial MC\text{-}syntax.T \ K2 \ x2 \ \partial MC\text{-}syntax.T \ K1 \ x1)$
by (*subst T-eq-prod*)
(*simp add*: *K1.T.measure-bind*[**where** *N=S*] *K2.T.measure-bind*[**where** *N=S*] *measure-return*)
also have ... = $(\int \omega 1. \int \omega 2. \ indicator \ \{\omega \in space \ K1.S. \ P1 \ \omega\} \ \omega 1 * indicator$
$\{\omega \in space \ K2.S. \ P2 \ \omega\} \ \omega 2 \ \partial K2.T \ x2 \ \partial K1.T \ x1)$
apply (*intro integral-cong-AE*)
apply *measurable*
using *K1.AE-T-enabled*
apply *eventually-elim*
apply (*intro integral-cong-AE*)
apply *measurable*
using *K2.AE-T-enabled*
apply *eventually-elim*
apply (*auto simp*: *space-stream-space* $szip_E$*-def K1.force-enabled K2.force-enabled*
smap-szip-snd[**where** $g=\lambda x. \ x$] *smap-szip-fst*[**where** $f=\lambda x. \ x$]
split: *split-indicator*)
done
also have ... = $\mathcal{P}(\omega \ in \ K1.T \ x1. \ P1 \ \omega) * \mathcal{P}(\omega \ in \ K2.T \ x2. \ P2 \ \omega)$
by *simp*
finally show *?thesis* **..**
qed

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*: $\bigwedge n \ s \ s'.$
$\mathcal{P}(\omega \ in \ M. \ \forall \ t \leq n. \ X \ t \ \omega = s \ t \ ) \neq 0 \implies$
$\mathcal{P}(\omega \ in \ M. \ X \ (Suc \ n) \ \omega = s' \mid \forall \ t \leq n. \ X \ t \ \omega = s \ t \ ) =$
$\mathcal{P}(\omega \ in \ M. \ X \ (Suc \ n) \ \omega = s' \mid X \ n \ \omega = s \ n \ )$

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* **.**

lift-definition *I* :: *'s pmf* **is** *distr M* (*count-space UNIV*) (*X 0*)
proof −
let *?X* = *distr M* (*count-space UNIV*) (*X 0*)
interpret *X*: *prob-space ?X*
by (*auto simp*: *prob-space-distr*)
have *AE x in ?X. measure ?X* {*x*} ≠ *0*
using *S* **by** (*subst X.AE-support-countable*) (*auto simp*: *AE-distr-iff intro*!: *exI*[*of - S*])
then show *prob-space ?X* ∧ *sets ?X* = *UNIV* ∧ (*AE x in ?X. measure ?X* {*x*} ≠ *0*)
by (*simp add*: *prob-space-distr AE-support-countable*)
qed

lemma *I-in-S*:
assumes *pmf I s* ≠ *0* **shows** *s* ∈ *S*
proof −
from ‹*pmf I s* ≠ *0*› **have** *0* ≠ $\mathcal{P}$(*x in M. X 0 x* = *s*)
by *transfer* (*auto simp*: *measure-distr vimage-def Int-def conj-commute*)
also have $\mathcal{P}$(*x in M. X 0 x* = *s*) = $\mathcal{P}$(*x in M. X 0 x* = *s* ∧ *s* ∈ *S*)
using *S*(*2*)[*of 0*] **by** (*intro M.finite-measure-eq-AE*) *auto*
finally show *?thesis*
by (*cases s* ∈ *S*) *auto*
qed

lift-definition *K* :: *'s* ⇒ *'s pmf* **is**
λs. with (*λn.* $\mathcal{P}$(*ω in M. X n ω* = *s*) ≠ *0*)
(*λn. distr* (*uniform-measure M* {*ω*∈*space M. X n ω* = *s*}) (*count-space UNIV*) (*X* (*Suc n*)))
(*uniform-measure* (*count-space UNIV*) {*s*})
proof (*rule withI*)
fix *s n* **assume** ∗: $\mathcal{P}$(*ω in M. X n ω* = *s*) ≠ *0*
let *?D* = *distr* (*uniform-measure M* {*ω*∈*space M. X n ω* = *s*}) (*count-space UNIV*) (*X* (*Suc n*))
have *D*: *prob-space ?D*
by (*intro prob-space.prob-space-distr prob-space-uniform-measure*)
(*auto simp*: *M.emeasure-eq-measure* ∗)
then interpret *D*: *prob-space ?D* **.**
have *sets-D*: *sets ?D* = *UNIV*

by *simp*
moreover have *AE x in ?D. measure ?D {x} ≠ 0*
unfolding *D.AE-support-countable[OF sets-D]*
proof (*intro exI[of - S] conjI*)
show *countable S* **by** (*rule S*)
show *AE x in ?D. x ∈ S*
using ∗ *S(2)[of Suc n]* **by** (*auto simp add*: *AE-distr-iff AE-uniform-measure M.emeasure-eq-measure*)
qed
ultimately show *prob-space ?D ∧ sets ?D = UNIV ∧ (AE x in ?D. measure ?D {x} ≠ 0)*
using *D* **by** *blast*
qed (*auto intro*!: *prob-space-uniform-measure AE-uniform-measureI*)

lemma *pmf-K*:
assumes *n*: *0 < 𝒫(ω in M. X n ω = s)*
shows *pmf (K s) t = 𝒫(ω in M. X (Suc n) ω = t | X n ω = s)*
proof (*transfer fixing*: *n s t*)
let *?P = λn. 𝒫(ω in M. X n ω = s) ≠ 0*
let *?D = λn. distr (uniform-measure M {ω∈space M. X n ω = s}) (count-space UNIV) (X (Suc n))*
let *?U = uniform-measure (count-space UNIV) {s}*
show *measure (with ?P ?D ?U) {t} = 𝒫(ω in M. X (Suc n) ω = t | X n ω = s)*
proof (*rule withI*)
fix *n′* **assume** *?P n′*
moreover have *X (Suc n′) −‘ {t} ∩ space M = {x∈space M. X (Suc n′) x = t}*
by *auto*
ultimately show *measure (?D n′) {t} = 𝒫(ω in M. X (Suc n) ω = t | X n ω = s)*
using *n M.measure-uniform-measure-eq-cond-prob[of λx. X (Suc n′) x = t λx. X n′ x = s]*
by (*auto simp*: *measure-distr M.emeasure-eq-measure simp del*: *measure-uniform-measure intro*!: *TH*)
qed (*insert n*, *simp*)
qed

lemma *pmf-K2*:
(⋀n. 𝒫(ω in M. X n ω = s) = 0) ⟹ pmf (K s) t = indicator {t} s
apply (*transfer fixing*: *s t*)
apply (*rule withI*)
apply (*auto split*: *split-indicator*)
done

end

sublocale *K*: *MC-syntax K* **.**

lemma *bind-I-K-eq-M*: *K.T′ I = distr M K.S (λω. to-stream (λn. X n ω)) (***is** -

```
= ?D)
proof (rule stream-space-eq-sstart)
  note streams-sets[measurable]
  note measurable-abs-UNIV[measurable (raw)]
  note sstart-sets[measurable]

  { fix s assume s ∈ S
    from K.AE-T-enabled[of s] have AE ω in K.T s. ω ∈ streams S
    proof eventually-elim
      fix ω assume K.enabled s ω from this ‹s∈S› show ω ∈ streams S
      proof (coinduction arbitrary: s ω)
        case streams
        then have 1: pmf (K s) (shd ω) ≠ 0
          by (simp add: K.enabled.simps[of s] set-pmf-iff)
        have shd ω ∈ S
        proof cases
          assume ∃ n. 0 < P(ω in M. X n ω = s)
          then obtain n where 0 < P(ω in M. X n ω = s) by auto
          with 1 have 2: P(ω' in M. X (Suc n) ω' = shd ω ∧ X n ω' = s) ≠ 0
            by (simp add: pmf-K cond-prob-def)
          show shd ω ∈ S
          proof (rule ccontr)
            assume shd ω ∉ S
            with S(2)[of Suc n] have P(ω' in M. X (Suc n) ω' = shd ω ∧ X n ω'
= s) = 0
              by (intro M.prob-eq-0-AE) auto
            with 2 show False by contradiction
          qed
        next
          assume ¬ (∃ n. 0 < P(ω in M. X n ω = s))
          then have pmf (K s) (shd ω) = indicator {shd ω} s
            by (intro pmf-K2) (auto simp: not-less measure-le-0-iff)
          with 1 ‹s∈S› show ?thesis
            by (auto split: split-indicator-asm)
        qed
        with streams show ?case
          by (cases ω) (auto simp: K.enabled.simps[of s])
      qed
    qed }
  note AE-streams = this

  show prob-space (K.T' I)
    by (rule K.prob-space-T')
  show prob-space ?D
    by (rule M.prob-space-distr) simp

  show AE x in K.T' I. x ∈ streams S
    by (auto simp add: K.AE-T' set-pmf-iff I-in-S AE-distr-iff streams-Stream
intro!: AE-streams)
```

show *AE x in ?D. x ∈ streams S*
by (*simp add*: *AE-distr-iff to-stream-in-streams AE-all-countable S*)
show *sets* (*K.T′ I*) = *sets* (*stream-space* (*count-space UNIV*))
by (*simp add*: *K.sets-T′*)
show *sets ?D* = *sets* (*stream-space* (*count-space UNIV*))
by *simp*

fix *xs′* **assume** *xs′* ≠ [] *xs′* ∈ *lists S*
then obtain *s xs* **where** *xs′*: *xs′* = *s* # *xs* **and** *s*: *s* ∈ *S* **and** *xs*: *xs* ∈ *lists S*
by (*auto simp*: *neq-Nil-conv del*: *in-listsD*)

have *emeasure* (*K.T′ I*) (*sstart S xs′*) = (∫⁺*s*. *emeasure* (*K.T s*) {ω∈*space K.S*. *s* ## ω ∈ *sstart S xs′*} ∂*I*)
by (*rule K.emeasure-T′*) *measurable*
also have ... = (∫⁺*s′*. *emeasure* (*K.T s*) (*sstart S xs*) ∗ *indicator* {*s*} *s′* ∂*I*)
by (*intro arg-cong2*[**where** *f*=*emeasure*] *nn-integral-cong*)
(*auto split*: *split-indicator simp*: *emeasure-distr vimage-def space-stream-space neq-Nil-conv xs′*)
also have ... = *pmf I s* ∗ *emeasure* (*K.T s*) (*sstart S xs*)
by (*auto simp add*: *max-def emeasure-pmf-single intro*: *mult-ac*)
also have *emeasure* (*K.T s*) (*sstart S xs*) = *ennreal* (∏*i*<*length xs*. *pmf* (*K* ((*s*#*xs*)!*i*)) (*xs*!*i*))
using *xs s*
proof (*induction arbitrary*: *s*)
case *Nil* **then show** *?case*
by (*simp add*: *K.T.emeasure-eq-1-AE AE-streams*)
next
case (*Cons t xs*)
have *emeasure* (*K.T s*) (*sstart S* (*t* # *xs*)) =
emeasure (*K.T s*) {*x*∈*space* (*K.T s*). *shd x* = *t* ∧ *stl x* ∈ *sstart S xs*}
by (*intro arg-cong2*[**where** *f*=*emeasure*]) (*auto simp*: *space-stream-space*)
also have ... = (∫⁺*t′*. *emeasure* (*K.T t′*) {*x*∈*space K.S*. *t′* = *t* ∧ *x* ∈ *sstart S xs*} ∂*K s*)
by (*subst K.emeasure-Collect-T*) *auto*
also have ... = (∫⁺*t′*. *emeasure* (*K.T t*) (*sstart S xs*) ∗ *indicator* {*t*} *t′* ∂*K s*)
by (*intro nn-integral-cong*) (*auto split*: *split-indicator simp*: *space-stream-space*)
also have ... = *emeasure* (*K.T t*) (*sstart S xs*) ∗ *pmf* (*K s*) *t*
by (*simp add*: *emeasure-pmf-single max-def*)
finally show *?case*
by (*simp add*: *lessThan-Suc-eq-insert-0 zero-notin-Suc-image prod.reindex Cons*
prod-nonneg ennreal-mult[*symmetric*])
qed
also have *pmf I s* ∗ *ennreal* (∏*i*<*length xs*. *pmf* (*K* ((*s*#*xs*)!*i*)) (*xs*!*i*)) =
𝒫(*x in M*. ∀*i*≤*length xs*. *X i x* = (*s* # *xs*) ! *i*)
using *xs s*
proof (*induction xs rule*: *rev-induct*)
case *Nil*

have *pmf I s = prob* {*x* ∈ *space M*. *X 0 x = s*}
by *transfer* (*simp add*: *vimage-def Int-def measure-distr conj-commute*)
then show *?case*
by *simp*
next
case (*snoc t xs*)
let *?l = length xs* **and** *?lt = length* (*xs* @ [*t*]) **and** *?xs′ = s # xs* @ [*t*]
have *ennreal* (*pmf I s*) ∗ (∏ *i*<*?lt*. *pmf* (*K* ((*?xs′*) ! *i*)) ((*xs* @ [*t*]) ! *i*)) =
(*ennreal* (*pmf I s*) ∗ (∏ *i*<*?l*. *pmf* (*K* ((*s # xs*) ! *i*)) (*xs* ! *i*))) ∗ *pmf* (*K* ((*s # xs*) ! *?l*)) *t*
by (*simp add*: *lessThan-Suc mult-ac nth-append append-Cons*[*symmetric*] *prod-nonneg ennreal-mult*[*symmetric*]
del: *append-Cons*)
also have ... = $\mathcal{P}$(*x in M*. ∀ *i*≤*?l*. *X i x* = (*s # xs*) ! *i*) ∗ *pmf* (*K* ((*s # xs*) ! *?l*)) *t*
using *snoc* **by** (*simp add*: *ennreal-mult*[*symmetric*])
also have ... = $\mathcal{P}$(*x in M*. ∀ *i*≤*?lt*. *X i x* = (*?xs′*) ! *i*)
proof *cases*
assume $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*) = *0*
moreover have $\mathcal{P}$(*x in M*. ∀ *i*≤*?lt*. *X i x* = (*?xs′*) ! *i*) ≤ $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*)
by (*intro M.finite-measure-mono*) (*auto simp*: *nth-append nth-Cons split*: *nat.split*)
moreover have $\mathcal{P}$(*x in M*. ∀ *i*≤*?l*. *X i x* = (*s # xs*) ! *i*) ≤ $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*)
by (*intro M.finite-measure-mono*) (*auto simp*: *nth-append nth-Cons split*: *nat.split*)
ultimately show *?thesis*
by (*simp add*: *measure-le-0-iff*)
next
assume $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*) ≠ *0*
then have ∗: *0* < $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*)
unfolding *less-le* **by** *simp*
moreover have $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*) ≤ $\mathcal{P}$(*ω in M*. *X ?l ω* = (*s # xs*) ! *?l*)
by (*intro M.finite-measure-mono*) (*auto simp*: *nth-append nth-Cons split*: *nat.split*)
ultimately have $\mathcal{P}$(*ω in M*. *X ?l ω* = (*s # xs*) ! *?l*) ≠ *0*
by *auto*
then have *pmf* (*K* ((*s # xs*) ! *?l*)) *t* = $\mathcal{P}$(*ω in M*. *X ?lt ω* = *?xs′* ! *?lt* | *X ?l ω* = (*s # xs*) ! *?l*)
by (*subst pmf-K*) (*auto simp*: *less-le*)
also have ... = $\mathcal{P}$(*ω in M*. *X ?lt ω* = *?xs′* ! *?lt* | ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*)
using ∗ *MC*[*of ?l* λ*i*. (*s # xs*) ! *i ?xs′* ! *?lt*] **by** *simp*
also have ... = $\mathcal{P}$(*ω in M*. ∀ *i*≤*?lt*. *X i ω* = *?xs′* ! *i*) / $\mathcal{P}$(*ω in M*. ∀ *i*≤*?l*. *X i ω* = (*s # xs*) ! *i*)
unfolding *cond-prob-def*
by (*intro arg-cong2*[**where** *f*=(/)] *arg-cong2*[**where** *f*=*measure*]) (*auto simp*:

nth-Cons nth-append split: *nat.splits*)
    finally show *?thesis*
      using ∗ **by** *simp*
   qed
   finally show *?case* **.**
  qed
  also have ... = *emeasure ?D* (*sstart S xs′*)
  proof −
   have *AE x in M*. ∀ *i*. *X i x* ∈ *S*
     using *S*(*2*) **by** (*simp add*: *AE-all-countable*)
   then have *AE x in M*. (∀ *i*≤*length xs*. *X i x* = (*s* # *xs*) ! *i*) = (*to-stream* (λ*n*. *X n x*) ∈ *sstart S xs′*)
   proof *eventually-elim*
     fix *x* **assume** ∀ *i*. *X i x* ∈ *S*
     then have *to-stream* (λ*n*. *X n x*) ∈ *streams S*
       by (*auto simp*: *streams-iff-snth to-stream-def*)
     then show (∀ *i*≤*length xs*. *X i x* = (*s* # *xs*) ! *i*) = (*to-stream* (λ*n*. *X n x*) ∈ *sstart S xs′*)
       by (*simp add*: *sstart-eq xs′ to-stream-def less-Suc-eq-le del*: *sstart.simps*(*1*) *in-sstart*)
   qed
   then show *?thesis*
   by (*auto simp*: *emeasure-distr M.emeasure-eq-measure intro*!: *M.finite-measure-eq-AE*)
  qed
  finally show *emeasure* (*K.T′ I*) (*sstart S xs′*) = *emeasure ?D* (*sstart S xs′*) **.**
qed (*rule S*)

end

lemma (**in** *MC-syntax*) *is-THDTMC*:
  fixes *I* :: *′s pmf*
  defines *U* ≡ (*SIGMA s*:*UNIV*. *K s*)∗ ‘‘ *I*
  shows *Time-Homogeneous-Discrete-Markov-Process* (*T′ I*) *U* (λ*n ω*. *ω* !! *n*)
proof −
  have [*measurable*]: *U* ∈ *sets* (*count-space UNIV*)
   by *auto*

  interpret *prob-space T′ I*
   by (*rule prob-space-T′*)

  { fix *s t I*
   have ⋀*t s*. *P*(*ω in T s*. *s* = *t*) = *indicator* {*t*} *s*
     using *T.prob-space* **by** (*auto split*: *split-indicator*)
   moreover then have ⋀*t t′ s*. *P*(*ω in T s*. *shd ω* = *t′* ∧ *s* = *t*) = *pmf* (*K t*) *t′* ∗ *indicator* {*t*} *s*
     by (*subst prob-T*) (*auto split*: *split-indicator simp*: *pmf.rep-eq*)
   ultimately have *P*(*ω in T′ I*. *shd* (*stl ω*) = *t* ∧ *shd ω* = *s*) = *P*(*ω in T′ I*. *shd ω* = *s*) ∗ *pmf* (*K s*) *t*
     by (*simp add*: *prob-T′ pmf.rep-eq*) **}**

note *start-eq* = *this*

{ fix *n s t* **assume** $\mathcal{P}(\omega\ in\ T'\ I.\ \omega\ !!\ n = s) \neq 0$
moreover have $\mathcal{P}(\omega\ in\ T'\ I.\ \omega\ !!\ (Suc\ n) = t \wedge \omega\ !!\ n = s) = \mathcal{P}(\omega\ in\ T'\ I.\ \omega\ !!\ n = s) * pmf\ (K\ s)\ t$
proof (*induction n arbitrary*: *I*)
case (*Suc n*) **then show** *?case*
by (*subst* (*1 2*) *prob-T'*) (*simp-all del*: *space-T add*: *T-eq-T'*)
qed (*simp add*: *start-eq*)
ultimately have $\mathcal{P}(\omega\ in\ T'\ I.\ stl\ \omega\ !!\ n = t \mid \omega\ !!\ n = s) = pmf\ (K\ s)\ t$
by (*simp add*: *cond-prob-def field-simps*) **}**
note *TH* = *this*

{ fix *n* ω' *t* **assume** $\mathcal{P}(\omega\ in\ T'\ I.\ \forall i{\leq}n.\ \omega\ !!\ i = \omega'\ i) \neq 0$
moreover have $\mathcal{P}(\omega\ in\ T'\ I.\ \omega\ !!\ (Suc\ n) = t \wedge (\forall i{\leq}n.\ \omega\ !!\ i = \omega'\ i)) = \mathcal{P}(\omega\ in\ T'\ I.\ \forall i{\leq}n.\ \omega\ !!\ i = \omega'\ i) * pmf\ (K\ (\omega'\ n))\ t$
proof (*induction n arbitrary*: *I* ω')
case (*Suc n*)
have $*$[*simp*]: $\bigwedge s\ P.\ measure\ (T'\ (K\ s))\ \{x.\ s = \omega'\ 0 \wedge P\ x\} = measure\ (T'\ (K\ (\omega'\ 0)))\ \{x.\ P\ x\} * indicator\ \{\omega'\ 0\}\ s$
by (*auto split*: *split-indicator*)
from *Suc*[*of* - $\lambda i.\ \omega'\ (Suc\ i)$] **show** *?case*
by (*subst* (*1 2*) *prob-T'*)
(*simp-all add*: *T-eq-T' all-Suc-split*[**where** $P{=}\lambda i.\ i \leq Suc\ n \longrightarrow Q\ i$ **for** *n Q*] *conj-commute conj-left-commute sets-eq-imp-space-eq*[*OF sets-T'*])
qed (*simp add*: *start-eq*)
ultimately have $\mathcal{P}(\omega\ in\ T'\ I.\ stl\ \omega\ !!\ n = t \mid \forall i{\leq}n.\ \omega\ !!\ i = \omega'\ i) = pmf\ (K\ (\omega'\ n))\ t$
by (*simp add*: *cond-prob-def field-simps*) **}**
note *MC* = *this*

{ fix *n* ω' **assume** $\mathcal{P}(\omega\ in\ T'\ I.\ \forall t{\leq}n.\ \omega\ !!\ t = \omega'\ t) \neq 0$
moreover have $\mathcal{P}(\omega\ in\ T'\ I.\ \forall t{\leq}n.\ \omega\ !!\ t = \omega'\ t) \leq \mathcal{P}(\omega\ in\ T'\ I.\ \omega\ !!\ n = \omega'\ n)$
by (*auto intro*!: *finite-measure-mono-AE simp*: *sets-T' sets-eq-imp-space-eq*[*OF sets-T'*])
ultimately have $\mathcal{P}(\omega\ in\ T'\ I.\ \omega\ !!\ n = \omega'\ n) \neq 0$
by (*auto simp*: *neq-iff not-less measure-le-0-iff*) **}**
note *MC'* = *this*

show *?thesis*
proof
show *countable U*
unfolding *U-def* **by** (*rule countable-reachable countable-Image countable-set-pmf*)+
show $\bigwedge t.\ (\lambda\omega.\ \omega\ !!\ t) \in measurable\ (T'\ I)\ (count\text{-}space\ UNIV)$
by (*subst measurable-cong-sets*[*OF sets-T' refl*]) *simp*
next
fix *n*
have $\forall x{\in}I.\ AE\ y\ in\ T\ x.\ (x\ \#\#\ y)\ !!\ n \in U$

```
    unfolding U-def
  proof (induction n arbitrary: I)
    case 0 then show ?case
      by auto
  next
    case (Suc n)
    { fix x assume x ∈ I
      have AE y in T x. y !! n ∈ (SIGMA x:UNIV. K x)* `` K x
        apply (subst AE-T-iff)
        apply (rule measurable-compose[OF measurable-snth], simp)
        apply (rule Suc)
        done
      moreover have (SIGMA x:UNIV. K x)* `` K x ⊆ (SIGMA x:UNIV. K x)*
`` I
        using ‹x ∈ I› by (auto intro: converse-rtrancl-into-rtrancl)
      ultimately have AE y in T x. y !! n ∈ (SIGMA x:UNIV. K x)* `` I
        by (auto simp: subset-eq) }
    then show ?case
      by simp
  qed
  then show AE x in T' I. x !! n ∈ U
    by (simp add: AE-T')
 qed (simp-all add: TH MC MC')
qed

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: ⋀s t. s ∈ S ⟹ t ∈ S ⟹ s + t ∈ S
 assumes s: s ∈ S s > 0
 shows eventually (λm. m * Gcd S ∈ S) sequentially
proof −
 define T where T = insert 0 (int ` S)
 with s S have int s ∈ T 0 ∈ T and T: r ∈ T ⟹ t ∈ T ⟹ r + t ∈ T for r t
   by (auto simp del: of-nat-add simp add: of-nat-add [symmetric])
 have Gcd T ∈ group-closure T
   by (rule Gcd-in-group-closure)
 also have group-closure T = {s − t | s t. s ∈ T ∧ t ∈ T}
 proof (auto intro: group-closure.base group-closure.diff)
   fix x assume x ∈ group-closure T
```

```
  then show ∃ s t. x = s − t ∧ s ∈ T ∧ t ∈ T
  proof induction
    case (base x) with ‹0 ∈ T› show ?case
      apply (rule-tac x=x in exI)
      apply (rule-tac x=0 in exI)
      apply auto
      done
  next
    case (diff x y)
    then obtain a b c d where
      a ∈ T b ∈ T x = a − b
      c ∈ T d ∈ T y = c − d
      by auto
    then show ?case
      apply (rule-tac x=a + d in exI)
      apply (rule-tac x=b + c in exI)
      apply (auto intro: T)
      done
  qed
qed
finally obtain s' t' :: int
  where s' ∈ T t' ∈ T Gcd T = s' − t'
  by blast
moreover define s and t where s = nat s' and t = nat t'
moreover have int (Gcd S) = − int t ⟷ S ⊆ {0} ∧ t = 0
  by auto (metis Gcd-dvd-nat dvd-0-right dvd-antisym nat-int nat-zminus-int)
ultimately have
  st: s = 0 ∨ s ∈ S t = 0 ∨ t ∈ S and Gcd-S: Gcd S = s − t
  using T-def by safe simp-all
with s
have t < s
  by (rule-tac ccontr) auto

{ fix s n have 0 < n ⟹ s ∈ S ⟹ n * s ∈ S
  proof (induct n)
    case (Suc n) then show ?case
      by (cases n) (auto intro: S)
  qed simp }
note cmult-S = this

show ?thesis
  unfolding eventually-sequentially
proof cases
  assume s = 0 ∨ t = 0
  with st Gcd-S s have *: Gcd S ∈ S
    by (auto simp: int-eq-iff)
  then show ∃ N. ∀ n≥N. n * Gcd S ∈ S by (auto intro!: exI[of - 1] cmult-S)
next
  assume ¬ (s = 0 ∨ t = 0)
```

with *st* **have** $s \in S\ t \in S\ t \neq 0$ **by** *auto*
then have *Gcd S dvd t* **by** *auto*
then obtain *a* **where** *a*: $t = Gcd\ S * a$ **..**
with ‹$t \neq 0$› **have** $0 < a$ **by** *auto*

show $\exists N.\ \forall n \geq N.\ n * Gcd\ S \in S$
proof (*safe intro*!: *exI*[*of* - $a * a$])
fix *n*
define *m* **where** $m = (n - a * a)\ div\ a$
define *r* **where** $r = (n - a * a)\ mod\ a$
with ‹$0 < a$›**have** $r < a$ **by** *simp*
moreover define *am* **where** $am = a + m$
ultimately have $r < am$ **by** *simp*
assume $a * a \leq n$ **then have** *n*: $n = a * a + (m * a + r)$
unfolding *m-def r-def* **by** *simp*
have $n * Gcd\ S = am * t + r * Gcd\ S$
unfolding *n a* **by** (*simp add*: *field-simps am-def*)
also have $\ldots = r * s + (am - r) * t$
unfolding ‹$Gcd\ S = s - t$›
using ‹$t < s$› ‹$r < am$› **by** (*simp add*: *field-simps diff-mult-distrib2*)
also have $\ldots \in S$
using ‹$s \in S$› ‹$t \in S$› ‹$r < am$›
by (*cases* $r = 0$) (*auto intro*!: *cmult-S S*)
finally show $n * Gcd\ S \in S$ **.**
qed
qed
qed

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*: $G\ s\ t = (\int^+ \omega.\ emeasure\ (count\text{-}space\ UNIV)\ \{i.\ (s\ \#\#\ \omega)\ !!\ i = t\}\ \partial T\ s)$
by (*simp add*: *G-def scount-eq-emeasure HLD-iff*)

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*]: $0 \leq p\ x\ y\ n$
by (*simp add*: *p-def*)

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

by (*simp add*: *p-def*)

lemma *p-x-x-0*[*simp*]: *p x x 0* = *1*
by (*simp add*: *p-def T.prob-space del*: *space-T*)

lemma *p-0*: *p x y 0* = (*if x* = *y then 1 else 0*)
by (*simp add*: *p-def T.prob-space del*: *space-T*)

lemma *p-in-reachable*: **assumes** (*x*, *y*) ∉ (*SIGMA x*:*UNIV*. *K x*)* **shows** *p x y n* = *0*
unfolding *p-def*
proof (*rule T.prob-eq-0-AE*)
from *AE-T-reachable* **show** *AE ω in T x*. (*x* ## *ω*) !! *n* ≠ *y*
proof *eventually-elim*
fix *ω* **assume** *alw* (*HLD* ((*SIGMA ω*:*UNIV*. *K ω*)* `` {*x*})) *ω*
then have *alw* (*HLD* (− {*y*})) *ω*
using *assms* **by** (*auto intro*: *alw-mono simp*: *HLD-iff*)
then show (*x* ## *ω*) !! *n* ≠ *y*
using *assms* **by** (*cases n*) (*auto simp*: *alw-HLD-iff-streams streams-iff-snth*)
qed
qed

lemma *p-Suc*: *ennreal* (*p x y* (*Suc n*)) = (∫⁺ *w*. *p w y n* ∂*K x*)
unfolding *p-def T.emeasure-eq-measure*[*symmetric*] **by** (*subst emeasure-Collect-T*) *simp-all*

lemma *p-Suc'*:
p x y (*Suc n*) = (∫ *x'*. *p x' y n* ∂*K x*)
using *p-Suc*[*of x y n*]
by (*subst* (*asm*) *nn-integral-eq-integral*)
(*auto simp*: *p-le-1 intro*!: *measure-pmf.integrable-const-bound*[**where** *B*=*1*])

lemma *p-add*: *p x y* (*n* + *m*) = (∫⁺ *w*. *p x w n* ∗ *p w y m* ∂*count-space UNIV*)
proof (*induction n arbitrary*: *x*)
case *0*
have [*simp*]: ⋀*w*. (*if x* = *w then 1 else 0*) ∗ *p w y m* = *ennreal* (*p x y m*) ∗ *indicator* {*x*} *w*
by *auto*
show *?case*
by (*simp add*: *p-0 one-ennreal-def*[*symmetric*] *max-def*)
next
case (*Suc n*)
define *X* **where** *X* = (*SIGMA x*:*UNIV*. *K x*)* `` *K x*
then have *X*: *countable X*
by (*blast intro*: *countable-Image countable-reachable countable-set-pmf*)

then interpret *X*: *sigma-finite-measure count-space X*
by (*rule sigma-finite-measure-count-space-countable*)
interpret *XK*: *pair-sigma-finite K x count-space X*

by *unfold-locales*

have *ennreal* $(p\ x\ y\ (Suc\ n + m)) = (\int^+ t.\ (\int^+ w.\ p\ t\ w\ n * p\ w\ y\ m\ \partial count\text{-}space\ UNIV)\ \partial K\ x)$
by (*simp add*: *p-Suc Suc*)
also have $\ldots = (\int^+ t.\ (\int^+ w.\ ennreal\ (p\ t\ w\ n * p\ w\ y\ m) * indicator\ X\ w\ \partial count\text{-}space\ UNIV)\ \partial K\ x)$
by (*auto intro*!: *nn-integral-cong-AE simp*: *AE-measure-pmf-iff AE-count-space Image-iff p-in-reachable X-def split*: *split-indicator*)
also have $\ldots = (\int^+ t.\ (\int^+ w.\ p\ t\ w\ n * p\ w\ y\ m\ \partial count\text{-}space\ X)\ \partial K\ x)$
by (*subst nn-integral-restrict-space*[*symmetric*]) (*simp-all add*: *restrict-count-space*)
also have $\ldots = (\int^+ w.\ (\int^+ t.\ p\ t\ w\ n * p\ w\ y\ m\ \partial K\ x)\ \partial count\text{-}space\ X)$
apply (*rule XK.Fubini'*[*symmetric*])
unfolding *measurable-split-conv*
apply (*rule measurable-compose-countable'*[*OF - measurable-snd X*])
apply (*rule measurable-compose*[*OF measurable-fst*])
apply *simp*
done
also have $\ldots = (\int^+ w.\ (\int^+ t.\ ennreal\ (p\ t\ w\ n * p\ w\ y\ m) * indicator\ X\ w\ \partial K\ x)\ \partial count\text{-}space\ UNIV)$
by (*simp add*: *nn-integral-restrict-space*[*symmetric*] *restrict-count-space nn-integral-multc*)
also have $\ldots = (\int^+ w.\ (\int^+ t.\ ennreal\ (p\ t\ w\ n * p\ w\ y\ m)\ \partial K\ x)\ \partial count\text{-}space\ UNIV)$
by (*auto intro*!: *nn-integral-cong-AE simp*: *AE-measure-pmf-iff AE-count-space Image-iff p-in-reachable X-def split*: *split-indicator*)
also have $\ldots = (\int^+ w.\ (\int^+ t.\ p\ t\ w\ n\ \partial K\ x) * p\ w\ y\ m\ \partial count\text{-}space\ UNIV)$
by (*simp add*: *nn-integral-multc*[*symmetric*] *ennreal-mult*)
finally show *?case*
by (*simp add*: *ennreal-mult p-Suc*)
qed

lemma *prob-reachable-le*:
assumes [*simp*]: $m \le n$
shows $p\ x\ y\ m * p\ y\ w\ (n - m) \le p\ x\ w\ n$
proof −
have $p\ x\ y\ m * p\ y\ w\ (n - m) = (\int^+ y'.\ ennreal\ (p\ x\ y\ m * p\ y\ w\ (n - m)) * indicator\ \{y\}\ y'\ \partial count\text{-}space\ UNIV)$
by *simp*
also have $\ldots \le p\ x\ w\ (m + (n - m))$
by (*subst p-add*)
(*auto intro*!: *nn-integral-mono split*: *split-indicator simp del*: *nn-integral-indicator-singleton*)
finally show *?thesis*
by *simp*
qed

lemma *G-eq-suminf*: $G\ x\ y = (\sum i.\ ennreal\ (p\ x\ y\ i))$
proof −
have $*$: $\bigwedge i\ \omega.\ indicator\ \{\omega \in space\ S.\ (x\ \#\#\ \omega)\ !!\ i = y\}\ \omega = indicator\ \{i.\ (x\ \#\#\ \omega)\ !!\ i = y\}\ i$

by (*auto simp*: *space-stream-space split*: *split-indicator*)

have $G\ x\ y = (\int^+ \omega.\ (\sum i.\ indicator\ \{\omega \in space\ (T\ x).\ (x\ \#\#\ \omega)\ !!\ i = y\}\ \omega)\ \partial T\ x)$
unfolding *G-eq* **by** (*simp add*: *nn-integral-count-space-nat*[*symmetric*] $*$)
also have $\ldots = (\sum i.\ ennreal\ (p\ x\ y\ i))$
by (*simp add*: *T.emeasure-eq-measure*[*symmetric*] *p-def nn-integral-suminf*)
finally show *?thesis* **.**
qed

lemma *G-eq-real-suminf*:
convergence-G x y (*1*::*real*) $\Longrightarrow$ $G\ x\ y = ennreal\ (\sum i.\ p\ x\ y\ i)$
unfolding *G-eq-suminf*
by (*intro suminf-ennreal ennreal-suminf-neq-top p-nonneg*)
(*auto simp*: *convergence-G-def p-def*)

lemma *convergence-norm-G*:
convergence-G x y z $\Longrightarrow$ *summable* $(\lambda n.\ p\ x\ y\ n * norm\ z\ \widehat{}\ n)$
unfolding *convergence-G-def* **.**

lemma *convergence-G*:
convergence-G x y (*z*::*'a*::{*banach*, *real-normed-div-algebra*}) $\Longrightarrow$ *summable* $(\lambda n.\ p\ x\ y\ n *_R z\ \widehat{}\ n)$
unfolding *convergence-G-def*
by (*rule summable-norm-cancel*) (*simp add*: *abs-mult norm-power*)

lemma *convergence-G-less-1*:
fixes *z* :: - :: {*banach*, *real-normed-field*}
assumes *z*: *norm z* $<$ *1* **shows** *convergence-G x y z*
unfolding *convergence-G-def*
proof (*rule summable-comparison-test*)
have $\bigwedge n.\ p\ x\ y\ n * norm\ (z\ \widehat{}\ n) \leq 1 * norm\ (z\ \widehat{}\ n)$
by (*intro mult-right-mono p-le-1*) *simp-all*
then show $\exists N.\ \forall n \geq N.\ norm\ (p\ x\ y\ n * norm\ z\ \widehat{}\ n) \leq norm\ z\ \widehat{}\ n$
by (*simp add*: *norm-power*)
qed (*simp add*: *z summable-geometric*)

lemma *lim-gf-G*: $((\lambda z.\ ennreal\ (gf\text{-}G\ x\ y\ z)) \longrightarrow G\ x\ y)\ (at\text{-}left\ (1::real))$
unfolding *gf-G-def G-eq-suminf real-scaleR-def*
by (*intro power-series-tendsto-at-left p-nonneg p-le-1 summable-power-series*)

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

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

lemma *f-Suc*: $x \neq y \Longrightarrow f\ x\ y\ (Suc\ n) = u\ x\ y\ n$
by (*simp add*: *u-def f-def*)

lemma *f-Suc-eq*: $f\ x\ x\ (Suc\ n) = 0$
by (*simp add*: *f-def*)

lemma *f-0*: $f\ x\ y\ 0 = (if\ x = y\ then\ 1\ else\ 0)$
using *T.prob-space* **by** (*simp add*: *f-def*)

lemma shows *u-nonneg*: $0 \leq u\ x\ y\ n$ **and** *u-le-1*: $u\ x\ y\ n \leq 1$
by (*simp-all add*: *u-def*)

lemma shows *f-nonneg*: $0 \leq f\ x\ y\ n$ **and** *f-le-1*: $f\ x\ y\ n \leq 1$
by (*simp-all add*: *f-def*)

lemma *U-nonneg*[*simp*]: $0 \leq U\ x\ y$
by (*simp add*: *U-def*)

lemma *U-le-1*: $U\ s\ t \leq 1$
by (*auto simp add*: *U-def intro*!: *antisym*)

lemma *U-cases*: $U\ s\ s = 1 \vee U\ s\ s < 1$
by (*auto simp add*: *U-def intro*!: *antisym*)

lemma *u-sums-U*: $u\ x\ y\ sums\ U\ x\ y$
unfolding *u-def*[*abs-def*] *U-def ev-iff-ev-at* **by** (*intro T.prob-sums*) (*auto intro*: *ev-at-unique*)

lemma *gf-U-eq-U*: $gf\text{-}U\ x\ y\ 1 = U\ x\ y$
using *u-sums-U*[*THEN sums-unique*] **by** (*simp add*: *gf-U-def U-def*)

lemma *f-sums-F*: $f\ x\ y\ sums\ F\ x\ y$
unfolding *f-def*[*abs-def*] *F-def ev-iff-ev-at*
by (*intro T.prob-sums*) (*auto intro*: *ev-at-unique*)

lemma *F-nonneg*[*simp*]: $0 \leq F\ x\ y$
by (*auto simp*: *F-def*)

lemma *F-le-1*: $F\ x\ y \leq 1$
by (*simp add*: *F-def*)

lemma *gf-F-eq-F*: $gf\text{-}F\ x\ y\ 1 = F\ x\ y$
using *f-sums-F*[*THEN sums-unique*] **by** (*simp add*: *gf-F-def F-def*)

lemma *gf-F-le-1*:
fixes *z* :: *real*
assumes *z*: *0* ≤ *z z* ≤ *1*
shows *gf-F x y z* ≤ *1*
proof −
have *gf-F x y z* ≤ *gf-F x y 1*
using *z* **unfolding** *gf-F-def*
by (*intro suminf-le*[*OF - summable-comparison-test*[*OF - sums-summable*[*OF f-sums-F*[*of x y*]]]] *mult-left-mono allI f-nonneg*)
(*simp-all add*: *power-le-one f-nonneg mult-right-le-one-le f-le-1 sums-summable*[*OF f-sums-F*[*of x y*]])
also have ... ≤ *1*
by (*simp add*: *gf-F-eq-F F-def*)
finally show *?thesis* **.**
qed

lemma *u-le-p*: *u x y n* ≤ *p x y* (*Suc n*)
unfolding *u-def p-def* **by** (*auto intro*!: *T.finite-measure-mono dest*: *ev-at-HLD-imp-snth*)

lemma *f-le-p*: *f x y n* ≤ *p x y n*
unfolding *f-def p-def* **by** (*auto intro*!: *T.finite-measure-mono dest*: *ev-at-HLD-imp-snth*)

lemma *convergence-norm-U*:
fixes *z* :: - :: *real-normed-div-algebra*
assumes *z*: *convergence-G x y z*
shows *summable* (λ*n*. *u x y n* ∗ *norm z* ^ *Suc n*)
using *summable-ignore-initial-segment*[*OF convergence-norm-G*[*OF z*], *of 1*]
by (*rule summable-comparison-test*[*rotated*])
(*auto simp add*: *u-nonneg abs-mult intro*!: *exI*[*of - 0*] *mult-right-mono u-le-p*)

lemma *convergence-norm-F*:
fixes *z* :: - :: *real-normed-div-algebra*
assumes *z*: *convergence-G x y z*
shows *summable* (λ*n*. *f x y n* ∗ *norm z* ^ *n*)
using *convergence-norm-G*[*OF z*]
by (*rule summable-comparison-test*[*rotated*])
(*auto simp add*: *f-nonneg abs-mult intro*!: *exI*[*of - 0*] *mult-right-mono f-le-p*)

lemma *gf-G-nonneg*:
fixes *z* :: *real*
shows *0* ≤ *z* ⟹ *z* < *1* ⟹ *0* ≤ *gf-G x y z*
unfolding *gf-G-def*
by (*intro suminf-nonneg convergence-G convergence-G-less-1*) *simp-all*

lemma *gf-F-nonneg*:
fixes *z* :: *real*
shows *0* ≤ *z* ⟹ *z* < *1* ⟹ *0* ≤ *gf-F x y z*
unfolding *gf-F-def*

using *convergence-norm-F*[*OF convergence-G-less-1*, *of z x y*]
by (*intro suminf-nonneg*) (*simp-all add*: *f-nonneg*)

lemma *convergence-U*:
fixes *z* :: - :: *banach*
shows *convergence-G x y z* ⟹ *summable* (λ*n*. *u x y n* ∗ *z* ^ *Suc n*)
by (*rule summable-norm-cancel*)
(*auto simp add*: *abs-mult u-nonneg power-abs dest*!: *convergence-norm-U*)

lemma *p-eq-sum-p-u*: *p x y* (*Suc n*) = (∑ *i*≤*n*. *p y y* (*n* − *i*) ∗ *u x y i*)
proof −
have ⋀ω. ω !! *n* = *y* ⟹ (∃ *i*. *i* ≤ *n* ∧ *ev-at* (*HLD* {*y*}) *i* ω)
proof (*induction n*)
case (*Suc n*)
then obtain *i* **where** *i* ≤ *n ev-at* (*HLD* {*y*}) *i* (*stl* ω)
by *auto*
then show *?case*
by (*auto intro*!: *exI*[*of - if HLD* {*y*} ω *then 0 else Suc i*])
qed (*simp add*: *HLD-iff*)
then have *p x y* (*Suc n*) = (∑ *i*≤*n*. *P*(ω *in T x*. *ev-at* (*HLD* {*y*}) *i* ω ∧ ω !! *n* = *y*))
unfolding *p-def* **by** (*intro T.prob-sum*) (*auto intro*: *ev-at-unique*)
also have ... = (∑ *i*≤*n*. *p y y* (*n* − *i*) ∗ *u x y i*)
proof (*intro sum.cong refl*)
fix *i* **assume** *i*: *i* ∈ {.. *n*}
then have ⋀ω. (*Suc i* ≤ *n* ⟶ ω !! (*n* − *Suc i*) = *y*) ⟷ ((*y* ## ω) !! (*n* − *i*) = *y*)
by (*auto simp*: *Stream-snth diff-Suc split*: *nat.split*)
from *i* **have** *i* ≤ *n* **by** *auto*
then have *P*(ω *in T x*. *ev-at* (*HLD* {*y*}) *i* ω ∧ ω !! *n* = *y*) =
(∫ ω′. *P*(ω *in T y*. (*y* ## ω) !! (*n* − *i*) = *y*) ∗
indicator {ω′∈*space* (*T x*). *ev-at* (*HLD* {*y*}) *i* ω′ } ω′ ∂*T x*)
by (*subst prob-T-split*[**where** *n*=*Suc i*])
(*auto simp*: *ev-at-shift ev-at-HLD-single-imp-snth shift-snth diff-Suc*
split: *split-indicator nat.split intro*!: *Bochner-Integration.integral-cong arg-cong2*[**where** *f*=*measure*]
simp del: *stake.simps integral-mult-right-zero*)
then show *P*(ω *in T x*. *ev-at* (*HLD* {*y*}) *i* ω ∧ ω !! *n* = *y*) = *p y y* (*n* − *i*) ∗ *u x y i*
by (*simp add*: *p-def u-def*)
qed
finally show *?thesis* **.**
qed

lemma *p-eq-sum-p-f*: *p x y n* = (∑ *i*≤*n*. *p y y* (*n* − *i*) ∗ *f x y i*)
by (*cases n*)
(*simp-all del*: *sum.atMost-Suc*
add: *f-0 p-0 p-eq-sum-p-u atMost-Suc-eq-insert-0 zero-notin-Suc-image sum.reindex*

f-Suc f-Suc-eq)

lemma *gf-G-eq-gf-F*:
assumes *z*: *norm z < 1*
shows *gf-G x y z = gf-F x y z * gf-G y y z*
proof −
have *gf-G x y z = (∑ n. ∑ i≤n. p y y (n − i) * f x y i * z^n)*
by (*simp add*: *gf-G-def p-eq-sum-p-f*[*of x y*] *sum-distrib-right*)
also have ... = (∑ *n*. ∑ *i*≤*n*. (*f x y i * z^i*) * (*p y y* (*n* − *i*) * *z*^(*n* − *i*)))
by (*intro arg-cong*[**where** *f*=*suminf*] *sum.cong ext atLeast0AtMost*[*symmetric*])
(*simp-all add*: *power-add*[*symmetric*])
also have ... = (∑ *n*. *f x y n * z^n*) * (∑ *n*. *p y y n * z^n*)
using *convergence-norm-F*[*OF convergence-G-less-1*[*OF z*]] *convergence-norm-G*[*OF convergence-G-less-1*[*OF z*]]
by (*intro Cauchy-product*[*symmetric*]) (*auto simp*: *f-nonneg abs-mult power-abs*)
also have ... = *gf-F x y z * gf-G y y z*
by (*simp add*: *gf-F-def gf-G-def*)
finally show *?thesis* **.**
qed

lemma *gf-G-eq-gf-U*:
fixes *z* :: *'z* :: {*banach*, *real-normed-field*}
assumes *z*: *convergence-G x x z*
shows *gf-G x x z = 1 / (1 − gf-U x x z) gf-U x x z ≠ 1*
proof −
{ fix *n*
have *p x x* (*Suc n*) *∗R z^Suc n* = (∑ *i*≤*n*. (*p x x* (*n* − *i*) * *u x x i*) *∗R z^Suc n*)
unfolding *scaleR-sum-left*[*symmetric*] **by** (*simp add*: *p-eq-sum-p-u*)
also have ... = (∑ *i*≤*n*. (*u x x i ∗R z^Suc i*) * (*p x x* (*n* − *i*) *∗R z*^(*n* − *i*)))
by (*intro sum.cong refl*) (*simp add*: *field-simps power-diff cong*: *disj-cong*)
finally have *p x x* (*Suc n*) *∗R z*^(*Suc n*) = (∑ *i*≤*n*. (*u x x i ∗R z^Suc i*) * (*p x x* (*n* − *i*) *∗R z*^(*n* − *i*)))
unfolding *atLeast0AtMost* **. }**
note *gfs-Suc-eq* = *this*

have *gf-G x x z = 1* + (∑ *n*. *p x x* (*Suc n*) *∗R z*^(*Suc n*))
unfolding *gf-G-def*
by (*subst suminf-split-initial-segment*[*OF convergence-G*[*OF z*], *of 1*]) *simp*
also have ... = *1* + (∑ *n*. ∑ *i*≤*n*. (*u x x i ∗R z^Suc i*) * (*p x x* (*n* − *i*) *∗R z*^(*n* − *i*)))
unfolding *gfs-Suc-eq* **..**
also have ... = *1* + *gf-U x x z * gf-G x x z*
unfolding *gf-U-def gf-G-def*
by (*subst Cauchy-product*)
(*auto simp*: *u-nonneg norm-power simp del*: *power-Suc*
intro!: *z convergence-norm-G convergence-norm-U*)
finally show *gf-G x x z = 1 / (1 − gf-U x x z) gf-U x x z ≠ 1*
apply −

```
    apply (cases gf-U x x z = 1)
    apply (auto simp add: field-simps)
    done
qed

lemma gf-U: (gf-U x y ⟶ U x y) (at-left 1)
proof −
  have ((λz. ennreal (∑ n. u x y n * z ^ n)) ⟶ (∑ n. ennreal (u x y n))) (at-left
1)
    using u-le-1 u-nonneg by (intro power-series-tendsto-at-left summable-power-series)
  also have (∑ n. ennreal (u x y n)) = ennreal (suminf (u x y))
    by (intro u-nonneg suminf-ennreal ennreal-suminf-neq-top sums-summable[OF
u-sums-U])
  also have suminf (u x y) = U x y
    using u-sums-U by (rule sums-unique[symmetric])
  finally have ((λz. ∑ n. u x y n * z ^ n) ⟶ U x y) (at-left 1)
    by (rule tendsto-ennrealD)
       (auto simp: u-nonneg u-le-1 intro!: suminf-nonneg summable-power-series
eventually-at-left-1)
  then have ((λz. z * (∑ n. u x y n * z ^ n)) ⟶ 1 * U x y) (at-left 1)
    by (intro tendsto-intros) simp
  then have ((λz. ∑ n. u x y n * z ^ Suc n) ⟶ 1 * U x y) (at-left 1)
    apply (rule filterlim-cong[OF refl refl, THEN iffD1, rotated])
    apply (rule eventually-at-left-1)
    apply (subst suminf-mult[symmetric])
    apply (auto intro!: summable-power-series u-le-1 u-nonneg)
    apply (simp add: field-simps)
    done
  then show ?thesis
    by (simp add: gf-U-def[abs-def] U-def)
qed

lemma gf-U-le-1: assumes z: 0 < z z < 1 shows gf-U x y z ≤ (1::real)
proof −
  note u = u-sums-U[of x y, THEN sums-summable]
  have gf-U x y z ≤ gf-U x y 1
    using z
    unfolding gf-U-def real-scaleR-def
   by (intro suminf-le allI mult-mono power-mono summable-comparison-test-ev[OF
- u] always-eventually)
      (auto simp: u-nonneg intro!: mult-left-le mult-le-one power-le-one)
  also have ... ≤ 1
    unfolding gf-U-eq-U by (rule U-le-1)
  finally show ?thesis .
qed

lemma gf-F: (gf-F x y ⟶ F x y) (at-left 1)
proof −
  have ((λz. ennreal (∑ n. f x y n * z ^ n)) ⟶ (∑ n. ennreal (f x y n))) (at-left
```

1)
  using *f-le-1 f-nonneg* **by** (*intro power-series-tendsto-at-left summable-power-series*)
 also have ($\sum$ *n. ennreal* (*f x y n*)) = *ennreal* (*suminf* (*f x y*))
   by (*intro f-nonneg suminf-ennreal ennreal-suminf-neq-top sums-summable*[*OF f-sums-F*])
 also have *suminf* (*f x y*) = *F x y*
   using *f-sums-F* **by** (*rule sums-unique*[*symmetric*])
 finally have ((*λz.* $\sum$ *n. f x y n* ∗ *z* ^ *n*) ⟶ *F x y*) (*at-left 1*)
   by (*rule tendsto-ennrealD*)
       (*auto simp*: *f-nonneg f-le-1 intro*!: *suminf-nonneg summable-power-series eventually-at-left-1*)
 then show *?thesis*
   by (*simp add*: *gf-F-def*[*abs-def*] *F-def*)
qed

lemma *U-bounded*: *0* ≤ *U x y U x y* ≤ *1*
 unfolding *U-def* **by** *simp-all*

4.3 Recurrent states

definition *recurrent* :: *'s* ⇒ *bool* **where**
 recurrent s ⟷ (*AE ω in T s. ev* (*HLD* {*s*}) *ω*)

lemma *recurrent-iff-U-eq-1*: *recurrent s* ⟷ *U s s* = *1*
   unfolding *recurrent-def U-def* **by** (*subst T.prob-Collect-eq-1*) *simp-all*

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

lemma *H-eq*:
 recurrent s ⟷ *H s s* = *1*
 ¬ *recurrent s* ⟷ *H s s* = *0*
 H s t = *U s t* ∗ *H t t*
proof −
 define *H'* **where** *H' t n* = {*ω*∈*space S. enat n* ≤ *scount* (*HLD* {*t*::*'s*}) *ω*} **for** *t n*
 have [*measurable*]: ⋀*y n. H' y n* ∈ *sets S*
   by (*simp add*: *H'-def*)
 let *?H'* = *λs t n. measure* (*T s*) (*H' t n*)
 { fix *x y* :: *'s* **and** *ω*
   have *Suc 0* ≤ *scount* (*HLD* {*y*}) *ω* ⟷ *ev* (*HLD* {*y*}) *ω*
     using *scount-eq-0-iff*[*of HLD* {*y*} *ω*]
     by (*cases scount* (*HLD* {*y*}) *ω rule*: *enat-coexhaust*)
      (*auto simp*: *not-ev-iff*[*symmetric*] *eSuc-enat*[*symmetric*] *enat-0 HLD-iff*[*abs-def*])
}
 then have *H'-1*: ⋀*x y. ?H' x y 1* = *U x y*
   unfolding *H'-def U-def* **by** *simp*

 { fix *n* **and** *x y* :: *'s*
   let *?U* = (*not* (*HLD* {*y*}) *suntil* (*HLD* {*y*} *aand nxt* (*λω. enat n* ≤ *scount*

```
(HLD {y}) ω)))
   { fix ω
     have enat (Suc n) ≤ scount (HLD {y}) ω ⟷ ?U ω
     proof
       assume enat (Suc n) ≤ scount (HLD {y}) ω
       with scount-eq-0-iff[of HLD {y} ω] have ev (HLD {y}) ω enat (Suc n) ≤
scount (HLD {y}) ω
         by (auto simp add: not-ev-iff[symmetric] eSuc-enat[symmetric])
       then show ?U ω
         by (induction rule: ev-induct-strong)
           (auto simp: scount-simps eSuc-enat[symmetric] intro: suntil.intros)
     next
       assume ?U ω then show enat (Suc n) ≤ scount (HLD {y}) ω
         by induction (auto simp: scount-simps  eSuc-enat[symmetric])
     qed }
   then have emeasure (T x) (H' y (Suc n)) = emeasure (T x) {ω∈space (T x).
?U ω}
     by (simp add: H'-def)
   also have ... = U x y * ?H' y y n
     by (subst emeasure-suntil-HLD) (simp-all add: T.emeasure-eq-measure U-def
H'-def ennreal-mult)
   finally have ?H' x y (Suc n) = U x y * ?H' y y n
     by (simp add: T.emeasure-eq-measure) }
  note H'-Suc = this

  { fix m and x :: 's
    have ?H' x x (Suc m) = U x x^Suc m
      using H'-1 H'-Suc by (induct m) auto }
  note H'-eq = this

  { fix x y
    have ?H' x y ⟶ measure (T x) (⋂i. H' y i)
    proof (rule T.finite-Lim-measure-decseq)
      show range (H' y) ⊆ T.events x
        by auto
    next
      show decseq (H' y)
        by (rule antimonoI) (simp add: subset-eq H'-def order-subst2)
    qed
    also have (⋂i. H' y i) = {ω∈space (T x). alw (ev (HLD {y})) ω}
     by (auto simp: H'-def scount-infinite-iff[symmetric]) (metis Suc-ile-eq enat.exhaust
neq-iff)
    finally have ?H' x y ⟶ H x y
      unfolding H-def . }
  note H'-lim = this

  from H'-lim[of s s, THEN LIMSEQ-Suc]
  have (λn. U s s ^ Suc n) ⟶ H s s
    by (simp add: H'-eq)
```

```
  then have lim-H: (λn. U s s ^ n) ⟶ H s s
    by (rule LIMSEQ-imp-Suc)

  have U s s < 1 ⟹ (λn. U s s ^ n) ⟶ 0
    by (rule LIMSEQ-realpow-zero) (simp-all add: U-def)
  with lim-H have U s s < 1 ⟹ H s s = 0
    by (blast intro: LIMSEQ-unique)
  moreover have U s s = 1 ⟹ (λn. U s s ^ n) ⟶ 1
    by simp
  with lim-H have U s s = 1 ⟹ H s s = 1
    by (blast intro: LIMSEQ-unique)
  moreover note recurrent-iff-U-eq-1 U-cases
  ultimately show recurrent s ⟷ H s s = 1 ¬ recurrent s ⟷ H s s = 0
    by (metis one-neq-zero)+

  from H'-lim[of s t, THEN LIMSEQ-Suc] H'-Suc[of s]
  have (λn. U s t * ?H' t t n) ⟶ H s t
    by simp
  moreover have (λn. U s t * ?H' t t n) ⟶ U s t * H t t
    by (intro tendsto-intros H'-lim)
  ultimately show H s t = U s t * H t t
    by (blast intro: LIMSEQ-unique)
qed

lemma recurrent-iff-G-infinite: recurrent x ⟷ G x x = ∞
proof −
  have ((λz. ennreal (gf-G x x z)) ⟶ G x x) (at-left 1)
    by (rule lim-gf-G)
 then have G: ((λz. ennreal (1 / (1 − gf-U x x z))) ⟶ G x x) (at-left (1::real))
    apply (rule filterlim-cong[OF refl refl, THEN iffD1, rotated])
    apply (rule eventually-at-left-1)
    apply (subst gf-G-eq-gf-U)
    apply (rule convergence-G-less-1)
    apply simp
    apply simp
    done

  { fix z :: real assume z: 0 < z z < 1
    have 1: summable (u x x)
      using u-sums-U by (rule sums-summable)
    have gf-U x x z ≠ 1
      using gf-G-eq-gf-U[OF convergence-G-less-1[of z]] z by simp
    moreover
    have gf-U x x z ≤ U x x
      unfolding gf-U-def gf-U-eq-U[symmetric]
      using z
      by (intro suminf-le)
         (auto simp add: 1 convergence-U convergence-G-less-1 u-nonneg simp del:
power-Suc
```

```
        intro!: mult-right-le-one-le power-le-one)
    ultimately have gf-U x x z < 1
      using U-bounded[of x x] by simp }
  note strict = this

  { assume U x x = 1
    moreover have ((λxa. 1 − gf-U x x xa :: real) ⟶ 1 − U x x) (at-left 1)
      by (intro tendsto-intros gf-U)
    moreover have eventually (λz. gf-U x x z < 1) (at-left (1::real))
      by (auto intro!: eventually-at-left-1 strict simp: ‹U x x = 1› gf-U-eq-U)
    ultimately have ((λz. ennreal (1 / (1 − gf-U x x z))) ⟶ top) (at-left 1)
      unfolding ennreal-tendsto-top-eq-at-top
      by (intro LIM-at-top-divide[where a=1] tendsto-const zero-less-one)
        (auto simp: field-simps)
    with G have G x x = top
      by (rule tendsto-unique[rotated]) simp }
  moreover
  { assume U x x < 1
    then have ((λxa. ennreal (1 / (1 − gf-U x x xa))) ⟶ 1 / (1 − U x x))
(at-left 1)
      by (intro tendsto-intros gf-U tendsto-ennrealI) simp
    from tendsto-unique[OF - G this] have G x x ≠ ∞
      by simp }
  ultimately show ?thesis
    using U-cases recurrent-iff-U-eq-1 by auto
qed

definition communicating :: ('s × '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 (rule ccontr)
  assume ∗: (x, y) ∉ acc

  { fix ω assume ev (HLD {y}) ω alw (HLD (acc `` {x})) ω from this ∗ have
False
      by induction (auto simp: HLD-iff) }
  with AE-T-reachable[of x] have U x y = 0
    unfolding U-def by (intro T.prob-eq-0-AE) auto
  with ‹0 < U x y› show False by auto
qed

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

using *assms* **proof** *induction*
case *base* **with** *T.prob-space*[*of x*] **show** *?case*
by (*auto intro*!: *exI*[*of - 0*])
next
have [*simp*]: $\bigwedge x\ y.$ (*if x = y then 1 else 0*::*real*) = *indicator* {*y*} *x*
by *simp*
case (*step w y*)
then obtain *n* **where** *0 < p x w n* **and** *0 < pmf* (*K w*) *y*
by (*auto simp*: *set-pmf-iff less-le*)
then have *0 < p x w n * pmf* (*K w*) *y*
by (*intro mult-pos-pos*)
also have ... ≤ *p x w n * p w y* (*Suc 0*)
by (*simp add*: *p-Suc' p-0 pmf.rep-eq*)
also have ... ≤ *p x y* (*Suc n*)
using *prob-reachable-le*[*of n Suc n x w y*] **by** *simp*
finally show *?case* **..**
qed

lemma *accI-pos*: *0 < p x y n* ⟹ (*x*, *y*) ∈ *acc*
proof (*induct n arbitrary*: *x*)
case (*Suc n*)
then have *less*: *0* < (∫ *x'*. *p x' y n* ∂*K x*)
by (*simp add*: *p-Suc'*)
have ∃ *x'*∈*K x*. *0 < p x' y n*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *AE x' in K x*. *p x' y n = 0*
by (*simp add*: *AE-measure-pmf-iff less-le*)
then have (∫ *x'*. *p x' y n* ∂*K x*) = (∫ *x'*. *0* ∂*K x*)
by (*intro integral-cong-AE*) *simp-all*
with *less* **show** *False* **by** *simp*
qed
with *Suc* **show** *?case*
by (*auto intro*: *converse-rtrancl-into-rtrancl*)
qed (*simp add*: *p-0 split*: *if-split-asm*)

lemma *recurrent-iffI-communicating*:
assumes (*x*, *y*) ∈ *communicating*
shows *recurrent x* ⟷ *recurrent y*
proof −
from *assms* **obtain** *n m* **where** *0 < p x y n 0 < p y x m*
by (*force simp*: *communicating-def dest*: *accD-pos*)
moreover
{ fix *x y n m* **assume** *0 < p x y n 0 < p y x m G y y* = ∞
then have ∞ = *ennreal* (*p x y n * p y x m*) * *G y y*
by (*auto intro*: *mult-pos-pos simp*: *ennreal-mult-top*)
also have *ennreal* (*p x y n * p y x m*) * *G y y* = (∑ *i*. *ennreal* (*p x y n * p y x m*) * *p y y i*)
unfolding *G-eq-suminf* **by** (*rule ennreal-suminf-cmult*[*symmetric*])

```
    also have ... ≤ (∑ i. ennreal (p x x (n + i + m)))
    proof (intro suminf-le allI)
      fix i
      have (p x y n * p y y ((n + i) - n)) * p y x ((n + i + m) - (n + i)) ≤ p x
y (n + i) * p y x ((n + i + m) - (n + i))
        by (intro mult-right-mono prob-reachable-le) simp-all
      also have ... ≤ p x x (n + i + m)
        by (intro prob-reachable-le) simp-all
      finally show ennreal (p x y n * p y x m) * p y y i ≤ ennreal (p x x (n + i
+ m))
        by (simp add: ac-simps ennreal-mult'[symmetric])
    qed auto
    also have ... ≤ (∑ i. ennreal (p x x (i + (n + m))))
      by (simp add: ac-simps)
    also have ... ≤ (∑ i. ennreal (p x x i))
      by (subst suminf-offset[of λi. ennreal (p x x i) n + m]) auto
    also have ... ≤ G x x
      unfolding G-eq-suminf by (auto intro!: suminf-le-pos)
    finally have G x x = ∞
      by (simp add: top-unique) }
  ultimately show ?thesis
    using recurrent-iff-G-infinite by blast
qed

lemma recurrent-acc:
  assumes recurrent x (x, y) ∈ acc
  shows U y x = 1 H y x = 1 recurrent y (x, y) ∈ communicating
proof −
  { fix w y assume step: (x, w) ∈ acc y ∈ K w U w x = 1 H w x = 1 recurrent w
x ≠ y
    have measure (K w) UNIV = U w x
      using step measure-pmf.prob-space[of K w] by simp
    also have ... = (∫ v. indicator {x} v + U v x * indicator (− {x}) v ∂K w)
      unfolding U-def
      by (subst prob-T)
        (auto intro!: Bochner-Integration.integral-cong arg-cong2[where f=measure]
AE-I2
              simp: ev-Stream T.prob-eq-1 split: split-indicator)
    also have ... = measure (K w) {x} + (∫ v. U v x * indicator (− {x}) v ∂K
w)
      by (subst Bochner-Integration.integral-add)
        (auto intro!: measure-pmf.integrable-const-bound[where B=1]
          simp: abs-mult mult-le-one U-bounded(2) measure-pmf.emeasure-eq-measure)
    finally have measure (K w) UNIV − measure (K w) {x} = (∫ v. U v x *
indicator (− {x}) v ∂K w)
      by simp
  also have measure (K w) UNIV − measure (K w) {x} = measure (K w) (UNIV
− {x})
      by (subst measure-pmf.finite-measure-Diff) auto
```

```
    finally have 0 = (∫ v. indicator (− {x}) v ∂K w) − (∫ v. U v x * indicator
(− {x}) v ∂K w)
      by (simp add: measure-pmf.emeasure-eq-measure Compl-eq-Diff-UNIV)
    also have ... = (∫ v. (1 − U v x) * indicator (− {x}) v ∂K w)
      by (subst Bochner-Integration.integral-diff[symmetric])
       (auto intro!: measure-pmf.integrable-const-bound[where B=1] Bochner-Integration.integral-cong
             simp: abs-mult mult-le-one U-bounded(2) split: split-indicator)
    also have ... ≥ (∫ v. (1 − U y x) * indicator {y} v ∂K w) (is - ≥ ?rhs)
      using ‹recurrent x›
      by (intro integral-mono measure-pmf.integrable-const-bound[where B=1])
       (auto simp: abs-mult mult-le-one U-bounded(2) recurrent-iff-U-eq-1 field-simps
             split: split-indicator)
    also (xtrans) have ?rhs = (1 − U y x) * pmf (K w) y
      by (simp add: measure-pmf.emeasure-eq-measure pmf.rep-eq)
    finally have (1 − U y x) * pmf (K w) y = 0
      by (auto intro!: antisym simp: U-bounded(2) mult-le-0-iff)
    with ‹y ∈ K w› have U y x = 1
      by (simp add: set-pmf-iff)
    then have U y x = 1 H y x = 1
      using H-eq(3)[of y x] H-eq(1)[of x] by (simp-all add: ‹recurrent x›)
    then have (y, x) ∈ acc
      by (intro accI-U) auto
    with step have (x, y) ∈ communicating
      by (auto simp add: communicating-def intro: rtrancl-trans)
    with ‹recurrent x› have recurrent y
      by (simp add: recurrent-iffI-communicating)
    note this ‹U y x = 1› ‹H y x = 1› ‹(x, y) ∈ communicating› }
  note enabled = this

  from ‹(x, y) ∈ acc›
  show U y x = 1 H y x = 1 recurrent y (x, y) ∈ communicating
  proof induction
    case base then show U x x = 1 H x x = 1 recurrent x (x, x) ∈ communicating
      using ‹recurrent x› H-eq(1)[of x] by (auto simp: recurrent-iff-U-eq-1 commu-
nicating-def)
  next
    case (step w y)
    with enabled[of w y] ‹recurrent x› H-eq(1)[of x]
    have U y x = 1 ∧ H y x = 1 ∧ recurrent y ∧ (x, y) ∈ communicating
      by (cases x = y) (auto simp: recurrent-iff-U-eq-1 communicating-def)
    then show U y x = 1 H y x = 1 recurrent y (x, y) ∈ communicating
      by auto
  qed
qed

lemma equiv-communicating: equiv UNIV communicating
  by (auto simp: equiv-def sym-def communicating-def refl-on-def trans-def)

lemma recurrent-class:
```

assumes *recurrent x*
shows *acc `` {x} = communicating `` {x}*
using *recurrent-acc(4)[OF ‹recurrent x›]* **by** (*auto simp*: *communicating-def*)

lemma *irreduccible-recurrent-class*:
assumes *recurrent x* **shows** *acc `` {x} ∈ UNIV // communicating*
unfolding *recurrent-class[OF ‹recurrent x›]* **by** (*rule quotientI*) *simp*

lemma *essential-classI*:
assumes *C*: *C ∈ UNIV // communicating*
assumes *eq*: *⋀x y. x ∈ C ⟹ (x, y) ∈ acc ⟹ y ∈ C*
shows *essential-class C*
by (*auto simp*: *essential-class-def intro*: *C*) (*metis eq*)

lemma *essential-recurrent-class*:
assumes *recurrent x* **shows** *essential-class* (*communicating `` {x}*)
unfolding *recurrent-class[OF ‹recurrent x›, symmetric]*
apply (*rule essential-classI*)
apply (*rule irreduccible-recurrent-class[OF assms]*)
apply (*auto simp*: *communicating-def*)
done

lemma *essential-classD2*:
essential-class C ⟹ x ∈ C ⟹ (x, y) ∈ acc ⟹ y ∈ C
unfolding *essential-class-def* **by** *auto*

lemma *essential-classD3*:
essential-class C ⟹ x ∈ C ⟹ y ∈ C ⟹ (x, y) ∈ communicating
unfolding *essential-class-def*
by (*auto elim*!: *quotientE simp*: *communicating-def*)

lemma *AE-acc*:
shows *AE ω in T x. ∀ m. (x, (x ## ω) !! m) ∈ acc*
using *AE-T-reachable*
by *eventually-elim* (*auto simp*: *alw-HLD-iff-streams streams-iff-snth Stream-snth split*: *nat.splits*)

lemma *finite-essential-class-imp-recurrent*:
assumes *C*: *essential-class C finite C* **and** *x*: *x ∈ C*
shows *recurrent x*
proof −
have *AE ω in T x. ∃ y∈C. alw* (*ev* (*HLD* {*y*})) *ω*
using *AE-T-reachable*
proof *eventually-elim*
fix *ω* **assume** *alw* (*HLD* (*acc `` {x}*)) *ω*
then have *alw* (*HLD C*) *ω*
by (*rule alw-mono*) (*auto simp*: *HLD-iff intro*: *assms essential-classD2*)
then show *∃ y∈C. alw* (*ev* (*HLD* {*y*})) *ω*
by (*rule pigeonhole-stream*) *fact*

qed
then have *1* = $\mathcal{P}(\omega$ *in* T *x*. $\exists$ *y*$\in$*C*. *alw* (*ev* (*HLD* {*y*})) ω)
by (*subst* (*asm*) *T.prob-Collect-eq-1*[*symmetric*]) (*auto simp*: ‹*finite C*›)
also have ... = *measure* (*T x*) ($\bigcup$ *y*$\in$*C*. {$\omega$$\in$*space* (*T x*). *alw* (*ev* (*HLD* {*y*})) ω})
by (*intro arg-cong2*[**where** *f*=*measure*]) *auto*
also have ... $\leq$ ($\sum$ *y*$\in$*C*. *H x y*)
unfolding *H-def* **using** ‹*finite C*› **by** (*rule T.finite-measure-subadditive-finite*) *auto*
also have ... = ($\sum$ *y*$\in$*C*. *U x y* * *H y y*)
by (*auto intro*!: *sum.cong H-eq*)
finally have $\exists$ *y*$\in$*C*. *recurrent y*
by (*rule-tac ccontr*) (*simp add*: *H-eq*(*2*))
then obtain *y* **where** *y* $\in$ *C recurrent y* **..**
from *essential-classD3*[*OF C*(*1*) *x this*(*1*)] *recurrent-acc*(*3*)[*OF this*(*2*)]
show *recurrent x*
by (*simp add*: *communicating-def*)
qed

lemma *irreducibleD*:
C $\in$ *UNIV* // *communicating* $\Longrightarrow$ *a* $\in$ *C* $\Longrightarrow$ *b* $\in$ *C* $\Longrightarrow$ (*a*, *b*) $\in$ *communicating*
by (*auto elim*!: *quotientE simp*: *communicating-def*)

lemma *irreducibleD2*:
C $\in$ *UNIV* // *communicating* $\Longrightarrow$ *a* $\in$ *C* $\Longrightarrow$ (*a*, *b*) $\in$ *communicating* $\Longrightarrow$ *b* $\in$ *C*
by (*auto elim*!: *quotientE simp*: *communicating-def*)

lemma *essential-class-iff-recurrent*:
finite C $\Longrightarrow$ *C* $\in$ *UNIV* // *communicating* $\Longrightarrow$ *essential-class C* $\longleftrightarrow$ ($\forall$ *x*$\in$*C*. *recurrent x*)
by (*metis finite-essential-class-imp-recurrent irreducibleD2 recurrent-acc*(*4*) *essential-classI*)

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

lemma *U'-neq-zero*[*simp*]: *U'* *x y* $\neq$ *0*
unfolding *U'-def* **by** (*simp add*: *nn-integral-add*)

definition *gf-U'* *x y z* = ($\sum$ *n*. *u x y n* * *Suc n* * *z* ^ *n*)

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

lemma *summable-gf-U'*:
assumes *z*: *norm z* < *1*
shows *summable* (λ*n*. *u x y n* * *Suc n* * *z* ^ *n*)
proof −
have *summable* (λ*n*. *n* * |*z*| ^ *n*)
proof (*rule root-test-convergence*)

```
    have (λn. root n n * |z|) ⟶ 1 * |z|
      by (intro tendsto-intros LIMSEQ-root)
    then show (λn. root n (norm (n * |z| ^ n))) ⟶ |z|
      by (rule filterlim-cong[THEN iffD1, rotated 3])
        (auto intro!: exI[of - 1]
                  simp add: abs-mult u-nonneg real-root-mult power-abs eventu-
ally-sequentially real-root-power)
  qed (insert z, simp add: abs-less-iff)
  note summable-mult[OF this, of 1 / |z|]
  from summable-ignore-initial-segment[OF this, of 1]
  show summable (λn. u x y n * Suc n * z ^ n)
    apply (rule summable-comparison-test[rotated])
    using z
    apply (auto intro!: exI[of - 1]
            simp: abs-mult u-nonneg power-abs Suc-le-eq gr0-conv-Suc field-simps
le-divide-eq u-le-1
            simp del: of-nat-Suc)
    done
qed

lemma gf-U'-nonneg[simp]: 0 < z ⟹ z < 1 ⟹ 0 ≤ gf-U' x y z
  unfolding gf-U'-def
  by (intro suminf-nonneg summable-gf-U') (auto simp: u-nonneg)

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
  unfolding gf-U-def[abs-def]  gf-U'-def real-scaleR-def u-def[symmetric]
  using z by (intro DERIV-power-series'[where R=1] summable-gf-U') auto

lemma sfirst-finiteI-recurrent:
  recurrent x ⟹ (x, y) ∈ acc ⟹ AE ω in T x. sfirst (HLD {y}) ω < ∞
  using recurrent-acc(1)[of y x] recurrent-acc[of x y]
    T.AE-prob-1[of x {ω∈space (T x). ev (HLD {y}) ω}]
   unfolding sfirst-finite U-def by (simp add: space-stream-space communicat-
ing-def)

lemma U'-eq-suminf:
  assumes x: recurrent x (x, y) ∈ acc
  shows U' x y = (∑ i. ennreal (u x y i * Suc i))
proof −
  have (∫+ω. eSuc (sfirst (HLD {y}) ω) ∂T x) =
      (∫+ω. (∑ i. ennreal (Suc i) * indicator {ω∈space (T y). ev-at (HLD {y}) i
ω} ω) ∂T x)
    using sfirst-finiteI-recurrent[OF x]
  proof (intro nn-integral-cong-AE, eventually-elim)
    fix ω assume sfirst (HLD {y}) ω < ∞
    then obtain n :: nat where [simp]: sfirst (HLD {y}) ω = n
      by auto
```

```
    show eSuc (sfirst (HLD {y}) ω) = (∑ i. ennreal (Suc i) * indicator {ω∈space
(T y). ev-at (HLD {y}) i ω} ω)
      by (subst suminf-cmult-indicator[where i=n])
         (auto simp: disjoint-family-on-def ev-at-unique space-stream-space
                     sfirst-eq-enat-iff[symmetric] ennreal-of-nat-eq-real-of-nat
               split: split-indicator)
  qed
  also have ... = (∑ i. ennreal (Suc i) * emeasure (T x) {ω∈space (T x). ev-at
(HLD {y}) i ω})
    by (subst nn-integral-suminf)
       (auto intro!: arg-cong[where f=suminf] nn-integral-cmult-indicator simp:
fun-eq-iff)
  finally show ?thesis
    by (simp add: U'-def u-def T.emeasure-eq-measure mult-ac ennreal-mult)
qed

lemma gf-U'-tendsto-U':
  assumes x: recurrent x (x, y) ∈ acc
  shows ((λz. ennreal (gf-U' x y z)) ⟶ U' x y) (at-left 1)
  unfolding U'-eq-suminf[OF x] gf-U'-def
  by (auto intro!: power-series-tendsto-at-left summable-gf-U' mult-nonneg-nonneg
u-nonneg simp del: of-nat-Suc)

lemma one-le-integral-t:
  assumes x: recurrent x shows 1 ≤ U' x x
  by (simp add: nn-integral-add T.emeasure-space-1 U'-def del: space-T)

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
  unfolding gf-U'-def
proof (subst suminf-pos-iff)
  show summable (λn. u x y n * real (Suc n) * z ^ n)
    using z by (intro summable-gf-U') simp
  show pos: ⋀n. 0 ≤ u x y n * real (Suc n) * z ^ n
    using u-nonneg z by auto
  show ∃ n. 0 < u x y n * real (Suc n) * z ^ n
  proof (rule ccontr)
    assume ¬ (∃ n. 0 < u x y n * real (Suc n) * z ^ n)
    with pos have ∀ n. u x y n * real (Suc n) * z ^ n = 0
      by (intro antisym allI) (simp-all add: not-less)
    with z have u x y = (λn. 0)
      by (intro ext) simp
    with u-sums-U[of x y, THEN sums-unique] ‹U x y ≠ 0› show False
      by simp
  qed
qed
```

lemma *inverse-gf-U'-tendsto*:
assumes *recurrent y*
shows $((\lambda x.\ -\ 1\ /\ -\ gf\text{-}U'\ y\ y\ x) \longrightarrow enn2real\ (1\ /\ U'\ y\ y))\ (at\text{-}left\ (1::real))$
proof *cases*
assume *inf*: $U'\ y\ y = \infty$
with *gf-U'-tendsto-U'*[*of y y*] ‹*recurrent y*›
have *LIM z* (*at-left 1*). *gf-U' y y z* :> *at-top*
by (*auto simp*: *ennreal-tendsto-top-eq-at-top U'-def*)
then have *LIM z* (*at-left 1*). *gf-U' y y z* :> *at-infinity*
by (*rule filterlim-mono*) (*auto simp*: *at-top-le-at-infinity*)
with *inf* **show** *?thesis*
by (*auto intro*!: *tendsto-divide-0*)
next
assume *fin*: $U'\ y\ y \neq \infty$
then obtain *r* **where** *r*: $U'\ y\ y = ennreal\ r$ **and** [*simp*]: $0 \leq r$
by (*cases U' y y*) (*auto simp*: *U'-def*)
then have *eq*: $enn2real\ (1\ /\ U'\ y\ y) = -\ 1\ /\ -\ r$ **and** $1 \leq r$
using *one-le-integral-t*[*OF* ‹*recurrent y*›]
by (*auto simp add*: *ennreal-1*[*symmetric*] *divide-ennreal simp del*: *ennreal-1*)
have $((\lambda z.\ ennreal\ (gf\text{-}U'\ y\ y\ z)) \longrightarrow ennreal\ r)\ (at\text{-}left\ 1)$
using *gf-U'-tendsto-U'*[*OF* ‹*recurrent y*›, *of y*] *r* **by** *simp*
then have *gf-U'*: $(gf\text{-}U'\ y\ y \longrightarrow r)\ (at\text{-}left\ (1::real))$
by (*rule tendsto-ennrealD*)
(*insert summable-gf-U'*, *auto intro*!: *eventually-at-left-1 suminf-nonneg simp*: *gf-U'-def u-nonneg*)
show *?thesis*
using ‹$1 \leq r$› **unfolding** *eq* **by** (*intro tendsto-intros gf-U'*) *simp*
qed

lemma *gf-G-pos*:
fixes *z* :: *real*
assumes *z*: $0 < z\ z < 1$ **and** $*$: $(x, y) \in acc$
shows $0 < gf\text{-}G\ x\ y\ z$
unfolding *gf-G-def*
proof (*subst suminf-pos-iff*)
show *summable* $(\lambda n.\ p\ x\ y\ n *_R z \mathbin{\hat{}} n)$
using *z* **by** (*intro convergence-G convergence-G-less-1*) *simp*
show *pos*: $\bigwedge n.\ 0 \leq p\ x\ y\ n *_R z \mathbin{\hat{}} n$
using *z* **by** (*auto intro*!: *mult-nonneg-nonneg p-nonneg*)
show $\exists n.\ 0 < p\ x\ y\ n *_R z \mathbin{\hat{}} n$
proof (*rule ccontr*)
assume $\neg\ (\exists n.\ 0 < p\ x\ y\ n *_R z \mathbin{\hat{}} n)$
with *pos* **have** $\forall n.\ p\ x\ y\ n * z \mathbin{\hat{}} n = 0$
by (*intro antisym allI*) (*simp-all add*: *not-less*)
with *z* **have** $\bigwedge n.\ p\ x\ y\ n = 0$
by *simp*
with $*$[*THEN accD-pos*] **show** *False*
by *simp*
qed

qed

lemma *pos-recurrentI-communicating*:
assumes *y*: *pos-recurrent y* **and** *x*: $(y, x) \in$ *communicating*
shows *pos-recurrent x*
proof −
from *y x* **have** *recurrent*: *recurrent y recurrent x* **and** *fin*: $U' \; y \; y \neq \infty$
by (*auto simp*: *pos-recurrent-def recurrent-iffI-communicating nn-integral-add*)
have *pos*: $0 <$ *enn2real* $(1 \; / \; U' \; y \; y)$
using *one-le-integral-t*[*OF* ‹*recurrent y*›] *fin*
by (*auto simp*: *U'-def enn2real-positive-iff less-top*[*symmetric*] *ennreal-zero-less-divide ennreal-divide-eq-top-iff*)

from *fin* **obtain** *r* **where** *r*: $U' \; y \; y =$ *ennreal r* **and** [*simp*]: $0 \leq r$
by (*cases* $U' \; y \; y$) (*auto simp*: *U'-def*)

from *x* **obtain** *n m* **where** $0 < p \; x \; y \; n \; 0 < p \; y \; x \; m$
by (*auto dest*!: *accD-pos simp*: *communicating-def*)

let *?L* = *at-left* (*1*::*real*)
have *le*: *eventually* $(\lambda z. \; p \; x \; y \; n * p \; y \; x \; m * z\hat{}(n + m) \leq (1 - \textit{gf-U} \; y \; y \; z) \; / \; (1 - \textit{gf-U} \; x \; x \; z))$ *?L*
proof (*rule eventually-at-left-1*)
fix *z* :: *real* **assume** *z*: $0 < z \; z < 1$
then have *conv*: $\bigwedge x.$ *convergence-G x x z*
by (*intro convergence-G-less-1*) *simp*
have *sums*: $(\lambda i. \; (p \; x \; y \; n * p \; y \; x \; m * z\hat{}(n + m)) * (p \; y \; y \; i * z\hat{}i))$ *sums* $((p \; x \; y \; n * p \; y \; x \; m * z\hat{}(n + m)) * \textit{gf-G} \; y \; y \; z)$
unfolding *gf-G-def*
by (*intro sums-mult summable-sums*) (*auto intro*!: *conv convergence-G*[**where** *'a=real*, *simplified*])
have $(\sum i. \; (p \; x \; y \; n * p \; y \; x \; m * z\hat{}(n + m)) * (p \; y \; y \; i * z\hat{}i)) \leq (\sum i. \; p \; x \; x \; (i + (n + m)) * z\hat{}(i + (n + m)))$
proof (*intro allI suminf-le sums-summable*[*OF sums*] *summable-ignore-initial-segment convergence-G*[**where** *'a=real*, *simplified*] *convergence-G-less-1*)
show *norm z* < 1 **using** *z* **by** *simp*
fix *i*
have $(p \; x \; y \; n * p \; y \; y \; ((n + i) - n)) * p \; y \; x \; ((n + i + m) - (n + i)) \leq p \; x \; y \; (n + i) * p \; y \; x \; ((n + i + m) - (n + i))$
by (*intro mult-right-mono prob-reachable-le*) *simp-all*
also have $\ldots \leq p \; x \; x \; (n + i + m)$
by (*intro prob-reachable-le*) *simp-all*
finally show $p \; x \; y \; n * p \; y \; x \; m * z \; \hat{} \; (n + m) * (p \; y \; y \; i * z \; \hat{} \; i) \leq p \; x \; x \; (i + (n + m)) * z \; \hat{} \; (i + (n + m))$
using *z* **by** (*auto simp add*: *ac-simps power-add intro*!: *mult-left-mono*)
qed
also have $\ldots \leq$ *gf-G x x z*
unfolding *gf-G-def*
using *z*

```
    apply (subst (2) suminf-split-initial-segment[where k=n + m])
    apply (intro convergence-G conv)
    apply (simp add: sum-nonneg)
    done
  finally have (p x y n * p y x m * z^(n + m)) * gf-G y y z ≤ gf-G x x z
    using sums-unique[OF sums] by simp
  then have (p x y n * p y x m * z^(n + m)) ≤ gf-G x x z / gf-G y y z
    using z gf-G-pos[of z y y] by (simp add: field-simps)
  also have ... = (1 - gf-U y y z) / (1 - gf-U x x z)
    unfolding gf-G-eq-gf-U[OF conv] using gf-G-eq-gf-U(2)[OF conv] by (simp
add: field-simps )
  finally show p x y n * p y x m * z^(n + m) ≤ (1 - gf-U y y z) / (1 - gf-U
x x z) .
 qed

 have U' x x ≠ ∞
 proof
  assume U' x x = ∞
  have ((λz. (1 - gf-U y y z) / (1 - gf-U x x z)) ⟶ 0) ?L
  proof (rule lhopital-left)
   show ((λz. 1 - gf-U y y z) ⟶ 0) ?L
     using gf-U[of y] recurrent-iff-U-eq-1[of y] ‹recurrent y› by (auto intro!:
tendsto-eq-intros)
   show ((λz. 1 - gf-U x x z) ⟶ 0) ?L
     using gf-U[of x] recurrent-iff-U-eq-1[of x] ‹recurrent x› by (auto intro!:
tendsto-eq-intros)
   show eventually (λz. 1 - gf-U x x z ≠ 0) ?L
   by (auto intro!: eventually-at-left-1 simp: gf-G-eq-gf-U(2) convergence-G-less-1)
   show eventually (λz. - gf-U' x x z ≠ 0) ?L
    using gf-U'-pos[of - x x] recurrent-iff-U-eq-1[of x] ‹recurrent x›
    by (auto intro!: eventually-at-left-1) (metis less-le)
   show eventually (λz. DERIV (λxa. 1 - gf-U x x xa) z :> - gf-U' x x z) ?L
    by (auto intro!: eventually-at-left-1 derivative-eq-intros DERIV-gf-U)
   show eventually (λz. DERIV (λxa. 1 - gf-U y y xa) z :> - gf-U' y y z) ?L
    by (auto intro!: eventually-at-left-1 derivative-eq-intros DERIV-gf-U)

   have (gf-U' y y ⟶ U' y y) ?L
    using ‹recurrent y› by (rule gf-U'-tendsto-U') simp
   then have *: (gf-U' y y ⟶ r) ?L
    by (auto simp add: r eventually-at-left-1 dest!: tendsto-ennrealD)
   moreover
   have (gf-U' x x ⟶ U' x x) ?L
    using ‹recurrent x› by (rule gf-U'-tendsto-U') simp
   then have LIM z ?L. - gf-U' x x z :> at-bot
   by (simp add: ennreal-tendsto-top-eq-at-top ‹U' x x = ∞› filterlim-uminus-at-top
          del: ennreal-of-enat-eSuc)
   then have LIM z ?L. - gf-U' x x z :> at-infinity
    by (rule filterlim-mono) (auto simp: at-bot-le-at-infinity)
   ultimately show ((λz. - gf-U' y y z / - gf-U' x x z) ⟶ 0) ?L
```

```
      by (intro tendsto-divide-0[where c=- r] tendsto-intros)
    qed
    moreover
    have ((λz. p x y n * p y x m * z^(n + m)) ⟶ p x y n * p y x m) ?L
      by (auto intro!: tendsto-eq-intros)
    ultimately have p x y n * p y x m ≤ 0
      using le by (rule tendsto-le[OF trivial-limit-at-left-real])
    with ‹0 < p x y n› ‹0 < p y x m› show False
      by (auto simp add: mult-le-0-iff)
  qed
  with ‹recurrent x› show ?thesis
    by (simp add: pos-recurrent-def nn-integral-add)
qed

lemma pos-recurrent-iffI-communicating:
  (y, x) ∈ communicating ⟹ pos-recurrent y ⟷ pos-recurrent x
  using pos-recurrentI-communicating[of x y] pos-recurrentI-communicating[of y x]
  by (auto simp add: communicating-def)

lemma U-le-F: U x y ≤ F x y
  by (auto simp: U-def F-def intro!: T.finite-measure-mono)

lemma not-empty-irreducible: C ∈ UNIV // communicating ⟹ C ≠ {}
  by (auto simp: quotient-def Image-def communicating-def)
```

4.4 Stationary distribution

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

lemma sets-stat[simp]: sets (stat C) = sets (count-space UNIV)
  by (simp add: stat-def sets-point-measure)

lemma space-stat[simp]: space (stat C) = UNIV
  by (simp add: stat-def space-point-measure)

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 −
  let ?L = at-left (1::real)
  from finite-sequence-to-countable-set[OF ‹countable C›]
  obtain A where A: ⋀i. A i ⊆ C ⋀i. A i ⊆ A (Suc i) ⋀i. finite (A i) ⋃ (range
A) = C
    by blast
  then have (λn. emeasure (stat C) (A n)) ⟶ emeasure (stat C) (⋃i. A i)
    by (intro Lim-emeasure-incseq) (auto simp: incseq-Suc-iff)
  then have emeasure (stat C) (⋃i. A i) ≤ 1
  proof (rule LIMSEQ-le[OF - tendsto-const], intro exI allI impI)
```

fix n
from $A(1,3)$ **have** $A\text{-}n$: *finite* $(A\ n)$
by *auto*

from C **have** $C \neq \{\}$
by (*simp add*: *essential-class-def not-empty-irreducible*)
then obtain x **where** $x \in C$ **by** *auto*

have $((\lambda z.\ (\sum y{\in}A\ n.\ gf\text{-}F\ x\ y\ z * ((1 - z) / (1 - gf\text{-}U\ y\ y\ z)))) \longrightarrow (\sum y{\in}A\ n.\ F\ x\ y * enn2real\ (1 / U'\ y\ y)))$ *?L*
proof (*intro tendsto-intros gf-F*, *rule lhopital-left*)
fix y **assume** $y \in A\ n$
with ‹$A\ n \subseteq C$› **have** $y \in C$
by *auto*
show $((-)\ 1 \longrightarrow 0)$ *?L*
by (*intro tendsto-eq-intros*) *simp-all*
have *recurrent y*
using *pos*[*THEN bspec*, *OF* ‹$y{\in}C$›] **by** (*simp add*: *pos-recurrent-def*)
then have $U\ y\ y = 1$
by (*simp add*: *recurrent-iff-U-eq-1*)

show $((\lambda x.\ 1 - gf\text{-}U\ y\ y\ x) \longrightarrow 0)$ *?L*
using *gf-U*[*of y y*] ‹$U\ y\ y = 1$› **by** (*intro tendsto-eq-intros*) *auto*
show *eventually* $(\lambda x.\ 1 - gf\text{-}U\ y\ y\ x \neq 0)$ *?L*
using *gf-G-eq-gf-U*(*2*)[*OF convergence-G-less-1*, **where** *'z=real*] **by** (*auto intro*!: *eventually-at-left-1*)
have *eventually* $(\lambda x.\ 0 < gf\text{-}U'\ y\ y\ x)$ *?L*
by (*intro eventually-at-left-1 gf-U'-pos*) (*simp-all add*: ‹$U\ y\ y = 1$›)
then show *eventually* $(\lambda x.\ -\ gf\text{-}U'\ y\ y\ x \neq 0)$ *?L*
by *eventually-elim simp*
show *eventually* $(\lambda x.\ DERIV\ (\lambda x.\ 1 - gf\text{-}U\ y\ y\ x)\ x :> -\ gf\text{-}U'\ y\ y\ x)$ *?L*
by (*auto intro*!: *eventually-at-left-1 derivative-eq-intros DERIV-gf-U*)
show *eventually* $(\lambda x.\ DERIV\ ((-)\ 1)\ x :> -\ 1)$ *?L*
by (*auto intro*!: *eventually-at-left-1 derivative-eq-intros*)
show $((\lambda x.\ -\ 1 / -\ gf\text{-}U'\ y\ y\ x) \longrightarrow enn2real\ (1 / U'\ y\ y))$ *?L*
using ‹*recurrent y*› **by** (*rule inverse-gf-U'-tendsto*)
qed
also have $(\sum y{\in}A\ n.\ F\ x\ y * enn2real\ (1 / U'\ y\ y)) = (\sum y{\in}A\ n.\ enn2real\ (1 / U'\ y\ y))$
proof (*intro sum.cong refl*)
fix y **assume** $y \in A\ n$
with ‹$A\ n \subseteq C$› **have** $y \in C$ **by** *auto*
with ‹$x \in C$› **have** $(x, y) \in communicating$
by (*rule essential-classD3*[*OF C*])
with ‹$y{\in}C$› **have** *recurrent y* $(y, x) \in acc$
using *pos*[*THEN bspec*, *of y*] **by** (*auto simp add*: *pos-recurrent-def communicating-def*)
then have $U\ x\ y = 1$
by (*rule recurrent-acc*)

with *F-le-1*[*of x y*] *U-le-F*[*of x y*] **have** *F x y = 1* **by** *simp*
then show *F x y * enn2real (1 / U' y y) = enn2real (1 / U' y y)*
by *simp*
qed
finally have *le*: $(\sum y{\in}A.\ enn2real\ (1\ /\ U'\ y\ y)) \leq 1$
proof (*rule tendsto-le*[*OF trivial-limit-at-left-real tendsto-const*], *intro eventually-at-left-1*)
fix *z* :: *real* **assume** *z*: *0 < z z < 1*
with ‹*x ∈ C*› **have** *norm z < 1*
by *auto*
then have *conv*: $\bigwedge x\ y.$ *convergence-G x y z*
by (*simp add*: *convergence-G-less-1*)
have $(\sum y{\in}A.\ gf\text{-}F\ x\ y\ z\ /\ (1 - gf\text{-}U\ y\ y\ z)) = (\sum y{\in}A.\ gf\text{-}G\ x\ y\ z)$
using ‹*norm z < 1*›
apply (*intro sum.cong refl*)
apply (*subst gf-G-eq-gf-F*)
apply *assumption*
apply (*subst gf-G-eq-gf-U*(*1*)[*OF conv*])
apply *auto*
done
also have $\ldots = (\sum y{\in}A.\ \sum n.\ p\ x\ y\ n * z\hat{}n)$
by (*simp add*: *gf-G-def*)
also have $\ldots = (\sum i.\ \sum y{\in}A.\ n.\ p\ x\ y\ i *_R z\hat{}i)$
by (*subst suminf-sum*[*OF convergence-G*[*OF conv*]]) *simp*
also have $\ldots \leq (\sum i.\ z\hat{}i)$
proof (*intro suminf-le summable-sum convergence-G conv summable-geometric allI*)
fix *l*
have $(\sum y{\in}A.\ n.\ p\ x\ y\ l *_R z \hat{}\ l) = (\sum y{\in}A.\ n.\ p\ x\ y\ l) * z \hat{}\ l$
by (*simp add*: *sum-distrib-right*)
also have $\ldots \leq z \hat{}\ l$
proof (*intro mult-left-le-one-le*)
have $(\sum y{\in}A.\ n.\ p\ x\ y\ l) = \mathcal{P}(\omega\ in\ T\ x.\ (x\ \#\#\ \omega)\ !!\ l \in A\ n)$
unfolding *p-def* **using** ‹*finite* (*A n*)›
by (*subst T.finite-measure-finite-Union*[*symmetric*])
(*auto simp*: *disjoint-family-on-def intro*!: *arg-cong2*[**where** *f*=*measure*])
then show $(\sum y{\in}A.\ n.\ p\ x\ y\ l) \leq 1$
by *simp*
qed (*insert z*, *auto simp*: *sum-nonneg*)
finally show $(\sum y{\in}A.\ n.\ p\ x\ y\ l *_R z \hat{}\ l) \leq z \hat{}\ l$ **.**
qed *fact*
also have $\ldots = 1\ /\ (1 - z)$
using *sums-unique*[*OF geometric-sums*, *OF* ‹*norm z < 1*›] **..**
finally have $(\sum y{\in}A.\ n.\ gf\text{-}F\ x\ y\ z\ /\ (1 - gf\text{-}U\ y\ y\ z)) \leq 1\ /\ (1 - z)$ **.**
then have $(\sum y{\in}A.\ n.\ gf\text{-}F\ x\ y\ z\ /\ (1 - gf\text{-}U\ y\ y\ z)) * (1 - z) \leq 1$
using *z* **by** (*simp add*: *field-simps*)
then have $(\sum y{\in}A.\ n.\ gf\text{-}F\ x\ y\ z\ /\ (1 - gf\text{-}U\ y\ y\ z) * (1 - z)) \leq 1$
by (*simp add*: *sum-distrib-right*)
then show $(\sum y{\in}A.\ n.\ gf\text{-}F\ x\ y\ z * ((1 - z)\ /\ (1 - gf\text{-}U\ y\ y\ z))) \leq 1$

```
      by simp
    qed

    from A-n have emeasure (stat C) (A n) = (∑ y∈A n. emeasure (stat C) {y})
      by (intro emeasure-eq-sum-singleton) simp-all
    also have ... = (∑ y∈A n. inverse (U' y y))
      unfolding stat-def U'-def using A(1)[of n]
      apply (intro sum.cong refl)
      apply (subst emeasure-point-measure-finite2)
        apply (auto simp: divide-ennreal-def Collect-conv-if)
      done
    also have ... = ennreal (∑ y∈A n. enn2real (1 / U' y y))
      apply (subst sum-ennreal[symmetric], simp)
    proof (intro sum.cong refl)
      fix y assume y ∈ A n
      with ‹A n ⊆ C› pos have pos-recurrent y
        by auto
      with one-le-integral-t[of y] obtain r where U' y y = ennreal r 1 ≤ U' y y
and [simp]: 0 ≤ r
        by (cases U' y y) (auto simp: pos-recurrent-def nn-integral-add)
      then show inverse (U' y y) = ennreal (enn2real (1 / U' y y))
          by (simp add: ennreal-1[symmetric] divide-ennreal inverse-ennreal in-
verse-eq-divide del: ennreal-1)
    qed
    also have ... ≤ 1
      using le by simp
    finally show emeasure (stat C) (A n) ≤ 1 .
  qed
  with A show ?thesis
    by simp
qed

lemma emeasure-stat-not-C:
  assumes y ∉ C
  shows emeasure (stat C) {y} = 0
  unfolding stat-def using ‹y ∉ C›
  by (subst emeasure-point-measure-finite2) auto

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
  unfolding stationary-distribution-def
proof (rule pmf-eqI antisym)+
  fix i
  show pmf (bind-pmf N K) i ≤ pmf N i
    by (simp add: pmf-bind le)
```

define Ω **where** $\Omega = N \cup (\bigcup i{\in}N.\ \textit{set-pmf}\ (K\ i))$
then have Ω: *countable* Ω
by (*auto intro*: *countable-set-pmf*)
then interpret N: *sigma-finite-measure count-space* Ω
by (*rule sigma-finite-measure-count-space-countable*)
interpret pN: *pair-sigma-finite N count-space* Ω
by *unfold-locales*

have *measurable-pmf*[*measurable*]: $(\lambda(x,\ y).\ \textit{pmf}\ (K\ x)\ y) \in \textit{borel-measurable}\ (N \bigotimes_M \textit{count-space}\ \Omega)$
unfolding *measurable-split-conv*
apply (*rule measurable-compose-countable'*[*OF - measurable-snd*])
apply (*rule measurable-compose*[*OF measurable-fst*])
apply (*simp-all add*: Ω)
done

{ assume $*$: $(\int y.\ \textit{pmf}\ (K\ y)\ i\ \partial N) < \textit{pmf}\ N\ i$
have $0 \le (\int y.\ \textit{pmf}\ (K\ y)\ i\ \partial N)$
by (*intro integral-nonneg-AE*) *simp*
with $*$ **have** i: $i \in \textit{set-pmf}\ N\ i \in \Omega$
by (*auto simp*: *set-pmf-iff* Ω*-def not-le*[*symmetric*])
from $*$ **have** $0 < \textit{pmf}\ N\ i - (\int y.\ \textit{pmf}\ (K\ y)\ i\ \partial N)$
by (*simp add*: *field-simps*)
also have $\ldots = (\int t.\ (\textit{pmf}\ N\ i - (\int y.\ \textit{pmf}\ (K\ y)\ i\ \partial N)) * \textit{indicator}\ \{i\}\ t\ \partial \textit{count-space}\ \Omega)$
by (*simp add*: *i*)
also have $\ldots \le (\int t.\ \textit{pmf}\ N\ t - \int y.\ \textit{pmf}\ (K\ y)\ t\ \partial N\ \partial \textit{count-space}\ \Omega)$
using *le*
by (*intro integral-mono integrable-diff*)
(*auto simp*: *i pmf-bind*[*symmetric*] *integrable-pmf field-simps split*: *split-indicator*)
also have $\ldots = (\int t.\ \textit{pmf}\ N\ t\ \partial \textit{count-space}\ \Omega) - (\int t.\ \int y.\ \textit{pmf}\ (K\ y)\ t\ \partial N\ \partial \textit{count-space}\ \Omega)$
by (*subst Bochner-Integration.integral-diff*) (*auto intro*!: *integrable-pmf simp*: *pmf-bind*[*symmetric*])
also have $(\int t.\ \int y.\ \textit{pmf}\ (K\ y)\ t\ \partial N\ \partial \textit{count-space}\ \Omega) = (\int y.\ \int t.\ \textit{pmf}\ (K\ y)\ t\ \partial \textit{count-space}\ \Omega\ \partial N)$
apply (*intro pN.Fubini-integral integrable-iff-bounded*[*THEN iffD2*] *conjI*)
apply (*auto simp add*: *N.nn-integral-fst*[*symmetric*] *nn-integral-eq-integral integrable-pmf*)
unfolding *less-top*[*symmetric*] **unfolding** *infinity-ennreal-def*[*symmetric*]
apply (*intro integrableD*)
apply (*auto intro*!: *measure-pmf.integrable-const-bound*[**where** *B=1*]
simp: *AE-measure-pmf-iff integral-nonneg-AE integral-pmf*)
done
also have $(\int y.\ \int t.\ \textit{pmf}\ (K\ y)\ t\ \partial \textit{count-space}\ \Omega\ \partial N) = (\int y.\ 1\ \partial N)$
by (*intro integral-cong-AE*)
(*auto simp*: *AE-measure-pmf-iff integral-pmf* Ω*-def intro*!: *measure-pmf.prob-eq-1*[*THEN iffD2*])

```
    finally have False
      using measure-pmf.prob-space[of N] by (simp add: integral-pmf field-simps
not-le[symmetric]) }
  then show pmf N i ≤ pmf (bind-pmf N K) i
    by (auto simp: pmf-bind not-le[symmetric])
qed

lemma stationary-distribution-iterate:
  assumes N: stationary-distribution N
  shows ennreal (pmf N y) = (∫⁺x. p x y n ∂N)
proof (induct n arbitrary: y)
  have [simp]: ⋀x y. ennreal (if x = y then 1 else 0) = indicator {y} x
    by simp
  case 0 then show ?case
    by (simp add: p-0 pmf.rep-eq measure-pmf.emeasure-eq-measure)
next
  case (Suc n) with N show ?case
    apply (simp add: nn-integral-eq-integral[symmetric] p-le-1 p-Suc'
                     measure-pmf.integrable-const-bound[where B=1])
    apply (subst nn-integral-bind[symmetric, where B=count-space UNIV])
    apply (auto simp: stationary-distribution-def measure-pmf-bind[symmetric]
                simp del: measurable-pmf-measure1)
    done
qed

lemma stationary-distribution-iterate':
  assumes stationary-distribution N
  shows measure N {y} = (∫ x. p x y n ∂N)
  using stationary-distribution-iterate[OF assms]
  by (subst (asm) nn-integral-eq-integral)
      (auto intro!: measure-pmf.integrable-const-bound[where B=1] simp: p-le-1
pmf.rep-eq)

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 −
  have integrable-K: ⋀f x. integrable N (λs. pmf (K s) (f x))
   by (rule measure-pmf.integrable-const-bound[where B=1]) (simp-all add: pmf-le-1)

  have measure-C: measure N C = 1 and ae-C: AE x in N. x ∈ C
    using N C measure-pmf.prob-eq-1[of C] by (auto simp: AE-measure-pmf-iff)

  have integrable-p: ⋀n y. integrable N (λx. p x y n)
   by (rule measure-pmf.integrable-const-bound[where B=1]) (simp-all add: p-le-1)

  { fix e :: real assume 0 < e
    then have [simp]: 0 ≤ e by simp
```

have $\exists A{\subseteq}C.$ *finite* $A \wedge 1 - e <$ *measure* N A
proof (*rule ccontr*)
assume *contr*: $\neg$ $(\exists A \subseteq C.$ *finite* $A \wedge 1 - e <$ *measure* N $A)$
from *finite-sequence-to-countable-set*[*OF* ‹*countable* C›]
obtain F **where** F: $\bigwedge i.$ F $i \subseteq C$ $\bigwedge i.$ F $i \subseteq F$ $(Suc$ $i)$ $\bigwedge i.$ *finite* $(F$ $i)$ $\bigcup$ $(range$ $F) = C$
by *blast*
then have $*$: $(\lambda n.$ *measure* N $(F$ $n)) \longrightarrow$ *measure* N $(\bigcup i.$ F $i)$
by (*intro measure-pmf.finite-Lim-measure-incseq*) (*auto simp*: *incseq-Suc-iff*)
with F *contr* **have** *measure* N $(\bigcup i.$ F $i) \leq 1 - e$
by (*intro LIMSEQ-le*[*OF* $*$ *tendsto-const*]) (*auto simp*: *not-less*)
with F ‹$0 < e$› **show** *False*
by (*simp add*: *measure-C*)
qed
then obtain A **where** $A \subseteq C$ *finite* A **and** e: $1 - e <$ *measure* N A **by** *auto*

{ fix y n **assume** $y \in C$
from $N(1)$ **have** *measure* N $\{y\} = (\int x.$ p x y n $\partial N)$
by (*rule stationary-distribution-iterate'*)
also have $\ldots \leq (\int x.$ p x y n $*$ *indicator* A x $+$ *indicator* $(C - A)$ x $\partial N)$
using *ae-C* ‹$A \subseteq C$›
by (*intro integral-mono-AE*)
(*auto elim*!: *eventually-mono*
intro!: *integral-add integral-indicator p-le-1 integrable-real-mult-indicator*
integrable-add
split: *split-indicator simp*: *integrable-p less-top*[*symmetric*] *top-unique*)
also have $\ldots = (\int x.$ p x y n $*$ *indicator* A x $\partial N) +$ *measure* N $(C - A)$
using *ae-C* ‹$A \subseteq C$›
apply (*subst Bochner-Integration.integral-add*)
apply (*auto elim*!: *eventually-mono*
intro!: *integral-add integral-indicator p-le-1 integrable-real-mult-indicator*
split: *split-indicator simp*: *integrable-p less-top*[*symmetric*] *top-unique*)
done
also have $\ldots \leq (\int x.$ p x y n $*$ *indicator* A x $\partial N) + e$
using e ‹$A \subseteq C$› **by** (*simp add*: *measure-pmf.finite-measure-Diff measure-C*)
finally have *measure* N $\{y\} \leq (\int x.$ p x y n $*$ *indicator* A x $\partial N) + e$ **.**
then have *emeasure* N $\{y\} \leq$ *ennreal* $(\int x.$ p x y n $*$ *indicator* A x $\partial N) + e$
by (*simp add*: *measure-pmf.emeasure-eq-measure ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)
also have $\ldots = (\int^+ x.$ *ennreal* $(p$ x y $n)$ $*$ *indicator* A x $\partial N) + e$
by (*subst nn-integral-eq-integral*[*symmetric*])
(*auto intro*!: *measure-pmf.integrable-const-bound*[**where** B=*1*]
simp: *abs-mult p-le-1 mult-le-one ennreal-indicator ennreal-mult*)
finally have *emeasure* N $\{y\} \leq (\int^+ x.$ *ennreal* $(p$ x y $n)$ $*$ *indicator* A x $\partial N)$ $+$ e **. }**
note *v-le* = *this*

{ fix y **and** z :: *real* **assume** y: $y \in C$ **and** z: $0 < z$ $z < 1$
have *summable-int-p*: *summable* $(\lambda n.$ $(\int$ $x.$ p x y n $*$ *indicator* A x $\partial N) *$ $(1$

$- z) * z \hat{} n)$
using ‹$y \in C$› z ‹$A \subseteq C$›
by (*auto intro*!: *summable-comparison-test*[*OF* - *summable-mult*[*OF summable-geometric*[*of* z], *of 1*]] *exI*[*of* - *0*] *mult-le-one*
measure-pmf.integral-le-const integrable-real-mult-indicator integrable-p AE-I2 p-le-1
simp: *abs-mult integral-nonneg-AE*)

from y z **have** *sums-y*: $(\lambda n.\ measure\ N\ \{y\} * (1 - z) * z \hat{} n)$ *sums measure N* $\{y\}$
using *sums-mult*[*OF geometric-sums*[*of* z], *of measure N* $\{y\} * (1 - z)$] **by** *simp*
then have *emeasure N* $\{y\} = ennreal\ (\sum n.\ (measure\ N\ \{y\} * (1 - z)) * z \hat{} n)$
by (*auto simp add*: *sums-unique*[*symmetric*] *measure-pmf.emeasure-eq-measure*)
also have $\ldots = (\sum n.\ emeasure\ N\ \{y\} * (1 - z) * z \hat{} n)$
using z *summable-mult*[*OF summable-geometric*[*of* z], *of measure-pmf.prob N* $\{y\} * (1 - z)$]
by (*subst suminf-ennreal*[*symmetric*])
(*auto simp*: *measure-pmf.emeasure-eq-measure ennreal-mult*[*symmetric*] *ennreal-suminf-neq-top*)
also have $\ldots \leq (\sum n.\ ((\int^+ x.\ ennreal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) + e) * (1 - z) * z \hat{} n)$
using ‹$y \in C$› z ‹$A \subseteq C$›
by (*intro suminf-le mult-right-mono v-le allI*)
(*auto simp*: *measure-pmf.emeasure-eq-measure*)
also have $\ldots = (\sum n.\ (\int^+ x.\ ennreal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * (1 - z) * z \hat{} n) + e$
using ‹$0 < e$› z *sums-mult*[*OF geometric-sums*[*of* z], *of* $e * (1 - z)$] ‹$0 < z$› ‹$z < 1$›
by (*simp add*: *distrib-right suminf-add*[*symmetric*] *ennreal-suminf-cmult*[*symmetric*] *ennreal-mult*[*symmetric*] *suminf-ennreal-eq sums-unique*[*symmetric*] *del*: *ennreal-suminf-cmult*)
also have $\ldots = (\sum n.\ ennreal\ (1 - z) * ((\int^+ x.\ ennreal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * z \hat{} n)) + e$
by (*simp add*: *ac-simps*)
also have $\ldots = ennreal\ (1 - z) * (\sum n.\ ((\int^+ x.\ ennreal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * z \hat{} n)) + e$
using z **by** (*subst ennreal-suminf-cmult*) *simp-all*
also have $(\sum n.\ ((\int^+ x.\ ennreal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * z \hat{} n)) = (\sum n.\ (\int^+ x.\ ennreal\ (p\ x\ y\ n * z \hat{} n) * indicator\ A\ x\ \partial N))$
using z **by** (*simp add*: *ac-simps nn-integral-cmult*[*symmetric*] *ennreal-mult*)
also have $\ldots = (\int^+ x.\ ennreal\ (gf\text{-}G\ x\ y\ z) * indicator\ A\ x\ \partial N)$
using z
apply (*subst nn-integral-suminf*[*symmetric*])
apply (*auto simp add*: *gf-G-def simp del*: *suminf-ennreal*
intro!: *ennreal-mult-right-cong suminf-ennreal2 nn-integral-cong*)
apply (*intro summable-comparison-test*[*OF* - *summable-mult*[*OF summable-geometric*[*of* z], *of 1*]] *impI*)

apply (*simp-all add*: *abs-mult p-le-1 mult-le-one power-le-one split*: *split-indicator*)
done
also have ... = ($\int^+ x$. *ennreal* (*gf-F x y z* * *gf-G y y z*) * *indicator A x* ∂N)
using *z* **by** (*intro nn-integral-cong*) (*simp add*: *gf-G-eq-gf-F*[*symmetric*])
also have ... = *ennreal* (*gf-G y y z*) * ($\int^+ x$. *ennreal* (*gf-F x y z*) * *indicator A x* ∂N)
using *z* **by** (*subst nn-integral-cmult*[*symmetric*]) (*simp-all add*: *gf-G-nonneg gf-F-nonneg ac-simps ennreal-mult*)
also have ... = *ennreal* (*1* / (*1* − *gf-U y y z*)) * ($\int^+ x$. *ennreal* (*gf-F x y z*) * *indicator A x* ∂N)
using *z* ‹*y* ∈ *C*› **by** (*subst gf-G-eq-gf-U*) (*auto intro*!: *convergence-G-less-1*)
finally have *emeasure N* {*y*} ≤ *ennreal* ((*1* − *z*) / (*1* − *gf-U y y z*)) * ($\int^+ x$. *gf-F x y z* * *indicator A x* ∂N) + *e*
using *z*
by (*subst* (*asm*) *mult.assoc*[*symmetric*])
(*simp add*: *ennreal-indicator*[*symmetric*] *ennreal-mult'*[*symmetric*] *gf-F-nonneg*)
then have *measure N* {*y*} ≤ (*1* − *z*) / (*1* − *gf-U y y z*) * ($\int x$. *gf-F x y z* * *indicator A x* ∂N) + *e*
using *z*
by (*subst* (*asm*) *nn-integral-eq-integral*[*OF measure-pmf.integrable-const-bound*[**where** *B=1*]])
(*auto simp*: *gf-F-nonneg gf-U-le-1 gf-F-le-1 measure-pmf.emeasure-eq-measure mult-le-one*
ennreal-mult''[*symmetric*] *ennreal-plus*[*symmetric*]
simp del: *ennreal-plus*) **}**
then have ∃ *A* ⊆ *C*. *finite A* ∧ (∀ *y*∈*C*. ∀ *z*. *0* < *z* ⟶ *z* < *1* ⟶ *measure N* {*y*} ≤ (*1* − *z*) / (*1* − *gf-U y y z*) * ($\int x$. *gf-F x y z* * *indicator A x* ∂N) + *e*)
using ‹*A* ⊆ *C*› ‹*finite A*› **by** *auto* **}**
note *eps* = *this*

{ fix *y A* **assume** *y* ∈ *C finite A A* ⊆ *C*
then have ((λ*z*. $\int x$. *gf-F x y z* * *indicator A x* ∂N) ⟶ $\int x$. *F x y* * *indicator A x* ∂N) (*at-left 1*)
by (*subst* (*1 2*) *integral-measure-pmf*[*of A*]) (*auto intro*!: *tendsto-intros gf-F simp*: *indicator-eq-0-iff*) **}**
note *int-gf-F* = *this*

have *all-recurrent*: ∀ *y*∈*C*. *recurrent y*
proof (*rule ccontr*)
assume ¬ (∀ *y*∈*C*. *recurrent y*)
then obtain *x* **where** *x* ∈ *C* ¬ *recurrent x* **by** *auto*
then have *transient*: ⋀*x*. *x* ∈ *C* ⟹ ¬ *recurrent x*
using *C* **by** (*auto simp*: *essential-class-def recurrent-iffI-communicating*[*symmetric*] *elim*!: *quotientE*)

{ fix *y* **assume** *y* ∈ *C*
with *transient* **have** *U y y* < *1*
by (*metis recurrent-iff-U-eq-1 U-cases*)
have *measure N* {*y*} ≤ *0*

```
    proof (rule dense-ge)
      fix e :: real assume 0 < e
      from eps[OF this] ‹y ∈ C› obtain A where
        A: finite A A ⊆ C and
        le: ⋀z. 0 < z ⟹ z < 1 ⟹ measure N {y} ≤ (1 − z) / (1 − gf-U y y
z) * (∫ x. gf-F x y z * indicator A x ∂N) + e
        by auto
      have ((λz. (1 − z) / (1 − gf-U y y z) * (∫ x. gf-F x y z * indicator A x
∂N) + e) ⟶
        (1 − 1) / (1 − U y y) * (∫ x. F x y * indicator A x ∂N) + e) (at-left
(1::real))
        using A ‹U y y < 1› ‹y ∈ C› by (intro tendsto-intros gf-U int-gf-F) auto
      then have 1: ((λz. (1 − z) / (1 − gf-U y y z) * (∫ x. gf-F x y z * indicator
A x ∂N) + e) ⟶ e) (at-left (1::real))
        by simp
      with le show measure N {y} ≤ e
        by (intro tendsto-le[OF trivial-limit-at-left-real - tendsto-const])
          (auto simp: eventually-at-left-1)
    qed
    then have measure N {y} = 0
      by (intro antisym measure-nonneg) }
  then have emeasure N C = 0
  by (subst emeasure-countable-singleton) (auto simp: measure-pmf.emeasure-eq-measure
nn-integral-0-iff-AE ae-C C)
  then show False
    using ‹measure N C = 1› by (simp add: measure-pmf.emeasure-eq-measure)
qed
then have ⋀x. x ∈ C ⟹ U x x = 1
  by (metis recurrent-iff-U-eq-1)

{ fix y assume y ∈ C
  then have U y y = 1 recurrent y
    using ‹y ∈ C ⟹ U y y = 1› all-recurrent by auto
  have measure N {y} ≤ enn2real (1 / U' y y)
  proof (rule field-le-epsilon)
    fix e :: real assume 0 < e
    from eps[OF ‹0 < e›] ‹y ∈ C› obtain A where
      A: finite A A ⊆ C and
      le: ⋀z. 0 < z ⟹ z < 1 ⟹ measure N {y} ≤ (1 − z) / (1 − gf-U y y z)
* (∫ x. gf-F x y z * indicator A x ∂N) + e
      by auto
    let ?L = at-left (1::real)
    have ((λz. (1 − z) / (1 − gf-U y y z) * (∫ x. gf-F x y z * indicator A x ∂N)
+ e) ⟶
      enn2real (1 / U' y y) * (∫ x. F x y * indicator A x ∂N) + e) ?L
    proof (intro tendsto-add tendsto-const tendsto-mult int-gf-F,
        rule lhopital-left[where f'=λx. − 1 and g'=λz. − gf-U' y y z])
      show ((−) 1 ⟶ 0) ?L ((λx. 1 − gf-U y y x) ⟶ 0) ?L
        using gf-U[of y y] by (auto intro!: tendsto-eq-intros simp: ‹U y y = 1›)
```

```
      show y ∈ C finite A A ⊆ C by fact+
      show eventually (λx. 1 − gf-U y y x ≠ 0) ?L
       using gf-G-eq-gf-U(2)[OF convergence-G-less-1, where 'z=real] by (auto
intro!: eventually-at-left-1)
      show ((λx. − 1 / − gf-U' y y x) ⟶ enn2real (1 / U' y y)) ?L
       using ‹recurrent y› by (rule inverse-gf-U'-tendsto)
      have eventually (λx. 0 < gf-U' y y x) ?L
       by (intro eventually-at-left-1 gf-U'-pos) (simp-all add: ‹U y y = 1›)
      then show eventually (λx. − gf-U' y y x ≠ 0) ?L
       by eventually-elim simp
      show eventually (λx. DERIV (λx. 1 − gf-U y y x) x :> − gf-U' y y x) ?L
       by (auto intro!: eventually-at-left-1 derivative-eq-intros DERIV-gf-U)
      show eventually (λx. DERIV ((−) 1) x :> − 1) ?L
       by (auto intro!: eventually-at-left-1 derivative-eq-intros)
    qed
    then have measure N {y} ≤ enn2real (1 / U' y y) * (∫ x. F x y * indicator
A x ∂N) + e
      by (rule tendsto-le[OF trivial-limit-at-left-real - tendsto-const]) (intro even-
tually-at-left-1 le)
     then have measure N {y} − e ≤ enn2real (1 / U' y y) * (∫ x. F x y *
indicator A x ∂N)
      by simp
    also have ... ≤ enn2real (1 / U' y y)
      using A
    by (intro mult-left-le measure-pmf.integral-le-const measure-pmf.integrable-const-bound[where
B=1])
       (auto simp: mult-le-one F-le-1 U'-def)
    finally show measure N {y} ≤ enn2real (1 / U' y y) + e
      by simp
  qed }
 note measure-y-le = this

 show pos: ∀ y∈C. pos-recurrent y
 proof (rule ccontr)
  assume ¬ (∀ y∈C. pos-recurrent y)
  then obtain x where x: x ∈ C ¬ pos-recurrent x by auto
  { fix y assume y ∈ C
   with x have ¬ pos-recurrent y
   using C by (auto simp: essential-class-def pos-recurrent-iffI-communicating[symmetric]
elim!: quotientE)
    with all-recurrent ‹y ∈ C› have enn2real (1 / U' y y) = 0
     by (simp add: pos-recurrent-def nn-integral-add)
    with measure-y-le[OF ‹y ∈ C›] have measure N {y} = 0
     by (auto intro!: antisym simp: pos-recurrent-def) }
  then have emeasure N C = 0
  by (subst emeasure-countable-singleton) (auto simp: C ae-C measure-pmf.emeasure-eq-measure
nn-integral-0-iff-AE)
  then show False
   using ‹measure N C = 1› by (simp add: measure-pmf.emeasure-eq-measure)
```

qed

{ fix A :: *'s set* **assume** [*simp*]: *countable A*
have *emeasure N A* = $(\int^+ x.$ *emeasure N* $\{x\}$ *∂count-space A*)
by (*intro emeasure-countable-singleton*) *auto*
also have ... ≤ $(\int^+ x.$ *emeasure* (*stat C*) $\{x\}$ *∂count-space A*)
proof (*intro nn-integral-mono*)
fix *y* **assume** *y* ∈ *space* (*count-space A*)
show *emeasure N* $\{y\}$ ≤ *emeasure* (*stat C*) $\{y\}$
proof *cases*
assume *y* ∈ *C*
with *pos* **have** *pos-recurrent y*
by *auto*
with *one-le-integral-t*[*of y*] **obtain** *r* **where** *r*: *U' y y* = *ennreal r 1* ≤ *U' y y* **and** [*simp*]: *0* ≤ *r*
by (*cases U' y y*) (*auto simp*: *pos-recurrent-def nn-integral-add*)

from *measure-y-le*[*OF* ‹*y* ∈ *C*›]
have *emeasure N* $\{y\}$ ≤ *ennreal* (*enn2real* (*1 / U' y y*))
by (*simp add*: *measure-pmf.emeasure-eq-measure*)
also have ... = *emeasure* (*stat C*) $\{y\}$
unfolding *stat-def* **using** ‹*y* ∈ *C*› *r*
by (*subst emeasure-point-measure-finite2*)
(*auto simp add*: *ennreal-1*[*symmetric*] *divide-ennreal inverse-ennreal inverse-eq-divide ennreal-mult*[*symmetric*]
simp del: *ennreal-1*)
finally show *emeasure N* $\{y\}$ ≤ *emeasure* (*stat C*) $\{y\}$
by *simp*
next
assume *y* ∉ *C*
with *ae-C* **have** *emeasure N* $\{y\}$ = *0*
by (*subst AE-iff-measurable*[*symmetric*, **where** *P*=λ*x*. *x* ≠ *y*]) (*auto elim*!: *eventually-mono*)
moreover have *emeasure* (*stat C*) $\{y\}$ = *0*
using *emeasure-stat-not-C*[*OF* ‹*y* ∉ *C*›] **.**
ultimately show *?thesis* **by** *simp*
qed
qed
also have ... = *emeasure* (*stat C*) *A*
by (*intro emeasure-countable-singleton*[*symmetric*]) *auto*
finally have *emeasure N A* ≤ *emeasure* (*stat C*) *A* **. }**
note *N-le-C* = *this*

from *stat-subprob*[*OF C*(*1*) ‹*countable C*› *pos*] *N-le-C*[*OF* ‹*countable C*›] ‹*measure N C* = *1*›
have *stat-C-eq-1*: *emeasure* (*stat C*) *C* = *1*
by (*auto simp add*: *measure-pmf.emeasure-eq-measure one-ennreal-def*)
moreover have *emeasure* (*stat C*) (*UNIV* − *C*) = *0*
by (*subst AE-iff-measurable*[*symmetric*, **where** *P*=λ*x*. *x* ∈ *C*])

```
    (auto simp: stat-def AE-point-measure sets-point-measure space-point-measure
          split: split-indicator cong del: AE-cong)
  ultimately have emeasure (stat C) (space (stat C)) = 1
    using plus-emeasure[of C stat C UNIV − C] by (simp add: Un-absorb1)
  interpret stat: prob-space stat C
    by standard fact

  show measure-pmf N = stat C
  proof (rule measure-eqI-countable-AE)
    show sets N = UNIV sets (stat C) = UNIV
      by auto
    show countable C AE x in N. x ∈ C and ae-stat: AE x in stat C. x ∈ C
    using C ae-C stat-C-eq-1 by (auto intro!: stat.AE-prob-1 simp: stat.emeasure-eq-measure)

    { assume ∃x. emeasure N {x} ≠ emeasure (stat C) {x}
      then obtain x where [simp]: emeasure N {x} ≠ emeasure (stat C) {x} by
auto
      with N-le-C[of {x}] have x: emeasure N {x} < emeasure (stat C) {x}
        by (auto simp: less-le)
      have 1 = emeasure N {x} + emeasure N (C − {x})
        using ae-C
        by (subst plus-emeasure) (auto intro!: measure-pmf.emeasure-eq-1-AE)
      also have ... < emeasure (stat C) {x} + emeasure (stat C) (C − {x})
        using x N-le-C[of C − {x}] C ae-C
        by (simp add: stat.emeasure-eq-measure measure-pmf.emeasure-eq-measure
                      ennreal-plus[symmetric] ennreal-less-iff
              del: ennreal-plus)
      also have ... = 1
        using ae-stat by (subst plus-emeasure) (auto intro!: stat.emeasure-eq-1-AE)
      finally have False by simp }
    then show ⋀x. emeasure N {x} = emeasure (stat C) {x} by auto
  qed
qed

lemma measure-point-measure-singleton:
  x ∈ A ⟹ measure (point-measure A X) {x} = enn2real (X x)
  unfolding measure-def by (subst emeasure-point-measure-finite2) auto

lemma stationary-distribution-imp-int-t:
  assumes C: essential-class C countable C stationary-distribution N N ⊆ C
  assumes x: x ∈ C shows U' x x = 1 / ennreal (pmf N x)
proof −
  from stationary-distributionD[OF C]
  have measure-pmf N = stat C and ∗: ∀x∈C. pos-recurrent x by auto
  show ?thesis
    unfolding ‹measure-pmf N = stat C› pmf.rep-eq stat-def
    using ∗[THEN bspec, OF x] x
    apply (simp add: measure-point-measure-singleton)
    apply (cases U' x x)
```

```
    subgoal for r
      by (cases r = 0)
         (simp-all add: divide-ennreal-def inverse-ennreal)
    apply simp
    done
qed

definition period-set x = {i. 0 < i ∧ 0 < p x x i }
definition period C = (SOME d. ∀ x∈C. d = Gcd (period-set x))

lemma Gcd-period-set-invariant:
  assumes c: (x, y) ∈ communicating
  shows Gcd (period-set x) = Gcd (period-set y)
proof −
  { fix x y n assume c: (x, y) ∈ communicating x ≠ y and n: n ∈ period-set x
    from c obtain l k where 0 < p x y l 0 < p y x k
      by (auto simp: communicating-def dest!: accD-pos)
    moreover with ‹x ≠ y› have l ≠ 0 ∧ k ≠ 0
      by (intro notI conjI) (auto simp: p-0)
    ultimately have pos: 0 < l 0 < k and l: 0 < p x y l and k: 0 < p y x k
      by auto

    from mult-pos-pos[OF k l] prob-reachable-le[of k k + l y x y] c
    have k-l: 0 < p y y (k + l)
      by simp
    then have Gcd (period-set y) dvd k + l
      using pos by (auto intro!: Gcd-dvd-nat simp: period-set-def)
    moreover
    from n have 0 < p x x n 0 < n by (auto simp: period-set-def)
    from mult-pos-pos[OF k this(1)] prob-reachable-le[of k k + n y x x] c
    have 0 < p y x (k + n)
      by simp
    from mult-pos-pos[OF this(1) l] prob-reachable-le[of k + n (k + n) + l y x y] c
    have 0 < p y y (k + n + l)
      by simp
    then have Gcd (period-set y) dvd (k + l) + n
      using pos by (auto intro!: Gcd-dvd-nat simp: period-set-def ac-simps)
    ultimately have Gcd (period-set y) dvd n
      by (metis dvd-add-left-iff add.commute) }
  note this[of x y] this[of y x] c
  moreover have (y, x) ∈ communicating
    using c by (simp add: communicating-def)
  ultimately show ?thesis
    by (auto intro: dvd-antisym Gcd-greatest Gcd-dvd)
qed

lemma period-eq:
  assumes C ∈ UNIV // communicating x ∈ C
  shows period C = Gcd (period-set x)
```

unfolding *period-def*
using *assms*
by (*rule-tac someI2*[**where** *a*=*Gcd* (*period-set x*)])
(*auto intro*!: *Gcd-period-set-invariant irreducibleD*)

definition *aperiodic C* ⟷ *C* ∈ *UNIV* // *communicating* ∧ *period C* = *1*

definition *not-ephemeral C* ⟷ *C* ∈ *UNIV* // *communicating* ∧ ¬ (∃ *x*. *C* = {*x*} ∧ *p x x 1* = *0*)

lemma *not-ephemeralD*:
assumes *C*: *not-ephemeral C x* ∈ *C*
shows ∃ *n*>*0*. *0* < *p x x n*
proof *cases*
assume ∃ *x*. *C* = {*x*}
with ‹*x* ∈ *C*› **have** *C* = {*x*} **by** *auto*
with *C p-nonneg*[*of x x 1*] **have** *0* < *p x x 1*
by (*auto simp*: *not-ephemeral-def less-le*)
with ‹*C* = {*x*}› **show** *?thesis* **by** *auto*
next
from *C* **have** *irr*: *C* ∈ *UNIV* // *communicating*
by (*auto simp*: *not-ephemeral-def*)
assume ¬(∃ *x*. *C* = {*x*})
then have ∀ *x*. *C* ≠ {*x*} **by** *auto*
with ‹*x* ∈ *C*› **obtain** *y* **where** *y* ∈ *C x* ≠ *y*
by *blast*
with *irreducibleD*[*OF irr*, *of x y*] *C* ‹*x* ∈ *C*› **have** *c*: (*x*, *y*) ∈ *communicating* **by** *auto*
with *accD-pos*[*of x y*] *accD-pos*[*of y x*]
obtain *k l* **where** *pos*: *0* < *p x y k 0* < *p y x l*
by (*auto simp*: *communicating-def*)
with ‹*x* ≠ *y*› **have** *l* ≠ *0*
by (*intro notI*) (*auto simp*: *p-0*)
have *0* < *p x y k* ∗ *p y x* (*k* + *l* − *k*)
using *pos* **by** *auto*
also have *p x y k* ∗ *p y x* (*k* + *l* − *k*) ≤ *p x x* (*k* + *l*)
using *prob-reachable-le*[*of k k* + *l x y x*] *c* **by** *auto*
finally show *?thesis*
using ‹*l* ≠ *0*› ‹*x* ∈ *C*› **by** (*auto intro*!: *exI*[*of* - *k* + *l*])
qed

lemma *not-ephemeralD-pos-period*:
assumes *C*: *not-ephemeral C*
shows *0* < *period C*
proof −
from *C not-empty-irreducible*[*of C*] **obtain** *x* **where** *x* ∈ *C*
by (*auto simp*: *not-ephemeral-def*)
from *not-ephemeralD*[*OF C this*]
obtain *n* **where** *n*: *0* < *p x x n 0* < *n* **by** *auto*

```
  have C': C ∈ UNIV // communicating
    using C by (auto simp: not-ephemeral-def)

  have period C ≠ 0
    unfolding period-eq [OF C' ‹x ∈ C›]
    using n by (auto simp: period-set-def)
  then show ?thesis by auto
qed


lemma period-posD:
  assumes C: C ∈ UNIV // communicating and 0 < period C x ∈ C
  shows ∃ n>0. 0 < p x x n
proof −
  from ‹0 < period C› have period C ≠ 0
    by auto
  then show ?thesis
    unfolding period-eq [OF C ‹x ∈ C›]
    unfolding period-set-def by auto
qed

lemma not-ephemeralD-pos-period':
  assumes C: C ∈ UNIV // communicating
  shows not-ephemeral C ⟷ 0 < period C
proof (auto dest!: not-ephemeralD-pos-period intro: C)
  from C not-empty-irreducible[of C] obtain x where x ∈ C
    by (auto simp: not-ephemeral-def)

  assume 0 < period C
  then show not-ephemeral C
    apply (auto simp: not-ephemeral-def C)
oops — should be easy to finish


lemma eventually-periodic:
  assumes C: C ∈ UNIV // communicating 0 < period C x ∈ C
  shows eventually (λm. 0 < p x x (m * period C)) sequentially
proof −
  from period-posD[OF assms] obtain n where n: 0 < p x x n 0 < n by auto
  have C': C ∈ UNIV // communicating
    using C by auto

  have period C ≠ 0
    unfolding period-eq [OF C' ‹x ∈ C›]
    using n by (auto simp: period-set-def)
  have eventually (λm. m * Gcd (period-set x) ∈ (period-set x)) sequentially
  proof (rule eventually-mult-Gcd)
    show n > 0 n ∈ period-set x
      using n by (auto simp add: period-set-def)
```

fix $k\ l$ **assume** $k \in$ *period-set* $x\ l \in$ *period-set* x
then have $0 < p\ x\ x\ k * p\ x\ x\ l\ 0 < l\ 0 < k$
by (*auto simp*: *period-set-def*)
moreover have $p\ x\ x\ k * p\ x\ x\ l \leq p\ x\ x\ (k + l)$
using *prob-reachable-le*[*of* $k\ k + l\ x\ x\ x$] ‹$x \in C$›
by *auto*
ultimately show $k + l \in$ *period-set* x
using ‹$0 < l$› **by** (*auto simp*: *period-set-def*)
qed
with *eventually-ge-at-top*[*of 1*] **show** *eventually* ($\lambda m.\ 0 < p\ x\ x\ (m * period\ C)$) *sequentially*
by *eventually-elim*
(*insert* ‹*period* $C \neq 0$› *period-eq*[*OF* C' ‹$x \in C$›, *symmetric*], *auto simp*: *period-set-def*)
qed

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 *safe*
fix x **assume** $x \in C$ *aperiodic* C
with *eventually-periodic*[*of* $C\ x$]
show *eventually* ($\lambda m.\ 0 < p\ x\ x\ m$) *sequentially*
by (*auto simp add*: *aperiodic-def*)
next
assume $\forall x{\in}C.$ *eventually* ($\lambda m.\ 0 < p\ x\ x\ m$) *sequentially* **and** C: $C \in UNIV\ //\ communicating$
moreover from *not-empty-irreducible*[*OF* C] **obtain** x **where** $x \in C$ **by** *auto*
ultimately obtain N **where** $\bigwedge M.\ \ M{\geq}N \Longrightarrow 0 < p\ x\ x\ M$
by (*auto simp*: *eventually-sequentially*)
then have $\{N <..\} \subseteq$ *period-set* x
by (*auto simp*: *period-set-def*)
from C **show** *aperiodic* C
unfolding *period-eq* [*OF* C ‹$x \in C$›] *aperiodic-def*
proof
show $Gcd\ (period\text{-}set\ x) = 1$
proof (*rule Gcd-eqI*)
from *one-dvd* **show** *1 dvd* q **for** q :: *nat* **.**
fix m
assume $\bigwedge q.\ q \in$ *period-set* $x \Longrightarrow m\ dvd\ q$
moreover from ‹$\{N <..\} \subseteq$ *period-set* x›
have $\{Suc\ N,\ Suc\ (Suc\ N)\} \subseteq$ *period-set* x
by *auto*
ultimately have $m\ dvd\ Suc\ (Suc\ N)$ **and** $m\ dvd\ Suc\ N$
by *auto*
then have $m\ dvd\ Suc\ (Suc\ N) - Suc\ N$
by (*rule dvd-diff-nat*)
then show *is-unit* m

```
      by simp
    qed simp
  qed
qed (simp add: aperiodic-def)

lemma stationary-distributionD-emeasure:
  assumes N: stationary-distribution N
  shows emeasure N A = (∫⁺s. emeasure (K s) A ∂N)
proof −
  have prob-space (measure-pmf N)
    by intro-locales
  then interpret subprob-space measure-pmf N
    by (rule prob-space-imp-subprob-space)
  show ?thesis
    unfolding measure-pmf.emeasure-eq-measure
    apply (subst N[unfolded stationary-distribution-def])
    apply (simp add: measure-pmf-bind)
    apply (subst measure-pmf.measure-bind[where N=count-space UNIV])
    apply (rule measurable-compose[OF - measurable-measure-pmf])
   apply (auto intro!: nn-integral-eq-integral[symmetric] measure-pmf.integrable-const-bound[where
B=1])
    done
qed

lemma communicatingD1:
  C ∈ UNIV // communicating ⟹ (a, b) ∈ communicating ⟹ a ∈ C ⟹ b ∈
C
  by (auto elim!: quotientE) (auto simp add: communicating-def)

lemma communicatingD2:
  C ∈ UNIV // communicating ⟹ (a, b) ∈ communicating ⟹ b ∈ C ⟹ a ∈
C
  by (auto elim!: quotientE) (auto simp add: communicating-def)

lemma acc-iff: (x, y) ∈ acc ⟷ (∃ n. 0 < p x y n)
  by (blast intro: accD-pos accI-pos)

lemma communicating-iff: (x, y) ∈ communicating ⟷ (∃ n. 0 < p x y n) ∧ (∃ n.
0 < p y x n)
  by (auto simp add: acc-iff communicating-def)

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
  unfolding p-def K1.p-def K2.p-def
```

by (*subst prod-eq-prob-T*)
(*auto intro*!: *arg-cong2*[**where** *f*=*measure*] *split*: *nat.splits simp*: *Stream-snth*)

lemma *P-accD*:
assumes ((*x1*, *x2*), (*y1*, *y2*)) ∈ *acc***shows** (*x1*, *y1*) ∈ *K1.acc* (*x2*, *y2*) ∈ *K2.acc*
using *assms* **by** (*auto simp*: *acc-iff K1.acc-iff K2.acc-iff p-eq-p1-p2 zero-less-mult-iff not-le*[*of 0*, *symmetric*]
cong: *conj-cong*)

lemma *aperiodicI-pair*:
assumes *C1*: *K1.aperiodic C1* **and** *C2*: *K2.aperiodic C2*
shows *aperiodic* (*C1* × *C2*)
unfolding *aperiodic-eventually-recurrent*
proof *safe*
from *C1*[*unfolded K1.aperiodic-eventually-recurrent*] *C2*[*unfolded K2.aperiodic-eventually-recurrent*]
have *C1*: *C1* ∈ *UNIV* // *K1.communicating* **and** *C2*: *C2* ∈ *UNIV* // *K2.communicating*
and
ev: ⋀*x*. *x* ∈ *C1* ⟹ *eventually* (λ*m*. *0* < *K1.p x x m*) *sequentially* ⋀*x*. *x* ∈ *C2*
⟹ *eventually* (λ*m*. *0* < *K2.p x x m*) *sequentially*
by *auto*
{ fix *x1 x2* **assume** *x*: *x1* ∈ *C1 x2* ∈ *C2*
from *ev*(*1*)[*OF x*(*1*)] *ev*(*2*)[*OF x*(*2*)]
show *eventually* (λ*m*. *0* < *p* (*x1*, *x2*) (*x1*, *x2*) *m*) *sequentially*
by *eventually-elim* (*simp add*: *p-eq-p1-p2 x*) **}**

{ fix *x1 x2 y1 y2*
assume *acc*: (*x1*, *y1*) ∈ *K1.acc* (*x2*, *y2*) ∈ *K2.acc x1* ∈ *C1 y1* ∈ *C1 x2* ∈ *C2*
y2 ∈ *C2*
then obtain *k l* **where** *0* < *K1.p x1 y1 l 0* < *K2.p x2 y2 k*
by (*auto dest*!: *K1.accD-pos K2.accD-pos*)
with *acc ev*(*1*)[*of y1*] *ev*(*2*)[*of y2*]
have *eventually* (λ*m*. *0* < *K1.p x1 y1 l* ∗ *K1.p y1 y1 m* ∧ *0* < *K2.p x2 y2 k*
∗ *K2.p y2 y2 m*) *sequentially*
by (*auto elim*: *eventually-elim2*)
then have *eventually* (λ*m*. *0* < *K1.p x1 y1* (*m* + *l*) ∧ *0* < *K2.p x2 y2* (*m* +
k)) *sequentially*
proof *eventually-elim*
fix *m* **assume** *0* < *K1.p x1 y1 l* ∗ *K1.p y1 y1 m* ∧ *0* < *K2.p x2 y2 k* ∗ *K2.p*
y2 y2 m
with *acc*
K1.prob-reachable-le[*of l l* + *m x1 y1 y1*]
K2.prob-reachable-le[*of k k* + *m x2 y2 y2*]
show *0* < *K1.p x1 y1* (*m* + *l*) ∧ *0* < *K2.p x2 y2* (*m* + *k*)
by (*auto simp add*: *ac-simps*)
qed
then have *eventually* (λ*m*. *0* < *K1.p x1 y1 m* ∧ *0* < *K2.p x2 y2 m*) *sequentially*
unfolding *eventually-conj-iff* **by** (*subst* (*asm*) (*1 2*) *eventually-sequentially-seg*)
(*auto elim*: *eventually-elim2*)
then obtain *N* **where** *0* < *K1.p x1 y1 N 0* < *K2.p x2 y2 N*

```
    by (auto simp: eventually-sequentially)
  with acc have 0 < p (x1, x2) (y1, y2) N
    by (auto simp add: p-eq-p1-p2)
  with acc have ((x1, x2), (y1, y2)) ∈ acc
    by (auto intro!: accI-pos) }
 note 1 = this

 { fix x1 x2 y1 y2 assume acc:((x1, x2), (y1, y2)) ∈ acc
   moreover from acc obtain k where 0 < p (x1, x2) (y1, y2) k by (auto
dest!: accD-pos)
   ultimately have (x1, y1) ∈ K1.acc ∧ (x2, y2) ∈ K2.acc
    by (subst (asm) p-eq-p1-p2)
       (auto intro!: K1.accI-pos K2.accI-pos simp: zero-less-mult-iff not-le[of 0,
symmetric]) }
 note 2 = this

 from K1.not-empty-irreducible[OF C1] K2.not-empty-irreducible[OF C2]
 obtain x1 x2 where xC: x1 ∈ C1 x2 ∈ C2 by auto
 show C1 × C2 ∈ UNIV // communicating
  apply (simp add: quotient-def Image-def)
  apply (safe intro!: exI[of - x1] exI[of - x2])
 proof −
  fix y1 y2 assume yC: y1 ∈ C1 y2 ∈ C2
  from K1.irreducibleD[OF C1 ‹x1 ∈ C1› ‹y1 ∈ C1›] K2.irreducibleD[OF C2
‹x2 ∈ C2› ‹y2 ∈ C2›]
  show ((x1, x2), (y1, y2)) ∈ communicating
   using 1[of x1 y1 x2 y2] 1[of y1 x1 y2 x2] xC yC
  by (auto simp: communicating-def K1.communicating-def K2.communicating-def)
 next
  fix y1 y2 assume ((x1, x2), (y1, y2)) ∈ communicating
  with 2[of x1 x2 y1 y2] 2[of y1 y2 x1 x2]
  have (x1, y1) ∈ K1.communicating (x2, y2) ∈ K2.communicating
  by (auto simp: communicating-def K1.communicating-def K2.communicating-def)
  with xC show y1 ∈ C1 y2 ∈ C2
   using K1.communicatingD1[OF C1] K2.communicatingD1[OF C2] by auto
 qed
qed

lemma stationary-distributionI-pair:
 assumes N1: K1.stationary-distribution N1
 assumes N2: K2.stationary-distribution N2
 shows stationary-distribution (pair-pmf N1 N2)
 unfolding stationary-distribution-def
 unfolding Kp-def pair-pmf-def
 apply (subst N1[unfolded K1.stationary-distribution-def])
 apply (subst N2[unfolded K2.stationary-distribution-def])
 apply (simp add: bind-assoc-pmf bind-return-pmf)
 apply (subst bind-commute-pmf[of N2])
 apply simp
```

done

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* $\subseteq$ *C*
assumes [*simp*]: *y* $\in$ *C*
shows ($\lambda n.$ $\int$ *x*. |*p y x n* − *pmf N x*| ∂*count-space C*) $\longrightarrow$ *0*
(**is** *?L* $\longrightarrow$ *0*)
proof −
from ‹*essential-class C*› **have** *C-comm*: *C* $\in$ *UNIV* // *communicating*
by (*simp add*: *essential-class-def*)

define *K′* **where** *K′* = (λ*Some x* $\Rightarrow$ *map-pmf Some* (*K x*) | *None* $\Rightarrow$ *map-pmf Some N*)

interpret *K2*: *MC-syntax K′* **.**
interpret *KN*: *MC-pair K K′* **.**

from *stationary-distributionD*[*OF C*(*2*,*3*) *N*]
have *pos*: $\bigwedge$*x*. *x* $\in$ *C* $\Longrightarrow$ *pos-recurrent x* **and** *measure-pmf N* = *stat C* **by** *auto*

have *pos*: $\bigwedge$*x*. *x* $\in$ *C* $\Longrightarrow$ *0* < *emeasure N* {*x*}
using *pos* **unfolding** *stat-def* ‹*measure-pmf N* = *stat C*›
by (*subst emeasure-point-measure-finite2*)
(*auto simp*: *U′-def pos-recurrent-def nn-integral-add ennreal-zero-less-divide less-top*)
then have *rpos*: $\bigwedge$*x*. *x* $\in$ *C* $\Longrightarrow$ *0* < *pmf N x*
by (*simp add*: *measure-pmf.emeasure-eq-measure pmf.rep-eq*)

have *eq*: $\bigwedge$*x y*. (*if x* = *y then 1 else 0*) = *indicator* {*y*} *x* **by** *auto*

have *intK*: $\bigwedge$*f x*. ($\int$ *x*. (*f x* :: *real*) ∂*K′* (*Some x*)) = ($\int$ *x*. *f* (*Some x*) ∂*K x*)
by (*simp add*: *K′-def integral-distr map-pmf-rep-eq*)

{ fix *m* **and** *x y* :: *′s*
have *K2.p* (*Some x*) (*Some y*) *m* = *p x y m*
by (*induct m arbitrary*: *x*)
(*auto intro*!: *integral-cong simp add*: *K2.p-Suc′ p-Suc′ intK K2.p-0 p-0*) **}**
note *K-p-eq* = *this*

{ fix *n* **and** *x* :: *′s* **have** *K2.p* (*Some x*) *None n* = *0*
by (*induct n arbitrary*: *x*) (*auto simp*: *K2.p-Suc′ K2.p-0 intK cong*: *integral-cong*) **}**
note *K-S-None* = *this*

from *not-empty-irreducible*[*OF C-comm*] **obtain** *c0* **where** *c0*: $c0 \in C$ **by** *auto*

have *K2-acc*: $\bigwedge x\ y.\ (Some\ x,\ y) \in K2.acc \longleftrightarrow (\exists\, z.\ y = Some\ z \wedge (x,\ z) \in acc)$
apply (*auto simp*: *K2.acc-iff acc-iff K-p-eq*)
apply (*case-tac y*)
apply (*auto simp*: *K-p-eq K-S-None*)
done

have *K2-communicating*: $\bigwedge c\ x.\ c \in C \Longrightarrow (Some\ c,\ x) \in K2.communicating \longleftrightarrow (\exists\, c' \in C.\ x = Some\ c')$
proof *safe*
fix *x c* **assume** $c \in C$ $(Some\ c,\ x) \in K2.communicating$
then show $\exists\, c' \in C.\ x = Some\ c'$
by (*cases x*)
(*auto simp*: *communicating-iff K2.communicating-iff K-p-eq K-S-None intro*!: *irreducibleD2*[*OF C-comm* ‹$c \in C$›])
next
fix *c c' x* **assume** $c \in C$ $c' \in C$
with *irreducibleD*[*OF C-comm this*] **show** $(Some\ c,\ Some\ c') \in K2.communicating$
by (*auto simp*: *K2.communicating-iff communicating-iff K-p-eq*)
qed

have *Some* ‘ $C \in UNIV\ //\ K2.communicating$
by (*auto simp add*: *quotient-def Image-def c0 K2-communicating intro*!: *exI*[*of* - *Some c0*])
then have *K2.essential-class* (*Some* ‘ *C*)
by (*rule K2.essential-classI*)
(*auto simp*: *K2-acc essential-classD2*[*OF* ‹*essential-class C*›])

have *K2.aperiodic* (*Some* ‘ *C*)
unfolding *K2.aperiodic-eventually-recurrent*
proof *safe*
fix *x* **assume** $x \in C$ **then show** *eventually* $(\lambda m.\ 0 < K2.p\ (Some\ x)\ (Some\ x)\ m)$ *sequentially*
using ‹*aperiodic C*› **unfolding** *aperiodic-eventually-recurrent*
by (*auto elim*!: *eventually-mono simp*: *K-p-eq*)
qed *fact*
then have *aperiodic*: *KN.aperiodic* $(C \times Some\ `\ C)$
by (*rule KN.aperiodicI-pair*[*OF* ‹*aperiodic C*›])

have *KN-essential*: *KN.essential-class* $(C \times Some\ `\ C)$
proof (*rule KN.essential-classI*)
show $C \times Some\ `\ C \in UNIV\ //\ KN.communicating$
using *aperiodic* **by** (*simp add*: *KN.aperiodic-def*)
next
fix *x y* **assume** $x \in C \times Some\ `\ C$ $(x,\ y) \in KN.acc$
with *KN.P-accD*[*of fst x snd x fst y snd y*]
show $y \in C \times Some\ `\ C$

```
    by (cases x y rule: prod.exhaust[case-product prod.exhaust])
      (auto simp: K2-acc essential-classD2[OF ‹essential-class C›])
  qed

  { fix n and x y :: 's
    have measure N {y} = P(ω in K2.T None. (None ## ω) !! (Suc n) = Some
y)
      unfolding stationary-distribution-iterate'[OF N(1), of y n]
      apply (subst K2.p-def[symmetric])
      apply (subst K2.p-Suc')
      apply (subst K'-def)
      apply (simp add: map-pmf-rep-eq integral-distr K-p-eq)
      done
    then have measure N {y} = P(ω in K2.T None. ω !! n = Some y)
      by simp }
  note measure-y-eq = this

  define D where D = {x::'s × 's option. Some (fst x) = snd x}

  have [measurable]:
    ⋀P::('s × 's option ⇒ bool). P ∈ measurable (count-space UNIV) (count-space
UNIV)
    by simp

  { fix n and x :: 's
    have P(ω in KN.T (y, None). ∃ i<n. snd (ω !! n) = Some x ∧ ev-at (HLD D)
i ω) =
      (∑ i<n. P(ω in KN.T (y, None). snd (ω !! n) = Some x ∧ ev-at (HLD D) i
ω))
      by (subst KN.T.finite-measure-finite-Union[symmetric])
        (auto simp: disjoint-family-on-def intro!: arg-cong2[where f=measure] dest:
ev-at-unique)
    also have ... = (∑ i<n. P(ω in KN.T (y, None). fst (ω !! n) = x ∧ ev-at
(HLD D) i ω))
    proof (intro sum.cong refl)
      fix i assume i: i ∈ {..< n}
     show P(ω in KN.T (y, None). snd (ω !! n) = Some x ∧ ev-at (HLD D) i ω)
=
        P(ω in KN.T (y, None). fst (ω !! n) = x ∧ ev-at (HLD D) i ω)
        apply (subst (1 2) KN.prob-T-split[where n=Suc i])
        apply (simp-all add: ev-at-shift snth-Stream del: stake.simps KN.space-T)
        unfolding ev-at-shift snth-Stream
      proof (intro Bochner-Integration.integral-cong refl)
        fix ω :: ('s × 's option) stream let ?s = λω'. stake (Suc i) ω @- ω'
       show P(ω' in KN.T (ω !! i). snd (?s ω' !! n) = Some x ∧ ev-at (HLD D) i
ω) =
          P(ω' in KN.T (ω !! i). fst (?s ω' !! n) = x ∧ ev-at (HLD D) i ω)
        proof cases
          assume ev-at (HLD D) i ω
```

from *ev-at-imp-snth*[*OF this*]
have *eq*: *snd* (*ω* !! *i*) = *Some* (*fst* (*ω* !! *i*))
by (*simp add*: *D-def HLD-iff*)

have $\mathcal{P}(\omega'$ *in KN.T* (*ω* !! *i*). *fst* (*ω*′ !! (*n* − *Suc i*)) = *x*) =
$\mathcal{P}(\omega'$ *in T* (*fst* (*ω* !! *i*)). *ω*′ !! (*n* − *Suc i*) = *x*) ∗ $\mathcal{P}(\omega'$ *in K2.T* (*snd* (*ω* !! *i*)). *True*)
by (*subst KN.prod-eq-prob-T*) *simp-all*
also have ... = *p* (*fst* (*ω* !! *i*)) *x* (*Suc* (*n* − *Suc i*))
using *K2.T.prob-space* **by** (*simp add*: *p-def*)
also have ... = *K2.p* (*snd* (*ω* !! *i*)) (*Some x*) (*Suc* (*n* − *Suc i*))
by (*simp add*: *K-p-eq eq*)
also have ... = $\mathcal{P}(\omega'$ *in T* (*fst* (*ω* !! *i*)). *True*) ∗ $\mathcal{P}(\omega'$ *in K2.T* (*snd* (*ω* !! *i*)). *ω*′ !! (*n* − *Suc i*) = *Some x*)
using *T.prob-space* **by** (*simp add*: *K2.p-def*)
also have ... = $\mathcal{P}(\omega'$ *in KN.T* (*ω* !! *i*). *snd* (*ω*′ !! (*n* − *Suc i*)) = *Some x*)
by (*subst KN.prod-eq-prob-T*) *simp-all*
finally show *?thesis* **using** ‹*ev-at* (*HLD D*) *i ω*› *i*
by (*simp del*: *stake.simps*)
qed *simp*
qed
qed
also have ... = $\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). (∃ *i*<*n*. *fst* (*ω* !! *n*) = *x* ∧ *ev-at* (*HLD D*) *i ω*))
by (*subst KN.T.finite-measure-finite-Union*[*symmetric*])
(*auto simp add*: *disjoint-family-on-def dest*: *ev-at-unique*
intro!: *arg-cong2*[**where** *f*=*measure*])
finally have *eq*: $\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). (∃ *i*<*n*. *snd* (*ω* !! *n*) = *Some x* ∧ *ev-at* (*HLD D*) *i ω*)) =
$\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). (∃ *i*<*n*. *fst* (*ω* !! *n*) = *x* ∧ *ev-at* (*HLD D*) *i ω*)) **.**

have *p y x* (*Suc n*) − *measure N* {*x*} = $\mathcal{P}(\omega$ *in T y*. *ω* !! *n* = *x*) − $\mathcal{P}(\omega$ *in K2.T None*. *ω* !! *n* = *Some x*)
unfolding *p-def* **by** (*subst measure-y-eq*) *simp-all*
also have $\mathcal{P}(\omega$ *in T y*. *ω* !! *n* = *x*) = $\mathcal{P}(\omega$ *in T y*. *ω* !! *n* = *x*) ∗ $\mathcal{P}(\omega$ *in K2.T None*. *True*)
using *K2.T.prob-space* **by** *simp*
also have ... = $\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). *fst* (*ω* !! *n*) = *x*)
by (*subst KN.prod-eq-prob-T*) *auto*
also have ... = $\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). (∃ *i*<*n*. *fst* (*ω* !! *n*) = *x* ∧ *ev-at* (*HLD D*) *i ω*)) +
$\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). *fst* (*ω* !! *n*) = *x* ∧ ¬ (∃ *i*<*n*. *ev-at* (*HLD D*) *i ω*))
by (*subst KN.T.finite-measure-Union*[*symmetric*])
(*auto intro*!: *arg-cong2*[**where** *f*=*measure*])
also have $\mathcal{P}(\omega$ *in K2.T None*. *ω* !! *n* = *Some x*) = $\mathcal{P}(\omega$ *in T y*. *True*) ∗ $\mathcal{P}(\omega$ *in K2.T None*. *ω* !! *n* = *Some x*)
using *T.prob-space* **by** *simp*
also have ... = $\mathcal{P}(\omega$ *in KN.T* (*y*, *None*). *snd* (*ω* !! *n*) = *Some x*)

by (*subst KN.prod-eq-prob-T*) *auto*
also have ... = $\mathcal{P}(\omega$ *in KN.T* (y, *None*). ($\exists\, i{<}n$. *snd* (ω !! n) = *Some* $x \wedge$ *ev-at* (*HLD D*) $i\ \omega$)) +
$\mathcal{P}(\omega$ *in KN.T* (y, *None*). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$))
by (*subst KN.T.finite-measure-Union*[*symmetric*])
(*auto intro*!: *arg-cong2*[**where** f=*measure*])
finally have | $p\ y\ x$ (*Suc* n) − *measure* N $\{x\}$ | =
| $\mathcal{P}(\omega$ *in KN.T* (y, *None*). *fst* (ω !! n) = $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$))
−
$\mathcal{P}(\omega$ *in KN.T* (y, *None*). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)) |
unfolding *eq* **by** (*simp add*: *field-simps*)
also have ... ≤ | $\mathcal{P}(\omega$ *in KN.T* (y, *None*). *fst* (ω !! n) = $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)) | +
| $\mathcal{P}(\omega$ *in KN.T* (y, *None*). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)) |
by (*rule abs-triangle-ineq4*)
also have ... ≤ $\mathcal{P}(\omega$ *in KN.T* (y, *None*). *fst* (ω !! n) = $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)) +
$\mathcal{P}(\omega$ *in KN.T* (y, *None*). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$))
by *simp*
finally have | $p\ y\ x$ (*Suc* n) − *measure* N $\{x\}$ | ≤ ... **. }**
note *mono* = *this*

{ fix n :: *nat*
have ($\int^+ x$. | $p\ y\ x$ (*Suc* n) − *measure* N $\{x\}$ | ∂*count-space* C) ≤
($\int^+ x$. *ennreal* ($\mathcal{P}(\omega$ *in KN.T* (y, *None*). *fst* (ω !! n) = $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$))) +
ennreal ($\mathcal{P}(\omega$ *in KN.T* (y, *None*). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$))) ∂*count-space* C)
using *mono* **by** (*intro nn-integral-mono*) (*simp add*: *ennreal-plus*[*symmetric*] *del*: *ennreal-plus*)
also have ... = ($\int^+ x$. $\mathcal{P}(\omega$ *in KN.T* (y, *None*). *fst* (ω !! n) = $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)) ∂*count-space* C) +
($\int^+ x$. $\mathcal{P}(\omega$ *in KN.T* (y, *None*). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)) ∂*count-space* C)
by (*subst nn-integral-add*) *auto*
also have ... = *emeasure* (*KN.T* (y, *None*)) ($\bigcup x{\in}C$. {$\omega{\in}$*space* (*KN.T* (y, *None*)). *fst* (ω !! n) = $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)}) +
emeasure (*KN.T* (y, *None*)) ($\bigcup x{\in}C$. {$\omega{\in}$*space* (*KN.T* (y, *None*)). *snd* (ω !! n) = *Some* $x \wedge \neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)})
by (*subst* (*1 2*) *emeasure-UN-countable*)
(*auto simp add*: *disjoint-family-on-def KN.T.emeasure-eq-measure* C)
also have ... ≤ *ennreal* ($\mathcal{P}(\omega$ *in KN.T* (y, *None*). $\neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$))) + *ennreal* ($\mathcal{P}(\omega$ *in KN.T* (y, *None*). $\neg$ ($\exists\, i{<}n$. *ev-at* (*HLD D*) $i\ \omega$)))
unfolding *KN.T.emeasure-eq-measure*
by (*intro add-mono*) (*auto intro*!: *KN.T.finite-measure-mono*)

```
    also have ... ≤ 2 * 𝒫(ω in KN.T (y, None). ¬ (∃ i<n. ev-at (HLD D) i ω))
      by (simp add: ennreal-plus[symmetric] del: ennreal-plus)
    finally have ?L (Suc n) ≤ 2 * 𝒫(ω in KN.T (y, None). ¬ (∃ i<n. ev-at (HLD
D) i ω))
      by (auto intro!: integral-real-bounded simp add: pmf.rep-eq) }
  note le-2 = this

  have c0-D: (c0, Some c0) ∈ D
    by (simp add: D-def c0)

  let ?N' = map-pmf Some N
  interpret NP: pair-prob-space N ?N' ..

  have pos-recurrent: ∀ x∈C × Some ` C. KN.pos-recurrent x
  proof (rule KN.stationary-distributionD(1)[OF KN-essential - KN.stationary-distributionI-pair[OF
N(1)]])
    show K2.stationary-distribution ?N'
      unfolding K2.stationary-distribution-def
      by (subst N(1)[unfolded stationary-distribution-def])
          (auto intro!: bind-pmf-cong simp: K'-def map-pmf-def bind-assoc-pmf
bind-return-pmf)
    show countable (C × Some`C)
      using C by auto
    show set-pmf (pair-pmf N (map-pmf Some N)) ⊆ C × Some ` C
      using ‹N ⊆ C› by auto
  qed

  from c0-D have 𝒫(ω in KN.T (y, None). alw (not (HLD D)) ω) ≤ 𝒫(ω in
KN.T (y, None). alw (not (HLD {(c0, Some c0)})) ω)
    apply (auto intro!: KN.T.finite-measure-mono)
    apply (rule alw-mono, assumption)
    apply (auto simp: HLD-iff)
    done
  also have ... = 0
    apply (rule KN.T.prob-eq-0-AE)
    apply (simp add: not-ev-iff[symmetric])
    apply (subst KN.AE-T-iff)
    apply simp
  proof
    fix t assume t: t ∈ KN.Kp (y, None)
    then obtain a b where t-eq: t = (a, Some b) a ∈ K y b ∈ N
      unfolding KN.Kp-def by (auto simp: K'-def)
    with ‹y ∈ C› have a ∈ C
      using essential-classD2[OF ‹essential-class C› ‹y ∈ C›] by auto
    have b ∈ C
      using ‹N ⊆ C› ‹b ∈ N› by auto

    from pos-recurrent[THEN bspec, of (c0, Some c0)]
    have recurrent-c0: KN.recurrent (c0, Some c0)
```

```
    by (simp add: KN.pos-recurrent-def c0)
  have C × Some ‘ C ∈ UNIV // KN.communicating
    using aperiodic by (simp add: KN.aperiodic-def)
  then have ((c0, Some c0), t) ∈ KN.communicating
    by (rule KN.irreducibleD) (simp-all add: t-eq c0 ‹b ∈ C› ‹a ∈ C›)
  then have ((c0, Some c0), t) ∈ KN.acc
    by (simp add: KN.communicating-def)
  then have KN.U t (c0, Some c0) = 1
    by (rule KN.recurrent-acc(1)[OF recurrent-c0])
  then show AE ω in KN.T t. ev (HLD {(c0, Some c0)}) (t ## ω)
    unfolding KN.U-def by (subst (asm) KN.T.prob-Collect-eq-1) (auto simp
add: ev-Stream)
 qed
 finally have P(ω in KN.T (y, None). alw (not (HLD D)) ω) = 0
  by (intro antisym measure-nonneg)

 have (λn. P(ω in KN.T (y, None). ¬ (∃ i<n. ev-at (HLD D) i ω))) ⟶
  measure (KN.T (y, None)) (⋂ n. {ω∈space (KN.T (y, None)). ¬ (∃ i<n. ev-at
(HLD D) i ω)})
  by (rule KN.T.finite-Lim-measure-decseq) (auto simp: decseq-def)
 also have (⋂ n. {ω∈space (KN.T (y, None)). ¬ (∃ i<n. ev-at (HLD D) i ω)})
=
  {ω∈space (KN.T (y, None)). alw (not (HLD D)) ω}
  by (auto simp: not-ev-iff[symmetric] ev-iff-ev-at)
 also have P(ω in KN.T (y, None). alw (not (HLD D)) ω) = 0 by fact
 finally have ∗: (λn. 2 ∗ P(ω in KN.T (y, None). ¬ (∃ i<n. ev-at (HLD D) i
ω))) ⟶ 0
  by (intro tendsto-eq-intros) auto

 show ?thesis
  apply (rule LIMSEQ-imp-Suc)
  apply (rule tendsto-sandwich[OF - - tendsto-const ∗])
  using le-2
  apply (simp-all add: integral-nonneg-AE)
  done
qed

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 −
 define D where D y n = |p x y n − pmf N y| for y n

 from stationary-distribution-imp-limit[OF assms(1,2,3,4,5,6)]
 have INT: (λn. ∫ y. D y n ∂count-space C) ⟶ 0
  unfolding D-def .
```

```
  { fix n
    have D y n ≤ (∫ z. D y n * indicator {y} z ∂count-space C)
      by simp
    also have ... ≤ (∫ y. D y n ∂count-space C)
      by (intro integral-mono)
         (auto split: split-indicator simp: D-def p-def disjoint-family-on-def
           intro!: Bochner-Integration.integrable-diff integrable-pmf T.integrable-measure)
    finally have D y n ≤ (∫ y. D y n ∂count-space C) . }
  note * = this

  have D-nonneg: ⋀n. 0 ≤ D y n by (simp add: D-def)

  have D y ⟶ 0
    by (rule tendsto-sandwich[OF - - tendsto-const INT])
       (auto simp: eventually-sequentially * D-nonneg)
  then show ?thesis
    using Lim-null[where l=pmf N y and net=sequentially and f=p x y]
    by (simp add: D-def [abs-def] tendsto-rabs-zero-iff)
qed

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 (rule essential-classI)
  { fix x y assume (x, y) ∈ acc x ∈ X
    then show y ∈ X
      by induct (auto dest: ED)}
  note accD = this
  from ‹X ≠ {}› obtain x where x ∈ X by auto
  from ‹x ∈ X› show X ∈ UNIV // communicating
    by (auto simp add: quotient-def Image-def communicating-def accI dest: accD
intro!: exI[of - x])
qed

end
```

5 Markov Decision Processes

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

lemma some-elem-ne: s ≠ {} ⟹ some-elem s ∈ s
  unfolding some-elem-def by (auto intro: someI)
```

5.1 Configurations

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

hide-const *cont*

codatatype *'s scheduler = Scheduler (action-sch: 's pmf) (cont-sch: 's ⇒ 's scheduler)*

lemma *equivp-rel-prod: equivp R ⟹ equivp Q ⟹ equivp (rel-prod R Q)*
by *(auto intro!: equivpI prod.rel-symp prod.rel-transp prod.rel-reflp elim: equivpE)*

coinductive *eq-scheduler :: 's scheduler ⇒ 's scheduler ⇒ bool*
where
⋀D. action-sch sc1 = D ⟹ action-sch sc2 = D ⟹
(∀ s∈D. eq-scheduler (cont-sch sc1 s) (cont-sch sc2 s)) ⟹ eq-scheduler sc1 sc2

lemma *eq-scheduler-refl[intro]: eq-scheduler sc sc*
by *(coinduction arbitrary: sc) auto*

quotient-type *'s cfg = 's × 's scheduler / rel-prod (=) eq-scheduler*
proof *(intro equivp-rel-prod equivpI reflpI sympI transpI)*
show *eq-scheduler sc1 sc2 ⟹ eq-scheduler sc2 sc1* **for** *sc1 sc2 :: 's scheduler*
by *(coinduction arbitrary: sc1 sc2) (auto elim: eq-scheduler.cases)*
show *eq-scheduler sc1 sc2 ⟹ eq-scheduler sc2 sc3 ⟹ eq-scheduler sc1 sc3*
for *sc1 sc2 sc3 :: 's scheduler*
by *(coinduction arbitrary: sc1 sc2 sc3)*
(subst (asm) (1 2) eq-scheduler.simps, auto)
qed *auto*

lift-definition *state :: 's cfg ⇒ 's* **is** *fst*
by *auto*

lift-definition *action :: 's cfg ⇒ 's pmf* **is** *λ(s, sc). action-sch sc*
by *(force elim: eq-scheduler.cases)*

lift-definition *cont :: 's cfg ⇒ 's ⇒ 's cfg* **is**
λ(s, sc) t. if t ∈ action-sch sc then (t, cont-sch sc t) else
(t, cont-sch sc (some-elem (action-sch sc)))
apply *(simp add: rel-prod-conv split: prod.splits)*
apply *(subst (asm) eq-scheduler.simps)*
apply *(auto simp: Let-def set-pmf-not-empty[THEN some-elem-ne])*
done

lift-definition *Cfg :: 's ⇒ 's pmf ⇒ ('s ⇒ 's cfg) ⇒ 's cfg* **is**
λs D c. (s, Scheduler D (λt. snd (c t)))
by *(auto simp: rel-prod-conv split-beta' eq-scheduler.simps[of Scheduler - -])*

lift-definition *cfg-corec :: 's ⇒ ('a ⇒ 's pmf) ⇒ ('a ⇒ 's ⇒ 'a) ⇒ 'a ⇒ 's cfg*

is
  $\lambda s\ D\ C\ x.\ (s,\ corec\text{-}scheduler\ D\ (\lambda x\ s.\ Inr\ (C\ x\ s))\ x)$.

lemma *state-cont*[*simp*]: $state\ (cont\ cfg\ s) = s$
  by *transfer* (*simp split*: *prod.split*)

lemma *state-Cfg*[*simp*]: $state\ (Cfg\ s\ d'\ c') = s$
  by *transfer simp*

lemma *action-Cfg*[*simp*]: $action\ (Cfg\ s\ d'\ c') = d'$
  by *transfer simp*

lemma *cont-Cfg*[*simp*]: $t \in set\text{-}pmf\ d' \Longrightarrow state\ (c'\ t) = t \Longrightarrow cont\ (Cfg\ s\ d'\ c')\ t = c'\ t$
  by *transfer* (*auto simp add*: *rel-prod-conv split*: *prod.split*)

lemma *state-cfg-corec*[*simp*]: $state\ (cfg\text{-}corec\ s\ d\ c\ x) = s$
  by *transfer auto*

lemma *action-cfg-corec*[*simp*]: $action\ (cfg\text{-}corec\ s\ d\ c\ x) = d\ x$
  by *transfer auto*

lemma *cont-cfg-corec*[*simp*]: $t \in set\text{-}pmf\ (d\ x) \Longrightarrow cont\ (cfg\text{-}corec\ s\ d\ c\ x)\ t = cfg\text{-}corec\ t\ d\ c\ (c\ x\ t)$
  by *transfer auto*

lemma *cfg-coinduct*[*consumes 1*, *case-names state action cont*, *coinduct pred*]:
  $X\ c\ d \Longrightarrow (\bigwedge c\ d.\ X\ c\ d \Longrightarrow state\ c = state\ d) \Longrightarrow (\bigwedge c\ d.\ X\ c\ d \Longrightarrow action\ c = action\ d) \Longrightarrow$
    $(\bigwedge c\ d\ t.\ X\ c\ d \Longrightarrow t \in set\text{-}pmf\ (action\ c) \Longrightarrow X\ (cont\ c\ t)\ (cont\ d\ t)) \Longrightarrow c = d$
proof (*transfer*, *clarsimp*)
  fix $X :: ('a \times 'a\ scheduler) \Rightarrow ('a \times 'a\ scheduler) \Rightarrow bool$ **and** *B s1 s2 sc1 sc2*
  assume *X*: $X\ (s1,\ sc1)\ (s2,\ sc2)$ **and** *rel-fun cr-cfg* (*rel-fun cr-cfg* (=)) *X B*
    and *1*: $\bigwedge s1\ sc1\ s2\ sc2.\ X\ (s1,\ sc1)\ (s2,\ sc2) \Longrightarrow s1 = s2$
    and *2*: $\bigwedge s1\ sc1\ s2\ sc2.\ X\ (s1,\ sc1)\ (s2,\ sc2) \Longrightarrow action\text{-}sch\ sc1 = action\text{-}sch\ sc2$
    and *3*: $\bigwedge s1\ sc1\ s2\ sc2\ t.\ X\ (s1,\ sc1)\ (s2,\ sc2) \Longrightarrow t \in set\text{-}pmf\ (action\text{-}sch\ sc2) \Longrightarrow$
      $X\ (t,\ cont\text{-}sch\ sc1\ t)\ (t,\ cont\text{-}sch\ sc2\ t)$
  from *X* **show** *eq-scheduler sc1 sc2*
    by (*coinduction arbitrary*: *s1 s2 sc1 sc2*)
      (*blast dest*: *2 3*)
qed

coinductive *rel-cfg* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ cfg \Rightarrow 'b\ cfg \Rightarrow bool$ **for** $P :: 'a \Rightarrow 'b \Rightarrow bool$
where
  $P\ (state\ cfg1)\ (state\ cfg2) \Longrightarrow$

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*)
by (*auto elim*: *rel-cfg.cases*)

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*)
by (*auto elim*: *rel-cfg.cases*)

lemma *rel-cfg-action*:
assumes *P*: *rel-cfg P cfg1 cfg2* **shows** *rel-pmf P* (*action cfg1*) (*action cfg2*)
proof (*rule pmf.rel-mono-strong*)
show *rel-pmf* (λ*s t*. *rel-cfg P* (*cont cfg1 s*) (*cont cfg2 t*)) (*action cfg1*) (*action cfg2*)
using *P* **by** (*rule rel-cfg-cont*)
qed (*auto dest*: *rel-cfg-state*)

lemma *rel-cfg-eq*: *rel-cfg* (=) *cfg1 cfg2* ⟷ *cfg1* = *cfg2*
proof *safe*
show *rel-cfg* (=) *cfg1 cfg2* ⟹ *cfg1* = *cfg2*
proof (*coinduction arbitrary*: *cfg1 cfg2*)
case *cont*
have *action cfg1* = *action cfg2*
using ‹*rel-cfg* (=) *cfg1 cfg2*› **by** (*auto dest*: *rel-cfg-action simp*: *pmf.rel-eq*)
then have *rel-pmf* (λ*s t*. *rel-cfg* (=) (*cont cfg1 s*) (*cont cfg2 t*)) (*action cfg1*) (*action cfg1*)
using *cont* **by** (*auto dest*: *rel-cfg-cont*)
then have *rel-pmf* (λ*s t*. *rel-cfg* (=) (*cont cfg1 s*) (*cont cfg2 t*) ∧ *s* = *t*) (*action cfg1*) (*action cfg1*)
by (*rule pmf.rel-mono-strong*) (*auto dest*: *rel-cfg-state*)
then have *pred-pmf* (λ*s*. *rel-cfg* (=) (*cont cfg1 s*) (*cont cfg2 s*)) (*action cfg1*)
unfolding *pmf.pred-rel* **by** (*rule pmf.rel-mono-strong*) (*auto simp*: *eq-onp-def*)
with ‹*t* ∈ *action cfg1*› **show** *?case*
by (*auto simp*: *pmf.pred-set*)
qed (*auto dest*: *rel-cfg-state rel-cfg-action simp*: *pmf.rel-eq*)
show *rel-cfg* (=) *cfg2 cfg2*
by (*coinduction arbitrary*: *cfg2*) (*auto intro*!: *rel-pmf-reflI*)
qed

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*]: $t \in (f\ s) \Longrightarrow cont\ (memoryless\text{-}on\ f\ s)\ t = memoryless\text{-}on\ f\ t$
by (*simp-all add*: *memoryless-on-def*)

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*)
by (*simp add*: *K-cfg-def*)

lemma *nn-integral-K-cfg*: $(\int^+ cfg.\ f\ cfg\ \partial K\text{-}cfg\ cfg) = (\int^+ s.\ f\ (cont\ cfg\ s)\ \partial action\ cfg)$
by (*simp add*: *K-cfg-def map-pmf-rep-eq nn-integral-distr*)

5.3 MDP Kernel and Induced Configurations

locale *Markov-Decision-Process* =
fixes *K* :: *'s* ⇒ *'s pmf set*
assumes *K-wf*: $\bigwedge s.\ K\ s \neq \{\}$
begin

definition $E = (SIGMA\ s{:}UNIV.\ \bigcup D{\in}K\ s.\ set\text{-}pmf\ D)$

coinductive *cfg-onp* :: *'s* ⇒ *'s cfg* ⇒ *bool* **where**
$\bigwedge s.\ state\ cfg = s \Longrightarrow action\ cfg \in K\ s \Longrightarrow (\bigwedge t.\ t \in action\ cfg \Longrightarrow cfg\text{-}onp\ t\ (cont\ cfg\ t)) \Longrightarrow$
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*
by (*auto simp*: *cfg-on-def intro*: *cfg-onp.intros elim*: *cfg-onp.cases*)

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 ⋀*cfg s t*. *P s cfg* ⟹ *t* ∈ *action cfg* ⟹ *P t* (*cont cfg t*)
shows *cfg* ∈ *cfg-on s*
using *assms cfg-onp.coinduct*[*of P s cfg*] **by** (*simp add*: *cfg-on-def*)

lemma *memoryless-on-cfg-onI*:
assumes ⋀*s*. *f s* ∈ *K s*

shows *memoryless-on f s* ∈ *cfg-on s*
by (*coinduction arbitrary*: *s*) (*auto intro*: *assms*)

lemma *cfg-of-cfg-onI*:
D ∈ *K s* ⟹ (⋀*t*. *t* ∈ *D* ⟹ *c t* ∈ *cfg-on t*) ⟹ *Cfg s D c* ∈ *cfg-on s*
by (*rule cfg-onI*) *auto*

definition *arb-act s* = (*SOME D*. *D* ∈ *K s*)

lemma *arb-actI*[*simp*]: *arb-act s* ∈ *K s*
by (*simp add*: *arb-act-def some-in-eq K-wf*)

lemma *cfg-on-not-empty*[*intro*, *simp*]: *cfg-on s* ≠ {}
by (*auto intro*: *memoryless-on-cfg-onI arb-actI*)

sublocale *MC*: *MC-syntax K-cfg* **.**

abbreviation *St* :: *'s stream measure* **where**
St ≡ *stream-space* (*count-space UNIV*)

5.4 Trace Space

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

sublocale *T*: *prob-space T cfg* **for** *cfg*
by (*simp add*: *T-def MC.T.prob-space-distr*)

lemma *space-T*[*simp*]: *space* (*T cfg*) = *space St*
by (*simp add*: *T-def*)

lemma *sets-T*[*simp*]: *sets* (*T cfg*) = *sets St*
by (*simp add*: *T-def*)

lemma *measurable-T1*[*simp*]: *measurable* (*T cfg*) *N* = *measurable St N*
by (*simp add*: *T-def*)

lemma *measurable-T2*[*simp*]: *measurable N* (*T cfg*) = *measurable N St*
by (*simp add*: *T-def*)

lemma *nn-integral-T*:
assumes [*measurable*]: *f* ∈ *borel-measurable St*
shows (∫⁺*X*. *f X* ∂*T cfg*) = (∫⁺*cfg'*. (∫⁺*x*. *f* (*state cfg'* ## *x*) ∂*T cfg'*) ∂*K-cfg cfg*)
by (*simp add*: *T-def MC.nn-integral-T*[*of - cfg*] *nn-integral-distr*)

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

fix A **assume** $A \in sets\ (T\ cfg)$
then show $emeasure\ (T\ cfg)\ A =$
$emeasure\ (measure\text{-}pmf\ (K\text{-}cfg\ cfg) \gg\!= (\lambda cfg'.\ distr\ (T\ cfg')\ St\ (\lambda\omega.\ state\ cfg'\ \#\#\ \omega)))\ A$
by (*subst emeasure-bind*[**where** $N{=}St$])
(*auto simp*: *space-subprob-algebra nn-integral-distr nn-integral-indicator*[*symmetric*] *nn-integral-T*[*of* - *cfg*]
simp del: *nn-integral-indicator intro*!: *prob-space-imp-subprob-space T.prob-space-distr*)
qed *simp*

lemma *T-memoryless-on*: $T\ (memoryless\text{-}on\ ct\ s) = MC\text{-}syntax.T\ ct\ s$
proof −
interpret *ct*: *MC-syntax ct* **.**
have $T \circ (memoryless\text{-}on\ ct) = MC\text{-}syntax.T\ ct$
proof (*rule ct.T-bisim*[*symmetric*])
fix s **show** $(T \circ memoryless\text{-}on\ ct)\ s =$
$measure\text{-}pmf\ (ct\ s) \gg\!= (\lambda s.\ distr\ ((T \circ memoryless\text{-}on\ ct)\ s)\ St\ ((\#\#)\ s))$
by (*auto simp add*: *T-eq*[*of memoryless-on ct s*] *K-cfg-def map-pmf-rep-eq bind-distr*[**where** $K{=}St$]
space-subprob-algebra T.prob-space-distr prob-space-imp-subprob-space intro!: *bind-measure-pmf-cong*)
qed (*simp-all*, *intro-locales*)
then show *?thesis* **by** (*simp add*: *fun-eq-iff*)
qed

lemma *nn-integral-T-lfp*:
assumes [*measurable*]: $case\text{-}prod\ g \in borel\text{-}measurable\ (count\text{-}space\ UNIV \bigotimes_M borel)$
assumes *cont-g*: $\bigwedge s.\ sup\text{-}continuous\ (g\ s)$
assumes *int-g*: $\bigwedge f\ cfg.\ f \in borel\text{-}measurable\ (stream\text{-}space\ (count\text{-}space\ UNIV))$
$\Longrightarrow$
$(\int^+\omega.\ g\ (state\ cfg)\ (f\ \omega)\ \partial T\ cfg) = g\ (state\ cfg)\ (\int^+\omega.\ f\ \omega\ \partial T\ cfg)$
shows $(\int^+\omega.\ lfp\ (\lambda f\ \omega.\ g\ (shd\ \omega)\ (f\ (stl\ \omega)))\ \omega\ \partial T\ cfg) =$
$lfp\ (\lambda f\ cfg.\ \int^+ t.\ g\ (state\ t)\ (f\ t)\ \partial K\text{-}cfg\ cfg)\ cfg$
proof (*rule nn-integral-lfp*)
show $\bigwedge s.\ sets\ (T\ s) = sets\ St$
$\bigwedge F.\ F \in borel\text{-}measurable\ St \Longrightarrow (\lambda a.\ g\ (shd\ a)\ (F\ (stl\ a))) \in borel\text{-}measurable\ St$
by *auto*
next
fix s **and** $F :: 's\ stream \Rightarrow ennreal$ **assume** $F \in borel\text{-}measurable\ St$
then show $(\int^+ a.\ g\ (shd\ a)\ (F\ (stl\ a))\ \partial T\ s) =$
$(\int^+ cfg.\ g\ (state\ cfg)\ (integral^N\ (T\ cfg)\ F)\ \partial K\text{-}cfg\ s)$
by (*rewrite nn-integral-T*) (*simp-all add*: *int-g*)
qed (*auto intro*!: *order-continuous-intros cont-g*[*THEN sup-continuous-compose*])

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*)
using *MC.emeasure-Collect-T*[*of* λ*x*. *P* (*smap state x*) *cfg*]
by (*simp add*: *nn-integral-distr emeasure-Collect-distr T-def*)

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*
using *T.emeasure-space-1* **by** (*simp add*: *E-sup-def*)

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*
by (*simp add*: *nn-integral-cmult E-sup-def SUP-mult-left-ennreal*)

lemma *E-sup-mono*:
(⋀ω. *f* ω ≤ *g* ω) ⟹ *E-sup s f* ≤ *E-sup s g*
unfolding *E-sup-def* **by** (*intro SUP-subset-mono order-refl nn-integral-mono*)

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 −
have *E-sup s* (λ*x*. *f x* + *g x*) = (⨆ *cfg*∈*cfg-on s*. (∫⁺*x*. *f x* ∂*T cfg*) + (∫⁺*x*. *g x* ∂*T cfg*))
by (*simp add*: *E-sup-def nn-integral-add*)
also have ... ≤ (⨆ *cfg*∈*cfg-on s*. ∫⁺*x*. *f x* ∂*T cfg*) + (⨆ *cfg*∈*cfg-on s*. (∫⁺*x*. *g x* ∂*T cfg*))
by (*auto simp*: *SUP-le-iff intro*!: *add-mono SUP-upper*)
finally show *?thesis*
by (*simp add*: *E-sup-def*)
qed

lemma *E-sup-add-left*:
assumes [*measurable*]: *f* ∈ *borel-measurable St*
shows *E-sup s* (λ*x*. *f x* + *c*) = *E-sup s f* + *c*
by (*simp add*: *nn-integral-add E-sup-def T.emeasure-space-1*[*simplified*] *ennreal-SUP-add-left*)

lemma *E-sup-add-right*:
f ∈ *borel-measurable St* ⟹ *E-sup s* (λ*x*. *c* + *f x*) = *c* + *E-sup s f*
using *E-sup-add-left*[*of f s c*] **by** (*simp add*: *add.commute*)

lemma *E-sup-SUP*:
assumes [*measurable*]: ⋀*i*. *f i* ∈ *borel-measurable St* **and** [*simp*]: *incseq f*
shows *E-sup s* (λ*x*. ⨆ *i*. *f i x*) = (⨆ *i*. *E-sup s* (*f i*))
by (*auto simp add*: *E-sup-def nn-integral-monotone-convergence-SUP intro*: *SUP-commute*)

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 −
let *?v* = $\lambda t.\ \int^+ x.\ f\ (state\ t\ \#\#\ x)\ \partial T\ t$
let *?p* = $\lambda t.$ *E-sup t* $(\lambda\omega.\ f\ (t\ \#\#\ \omega))$
have *E-sup s f* = $(\bigsqcup cfg{\in}cfg\text{-}on\ s.\ \int^+ t.$ *?v t* ∂*K-cfg cfg*)
unfolding *E-sup-def* **by** (*intro SUP-cong refl*) (*subst nn-integral-T*, *simp-all add*: *cfg-on-def*)
also have ... = $(\bigsqcup D{\in}K\ s.\ \int^+ t.$ *?p t* ∂*measure-pmf D*)
proof (*intro antisym SUP-least*)
fix *cfg* :: *'s cfg* **assume** *cfg*: *cfg* $\in$ *cfg-on s*
then show $(\int^+ t.$ *?v t* ∂*K-cfg cfg*$) \le ($*SUP* $D{\in}K\ s.\ \int^+ t.$ *?p t* ∂*measure-pmf D*)
by (*auto simp*: *E-sup-def nn-integral-K-cfg AE-measure-pmf-iff*
intro!: *nn-integral-mono-AE SUP-upper2*)
next
fix *D* **assume** *D*: *D* $\in$ *K s* **show** $(\int^+ t.$ *?p t* $\partial D) \le ($*SUP cfg* $\in$ *cfg-on s.* $\int^+ t.$ *?v t* ∂*K-cfg cfg*)
proof *cases*
assume *p-finite*: $\forall\, t{\in}D.$ *?p t* $< \infty$
show *?thesis*
proof (*rule ennreal-le-epsilon*)
fix *e* :: *real* **assume** *0* $<$ *e*
have $\forall\, t{\in}D.\ \exists\, cfg{\in}cfg\text{-}on\ t.$ *?p t* $\le$ *?v cfg* + *e*
proof
fix *t* **assume** *t* $\in$ *D*
moreover have (*SUP cfg* $\in$ *cfg-on t. ?v cfg*) = *?p t*
unfolding *E-sup-def* **by** (*simp add*: *cfg-on-def*)
ultimately have (*SUP cfg* $\in$ *cfg-on t. ?v cfg*) $\neq \infty$
using *p-finite* **by** *auto*
from *SUP-approx-ennreal*[*OF* ‹*0*<*e*› *- refl this*]
show $\exists\, cfg{\in}cfg\text{-}on\ t.$ *?p t* $\le$ *?v cfg* + *e*
by (*auto simp add*: *E-sup-def intro*: *less-imp-le*)
qed
then obtain *cfg'* **where** *v-cfg'*: $\bigwedge t.\ t \in D \Longrightarrow$ *?p t* $\le$ *?v* (*cfg' t*) + *e* **and**
cfg-on-cfg': $\bigwedge t.\ t \in D \Longrightarrow$ *cfg' t* $\in$ *cfg-on t*
unfolding *Bex-def bchoice-iff* **by** *blast*

let *?cfg* = *Cfg s D cfg'*
have *cfg*: *K-cfg ?cfg* = *map-pmf cfg' D*
by (*auto simp add*: *K-cfg-def fun-eq-iff cfg-on-cfg' intro*!: *map-pmf-cong*)

have $(\int^+ t.$ *?p t* $\partial D) \le (\int^+ t.$ *?v* (*cfg' t*) + *e* $\partial D)$
by (*intro nn-integral-mono-AE*) (*simp add*: *v-cfg' AE-measure-pmf-iff*)
also have ... = $(\int^+ t.$ *?v* (*cfg' t*) $\partial D) +$ *e*
using ‹*0* < *e*› *measure-pmf.emeasure-space-1*[*of D*]
by (*subst nn-integral-add*) (*auto intro*: *cfg-on-cfg'*)

```
      also have (∫⁺t. ?v (cfg' t) ∂D) = (∫⁺t. ?v t ∂K-cfg ?cfg)
        by (simp add: cfg map-pmf-rep-eq nn-integral-distr)
      also have ... ≤ (SUP cfg∈cfg-on s. (∫⁺t. ?v t ∂K-cfg cfg))
        by (auto intro!: SUP-upper intro!: cfg-of-cfg-onI D cfg-on-cfg')
      finally show (∫⁺ t. ?p t ∂D) ≤ (SUP cfg ∈ cfg-on s. ∫⁺ t. ?v t ∂K-cfg
cfg) + e
        by (blast intro: add-mono)
    qed
  next
    assume ¬ (∀ t∈D. ?p t < ∞)
    then obtain t where t ∈ D ?p t = ∞
      by (auto simp: not-less top-unique)
    then have ∞ = pmf (D) t * ?p t
      by (auto simp: ennreal-mult-top set-pmf-iff)
    also have ... = (SUP cfg ∈ cfg-on t. pmf (D) t * ?v cfg)
      unfolding E-sup-def
      by (auto simp: SUP-mult-left-ennreal[symmetric])
    also have ... ≤ (SUP cfg ∈ cfg-on s. ∫⁺ t. ?v t ∂K-cfg cfg)
      unfolding E-sup-def
    proof (intro SUP-least SUP-upper2)
      fix cfg :: 's cfg assume cfg: cfg ∈ cfg-on t

      let ?cfg = Cfg s D ((memoryless-on arb-act) (t := cfg))
      have C: K-cfg ?cfg = map-pmf ((memoryless-on arb-act) (t := cfg)) D
        by (auto simp add: K-cfg-def fun-eq-iff intro!: map-pmf-cong simp: cfg)

      show ?cfg ∈ cfg-on s
        by (auto intro!: cfg-of-cfg-onI D cfg memoryless-on-cfg-onI)
      have ennreal (pmf (D) t) * (∫⁺ x. f (state cfg ## x) ∂T cfg) =
        (∫⁺t'. (∫⁺ x. f (state cfg ## x) ∂T cfg) * indicator {t} t' ∂D)
        by (auto simp add:  max-def emeasure-pmf-single intro: mult-ac)
      also have ... = (∫⁺cfg. ?v cfg * indicator {t} (state cfg) ∂K-cfg ?cfg)
        unfolding C using cfg
        by (auto simp add: nn-integral-distr map-pmf-rep-eq split: split-indicator
              simp del: nn-integral-indicator-singleton
              intro!: nn-integral-cong)
      also have ... ≤ (∫⁺cfg. ?v cfg ∂K-cfg ?cfg)
        by (auto intro!: nn-integral-mono  split: split-indicator)
      finally show ennreal (pmf (D) t) * (∫⁺ x. f (state cfg ## x) ∂T cfg)
        ≤ (∫⁺ t. ∫⁺ x. f (state t ## x) ∂T t ∂K-cfg ?cfg) .
    qed
    finally show ?thesis
      by (simp add: top-unique del: Sup-eq-top-iff SUP-eq-top-iff)
  qed
 qed
 finally show ?thesis .
qed

lemma E-sup-bot: E-sup s ⊥ = 0
```

by (*auto simp add*: *E-sup-def bot-ennreal*)

lemma *E-sup-lfp*:
fixes *g*
defines $l \equiv \lambda f\ \omega.\ g\ (shd\ \omega)\ (f\ (stl\ \omega))$
assumes *measurable-g*[*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 St* ⟹
$(\int^+ \omega.\ g\ (state\ cfg)\ (f\ \omega)\ \partial T\ cfg) = g\ (state\ cfg)\ (integral^N\ (T\ cfg)\ f)$
shows $(\lambda s.\ E\text{-}sup\ s\ (lfp\ l)) = lfp\ (\lambda f\ s.\ \bigsqcup D{\in}K\ s.\ \int^+ t.\ g\ t\ (f\ t)\ \partial measure\text{-}pmf\ D)$
proof (*rule lfp-transfer-bounded*[**where** $\alpha{=}\lambda F\ s.\ E\text{-}sup\ s\ F$ **and** $f{=}l$ **and** $P{=}\lambda f.\ f \in borel\text{-}measurable\ St$])
show *sup-continuous* $(\lambda f\ s.\ \bigsqcup x{\in}K\ s.\ \int^+ t.\ g\ t\ (f\ t)\ \partial measure\text{-}pmf\ x)$
using *cont-g*[*THEN sup-continuous-compose*] **by** (*auto intro*!: *order-continuous-intros*)
show *sup-continuous l*
using *cont-g*[*THEN sup-continuous-compose*] **by** (*auto intro*!: *order-continuous-intros simp*: *l-def*)
show $\bigwedge F.\ (\lambda s.\ E\text{-}sup\ s\ \bot) \le (\lambda s.\ \bigsqcup D{\in}K\ s.\ \int^+ t.\ g\ t\ (F\ t)\ \partial measure\text{-}pmf\ D)$
using *K-wf* **by** (*auto simp*: *E-sup-bot le-fun-def intro*: *SUP-upper2*)
next
fix *f* :: *'s stream* ⇒ *ennreal* **assume** *f*: *f* ∈ *borel-measurable St*
moreover
have $E\text{-}sup\ s\ (\lambda\omega.\ g\ s\ (f\ \omega)) = g\ s\ (E\text{-}sup\ s\ f)$ **for** *s*
unfolding *E-sup-def* **using** *int-g*[*OF f*]
by (*subst SUP-sup-continuous-ennreal*[*OF cont-g*, *symmetric*])
(*auto intro*!: *SUP-cong simp del*: *cfg-onD-state dest*: *cfg-onD-state*[*symmetric*])
ultimately show $(\lambda s.\ E\text{-}sup\ s\ (l\ f)) = (\lambda s.\ \bigsqcup D{\in}K\ s.\ \int^+ t.\ g\ t\ (E\text{-}sup\ t\ f)\ \partial measure\text{-}pmf\ D)$
by (*subst E-sup-iterate*) (*auto simp*: *l-def int-g fun-eq-iff intro*!: *SUP-cong nn-integral-cong*)
qed (*auto simp*: *bot-fun-def l-def SUP-apply*[*abs-def*] *E-sup-SUP*)

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

lemma *P-sup-eq-E-sup*:
assumes [*measurable*]: *Measurable.pred St P*
shows *P-sup s P* = *E-sup s* (*indicator* $\{x{\in}space\ St.\ P\ x\}$)
by (*auto simp add*: *P-sup-def E-sup-def intro*!: *SUP-cong nn-integral-cong*)

lemma *P-sup-True*[*simp*]: *P-sup t* (λω. *True*) = *1*
using *T.emeasure-space-1*
by (*auto simp add*: *P-sup-def SUP-constant*)

lemma *P-sup-False*[*simp*]: *P-sup t* (λω. *False*) = *0*
by (*auto simp add*: *P-sup-def SUP-constant*)

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* $(\lambda x.\ \exists\, i.\ P\ i\ x) = (\bigsqcup i.$ *P-sup s* $(P\ i))$
proof −
have *P-sup s* $(\lambda x.\ \bigsqcup i.\ P\ i\ x) = (\bigsqcup$ *cfg*∈*cfg-on s. emeasure* $(T\ cfg)\ (\bigcup i.\ \{x \in$ *space St. P i x*$\}))$
by (*auto simp*: *P-sup-def intro*!: *SUP-cong arg-cong2*[**where** *f*=*emeasure*])
also have $\ldots = (\bigsqcup$ *cfg*∈*cfg-on s.* $\bigsqcup i.$ *emeasure* $(T\ cfg)\ \{x \in$ *space St. P i x*$\})$
using ‹*mono P*› **by** (*auto intro*!: *SUP-cong SUP-emeasure-incseq*[*symmetric*] *simp*: *mono-def le-fun-def*)
also have $\ldots = (\bigsqcup i.$ *P-sup s* $(P\ i))$
by (*subst SUP-commute*) (*simp add*: *P-sup-def*)
finally show *?thesis*
by *simp*
qed

lemma *P-sup-lfp*:
assumes *Q*: *sup-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-sup s* $(\lambda x.$ *lfp Q* $(f\ x)) = (\bigsqcup i.$ *P-sup s* $(\lambda x.\ (Q\ \widehat{\ }\ i)\ \bot\ (f\ x)))$
unfolding *sup-continuous-lfp*[*OF Q*]
apply *simp*
proof (*rule P-sup-SUP*)
fix *i* **show** *Measurable.pred St* $(\lambda x.\ (Q\ \widehat{\ }\ i)\ \bot\ (f\ x))$
apply (*intro measurable-compose*[*OF f*])
by (*induct i*) (*auto intro*!: *Q-m*)
qed (*intro mono-funpow sup-continuous-mono*[*OF Q*] *mono-compose*[**where** *f*=*f*])

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* $(\lambda\omega.\ P\ (t\ \#\#\ \omega))\ \partial$*measure-pmf D*$)$
proof −
have [*simp*]: $\bigwedge x\ s.$ *indicator* $\{x \in$ *space St. P x*$\}\ (x\ \#\#\ s) =$ *indicator* $\{s \in$ *space St. P* $(x\ \#\#\ s)\}\ s$
by (*auto simp*: *space-stream-space split*: *split-indicator*)
show *?thesis*
using *E-sup-iterate*[*of indicator* $\{x \in$*space St. P x*$\}\ s$] **by** (*auto simp*: *P-sup-eq-E-sup*)
qed

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

lemma *E-inf-const*: $0 \le c \Longrightarrow$ *E-inf s* $(\lambda\text{-}.\ c) = c$
using *T.emeasure-space-1* **by** (*simp add*: *E-inf-def*)

lemma *E-inf-mono*:
$(\bigwedge\omega.\ f\ \omega \le g\ \omega) \Longrightarrow$ *E-inf s f* $\le$ *E-inf s g*
unfolding *E-inf-def* **by** (*intro INF-superset-mono order-refl nn-integral-mono*)

lemma *E-inf-iterate*:
assumes [*measurable*]: $f \in$ *borel-measurable St*
shows *E-inf s f* $= (\prod D{\in}K\ s.\ \int^+ t.$ *E-inf t* $(\lambda\omega.\ f\ (t\ \#\#\ \omega))\ \partial$*measure-pmf D*)
proof −
let *?v* $= \lambda t.\ \int^+ x.\ f\ ($*state t* $\#\#\ x)\ \partial T\ t$
let *?p* $= \lambda t.$ *E-inf t* $(\lambda\omega.\ f\ (t\ \#\#\ \omega))$
have *E-inf s f* $= (\prod$ *cfg*$\in$*cfg-on s.* $\int^+ t.$ *?v t* ∂*K-cfg cfg*)
unfolding *E-inf-def* **by** (*intro INF-cong refl*) (*subst nn-integral-T*, *simp-all add*: *cfg-on-def*)
also have $\ldots = (\prod D{\in}K\ s.\ \int^+ t.$ *?p t* ∂*measure-pmf D*)
proof (*intro antisym INF-greatest*)
fix *cfg* :: *'s cfg* **assume** *cfg*: *cfg* $\in$ *cfg-on s*
then show (*INF* $D{\in}K\ s.\ \int^+ t.$ *?p t* ∂*measure-pmf D*) $\leq (\int^+ t.$ *?v t* ∂*K-cfg cfg*)
by (*auto simp add*: *E-inf-def nn-integral-K-cfg AE-measure-pmf-iff intro*!: *nn-integral-mono-AE INF-lower2*)
next
fix *D* **assume** *D*: $D \in K\ s$ **show** (*INF cfg* $\in$ *cfg-on s.* $\int^+ t.$ *?v t* ∂*K-cfg cfg*) $\leq (\int^+ t.$ *?p t* ∂D)
proof (*rule ennreal-le-epsilon*)
fix *e* :: *real* **assume** $0 < e$
have $\forall\, t{\in}D.\ \exists$ *cfg*$\in$*cfg-on t. ?v cfg* $\leq$ *?p t* $+ e$
proof
fix *t* **assume** $t \in D$
show $\exists$ *cfg*$\in$*cfg-on t. ?v cfg* $\leq$ *?p t* $+ e$
proof *cases*
assume *?p t* $= \infty$ **with** *cfg-on-not-empty*[*of t*] **show** *?thesis*
by (*auto simp*: *top-add simp del*: *cfg-on-not-empty*)
next
assume *p-finite*: *?p t* $\neq \infty$
note ‹$t \in D$›
moreover have (*INF cfg* $\in$ *cfg-on t. ?v cfg*) $=$ *?p t*
unfolding *E-inf-def* **by** (*simp add*: *cfg-on-def*)
ultimately have (*INF cfg* $\in$ *cfg-on t. ?v cfg*) $\neq \infty$
using *p-finite* **by** *auto*
from *INF-approx-ennreal*[*OF* ‹$0 < e$› *refl this*]
show $\exists$ *cfg*$\in$*cfg-on t. ?v cfg* $\leq$ *?p t* $+ e$
by (*auto simp*: *E-inf-def intro*: *less-imp-le*)
qed
qed
then obtain *cfg'* **where** *v-cfg'*: $\bigwedge t.\ t \in D \Longrightarrow$ *?v* (*cfg'* *t*) $\leq$ *?p t* $+ e$ **and**
cfg-on-cfg': $\bigwedge t.\ t \in D \Longrightarrow$ *cfg' t* $\in$ *cfg-on t*
unfolding *Bex-def bchoice-iff* **by** *blast*

let *?cfg* $=$ *Cfg s D cfg'*

have *cfg*: *K-cfg ?cfg* $=$ *map-pmf cfg' D*
by (*auto simp add*: *K-cfg-def cfg-on-cfg' intro*!: *map-pmf-cong*)

have *?cfg* ∈ *cfg-on s*
by (*auto intro*: *D cfg-on-cfg′ cfg-of-cfg-onI*)
then have (*INF cfg* ∈ *cfg-on s*. ∫⁺ *t*. *?v t* ∂*K-cfg cfg*) ≤ (∫⁺ *t*. *?p t* + *e* ∂*D*)
by (*rule INF-lower2*) (*auto simp*: *cfg map-pmf-rep-eq nn-integral-distr v-cfg′ AE-measure-pmf-iff intro*!: *nn-integral-mono-AE*)
also have . . . = (∫⁺ *t*. *?p t* ∂*D*) + *e*
using ‹*0* < *e*› **by** (*simp add*: *nn-integral-add measure-pmf.emeasure-space-1*[*simplified*])
finally show (*INF cfg* ∈ *cfg-on s*. ∫⁺ *t*. *?v t* ∂*K-cfg cfg*) ≤ (∫⁺ *t*. *?p t* ∂*D*) + *e* **.**
qed
qed
finally show *?thesis* **.**
qed

lemma *emeasure-T-const*[*simp*]: *emeasure* (*T s*) (*space St*) = *1*
using *T.emeasure-space-1*[*of s*] **by** *simp*

lemma *E-inf-greatest*:
(⋀*cfg*. *cfg* ∈ *cfg-on s* ⟹ *x* ≤ (∫⁺*x*. *f x* ∂*T cfg*)) ⟹ *x* ≤ *E-inf s f*
unfolding *E-inf-def* **by** (*rule INF-greatest*)

lemma *E-inf-lower2*:
cfg ∈ *cfg-on s* ⟹ (∫⁺*x*. *f x* ∂*T cfg*) ≤ *x* ⟹ *E-inf s f* ≤ *x*
unfolding *E-inf-def* **by** (*rule INF-lower2*)

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

lemma *E-inf-lfp*:
fixes *g*
defines *l* ≡ λ*f ω*. *g* (*shd ω*) (*f* (*stl ω*))
assumes *measurable-g*[*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 St* ⟹
(∫⁺ *ω*. *g* (*state cfg*) (*f ω*) ∂*T cfg*) = *g* (*state cfg*) (*integral*^{N} (*T cfg*) *f*)
assumes *K-finite*: ⋀*s*. *finite* (*K s*)
shows (λ*s*. *E-inf s* (*lfp l*)) = *lfp* (λ*f s*. ∏ *D*∈*K s*. ∫⁺*t*. *g t* (*f t*) ∂*measure-pmf D*)
proof (*rule antisym*)
let *?F* = λ*F s*. ∏ *D*∈*K s*. ∫⁺ *t*. *g t* (*F t*) ∂*measure-pmf D*
let *?I* = λ*D*. (∫⁺*t*. *g t* (*lfp ?F t*) ∂*measure-pmf D*)
have *mono-F*: *mono ?F*
using *sup-continuous-mono*[*OF cont-g*]
by (*force intro*!: *INF-mono nn-integral-mono monoI simp*: *mono-def le-fun-def*)
define *ct* **where** *ct s* = (*SOME D*. *D* ∈ *K s* ∧ (*lfp ?F s* = *?I D*)) **for** *s*
{ fix *s*
have *finite* (*?I* ‘ *K s*)
by (*auto intro*: *K-finite*)
then obtain *D* **where** *D* ∈ *K s ?I D* = *Min* (*?I* ‘ *K s*)

```
    by (auto simp: K-wf dest!: Min-in)
  note this(2)
  also have ... = (INF D ∈ K s. ?I D)
    using K-wf by (subst Min-Inf) (auto intro: K-finite)
  also have ... = lfp ?F s
    by (rewrite in - = ⌑ lfp-unfold[OF mono-F]) auto
  finally have ∃ D. D ∈ K s ∧ (lfp ?F s = ?I D)
    using ‹D ∈ K s› by auto
  then have ct s ∈ K s ∧ (lfp ?F s = ?I (ct s))
    unfolding ct-def by (rule someI-ex)
  then have ct s ∈ K s lfp ?F s = ?I (ct s)
    by auto }
 note ct = this
 then have ct-cfg-on[simp]: ⋀s. memoryless-on ct s ∈ cfg-on s
  by (intro memoryless-on-cfg-onI) simp
 then show (λs. E-inf s (lfp l)) ≤ lfp ?F
 proof (intro le-funI, rule E-inf-lower2)
  fix s
  define P where P f cfg = ∫⁺ t. g (state t) (f t) ∂K-cfg cfg for f cfg
  have integral^N (T (memoryless-on ct s)) (lfp l) = lfp P (memoryless-on ct s)
   unfolding P-def l-def using measurable-g cont-g int-g by (rule nn-integral-T-lfp)
  also have ... = (SUP i. (P ^^ i) ⊥) (memoryless-on ct s)
    by (rewrite sup-continuous-lfp)
      (auto intro!: order-continuous-intros cont-g[THEN sup-continuous-compose]
simp: P-def)
  also have ... = (SUP i. (P ^^ i) ⊥ (memoryless-on ct s))
    by (simp add: image-comp)
  also have ... ≤ lfp ?F s
  proof (rule SUP-least)
    fix i show (P ^^ i) ⊥ (memoryless-on ct s) ≤ lfp ?F s
    proof (induction i arbitrary: s)
      case 0 then show ?case
        by simp
    next
      case (Suc n)
      have (P ^^ Suc n) ⊥ (memoryless-on ct s) =
        (∫⁺ t. g t ((P ^^ n) ⊥ (memoryless-on ct t)) ∂ct s)
       by (auto simp add: P-def K-cfg-def AE-measure-pmf-iff intro!: nn-integral-cong-AE)
      also have ... ≤ (∫⁺ t. g t (lfp ?F t) ∂ct s)
      by (intro nn-integral-mono sup-continuous-mono[OF cont-g, THEN monoD]
Suc)
      also have ... = lfp ?F s
        by (rule ct(2) [symmetric])
      finally show ?case .
    qed
  qed
  finally show integral^N (T (memoryless-on ct s)) (lfp l) ≤ lfp ?F s .
 qed
```

```
  have cont-l: sup-continuous l
   by (auto simp: l-def intro!: order-continuous-intros cont-g[THEN sup-continuous-compose])

  show lfp ?F ≤ (λs. E-inf s (lfp l))
  proof (intro lfp-lowerbound le-funI)
    fix s show (∏x∈K s. ∫+ t. g t (E-inf t (lfp l)) ∂measure-pmf x) ≤ E-inf s (lfp
l)
    proof (rewrite in - ≤ ⌑ E-inf-iterate)
      show l: lfp l ∈ borel-measurable St
        using cont-l by (rule borel-measurable-lfp) (simp add: l-def)
      show (∏D∈K s. ∫+ t. g t (E-inf t (lfp l)) ∂measure-pmf D) ≤
        (∏D∈K s. ∫+ t. E-inf t (λω. lfp l (t ## ω)) ∂measure-pmf D)
      proof (rule INF-mono nn-integral-mono bexI)+
        fix t D assume D ∈ K s
        { fix cfg assume cfg ∈ cfg-on t
          have (∫+ ω. g (state cfg) (lfp l ω) ∂T cfg) = g (state cfg) (∫+ ω. (lfp l
ω) ∂T cfg)
            using l by (rule int-g)
          with ‹cfg ∈ cfg-on t› have ∗: (∫+ ω. g t (lfp l ω) ∂T cfg) = g t (∫+ ω.
(lfp l ω) ∂T cfg)
            by simp }
        then
         have ∗: g t (∏cfg∈cfg-on t. integral^N (T cfg) (lfp l)) ≤ (∏cfg∈cfg-on t.
∫+ ω. g t (lfp l ω) ∂T cfg)
          apply simp
          apply (rule INF-greatest)
          apply (rule sup-continuous-mono[OF cont-g, THEN monoD])
          apply (rule INF-lower)
          apply assumption
          done
        show g t (E-inf t (lfp l)) ≤ E-inf t (λω. lfp l (t ## ω))
          apply (rewrite in - ≤ ⌑ lfp-unfold[OF sup-continuous-mono[OF cont-l]])
          apply (rewrite in - ≤ ⌑ l-def)
          apply (simp add: E-inf-def ∗)
          done
      qed
    qed
  qed
qed

definition P-inf s P = (∏cfg∈cfg-on s. emeasure (T cfg) {x∈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∈space St. P x})
  by (auto simp add: P-inf-def E-inf-def intro!: SUP-cong nn-integral-cong)

lemma P-inf-True[simp]: P-inf t (λω. True) = 1
  using T.emeasure-space-1
```

by (*auto simp add*: *P-inf-def SUP-constant*)

lemma *P-inf-False*[*simp*]: *P-inf t* (λω. *False*) = *0*
by (*auto simp add*: *P-inf-def SUP-constant*)

lemma *P-inf-INF*:
fixes *P* :: *nat* ⇒ *'s stream* ⇒ *bool*
assumes *decseq P* **and** *P*[*measurable*]: ⋀*i*. *Measurable.pred St* (*P i*)
shows *P-inf s* (λ*x*. ∀ *i*. *P i x*) = (∏ *i*. *P-inf s* (*P i*))
proof −
have *P-inf s* (λ*x*. ∏ *i*. *P i x*) = (∏ *cfg*∈*cfg-on s*. *emeasure* (*T cfg*) (⋂ *i*. {*x*∈*space St*. *P i x*}))
by (*auto simp*: *P-inf-def intro*!: *INF-cong arg-cong2*[**where** *f*=*emeasure*])
also have ... = (∏ *cfg*∈*cfg-on s*. ∏ *i*. *emeasure* (*T cfg*) {*x*∈*space St*. *P i x*})
using ‹*decseq P*›
by (*auto intro*!: *INF-cong INF-emeasure-decseq*[*symmetric*]
simp: *decseq-def monotone-def le-fun-def*)
also have ... = (∏ *i*. *P-inf s* (*P i*))
by (*subst INF-commute*) (*simp add*: *P-inf-def*)
finally show *?thesis*
by *simp*
qed

lemma *P-inf-gfp*:
assumes *Q*: *inf-continuous Q*
assumes *f*: *f* ∈ *measurable St M*
assumes *Q-m*: ⋀*P*. *Measurable.pred M P* ⟹ *Measurable.pred M* (*Q P*)
shows *P-inf s* (λ*x*. *gfp Q* (*f x*)) = (∏ *i*. *P-inf s* (λ*x*. (*Q* ^^ *i*) ⊤ (*f x*)))
unfolding *inf-continuous-gfp*[*OF Q*]
apply *simp*
proof (*rule P-inf-INF*)
fix *i* **show** *Measurable.pred St* (λ*x*. (*Q* ^^ *i*) ⊤ (*f x*))
apply (*intro measurable-compose*[*OF f*])
by (*induct i*) (*auto intro*!: *Q-m*)
next
show *decseq* (λ*i x*. (*Q* ^^ *i*) ⊤ (*f x*))
using *inf-continuous-mono*[*OF Q*, *THEN funpow-increasing*[*rotated*]]
unfolding *decseq-def monotone-def le-fun-def* **by** *auto*
qed

lemma *P-inf-iterate*:
assumes [*measurable*]: *Measurable.pred St P*
shows *P-inf s P* = (∏ *D*∈*K s*. ∫⁺ *t*. *P-inf t* (λω. *P* (*t* ## ω)) ∂*measure-pmf D*)
proof −
have [*simp*]: ⋀*x s*. *indicator* {*x* ∈ *space St*. *P x*} (*x* ## *s*) = *indicator* {*s* ∈ *space St*. *P* (*x* ## *s*)} *s*
by (*auto simp*: *space-stream-space split*: *split-indicator*)
show *?thesis*

using *E-inf-iterate*[*of indicator* {*x*∈*space St*. *P x*} *s*] **by** (*auto simp*: *P-inf-eq-E-inf*)
qed

end

5.5 Finite MDPs

locale *Finite-Markov-Decision-Process* = *Markov-Decision-Process K* **for** *K* :: *'s* ⇒ *'s pmf set* +
fixes *S* :: *'s set*
assumes *S-not-empty*: *S* ≠ {}
assumes *S-finite*: *finite S*
assumes *K-closed*: ⋀*s*. *s* ∈ *S* ⟹ (⋃ *D*∈*K s*. *set-pmf D*) ⊆ *S*
assumes *K-finite*: ⋀*s*. *s* ∈ *S* ⟹ *finite* (*K s*)
begin

lemma *action-closed*: *s* ∈ *S* ⟹ *cfg* ∈ *cfg-on s* ⟹ *t* ∈ *action cfg* ⟹ *t* ∈ *S*
using *cfg-onD-action*[*of cfg s*] *K-closed*[*of s*] **by** *auto*

lemma *set-pmf-closed*: *s* ∈ *S* ⟹ *D* ∈ *K s* ⟹ *t* ∈ *D* ⟹ *t* ∈ *S*
using *K-closed* **by** *auto*

lemma *Pi-closed*: *ct* ∈ *Pi S K* ⟹ *s* ∈ *S* ⟹ *t* ∈ *ct s* ⟹ *t* ∈ *S*
using *set-pmf-closed* **by** *auto*

lemma *E-closed*: *s* ∈ *S* ⟹ (*s*, *t*) ∈ *E* ⟹ *t* ∈ *S*
using *K-closed* **by** (*auto simp*: *E-def*)

lemma *set-pmf-finite*: *s* ∈ *S* ⟹ *D* ∈ *K s* ⟹ *finite D*
using *K-closed* **by** (*intro finite-subset*[*OF - S-finite*]) *auto*

definition *valid-cfg* = (⋃ *s*∈*S*. *cfg-on s*)

lemma *valid-cfgI*: *s* ∈ *S* ⟹ *cfg* ∈ *cfg-on s* ⟹ *cfg* ∈ *valid-cfg*
by (*auto simp*: *valid-cfg-def*)

lemma *valid-cfgD*: *cfg* ∈ *valid-cfg* ⟹ *cfg* ∈ *cfg-on* (*state cfg*)
by (*auto simp*: *valid-cfg-def*)

lemma
shows *valid-cfg-state-in-S*: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∈ *S*
and *valid-cfg-action*: *cfg* ∈ *valid-cfg* ⟹ *s* ∈ *action cfg* ⟹ *s* ∈ *S*
and *valid-cfg-cont*: *cfg* ∈ *valid-cfg* ⟹ *s* ∈ *action cfg* ⟹ *cont cfg s* ∈ *valid-cfg*
by (*auto simp*: *valid-cfg-def intro*!: *bexI*[*of - s*] *intro*: *action-closed*)

lemma *valid-K-cfg*[*intro*]: *cfg* ∈ *valid-cfg* ⟹ *cfg'* ∈ *K-cfg cfg* ⟹ *cfg'* ∈ *valid-cfg*
by (*auto simp add*: *K-cfg-def valid-cfg-cont*)

definition *simple ct* = *memoryless-on* (λ*s*. *if s* ∈ *S then ct s else arb-act s*)

lemma *simple-cfg-on*[*simp*]: *ct* ∈ *Pi S K* ⟹ *simple ct s* ∈ *cfg-on s*
by (*auto simp*: *simple-def intro*!: *memoryless-on-cfg-onI*)

lemma *simple-valid-cfg*[*simp*]: *ct* ∈ *Pi S K* ⟹ *s* ∈ *S* ⟹ *simple ct s* ∈ *valid-cfg*
by (*auto intro*: *valid-cfgI*)

lemma *cont-simple*[*simp*]: *s* ∈ *S* ⟹ *t* ∈ *set-pmf* (*ct s*) ⟹ *cont* (*simple ct s*) *t* = *simple ct t*
by (*simp add*: *simple-def*)

lemma *state-simple*[*simp*]: *state* (*simple ct s*) = *s*
by (*simp add*: *simple-def*)

lemma *action-simple*[*simp*]: *s* ∈ *S* ⟹ *action* (*simple ct s*) = *ct s*
by (*simp add*: *simple-def*)

lemma *simple-valid-cfg-iff*: *ct* ∈ *Pi S K* ⟹ *simple ct s* ∈ *valid-cfg* ⟷ *s* ∈ *S*
using *cfg-onD-state*[*of simple ct s*] **by** (*auto simp add*: *valid-cfg-def intro*!: *bexI*[*of* - *s*])

end

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

inductive-set *directed-towards* :: *'a set* ⇒ (*'a* × *'a*) *set* ⇒ *'a set* **for** *A r* **where**
start: ⋀*x*. *x* ∈ *A* ⟹ *x* ∈ *directed-towards A r*
| *step*: ⋀*x y*. *y* ∈ *directed-towards A r* ⟹ (*x*, *y*) ∈ *r* ⟹ *x* ∈ *directed-towards A r*

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

lemma *directed-towards-mono*:
assumes *s* ∈ *directed-towards A F F* ⊆ *G* **shows** *s* ∈ *directed-towards A G*
using *assms* **by** *induct* (*auto intro*: *directed-towards.intros*)

lemma *directed-eq-rtrancl*: *x* ∈ *directed-towards A r* ⟷ (∃ *a*∈*A*. (*x*, *a*) ∈ *r**)
proof
assume *x* ∈ *directed-towards A r* **then show** ∃ *a*∈*A*. (*x*, *a*) ∈ *r**
by *induction* (*auto intro*: *converse-rtrancl-into-rtrancl*)
next
assume ∃ *a*∈*A*. (*x*, *a*) ∈ *r**
then obtain *a* **where** (*x*, *a*) ∈ *r** *a* ∈ *A* **by** *auto*
then show *x* ∈ *directed-towards A r*
by (*induction rule*: *converse-rtrancl-induct*)
(*auto intro*: *directed-towards.start directed-towards.step*)
qed

lemma *directed-eq-rtrancl-Image*: *directed-towards A r* = $(r^*)^{-1}$ `` *A*
unfolding *set-eq-iff directed-eq-rtrancl Image-iff* **by** *simp*

locale *Reachability-Problem* = *Finite-Markov-Decision-Process K S* **for** *K* :: *'s* ⇒ *'s pmf set* **and** *S* +
fixes *S1 S2* :: *'s set*
assumes *S1*: *S1* ⊆ *S* **and** *S2*: *S2* ⊆ *S* **and** *S1-S2*: *S1* ∩ *S2* = {}
begin

lemma [*measurable*]:
S ∈ *sets* (*count-space UNIV*) *S1* ∈ *sets* (*count-space UNIV*) *S2* ∈ *sets* (*count-space UNIV*)
by *auto*

definition
v = (λ*cfg*∈*valid-cfg*. *emeasure* (*T cfg*) {*x*∈*space St*. (*HLD S1 suntil HLD S2*) (*state cfg ## x*)})

lemma *v-eq*: *cfg* ∈ *valid-cfg* ⟹
v cfg = *emeasure* (*T cfg*) {*x*∈*space St*. (*HLD S1 suntil HLD S2*) (*state cfg ## x*)}
by (*auto simp add*: *v-def*)

lemma *real-v*: *cfg* ∈ *valid-cfg* ⟹ *enn2real* (*v cfg*) = $\mathcal{P}$(ω *in T cfg*. (*HLD S1 suntil HLD S2*) (*state cfg ## ω*))
by (*auto simp add*: *v-def T.emeasure-eq-measure*)

lemma *v-le-1*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* ≤ *1*
by (*auto simp add*: *v-def T.emeasure-eq-measure*)

lemma *v-neq-Pinf*[*simp*]: *cfg* ∈ *valid-cfg* ⟹ *v cfg* ≠ *top*
by (*auto simp add*: *v-def*)

lemma *v-1-AE*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* = *1* ⟷ (*AE ω in T cfg*. (*HLD S1 suntil HLD S2*) (*state cfg ## ω*))
unfolding *v-eq T.emeasure-eq-measure ennreal-eq-1 space-T*[*symmetric*, *of cfg*]
by (*rule T.prob-Collect-eq-1*) *simp*

lemma *v-0-AE*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* = *0* ⟷ (*AE x in T cfg*. *not* (*HLD S1 suntil HLD S2*) (*state cfg ## x*))
unfolding *v-eq T.emeasure-eq-measure space-T*[*symmetric*, *of cfg*] *ennreal-eq-zero-iff*[*OF measure-nonneg*]
by (*rule T.prob-Collect-eq-0*) *simp*

lemma *v-S2*[*simp*]: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∈ *S2* ⟹ *v cfg* = *1*
using *S2* **by** (*subst v-1-AE*) (*auto simp*: *suntil-Stream*)

lemma *v-nS12*[*simp*]: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∉ *S1* ⟹ *state cfg* ∉ *S2* ⟹ *v*

```
  cfg = 0
  by (subst v-0-AE) (auto simp: suntil-Stream)

lemma v-nS[simp]: cfg ∉ valid-cfg ⟹ v cfg = undefined
  by (auto simp add: v-def)

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 −
  have [simp]: state cfg ∉ S2
    using cfg-S1 S1-S2 S1 by blast
  show ?thesis
     by (auto simp: v-eq emeasure-Collect-T[of - cfg] K-cfg-def map-pmf-rep-eq
nn-integral-distr
                  AE-measure-pmf-iff suntil-Stream[of - - state cfg]
                  valid-cfg-cont
           intro!: nn-integral-cong-AE)
qed

lemma real-v-integrable:
  integrable (action cfg) (λs. enn2real (v (cont cfg s)))
  by (rule measure-pmf.integrable-const-bound[where B=max 1 (enn2real unde-
fined)])
     (auto simp add: v-def measure-def[symmetric] le-max-iff-disj)

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
 by (subst integral-eq-nn-integral)
    (auto simp: AE-measure-pmf-iff v-eq T.emeasure-eq-measure valid-cfg-cont
          intro!: arg-cong[where f=enn2real] nn-integral-cong-AE)

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 −
  from ∗ valid[OF ∗]
  have AE x in MC-syntax.T K-cfg cfg. ¬ (HLD S1 suntil HLD S2) (state cfg ##
smap state x)
    unfolding stream.map[symmetric] suntil-smap hld-smap'
  proof (coinduction arbitrary: cfg rule: MC.AE-not-suntil-coinduct-strong)
    case (ψ cfg) then show ?case
      by (auto simp del: cfg-onD-state dest: nS2)
```

next
case (φ *cfg'* *cfg*)
then have $*$: *P cfg state cfg* $\in$ *S1 cfg'* $\in$ *K-cfg cfg* **and** [*simp*, *intro*]: *cfg* $\in$ *valid-cfg*
by *auto*
with *cont*[*OF* $*$] **show** *?case*
by (*subst* (*asm*) *v-0-AE*)
(*auto simp*: *suntil-Stream T-def AE-distr-iff suntil-smap hld-smap' cong del*: *AE-cong*)
qed
then have *AE* ω *in T cfg.* $\neg$ (*HLD S1 suntil HLD S2*) (*state cfg* ## ω)
unfolding *T-def* **by** (*subst AE-distr-iff*) *simp-all*
with *valid*[*OF* $*$] **show** *?thesis*
by (*simp add*: *v-0-AE*)
qed

definition $p = (\lambda s{\in}S.$ *P-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-on s*)
by (*auto simp add*: *p-def v-def P-sup-def T.emeasure-eq-measure intro*: *valid-cfgI intro*!: *SUP-cong cong*: *SUP-cong-simp*)

lemma *v-le-p*: *cfg* $\in$ *valid-cfg* $\Longrightarrow$ *v cfg* $\leq$ *p* (*state cfg*)
by (*subst p-eq-SUP-v*) (*auto intro*!: *SUP-upper dest*: *valid-cfgD valid-cfg-state-in-S*)

lemma *p-eq-0-imp*: *cfg* $\in$ *valid-cfg* $\Longrightarrow$ *p* (*state cfg*) = *0* $\Longrightarrow$ *v cfg* = *0*
using *v-le-p*[*of cfg*] **by** (*auto intro*: *antisym*)

lemma *p-eq-0-iff*: $s \in S \Longrightarrow p\ s = 0 \longleftrightarrow (\forall\, cfg{\in}cfg\text{-}on\ s.\ v\ cfg = 0)$
unfolding *p-eq-SUP-v* **by** (*subst SUP-eq-iff*) *auto*

lemma *p-le-1*: $s \in S \Longrightarrow p\ s \leq 1$
by (*auto simp*: *p-eq-SUP-v intro*!: *SUP-least v-le-1 intro*: *valid-cfgI*)

lemma *p-undefined*[*simp*]: $s \notin S \Longrightarrow p\ s =$ *undefined*
by (*simp add*: *p-def*)

lemma *p-not-inf*[*simp*]: $s \in S \Longrightarrow p\ s \neq$ *top*
using *p-le-1*[*of s*] **by** (*auto simp*: *top-unique*)

lemma *p-S1*: $s \in S1 \Longrightarrow p\ s = (\bigsqcup D{\in}K\ s.\ \int^+ t.\ p\ t\ \partial measure\text{-}pmf\ D)$
using *S1 S1-S2 K-closed*[*of s*] **unfolding** *p-def*
by (*simp add*: *P-sup-iterate*[*of - s*] *subset-eq set-eq-iff suntil-Stream*[*of - - s*])
(*auto intro*!: *SUP-cong nn-integral-cong-AE simp add*: *AE-measure-pmf-iff*)

lemma *p-S2*[*simp*]: $s \in S2 \Longrightarrow p\ s = 1$
using *S2* **by** (*auto simp*: *v-S2*[*OF valid-cfgI*] *p-eq-SUP-v*)

lemma *p-nS12*: $s \in S \Longrightarrow s \notin S1 \Longrightarrow s \notin S2 \Longrightarrow p\ s = 0$
by (*auto simp*: *p-eq-SUP-v v-nS12*[*OF valid-cfgI*])

lemma *p-pos*:
assumes $(s, t) \in (SIGMA\ s{:}S1.\ \bigcup D{\in}K\ s.\ set\text{-}pmf\ D)^*$ $t \in S2$ **shows** $0 < p\ s$
using *assms* **proof** (*induction rule*: *converse-rtrancl-induct*)
case (*step s t'*)
then obtain *D* **where** $s \in S1\ D \in K\ s\ t' \in D\ 0 < p\ t'$
by *auto*
with *S1 set-pmf-closed*[*of s D*] **have** *in-S*: $\bigwedge t.\ t \in D \Longrightarrow t \in S$
by *auto*
from ‹$t' \in D$› ‹$0 < p\ t'$› **have** $0 < pmf\ D\ t' * p\ t'$
by (*auto simp add*: *ennreal-zero-less-mult-iff pmf-positive*)
also have ... $\leq (\int{}^+ t.\ p\ t' * indicator\ \{t'\}\ t \partial D)$
using *in-S*[*OF* ‹$t' \in D$›]
by (*subst nn-integral-cmult-indicator*) (*auto simp*: *ac-simps emeasure-pmf-single*)
also have ... $\leq (\int{}^+ t.\ p\ t\ \partial D)$
by (*auto intro*!: *nn-integral-mono-AE split*: *split-indicator simp*: *in-S AE-measure-pmf-iff*
simp del: *nn-integral-indicator-singleton*)
also have ... $\leq p\ s$
using ‹$s \in S1$› ‹$D \in K\ s$› **by** (*auto intro*: *SUP-upper simp add*: *p-S1*)
finally show *?case* **.**
qed *simp*

definition *F-sup* :: $('s \Rightarrow ennreal) \Rightarrow 's \Rightarrow ennreal$ **where**
F-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*: $(\bigwedge s.\ s \in S \Longrightarrow f\ s = g\ s) \Longrightarrow F\text{-}sup\ f\ s = F\text{-}sup\ g\ s$
using *K-closed*[*of s*]
by (*auto simp*: *F-sup-def AE-measure-pmf-iff subset-eq*
intro!: *SUP-cong nn-integral-cong-AE*)

lemma *continuous-F-sup*: *sup-continuous F-sup*
unfolding *sup-continuous-def fun-eq-iff F-sup-def*[*abs-def*]
by (*auto simp*: *SUP-apply*[*abs-def*] *nn-integral-monotone-convergence-SUP intro*:
SUP-commute)

lemma *mono-F-sup*: *mono F-sup*
by (*intro sup-continuous-mono continuous-F-sup*)

lemma *lfp-F-sup-iterate*: *lfp F-sup* = $(SUP\ i.\ (F\text{-}sup\ \widehat{}\widehat{}\ i)\ (\lambda x{\in}S.\ 0))$
proof −
{ have $(SUP\ i.\ (F\text{-}sup\ \widehat{}\widehat{}\ i)\ \bot) = (SUP\ i.\ (F\text{-}sup\ \widehat{}\widehat{}\ i)\ (\lambda x{\in}S.\ 0))$
proof (*rule SUP-eq*)
fix *i* **show** $\exists j{\in}UNIV.\ (F\text{-}sup\ \widehat{}\widehat{}\ i)\ \bot \leq (F\text{-}sup\ \widehat{}\widehat{}\ j)\ (\lambda x{\in}S.\ 0)$
by (*intro bexI*[*of - i*] *funpow-mono mono-F-sup*) *auto*
have $*$: $(\lambda x{\in}S.\ 0) \leq F\text{-}sup\ \bot$
using *K-wf* **by** (*auto simp*: *F-sup-def le-fun-def*)

```
    show ∃j∈UNIV. (F-sup ^^ i) (λx∈S. 0) ≤ (F-sup ^^ j) ⊥
      by (auto intro!: exI[of - Suc i] funpow-mono mono-F-sup ∗
            simp del: funpow.simps simp add: funpow-Suc-right le-funI)
   qed }
  then show ?thesis
   by (auto simp: sup-continuous-lfp continuous-F-sup)
qed

lemma p-eq-lfp-F-sup: p = lfp F-sup
proof −
  { fix s assume s ∈ S let ?F = λP. HLD S2 or (HLD S1 aand nxt P)
    have P-sup s (λω. (HLD S1 suntil HLD S2) (s ## ω)) = (⨆i. P-sup s (λω.
(?F ^^ i) ⊥ (s ## ω)))
    proof (simp add: suntil-def, rule P-sup-lfp)
      show (##) s ∈ measurable St St
       by simp

      fix P assume P: Measurable.pred St P
      show Measurable.pred St (HLD S2 or (HLD S1 aand (λω. P (stl ω))))
     by (intro pred-intros-logic measurable-compose[OF - P] measurable-compose[OF
measurable-shd]) auto
    qed (auto simp: sup-continuous-def)
    also have ... = (SUP i. (F-sup ^^ i) (λx∈S. 0) s)
    proof (rule SUP-cong)
      fix i from ‹s ∈ S› show P-sup s (λω. (?F ^^ i) ⊥ (s##ω)) = (F-sup ^^ i)
(λx∈S. 0) s
      proof (induct i arbitrary: s)
       case (Suc n) show ?case
       proof (subst P-sup-iterate)

         show Measurable.pred St (λω. (?F ^^ Suc n) ⊥ (s ## ω))
            apply (intro measurable-compose[OF measurable-Stream[OF measur-
able-const measurable-ident-sets[OF refl]] measurable-predpow])
           apply simp
           apply (simp add: bot-fun-def[abs-def])
             apply (intro pred-intros-logic measurable-compose[OF measurable-stl]
measurable-compose[OF measurable-shd])
           apply auto
           done
       next
         show (⨆D∈K s. ∫+ t. P-sup t (λω. (?F ^^ Suc n) ⊥ (s ## t ## ω))
∂measure-pmf D) =
           (F-sup ^^ Suc n) (λx∈S. 0) s
           unfolding funpow.simps comp-def
           using S1 S2 ‹s ∈ S›
           by (subst F-sup-cong[OF Suc(1)[symmetric]])
              (auto simp add: F-sup-def measure-pmf.emeasure-space-1[simplified]
K-wf subset-eq)
       qed
```

qed *simp*
qed *simp*
finally have *lfp F-sup s = P-sup s* (λω. (*HLD S1 suntil HLD S2*) (*s ## ω*))
by (*simp add*: *lfp-F-sup-iterate image-comp*) **}**
moreover have ⋀*s*. *s* ∉ *S* ⟹ *lfp F-sup s = undefined*
by (*subst lfp-unfold*[*OF mono-F-sup*]) (*auto simp add*: *F-sup-def*)
ultimately show *?thesis*
by (*auto simp*: *p-def*)
qed

definition $S_e = \{s{\in}S.\ p\ s = 0\}$

lemma S_e: $S_e \subseteq S$
by (*auto simp add*: S_e*-def*)

lemma *v-*S_e: *cfg* ∈ *valid-cfg* ⟹ *state cfg* ∈ S_e ⟹ *v cfg = 0*
using *p-eq-0-imp*[*of cfg*] **by** (*auto simp*: S_e*-def*)

lemma S_e*-nS2*: $S_e \cap S2 = \{\}$
by (*auto simp*: S_e*-def*)

lemma S_e*-E1*: *s* ∈ S_e ∩ *S1* ⟹ (*s*, *t*) ∈ *E* ⟹ *t* ∈ S_e
unfolding S_e*-def* **using** *S1*
by (*auto simp*: *p-S1 SUP-eq-iff K-wf nn-integral-0-iff-AE AE-measure-pmf-iff E-def*
intro: *set-pmf-closed antisym*
cong: *rev-conj-cong*)

lemma S_e*-E2*: *s* ∈ *S1* ⟹ (⋀*t*. (*s*, *t*) ∈ *E* ⟹ *t* ∈ S_e) ⟹ *s* ∈ S_e
unfolding S_e*-def* **using** *S1 S1-S2*
by (*force simp*: *p-S1 SUP-eq-iff K-wf nn-integral-0-iff-AE AE-measure-pmf-iff E-def*
cong: *rev-conj-cong*)

lemma S_e*-E-iff*: *s* ∈ *S1* ⟹ *s* ∈ S_e ⟷ (∀ *t*. (*s*, *t*) ∈ *E* ⟶ *t* ∈ S_e)
using S_e*-E1*[*of s*] S_e*-E2*[*of s*] **by** *blast*

definition $S_r = S - (S_e \cup S2)$

lemma S_r: $S_r \subseteq S$
by (*auto simp*: S_r*-def*)

lemma S_r*-S1*: $S_r \subseteq S1$
by (*auto simp*: *p-nS12* S_r*-def* S_e*-def*)

lemma S_r*-eq*: $S_r = S1 - S_e$
using *S1-S2 S1 S2* **by** (*auto simp add*: S_r*-def* S_e*-def p-nS12*)

lemma *v-neq-0-imp*: *cfg* ∈ *valid-cfg* ⟹ *v cfg* ≠ *0* ⟹ *state cfg* ∈ $S_r \cup S2$

using *p-eq-0-imp*[*of cfg*] **by** (*auto simp add*: S_r*-def* S_e*-def valid-cfg-state-in-S*)

lemma *valid-cfg-action-in-K*: *cfg* ∈ *valid-cfg* ⟹ *action cfg* ∈ *K* (*state cfg*)
by (*auto dest*!: *valid-cfgD*)

lemma *K-cfg-E*: *cfg* ∈ *valid-cfg* ⟹ *cfg'* ∈ *K-cfg cfg* ⟹ (*state cfg*, *state cfg'*) ∈ *E*
by (*auto simp*: *E-def K-cfg-def valid-cfg-action-in-K*)

lemma S_r*-directed-towards-S2*:
assumes *s*: *s* ∈ S_r
shows *s* ∈ *directed-towards S2* {(*s*, *t*) | *s t*. *s* ∈ S_r ∧ (*s*, *t*) ∈ *E*} (**is** *s* ∈ *?D*)
proof −
{ fix *cfg* **assume** *s* ∉ *?D cfg* ∈ *cfg-on s*
with *s* S_r **have** *state cfg* ∈ S_r *state cfg* ∉ *?D cfg* ∈ *valid-cfg*
by (*auto intro*: *valid-cfgI*)
then have *v cfg* = *0*
proof (*coinduction arbitrary*: *cfg rule*: *v-eq-0-coinduct*)
case (*cont cfg' cfg*)
with *v-neq-0-imp*[*of cfg'*] **show** *?case*
by (*auto intro*: *directed-towards.intros K-cfg-E*)
qed (*auto intro*: *directed-towards.intros*) **}**
with *p-eq-0-iff*[*of s*] *s* **show** *?thesis*
unfolding S_r*-def* S_e*-def* **by** *blast*
qed

definition *proper ct* ⟷ *ct* ∈ Pi_E *S K* ∧ (∀ *s*∈S_r. *v* (*simple ct s*) > *0*)

lemma S_r*-nS2*: *s* ∈ S_r ⟹ *s* ∉ *S2*
by (*auto simp*: S_r*-def*)

lemma *properD1*: *proper ct* ⟹ *ct* ∈ Pi_E *S K*
by (*auto simp*: *proper-def*)

lemma *proper-eq*:
assumes *ct*[*simp*, *intro*]: *ct* ∈ Pi_E *S K*
shows *proper ct* ⟷ S_r ⊆ *directed-towards S2* (*SIGMA s*:S_r. *ct s*)
(**is** - ⟷ - ⊆ *?D*)
proof −
have ∗[*simp*]: ⋀*s*. *s* ∈ S_r ⟹ *s* ∈ *S* **and** *ct'*: *ct* ∈ *Pi S K*
using *ct* **by** (*auto simp*: S_r*-def simp del*: *ct*)
{ fix *s t* **have** *s* ∈ *S* ⟹ *t* ∈ *ct s* ⟹ *t* ∈ *S*
using *K-closed*[*of s*] *ct'* **by** (*auto simp add*: *subset-eq*) **}**
note *ct-closed* = *this*

let *?C* = *simple ct*
from *ct* **have** *valid-C*[*simp*]: ⋀*s*. *s* ∈ *S* ⟹ *?C s* ∈ *valid-cfg*
by (*auto simp add*: *PiE-def*)
{ fix *s* **assume** *s* ∈ *?D*

```
    then have 0 < v (?C s)
    proof induct
      case (step s t)
      then have s: s ∈ S_r and t: t ∈ ct s and [simp]: s ∈ S
        by auto
      with S_r-S1 ct have v (?C s) = (∫⁺t. v (?C t) ∂ct s)
        by (subst v-S1) (auto intro!: nn-integral-cong-AE AE-pmfI)
      also have ... ≠ 0
        using ct t step
      by (subst nn-integral-0-iff-AE) (auto simp add: AE-measure-pmf-iff zero-less-iff-neq-zero)
      finally show ?case
        using ct by (auto simp add: less-le)
    qed (subst v-S2, insert S2, auto) }
  moreover
  { fix s assume s: s ∉ ?D s ∈ S_r
    with ct' have C: ?C s ∈ cfg-on s and [simp]: s ∈ S
      by auto
    from s have v (?C s) = 0
    proof (coinduction arbitrary: s rule: v-eq-0-coinduct)
      case (cont cfg s)
      with S1 obtain t where cfg = ?C t t ∈ ct s s ∈ S
        by (auto simp: set-K-cfg subset-eq)
      with cont(1,2) v-neq-0-imp[of ?C t] ct-closed[of s t] show ?case
        by (intro exI[of - t] disjCI) (auto intro: directed-towards.intros)
    qed (auto simp: S_r-nS2) }
  ultimately show ?thesis
    unfolding proper-def using ct by (force simp del: v-nS v-S2 v-nS12 ct)
qed

lemma exists-proper:
  obtains ct where proper ct
proof atomize-elim
  define r where r = rec-nat S2 (λ- S'. {s∈S_r. ∃ t∈S'. (s, t) ∈ E})
  then have [simp]: r 0 = S2 ⋀n. r (Suc n) = {s∈S_r. ∃ t∈r n. (s, t) ∈ E}
    by simp-all

  { fix s assume s ∈ S_r
    then have s ∈ directed-towards S2 {(s, t) | s t. s ∈ S_r ∧ (s, t) ∈ E}
      by (rule S_r-directed-towards-S2)
    from this ‹s∈S_r› have ∃ n. s ∈ r n
    proof induction
      case (step s t)
      show ?case
      proof cases
        assume t ∈ S2 with step.prems step.hyps show ?thesis
          by (intro exI[of - Suc 0]) force
      next
        assume t ∉ S2
        with step obtain n where t ∈ r n t ∈ S_r
```

```
        by (auto elim: directed-towards.cases)
      with ‹t∈S_r› step.hyps show ?thesis
        by (intro exI[of - Suc n]) force
    qed
  qed (simp add: S_r-def) }
 note r = this

 { fix s assume s ∈ S
  have ∃ D∈K s. s ∈ S_r ⟶ (∃ t∈D. ∃ n. t ∈ r n ∧ (∀ m. s ∈ r m ⟶ n < m))
  proof cases
    assume s: s ∈ S_r
    define n where n = (LEAST n. s ∈ r n)
    then have s ∈ r n and n: ⋀i. i < n ⟹ s ∉ r i
      using r s by (auto intro: LeastI-ex dest: not-less-Least)
    with s have n ≠ 0
      by (intro notI) (auto simp: S_r-def)
    then obtain n' where n = Suc n'
      by (cases n) auto
    with ‹s ∈ r n› obtain t D where D ∈ K s t ∈ D t ∈ r n'
      by (auto simp: E-def)
    with n ‹n = Suc n'› s show ?thesis
    by (auto intro!: bexI[of - D] bexI[of - t] exI[of - n'] simp: not-less-eq[symmetric])
  qed (insert K-wf ‹s∈S›, auto) }
 then obtain ct where ct: ⋀s. s ∈ S ⟹ ct s ∈ K s
  ⋀s. s ∈ S ⟹ s ∈ S_r ⟹ ∃ t∈ct s. ∃ n. t ∈ r n ∧ (∀ m. s ∈ r m ⟶ n < m)
  by metis
 then have ∗: restrict ct S ∈ Pi_E S K
  by auto

 moreover
 { fix s assume s ∈ S_r
  then obtain n where s ∈ r n
    by (metis r)
  with ‹s ∈ S_r› have s ∈ directed-towards S2 (SIGMA s : S_r. ct s)
  proof (induction n arbitrary: s rule: less-induct)
    case (less n s)
    moreover with S_r have s ∈ S by auto
    ultimately obtain t m where t ∈ ct s t ∈ r m m < n
      using ct[of s] by (auto simp: E-def)
    with less.IH[of m t] ‹s ∈ S_r› show ?case
      by (cases m) (auto intro: directed-towards.intros)
  qed }

 ultimately show ∃ ct. proper ct
  using S_r S2
  by (auto simp: proper-eq[OF ∗] subset-eq
        intro!: exI[of - restrict ct S]
        cong: Sigma-cong)
qed
```

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* (λ*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 −
have *ct-closed*: $\bigwedge$*s t*. *s* $\in$ *S* $\Longrightarrow$ *t* $\in$ *ct s* $\Longrightarrow$ *t* $\in$ *S*
using *ct K-closed* **by** (*auto simp*: *proper-def PiE-iff*)
have *ct-Pi*: *ct* $\in$ *Pi S K*
using *ct* **by** (*auto simp*: *proper-def*)

have *finite* S_r
using *S-finite* **by** (*auto simp*: S_r*-def*)
then show *?thesis*
proof (*induct rule*: *finite-induct-select*)
case (*select X*)
then obtain *l* **where** *l*: *l* $\in$ *X* $\to$ *X* $\cup$ *S2* **and** *desc*: $\bigwedge$*s*. *s* $\in$ *X* $\Longrightarrow$ *l-desc X ct l s*
by *auto*
obtain *x* **where** *x*: *x* $\in$ S_r − *X*
using ‹*X* $\subset$ S_r› **by** *auto*
then have *x* $\in$ *S*
by (*auto simp*: S_r*-def*)

let *?C* = *simple ct*
let *?v* = λ*s*. *v* (*?C s*) **and** *?E* = λ*s*. *set-pmf* (*ct s*)
let *?M* = λ*s*. *maximal ?v* (*?E s*)

have *finite-E*[*simp*]: $\bigwedge$*s*. *s* $\in$ *S* $\Longrightarrow$ *finite* (*?E s*)
using *K-closed ct* **by** (*intro finite-subset*[*OF - S-finite*]) (*auto simp*: *proper-def subset-eq*)

have *valid-C*[*simp*]: $\bigwedge$*s*. *s* $\in$ *S* $\Longrightarrow$ *?C s* $\in$ *valid-cfg*
using *ct* **by** (*auto simp*: *proper-def intro*!: *simple-valid-cfg*)

have *E-ne*[*simp*]: $\bigwedge$*s*. *?E s* $\neq$ {}
by (*rule set-pmf-not-empty*)

have $\exists\, s{\in}S_r - X.\ \exists\, t{\in}$*?M s*. *t* $\in$ *S2* $\cup$ *X*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *not-M*: $\bigwedge$*s*. *s* $\in$ S_r − *X* $\Longrightarrow$ *?M s* $\cap$ (*S2* $\cup$ *X*) = {}
by *auto*

let *?*S_m = *maximal ?v* (S_r − *X*)

have *finite* $(S_r - X)$ $S_r - X \neq \{\}$
  using ‹$X \subset S_r$› **by** (*auto intro*!: *finite-subset*[*OF - S-finite*] *simp*: S_r*-def*)
from *maximal-ne*[*OF this*] **obtain** s_m **where** s_m: $s_m \in ?S_m$
  by *force*

have $\exists\, s_0{\in}?S_m.\ \exists\, t{\in}?E\ s_0.\ t \notin ?S_m$
proof (*rule ccontr*)
  assume ¬ *?thesis*
  then have S_m: $\bigwedge s_0\ t.\ s_0 \in ?S_m \Longrightarrow t \in ?E\ s_0 \Longrightarrow t \in ?S_m$ **by** *blast*
  from ‹$s_m \in ?S_m$› **have** [*simp*]: $s_m \in S$ **and** $s_m \in S_r$
    by (*auto simp*: S_r*-def dest*: *maximalD1*)

  from ‹$s_m \in ?S_m$› **have** $v\ (?C\ s_m) = 0$
  proof (*coinduction arbitrary*: s_m *rule*: *v-eq-0-coinduct*)
    case (*cont t* s_m) **with** *S1* **show** *?case*
      by (*intro exI*[*of - state t*] *disjCI conjI* S_m[*of* s_m *state t*])
        (*auto simp*: *set-K-cfg*)
  qed (*auto simp*: S_r*-def ct-Pi dest*!: *maximalD1*)
  with ‹$s_m \in S_r$› ‹*proper ct*› **show** *False*
    by (*auto simp*: *proper-def*)
qed
then obtain s_0 t **where** $s_0 \in ?S_m$ **and** t: $t \in ?E\ s_0\ t \notin ?S_m$
  by *metis*
with S_r*-S1* **have** s_0: $s_0 \in S_r - X$ **and** [*simp*]: $s_0 \in S$ **and** $s_0 \in S1$
  by (*auto simp*: S_r*-def dest*: *maximalD1*)

from ‹*proper ct*› ‹$s_0 \in S$› s_0 **have** *?v* $s_0 \neq 0$
  by (*auto simp add*: *proper-def*)
then have $0 <$ *?v* s_0 **by** (*simp add*: *zero-less-iff-neq-zero*)

{ fix t **assume** $t \in S_e \cup S2 \cup X$ $t \in ?E\ s_0$ **and** *?v* $s_0 \leq$ *?v* t
  moreover have $t \in S_e \Longrightarrow$ *?v* $t = 0$
    by (*simp add*: *p-eq-0-imp* S_e*-def ct-Pi*)
  ultimately have t: $t \in S2 \cup X$ $t \in ?E\ s_0$
    using ‹$0 <$ *?v* s_0› **by** (*auto simp*: S_e*-def*)

  have *maximal ?v* (*?E* $s_0 \cap (S2 \cup X)) \neq \{\}$
    using *finite-E t* **by** (*intro maximal-ne*) *auto*
  moreover
  { fix x y **assume** x: $x \in S2 \cup X$ $x \in ?E\ s_0$
      and ∗: $\forall\, y{\in}?E\ s_0 \cap (S2 \cup X).$ *?v* $y \leq$ *?v* x **and** y: $y \in ?E\ s_0$
    with *S2* ‹$s_0 \in S$›[*THEN ct-closed*] **have** [*simp*]: $x \in S$ $y \in S$
      by *auto*

    have *?v* $y \leq$ *?v* x
    proof *cases*
      assume $y \in S_r - X$
      then have *?v* $y \leq$ *?v* s_0

using ‹$s_0 \in ?S_m$› **by** (*auto intro*: *maximalD2*)
also note ‹*?v* $s_0 \leq$ *?v* t›
also have *?v* $t \leq$ *?v* x
using $*$ t **by** *auto*
finally show *?thesis* **.**
next
assume $y \notin S_r - X$ **with** $y *$ **show** *?thesis*
by (*auto simp*: S_r*-def v-*S_e[*of ?C y*] *ct-Pi*)
qed **}**
then have *maximal ?v* (*?E* $s_0 \cap$ (*S2* $\cup$ *X*)) $\subseteq$ *maximal ?v* (*?E* s_0)
by (*auto simp*: *maximal-def*)
moreover note *not-M*[*OF* s_0]
ultimately have *False*
by (*blast dest*: *maximalD1*) **}**
then have *less-*s_0: $\bigwedge t.\ t \in S_e \cup S2 \cup X \Longrightarrow t \in ?E\ s_0 \Longrightarrow ?v\ t < ?v\ s_0$
by (*auto simp add*: *not-le*[*symmetric*])

let *?K* = *ct* s_0

have *?v* $s_0 = (\int^+ x.\ ?v\ x\ \partial ?K)$
using *v-S1*[*of ?C* s_0] ‹$s_0 \in S1$› ‹$s_0 \in S$›
by (*auto simp add*: *ct-Pi intro*!: *nn-integral-cong-AE AE-pmfI*)
also have $\ldots < (\int^+ x.\ ?v\ s_0\ \partial ?K)$
proof (*intro nn-integral-less*)
have $(\int^+ x.\ ?v\ x\ \partial ?K) \leq (\int^+ x.\ 1\ \partial ?K)$
using *ct ct-closed*[*of* s_0]
by (*intro nn-integral-mono-AE*)
(*auto intro*!: *v-le-1 simp*: *AE-measure-pmf-iff proper-def ct-Pi*)
then show $(\int^+ x.\ ?v\ x\ \partial ?K) \neq \infty$
by (*auto simp*: *top-unique*)
have *?v* $t <$ *?v* s_0
proof *cases*
assume $t \in S_e \cup S2 \cup X$ **then show** *?thesis*
using *less-*s_0[*of t*] t **by** *simp*
next
assume $t \notin S_e \cup S2 \cup X$
with *t*(*1*) *ct-closed*[*of* s_0 t] **have** $t \in S_r - X$
unfolding S_r*-def* **by** (*auto simp*: *E-def*)
with *t*(*2*) **show** *?thesis*
using ‹$s_0 \in ?S_m$› **by** (*auto simp*: *maximal-def not-le intro*: *less-le-trans*)
qed
then show $\neg$ (*AE x in ?K.* *?v* $s_0 \leq$ *?v* x)
using t **by** (*auto simp*: *not-le AE-measure-pmf-iff E-def cong del*: *AE-cong intro*!: *exI*[*of - t*])

show *AE x in ?K.* *?v* $x \leq$ *?v* s_0
proof (*subst AE-measure-pmf-iff*, *safe*)
fix t **assume** *t*: $t \in ?E\ s_0$
show *?v* $t \leq$ *?v* s_0

proof *cases*
assume $t \in S_e \cup S2 \cup X$ **then show** *?thesis*
using *less-*s_0[*of t*] t **by** *simp*
next
assume $t \notin S_e \cup S2 \cup X$ **with** t ‹$s_0 \in ?S_m$› ‹$s_0 \in S$› **show** *?thesis*
by (*elim maximalD2*) (*auto simp*: S_r*-def intro*!: *ct-closed*[*of - t*])
qed
qed
qed (*insert ct-closed*[*of* s_0], *auto simp*: *AE-measure-pmf-iff*)
also have ... = *?v* s_0
using ‹$s_0 \in S$› *measure-pmf.emeasure-space-1*[*of ct* s_0] **by** *simp*
finally show *False*
by *simp*
qed
then obtain s t **where** s: $s \in S_r - X$ **and** t: $t \in S2 \cup X$ $t \in ?M\ s$
by *auto*
with *S2* ‹$X \subset S_r$› **have** $s \notin S2$ **and** $s \in S \wedge s \notin S2$ **and** $s \notin X$**and** [*simp*]: $t \in S$
by (*auto simp add*: S_r*-def*)
define l' **where** $l' = l(s := t)$
then have l'*-s*[*simp*, *intro*]: $l'\ s = t$
by *simp*

let *?D* = $\lambda X\ l.$ *directed-towards S2* (*SIGMA* $s : X.$ $\{l\ s\}$)
{ fix s' **assume** $s' \in ?D\ X\ l\ s' \in X$
from *this*(*1*) **have** $s' \in ?D$ (*insert* s X) l'
by (*rule directed-towards-mono*) (*auto simp*: l'*-def* ‹$s \notin X$›) **}**
note *directed-towards-*l' = *this*

show *?case*
proof (*intro bexI ballI*, *elim insertE*)
show $s \in S_r - X$ **by** *fact*
show $l' \in$ *insert* s $X \to$ *insert* s $X \cup S2$
using s t l **by** (*auto simp*: l'*-def*)
next
fix s' **assume** s': $s' \in X$
moreover
from *desc*[*OF* s'] **have** $s' \in ?D\ X\ l$ **and** $*$: $?v\ s' \leq ?v\ (l\ s')\ l\ s' \in ?M\ s'$
by (*auto simp*: *l-desc-def*)
moreover have $l'\ s' = l\ s'$
using ‹$s' \in X$› s **by** (*auto simp add*: l'*-def*)
ultimately show *l-desc* (*insert* s X) *ct* l' s'
by (*auto simp*: *l-desc-def intro*!: *directed-towards-*l')
next
fix s' **assume** $s' = s$
show *l-desc* (*insert* s X) *ct* l' s'
unfolding ‹$s' = s$› *l-desc-def* l'*-s*
proof (*intro conjI*)
show $s \in ?D$ (*insert* s X) l'

```
      proof cases
        assume t ∉ S2
        with t have t ∈ X by auto
        with desc have t ∈ ?D X l
          by (simp add: l-desc-def)
        then show ?thesis
          by (force intro: directed-towards.step[OF directed-towards-l'] ‹t ∈ X›)
      qed (force intro: directed-towards.step directed-towards.start)

      from ‹s ∈ S_r - X› S_r-S1 have [simp]: s ∈ S1 s ∈ S
        by (auto simp: S_r-def)
      show ?v s ≤ ?v t
        using t(2)[THEN maximalD2] ct
        by (auto simp add: v-S1 AE-measure-pmf-iff proper-def Pi-iff PiE-def
              intro!: measure-pmf.nn-integral-le-const)
    qed fact
  qed
 qed simp
qed

lemma F-v-memoryless:
  obtains ct where ct ∈ Pi_E S K v∘simple ct = F-sup (v∘simple ct)
proof atomize-elim
  define R where R = {(ct(s := D), ct) | ct s D.
    ct ∈ Pi_E S K ∧ proper ct ∧ s ∈ S_r ∧ D ∈ K s ∧ v (simple ct s) < (∫⁺t. v
(simple ct t) ∂D) }

  { fix ct ct' assume ct-ct': (ct', ct) ∈ R
    let ?v = λs. v (simple ct s) and ?v' = λs. v (simple ct' s)

    from ct-ct' obtain s D where ct ∈ Pi_E S K proper ct and s: s ∈ S_r and D:
D ∈ K s
      and not-maximal: ?v s < (∫⁺t. ?v t ∂D) and ct'-eq: ct' = ct(s := D)
      by (auto simp: R-def)
    with S_r-S1 have ct: ct ∈ Pi S K and s ∈ S and s ∈ S1
      by (auto simp: S_r-def)
    then have valid-ct[simp]: ⋀s. s ∈ S ⟹ simple ct s ∈ cfg-on s
      by simp

    from ct'-eq have [simp]: ct' s = D ⋀t. t ≠ s ⟹ ct' t = ct t
      by simp-all

    from ct-ct' S_r have ct'-E: ct' ∈ Pi_E S K
      by (auto simp: ct'-eq R-def)
    from ct s D have ct': ct' ∈ Pi S K
      by (auto simp: ct'-eq)
    then have valid-ct'[simp]: ⋀s. s ∈ S ⟹ simple ct' s ∈ cfg-on s
      by simp
```

from *exists-l-desc*[*OF* ‹*proper ct*›]
obtain *l* **where** *l*: $l \in S_r \to S_r \cup S2$ **and** $\bigwedge s.\ s \in S_r \Longrightarrow$ *l-desc* S_r *ct l s*
by *auto*
then have *directed-l*: $\bigwedge s.\ s \in S_r \Longrightarrow s \in$ *directed-towards S2* (*SIGMA* $s{:}S_r.$ $\{l\ s\}$)
and *v-l-mono*: $\bigwedge s.\ s \in S_r \Longrightarrow ?v\ s \leq ?v\ (l\ s)$
and *l-in-Ea*: $\bigwedge s.\ s \in S_r \Longrightarrow l\ s \in ct\ s$
by (*auto simp*: *l-desc-def dest*!: *maximalD1*)

let *?E* = $\lambda ct.$ *SIGMA* $s{:}S_r.$ *ct s*
let *?D* = $\lambda ct.$ *directed-towards S2* (*?E ct*)

have *finite-E*[*simp*]: $\bigwedge s.\ s \in S \Longrightarrow$ *finite* ($ct'\ s$)
using ct' *K-closed* **by** (*intro rev-finite-subset*[*OF S-finite*]) *auto*

have *maximal ?v* ($ct'\ s$) $\neq \{\}$
using ct' *D* ‹$s{\in}S$› *finite-E*[*of s*] **by** (*intro maximal-ne set-pmf-not-empty*) (*auto simp del*: *finite-E*)
then obtain s' **where** s': $s' \in$ *maximal ?v* ($ct'\ s$)
by *blast*
with *K-closed*[*OF* ‹$s \in S$›] *D* **have** $s' \in S$
by (*auto dest*!: *maximalD1*)

have $s' \neq s$
proof
assume [*simp*]: $s' = s$
have *?v s* < $(\int^+ t.$ *?v t* $\partial D)$
by *fact*
also have ... $\leq (\int^+ t.$ *?v s* $\partial D)$
using ‹$s \in S$› s' *D* **by** (*intro nn-integral-mono-AE*) (*auto simp*: *AE-measure-pmf-iff intro*: *maximalD2*)
finally show *False*
using *measure-pmf.emeasure-space-1*[*of D*] **by** (*simp add*: ‹$s \in S$› *ct*)
qed

have *p* $s' \neq 0$
proof
assume *p* $s' = 0$
then have *?v* $s' = 0$
using *v-le-p*[*of simple ct* s'] *ct* ‹$s' \in S$› **by** (*auto intro*!: *antisym ct*)
then have $(\int^+ t.$ *?v t* $\partial D) = 0$
using *maximalD2*[*OF* s'] **by** (*subst nn-integral-0-iff-AE*) (*auto simp*: ‹$s{\in}S$› *D AE-measure-pmf-iff*)
then have *?v s* < 0
using *not-maximal* **by** *auto*
then show *False*
using ‹$s{\in}S$› **by** (*simp add*: *ct*)
qed
with ‹$s' \in S$› **have** $s' \in S2 \cup S_r$

```
    by (auto simp: S_r-def S_e-def)

  have l-acyclic: (s', s) ∉ (SIGMA s:S_r. {l s})^+
  proof
    assume (s', s) ∈ (SIGMA s:S_r. {l s})^+
    then have ?v s' ≤ ?v s
      by induct (blast intro: order-trans v-l-mono)+
    also have ... < (∫⁺t. ?v t ∂D)
      using not-maximal .
    also have ... ≤ (∫⁺t. ?v s' ∂D)
    using s' by (intro nn-integral-mono-AE) (auto simp: ‹s ∈ S› D AE-measure-pmf-iff
intro: maximalD2)
    finally show False
      using measure-pmf.emeasure-space-1[of D] by (simp add:‹s' ∈ S› ct)
  qed

  from ‹s' ∈ S2 ∪ S_r› have s' ∈ ?D ct'
  proof
    assume s' ∈ S_r
    then have l s' ∈ directed-towards S2 (SIGMA s:S_r. {l s})
      using l directed-l[of l s'] by (auto intro: directed-towards.start)
    moreover from ‹s' ∈ S_r› have (s', l s') ∈ (SIGMA s:S_r. {l s})^+
      by auto
    ultimately have l s' ∈ ?D ct'
    proof induct
      case (step t t')
      then have t: t ≠ s t ∈ S_r t' = l t
        using l-acyclic by auto

      from step have (s', t') ∈ (SIGMA s:S_r. {l s})^+
        by (blast intro: trancl-into-trancl)
      from step(2)[OF this] show ?case
        by (rule directed-towards.step) (simp add: l-in-Ea t)
    qed (rule directed-towards.start)
    then show s' ∈ ?D ct'
      by (rule directed-towards.step)
         (simp add: l-in-Ea ‹s' ∈ S_r› ‹s ∈ S_r› ‹s' ≠ s›)
  qed (rule directed-towards.start)

  have proper: proper ct'
    unfolding proper-eq[OF ct'-E]
  proof
    fix t assume t ∈ S_r
    from directed-l[OF this] show t ∈ ?D ct'
    proof induct
      case (step t t')
      show ?case
      proof cases
        assume t = s
```

```
        with ‹s ∈ S_r› s'[THEN maximalD1] have (t, s') ∈ ?E ct'
          by auto
        with ‹s' ∈ ?D ct'› show ?thesis
          by (rule directed-towards.step)
      next
        assume t ≠ s
        with step have (t, t') ∈ ?E ct'
          by (auto simp: l-in-Ea)
        with step.hyps(2) show ?thesis
          by (rule directed-towards.step)
      qed
    qed (rule directed-towards.start)
  qed

  have ?v ≤ ?v'
  proof (intro le-funI leI notI)
    fix t' assume *: ?v' t' < ?v t'
    then have t' ∈ S
      by (metis v-nS simple-valid-cfg-iff ct' ct order.irrefl)

    define Δ where Δ t = enn2real (?v t) − enn2real (?v' t) for t
    with * ‹t' ∈ S› have 0 < Δ t'
      by (cases ?v t' ?v' t' rule: ennreal2-cases) (auto simp add: ct' ct en-
nreal-less-iff)

    { fix t assume t: t ∈ maximal Δ S
      with ‹t' ∈ S› have Δ t' ≤ Δ t
        by (auto intro: maximalD2)
      with ‹0 < Δ t'› have 0 < Δ t by simp
      with t have t ∈ S_r
        by (auto simp add: S_r-def v-S_e ct ct' Δ-def dest!: maximalD1) }
    note max-is-S_r = this

    { fix s assume s ∈ S
      with v-le-1[of simple ct' s] v-le-1[of simple ct s]
      have |Δ s| ≤ 1
        by (cases ?v s ?v' s rule: ennreal2-cases) (auto simp: Δ-def ct ct') }
    note Δ-le-1[simp] = this
    then have ennreal-Δ: ⋀s. s ∈ S ⟹ Δ s = ?v s − ?v' s
     by (auto simp add: Δ-def v-def T.emeasure-eq-measure ct ct' ennreal-minus)

    from ‹s ∈ S› S-finite have maximal Δ S ≠ {}
      by (intro maximal-ne) auto
    then obtain t where t ∈ maximal Δ S by auto
    from max-is-S_r[OF this] proper have t ∈ ?D ct'
      unfolding proper-eq[OF ct'-E] by auto
    from this ‹t ∈ maximal Δ S› show False
    proof induct
      case (start t)
```

```
    then have t ∈ S_r
      by (intro max-is-S_r)
    with ‹t ∈ S2› show False
      by (auto simp: S_r-def)
  next
    case (step t t')
    then have t': t' ∈ ct' t and t ∈ S_r and t: t ∈ maximal Δ S
      by (auto intro: max-is-S_r simp: comp-def)
    then have t' ∈ S t ∈ S1 t ∈ S
      using S_r-S1 S1
      by (auto simp: Pi-closed[OF ct'])

    have Δ t ≤ Δ t'
    proof (intro leI notI)
      assume less: Δ t' < Δ t
      have (∫ s. Δ s ∂ct' t) < (∫ s. Δ t ∂ct' t)
      proof (intro measure-pmf.integral-less-AE)
        show emeasure (ct' t) {t'} ≠ 0 {t'} ∈ sets (ct' t)
          AE s in ct' t. s∈{t'} ⟶ Δ s ≠ Δ t
          using t' less by (auto simp add: emeasure-pmf-single-eq-zero-iff)
        show AE s in ct' t. Δ s ≤ Δ t
          using ct' ct t D
       by (auto simp add: AE-measure-pmf-iff ct ‹t∈S› Pi-iff E-def Pi-closed[OF
ct']
                  intro!: maximalD2[of t Δ] intro: Pi-closed[OF ct'] maximalD1)
        show integrable (ct' t) (λ-. Δ t) integrable (ct' t) Δ
          using ct ct' ‹t ∈ S› D
        by (auto intro!: measure-pmf.integrable-const-bound[where B=1] Δ-le-1
                  simp: AE-measure-pmf-iff dest: Pi-closed)
      qed
      also have ... = Δ t
        using measure-pmf.prob-space[of ct' t] by simp
     also have Δ t ≤ (∫ s. enn2real (?v s) ∂ct' t) − (∫ s. enn2real (?v' s) ∂ct'
t)
      proof −
        have ?v t ≤ (∫⁺s. ?v s ∂ct' t)
        proof cases
          assume t = s with not-maximal show ?thesis by simp
        next
          assume t ≠ s with S1 ‹t∈S1› ‹t ∈ S› ct ct' show ?thesis
            by (subst v-S1) (auto intro!: nn-integral-mono-AE AE-pmfI)
        qed
        also have ... = ennreal (∫ s. enn2real (?v s) ∂ct' t)
          using ct ct' ‹t∈S›
          by (intro measure-pmf.ennreal-integral-real[symmetric, where B=1])
             (auto simp: AE-measure-pmf-iff one-ennreal-def[symmetric]
                   intro!: v-le-1 simple-valid-cfg intro: Pi-closed)
        finally have enn2real (?v t) ≤ (∫ s. enn2real (?v s) ∂ct' t)
          using ct ‹t∈S› by (simp add: v-def T.emeasure-eq-measure)
```

```
          moreover
          { have ?v' t = (∫⁺s. ?v' s ∂ct' t)
                using ct ct' ‹t ∈ S› ‹t ∈ S1› S1 by (subst v-S1) (auto intro!:
nn-integral-cong-AE AE-pmfI)
            also have ... = ennreal (∫ s. enn2real (?v' s) ∂ct' t)
              using ct' ‹t∈S›
             by (intro measure-pmf.ennreal-integral-real[symmetric, where B=1])
                (auto simp: AE-measure-pmf-iff one-ennreal-def[symmetric]
                      intro!: v-le-1 simple-valid-cfg intro: Pi-closed)
            finally have enn2real (?v' t) = (∫ s. enn2real (?v' s) ∂ct' t)
              using ct' ‹t∈S› by (simp add: v-def T.emeasure-eq-measure) }
          ultimately show ?thesis
            using ‹t ∈ S› by (simp add: Δ-def ennreal-minus-mono)
        qed
        also have ... = (∫ s. Δ s ∂ct' t)
          unfolding Δ-def using Pi-closed[OF ct ‹t∈S›] Pi-closed[OF ct' ‹t∈S›]
ct ct'
       by (intro Bochner-Integration.integral-diff[symmetric] measure-pmf.integrable-const-bound[where
B=1])
             (auto simp: AE-measure-pmf-iff real-v)
        finally show False
          by simp
      qed
      with t[THEN maximalD2] ‹t ∈ S› ‹t' ∈ S› have Δ t = Δ t'
        by (auto intro: antisym)
      with t ‹t' ∈ S› have t' ∈ maximal Δ S
        by (auto simp: maximal-def)
      then show ?case
        by fact
    qed
  qed
  moreover have ?v s < ?v' s
  proof −
    have ?v s < (∫⁺t. ?v t ∂D)
      by fact
    also have ... ≤ (∫⁺t. ?v' t ∂D)
      using ‹?v ≤ ?v'› ‹s∈S› D ct ct'
      by (intro nn-integral-mono) (auto simp: le-fun-def)
    also have ... = ?v' s
    using ‹s∈S1› S1 ct' ‹s ∈ S› by (subst (2) v-S1) (auto intro!: nn-integral-cong-AE
AE-pmfI)
    finally show ?thesis .
  qed
  ultimately have ?v < ?v'
    by (auto simp: less-le le-fun-def fun-eq-iff)
  note this proper ct' }
 note v-strict = this(1) and proper = this(2) and sc'-R = this(3)

 have finite (Pi_E S K × Pi_E S K)
```

by (*intro finite-PiE S-finite K-finite finite-SigmaI*)
then have *finite R*
by (*rule rev-finite-subset*) (*auto simp add*: *PiE-iff* S_r*-def R-def intro*: *extensional-arb*)
moreover
from *v-strict* **have** *acyclic R*
by (*rule acyclicI-order*)
ultimately have *wf R*
by (*rule finite-acyclic-wf*)

from *exists-proper* **obtain** *ct'* **where** *ct'*: *proper ct'* **.**
define *ct* **where** *ct* = *restrict ct' S*
with *ct'* **have** *sc-Pi*: *ct* ∈ *Pi S K* **and** *ct'* ∈ *Pi S K*
by (*auto simp*: *proper-def*)
then have *ct*: *ct* ∈ {*ct* ∈ Pi_E *S K*. *proper ct*}
using *ct' directed-towards-mono*[**where** *F*=*SIGMA s*:S_r. *ct' s* **and** *G*=*SIGMA s*:S_r. *ct s*]
apply *simp*
apply (*subst proper-eq*)
by (*auto simp*: *ct-def proper-eq*[*OF properD1*[*OF ct'*]] *subset-eq* S_r*-def*)

show ∃ *ct*. *ct* ∈ Pi_E *S K* ∧ *v*∘*simple ct* = *F-sup* (*v*∘*simple ct*)
proof (*rule wfE-min*[*OF* ‹*wf R*› *ct*])
fix *ct* **assume** *ct*: *ct* ∈ {*ct* ∈ Pi_E *S K*. *proper ct*}
then have *ct* ∈ *Pi S K proper ct*
by (*auto simp*: *proper-def*)
assume *min*: ⋀*ct'*. (*ct'*, *ct*) ∈ *R* ⟹ *ct'* ∉ {*ct* ∈ Pi_E *S K*. *proper ct*}
let *?v* = λ*s*. *v* (*simple ct s*)
{ fix *s* **assume** *s* ∈ *S s* ∈ *S1 s* ∉ *S2*
with *ct* **have** *ct s* ∈ *K s ?v s* ≤ $integral^N$ (*ct s*) *?v*
by (*auto simp*: *v-S1 PiE-def intro*!: *nn-integral-mono-AE AE-pmfI*)
moreover
{ have *0* ≤ *?v s*
using ‹*s*∈*S*› *ct* **by** (*simp add*: *PiE-def*)
also assume *v-less*: *?v s* < (∐ *D*∈*K s*. ∫⁺ *s*. *v* (*simple ct s*) ∂*measure-pmf D*)
also have ... ≤ *p s*
unfolding *p-S1*[*OF* ‹*s*∈*S1*›] **using** ‹*s*∈*S*› *ct v-le-p*[*OF simple-valid-cfg*, *OF* ‹*ct* ∈ *Pi S K*›]
by (*auto intro*!: *SUP-mono nn-integral-mono-AE bexI*
simp: *PiE-def AE-measure-pmf-iff set-pmf-closed*)
finally have *s* ∈ S_r
using ‹*s*∈*S*› ‹*s*∉*S2*› **by** (*simp add*: S_r*-def* S_e*-def*)

from *v-less* **obtain** *D* **where** *D* ∈ *K s ?v s* < $integral^N$ *D ?v*
by (*auto simp*: *less-SUP-iff*)
with *ct* ‹*s*∈*S*› ‹*s*∈S_r› **have** (*ct*(*s*:=*D*), *ct*) ∈ *R ct*(*s*:=*D*) ∈ Pi_E *S K*
unfolding *R-def* **by** (*auto simp*: *PiE-def extensional-def*)
from *proper*[*OF this*(*1*)] *min*[*OF this*(*1*)] *ct* ‹*D* ∈ *K s*› ‹*s*∈*S*› *this*(*2*)

```
      have False
        by simp }
    ultimately have ?v s = (⨆D∈K s. ∫⁺ s. ?v s ∂measure-pmf D)
      by (auto intro: antisym SUP-upper2[where i=ct s] leI)
    also have ... = (⨆D∈K s. integral^N (measure-pmf D) (λs∈S. ?v s))
        using ‹s∈S› by (auto intro!: SUP-cong nn-integral-cong v-nS simp: ct
simple-valid-cfg-iff ‹ct ∈ Pi S K›)
    finally have ?v s = (⨆D∈K s. integral^N (measure-pmf D) (λs∈S. ?v s)) . }
  then have ?v = F-sup ?v
    unfolding F-sup-def using ct
    by (auto intro!: ext v-S2 simple-cfg-on v-nS v-nS12 SUP-cong nn-integral-cong
          simp: PiE-def simple-valid-cfg-iff)
  with ct show ?thesis
    by (auto simp: comp-def)
 qed
qed

lemma p-v-memoryless:
 obtains ct where ct ∈ Pi_E S K p = v∘simple ct
proof −
 obtain ct where ct-PiE: ct ∈ Pi_E S K and eq: v∘simple ct = F-sup (v∘simple
ct)
   by (rule F-v-memoryless)
 then have ct: ct ∈ Pi S K
   by (simp add: PiE-def)
 have p = v∘simple ct
 proof (rule antisym)
   show p ≤ v∘simple ct
     unfolding p-eq-lfp-F-sup by (rule lfp-lowerbound) (metis order-refl eq)
   show v∘simple ct ≤ p
   proof (rule le-funI)
     fix s show (v∘simple ct) s ≤ p s
       using v-le-p[of simple ct s]
       by (cases s ∈ S) (auto simp del: simp add: v-def ct)
   qed
 qed
 with ct-PiE that show thesis by auto
qed

definition n = (λs∈S. P-inf s (λω. (HLD S1 suntil HLD S2) (s ## ω)))

lemma n-eq-INF-v: s ∈ S ⟹ n s = (∏cfg∈cfg-on s. v cfg)
 by (auto simp add: n-def v-def P-inf-def T.emeasure-eq-measure valid-cfgI intro!:
INF-cong)

lemma n-le-v: s ∈ S ⟹ cfg ∈ cfg-on s ⟹ n s ≤ v cfg
 by (subst n-eq-INF-v) (blast intro!: INF-lower)+

lemma n-eq-1-imp: s ∈ S ⟹ cfg ∈ cfg-on s ⟹ n s = 1 ⟹ v cfg = 1
```

using *n-le-v*[*of s cfg*] *v-le-1*[*of cfg*] **by** (*auto intro*: *antisym valid-cfgI*)

lemma *n-eq-1-iff*: $s \in S \Longrightarrow n\ s = 1 \longleftrightarrow (\forall\, cfg{\in}cfg\text{-}on\ s.\ v\ cfg = 1)$
apply *rule*
apply (*metis n-eq-1-imp*)
apply (*auto simp*: *n-eq-INF-v intro*!: *INF-eqI*)
done

lemma *n-le-1*: $s \in S \Longrightarrow n\ s \leq 1$
by (*auto simp*: *n-eq-INF-v intro*!: *INF-lower2*[*OF simple-cfg-on*[*of arb-act*]] *v-le-1*)

lemma *n-undefined*[*simp*]: $s \notin S \Longrightarrow n\ s = undefined$
by (*simp add*: *n-def*)

lemma *n-eq-0*: $s \in S \Longrightarrow cfg \in cfg\text{-}on\ s \Longrightarrow v\ cfg = 0 \Longrightarrow n\ s = 0$
using *n-le-v*[*of s cfg*] **by** *auto*

lemma *n-not-inf*[*simp*]: $s \in S \Longrightarrow n\ s \neq top$
using *n-le-1*[*of s*] **by** (*auto simp*: *top-unique*)

lemma *n-S1*: $s \in S1 \Longrightarrow n\ s = (\prod D{\in}K\ s.\ \int^+ t.\ n\ t\ \partial measure\text{-}pmf\ D)$
using *S1 S1-S2* **unfolding** *n-def*
apply *auto*
apply (*subst P-inf-iterate*)
apply (*auto intro*!: *nn-integral-cong-AE INF-cong intro*: *set-pmf-closed*
simp: *AE-measure-pmf-iff suntil-Stream set-eq-iff*)
done

lemma *n-S2*[*simp*]: $s \in S2 \Longrightarrow n\ s = 1$
using *S2* **by** (*auto simp add*: *n-eq-INF-v valid-cfgI*)

lemma *n-nS12*: $s \in S \Longrightarrow s \notin S1 \Longrightarrow s \notin S2 \Longrightarrow n\ s = 0$
by (*auto simp add*: *n-eq-INF-v valid-cfgI*)

lemma *n-pos*:
assumes *P s s* ∈ *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 $0 < n\ s$
using ‹*wf R*› ‹*P s*› ‹*s*∈*S1*›
proof (*induction s*)
case (*less s*)
with *S1* **have** [*simp*]: $s \in S$ **by** *auto*
let *?I* = $\lambda D{::}'s\ pmf.\ \int^+ t.\ n\ t\ \partial D$
have $0 < Min\ (?I`K\ s)$
proof (*safe intro*!: *Min-gr-iff* [*THEN iffD2*])
fix *D* **assume** [*simp*]: $D \in K\ s$
from *cont*[*OF* ‹*P s*› ‹*s* ∈ *S1*› ‹*D* ∈ *K s*›]
obtain *w* **where** *w*: $w \in D\ 0 < n\ w$

```
    by (force intro: less.IH)
  have in-S: ⋀t. t ∈ D ⟹ t ∈ S
    using set-pmf-closed[OF ‹s ∈ S› ‹D ∈ K s›] by auto
  from w have 0 < pmf D w * n w
    by (simp add: pmf-positive ennreal-zero-less-mult-iff)
  also have ... = (∫⁺t. n w * indicator {w} t ∂D)
    by (subst nn-integral-cmult-indicator)
       (auto simp: ac-simps emeasure-pmf-single in-S ‹w ∈ D›)
  also have ... ≤ (∫⁺t. n t ∂D)
   by (intro nn-integral-mono-AE) (auto split: split-indicator simp: AE-measure-pmf-iff
in-S)
  finally show 0 < (∫⁺t. n t ∂D) .
 qed (insert K-wf K-finite ‹s∈S›, auto)
 also have ... = n s
  unfolding n-S1[OF ‹s ∈ S1›]
  using K-wf K-finite ‹s∈S› by (intro Min-Inf) auto
 finally show 0 < n s .
qed

definition F-inf :: ('s ⇒ ennreal) ⇒ ('s ⇒ ennreal) where
  F-inf f = (λs∈S. if s ∈ S2 then 1 else if s ∈ S1 then (∏D∈K s. ∫⁺ t. f t
∂measure-pmf D) else 0)

lemma F-inf-n: F-inf n = n
 by (simp add: F-inf-def n-nS12 n-S1 fun-eq-iff)

lemma F-inf-nS[simp]: s ∉ S ⟹ F-inf f s = undefined
 by (simp add: F-inf-def)

lemma mono-F-inf: mono F-inf
  by (auto intro!: INF-superset-mono nn-integral-mono simp: mono-def F-inf-def
le-fun-def)

lemma S1-nS2: s ∈ S1 ⟹ s ∉ S2
 using S1-S2 by auto

lemma n-eq-lfp-F-inf: n = lfp F-inf
proof (intro antisym lfp-lowerbound le-funI)
 fix s let ?I = λD. (∫⁺t. lfp F-inf t ∂measure-pmf D)
 define ct where ct s = (SOME D. D ∈ K s ∧ (s ∈ S1 ⟶ lfp F-inf s = ?I D))
for s
 { fix s assume s: s ∈ S
   then have finite (?I ‘ K s)
     by (auto intro: K-finite)
   with s obtain D where D ∈ K s (∫⁺t. lfp F-inf t ∂D) = Min (?I ‘ K s)
     by (auto simp: K-wf dest!: Min-in)
   note this(2)
   also have ... = (INF D ∈ K s. ?I D)
     using s K-wf by (subst Min-Inf) (auto intro: K-finite)
```

```
    also have s ∈ S1 ⟹ ... = lfp F-inf s
      using s S1-S2 by (subst (3) lfp-unfold[OF mono-F-inf]) (auto simp add:
F-inf-def)
    finally have ∃ D. D ∈ K s ∧ (s ∈ S1 ⟶ lfp F-inf s = ?I D)
      using ‹D ∈ K s› by auto
    then have ct s ∈ K s ∧ (s ∈ S1 ⟶ lfp F-inf s = ?I (ct s))
      unfolding ct-def by (rule someI-ex)
    then have ct s ∈ K s s ∈ S1 ⟹ lfp F-inf s = ?I (ct s)
      by auto }
  note ct = this
  then have Pi-ct: ct ∈ Pi S K
    by auto
  then have valid-ct[simp]: ⋀s. s ∈ S ⟹ simple ct s ∈ valid-cfg
    by simp
  let ?F = λP. HLD S2 or (HLD S1 aand nxt P)
  define P where P s n =
     emeasure (T (simple ct s)) {x∈space (T (simple ct s)). (?F ^^ n) (λx. False)
(s ## x)}
    for s n
  { assume s ∈ S
    with S1 have [simp, measurable]: s ∈ S by auto
    then have n s ≤ v (simple ct s)
      by (intro n-le-v) (auto intro: simple-cfg-on[OF Pi-ct])
    also have ... = emeasure (T (simple ct s)) {x∈space (T (simple ct s)). lfp ?F
(s ## x)}
      using S1-S2
      by (simp add: v-eq[OF simple-valid-cfg[OF Pi-ct ‹s∈S›]])
         (simp add: suntil-lfp space-T[symmetric, of simple ct s] del: space-T)
    also have ... = (⨆ n. P s n) unfolding P-def
      apply (rule emeasure-lfp2[where P=λM. ∃ s. M = T (simple ct s) and
M=T (simple ct s)])
      apply (intro exI[of - s] refl)
      apply (auto simp: sup-continuous-def) []
      apply auto []
    proof safe
      fix A s assume ⋀N. ∃ s. N = T (simple ct s) ⟹ Measurable.pred N A
      then have ⋀s. Measurable.pred (T (simple ct s)) A
        by metis
      then have ⋀s. Measurable.pred St A
        by simp
      then show Measurable.pred (T (simple ct s)) (λxs. HLD S2 xs ∨ HLD S1 xs
∧ nxt A xs)
        by simp
    qed
    also have ... ≤ lfp F-inf s
    proof (intro SUP-least)
      fix n from ‹s∈S› show P s n ≤ lfp F-inf s
      proof (induct n arbitrary: s)
        case 0 with S1 show ?case
```

```
      by (subst lfp-unfold[OF mono-F-inf]) (auto simp: P-def)
    next
      case (Suc n)

      show ?case
      proof cases
        assume s ∈ S1 with S1-S2 S1 have s[simp]: s ∉ S2 s ∈ S s ∈ S1 by
auto
        have P s (Suc n) = (∫⁺t. P t n ∂ct s)
          unfolding P-def space-T
          apply (subst emeasure-Collect-T)
            apply (rule measurable-compose[OF measurable-Stream[OF measur-
able-const measurable-ident-sets[OF refl]]])
          apply (measurable, assumption)
          apply (auto simp: K-cfg-def map-pmf-rep-eq nn-integral-distr
                      intro!: nn-integral-cong-AE AE-pmfI)
          done
        also have ... ≤ (∫⁺t. lfp F-inf t ∂ct s)
          using Pi-closed[OF Pi-ct ‹s ∈ S›]
          by (auto intro!: nn-integral-mono-AE Suc simp: AE-measure-pmf-iff)
        also have ... = lfp F-inf s
          by (intro ct(2)[symmetric]) auto
        finally show ?thesis .
      next
        assume s ∉ S1 with S2 ‹s ∈ S› show ?case
          using T.emeasure-space-1[of simple ct s]
          by (subst lfp-unfold[OF mono-F-inf]) (auto simp: F-inf-def P-def)
      qed
    qed
  qed
  finally have n s ≤ lfp F-inf s . }
 moreover have s ∉ S ⟹ n s ≤ lfp F-inf s
  by (subst lfp-unfold[OF mono-F-inf]) (simp add: n-def F-inf-def)
 ultimately show n s ≤ lfp F-inf s
  by blast
qed (simp add: F-inf-n)


lemma real-n: s ∈ S ⟹ ennreal (enn2real (n s)) = n s
 by (cases n s) simp-all

lemma real-p: s ∈ S ⟹ ennreal (enn2real (p s)) = p s
 by (cases p s) simp-all

lemma p-ub:
 fixes x
 assumes s ∈ S
 assumes solution: ⋀s D. s ∈ S1 ⟹ D ∈ K s ⟹ (∑ t∈S. pmf D t * x t) ≤ x s
 assumes solution-0: ⋀s. s ∈ S ⟹ p s = 0 ⟹ x s = 0
```

```
  assumes solution-S2: ⋀s. s ∈ S2 ⟹ x s = 1
  shows enn2real (p s) ≤ x s (is ?y s ≤ -)
proof −
  let ?p = λs. enn2real (p s)
  from p-v-memoryless obtain sc where sc ∈ Pi_E S K and p-eq: p = v ∘ simple
sc
    by auto
  then have sch: ⋀s. s ∈ S ⟹ sc s ∈ K s and sc-Pi: sc ∈ Pi S K
    by (auto simp: PiE-iff)

  interpret sc: MC-syntax sc .

  define N where N = {s∈S. p s = 0} ∪ S2
  { fix s assume s ∈ S s ∉ N
    with p-nS12 have s ∈ S1
      by (auto simp add: N-def) }
  note N = this

  have N-S: N ⊆ S
    using S2 by (auto simp: N-def)

  have finite-sc[intro]: s ∈ S ⟹ finite (sc s) for s
    using ‹sc ∈ Pi_E S K› by (auto simp: PiE-iff intro: set-pmf-finite)


  show ?thesis
  proof cases
    assume s ∈ S − N
    then show ?thesis
    proof (rule mono-les)
      show (⋃x∈S − N. set-pmf (sc x)) ⊆ S − N ∪ N
        using Pi-closed[OF sc-Pi] by auto
      show finite ((λs. ?p s − x s) ` (S − N ∪ N))
        using N-S by (intro finite-imageI finite-subset[OF - S-finite]) auto
    next
      fix s assume s ∈ N then show ?p s ≤ x s
        by (auto simp: N-def solution-S2 solution-0)
    next
      fix s assume s: s ∈ S − N
      then show integrable (sc s) x integrable (sc s) ?p
        by (auto intro!: integrable-measure-pmf-finite set-pmf-finite sch)

      from s have s ∈ S1 s ∈ S
        using p-nS12[of s] by (auto simp: N-def)
      then show ?p s ≤ (∫ t. ?p t ∂sc s) + 0
        unfolding p-eq using real-v-integral-eq[of simple sc s]
     by (auto simp add: v-S1 sc-Pi intro!: integral-mono-AE integrable-measure-pmf-finite
AE-pmfI)
      show (∫ t. x t ∂sc s) + 0 ≤ x s
```

```
      using solution[OF ‹s ∈ S1› sch[OF ‹s ∈ S›]]
      by (subst integral-measure-pmf[where A=S])
        (auto intro: S-finite Pi-closed[OF sc-Pi] ‹s ∈ S› simp: ac-simps)

    define X where X = (SIGMA x:UNIV. sc x)
    show ∃ t∈N. (s, t) ∈ X*
    proof (rule ccontr)
      assume ¬ ?thesis
      then have *: ∀ t∈N. (s, t) ∉ X*
        by auto
      with ‹s∈S› have v (simple sc s) = 0
      proof (coinduction arbitrary: s rule: v-eq-0-coinduct)
        case (valid t) with sch show ?case
          by auto
      next
        case (nS2 s) then show ?case
          by (auto simp: N-def)
      next
        case (cont cfg s)
        then have (s, state cfg) ∈ X*
          by (auto simp: X-def set-K-cfg)
        with cont show ?case
          by (auto simp: set-K-cfg intro!: exI intro: Pi-closed[OF sc-Pi])
            (blast intro: rtrancl-trans)
      qed
      then have p s = 0
        unfolding p-eq by simp
      with ‹s∈S› have s∈N
        by (auto simp: N-def)
      with * show False
        by auto
    qed
  qed
next
  assume s ∉ S − N with ‹s ∈ S› show ?p s ≤ x s
    by (auto simp: N-def solution-0 solution-S2)
qed
qed

lemma n-lb:
  fixes x
  assumes s ∈ S
  assumes solution: ⋀s D. s ∈ S1 ⟹ D ∈ K s ⟹ x s ≤ (∑ t∈S. pmf D t * x t)
  assumes solution-n0: ⋀s. s ∈ S ⟹ n s = 0 ⟹ x s = 0
  assumes solution-S2: ⋀s. s ∈ S2 ⟹ x s = 1
  shows x s ≤ enn2real (n s) (is - ≤ ?y s)
proof −
  let ?I = λD::'s pmf. ∫⁺x. n x ∂D
  { fix s assume s ∈ S1
```

```
  with S1 S1-S2 have n s = (∏ D∈K s. ?I D)
    by (subst n-eq-lfp-F-inf, subst lfp-unfold[OF mono-F-inf])
       (auto simp add: F-inf-def n-eq-lfp-F-inf)
  moreover have (∏ D∈K s. ∫⁺x. n x ∂measure-pmf D) = Min (?I‘K s)
    using ‹s ∈ S1› S1 K-wf
    by (intro cInf-eq-Min finite-imageI K-finite) auto
  moreover have Min (?I‘K s) ∈ ?I‘K s
    using ‹s ∈ S1› S1 K-wf by (intro Min-in finite-imageI K-finite) auto
  ultimately have ∃ D∈K s. (∫⁺x. n x ∂D) = n s
    by auto }
then have ⋀s. s ∈ S ⟹ ∃ D∈K s. s ∈ S1 ⟶ (∫⁺x. n x ∂D) = n s
  using K-wf by auto
then obtain sc where sch: ⋀s. s ∈ S ⟹ sc s ∈ K s
  and n-sc: ⋀s. s ∈ S1 ⟹ (∫⁺x. n x ∂sc s) = n s
  by (metis S1 subsetD)
then have sc-Pi: sc ∈ Pi S K
  by auto

define N where N = {s∈S. n s = 0} ∪ S2
with S2 have N-S: N ⊆ S
  by auto
{ fix s assume s ∈ S s ∉ N
  with n-nS12 have s ∈ S1
    by (auto simp add: N-def) }
note N = this

let ?n = λs. enn2real (n s)
show ?thesis
proof cases
  assume s ∈ S − N
  then show ?thesis
  proof (rule mono-les)
    show (⋃ x∈S − N. set-pmf (sc x)) ⊆ S − N ∪ N
      using Pi-closed[OF sc-Pi] by auto
    show finite ((λs. x s − ?n s) ‘ (S − N ∪ N))
      using N-S by (intro finite-imageI finite-subset[OF - S-finite]) auto
  next
    fix s assume s ∈ N then show x s ≤ ?n s
      by (auto simp: N-def solution-S2 solution-n0)
  next
    fix s assume s: s ∈ S − N
    then show integrable (sc s) x integrable (sc s) ?n
      by (auto intro!: integrable-measure-pmf-finite set-pmf-finite sch)

    from s have s ∈ S1 s ∈ S
      using n-nS12[of s] by (auto simp: N-def)
    then have (∫ t. ?n t ∂sc s) = ?n s
      apply (subst n-sc[symmetric, of s])
      apply simp-all
```

```
      apply (subst integral-eq-nn-integral)
      apply (auto simp: Pi-closed[OF sc-Pi] AE-measure-pmf-iff
                  intro!: arg-cong[where f=enn2real] nn-integral-cong-AE real-n)
      done
    then show (∫ t. ?n t ∂sc s) + 0 ≤ ?n s
      by simp

    show x s ≤ (∫ t. x t ∂sc s) + 0
      using solution[OF ‹s ∈ S1› sch[OF ‹s ∈ S›]]
      by (subst integral-measure-pmf[where A=S])
         (auto intro: S-finite Pi-closed[OF sc-Pi] ‹s ∈ S› simp: ac-simps)

    define X where X = (SIGMA x:UNIV. sc x)
    show ∃ t∈N. (s, t) ∈ X*
    proof (rule ccontr)
      assume ¬ ?thesis
      then have *: ∀ t∈N. (s, t) ∉ X*
        by auto
      with ‹s∈S› have v (simple sc s) = 0
      proof (coinduction arbitrary: s rule: v-eq-0-coinduct)
        case (valid t) with sch show ?case
          by auto
      next
        case (nS2 s) then show ?case
          by (auto simp: N-def)
      next
        case (cont cfg s)
        then have (s, state cfg) ∈ X*
          by (auto simp: X-def set-K-cfg)
        with cont show ?case
          by (auto simp: set-K-cfg intro!: exI intro: Pi-closed[OF sc-Pi])
             (blast intro: rtrancl-trans)
      qed
      from n-eq-0[OF ‹s ∈ S› simple-cfg-on this] have n s = 0
        by (auto simp: sc-Pi)
      with ‹s∈S› have s∈N
        by (auto simp: N-def)
      with * show False
        by auto
    qed
  qed
next
  assume s ∉ S − N with ‹s ∈ S› show x s ≤ ?n s
    by (auto simp: N-def solution-n0 solution-S2)
qed
qed

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-emb UNIV M* {..*n*} (Pi_E {..*n*} *A*)) = Q (*prod-emb UNIV M* {..*n*} (Pi_E {..*n*} *A*))
assumes *A*: *finite-measure P*
shows $P = Q$
proof (*rule measure-eqI-PiM-infinite*[*OF* $*$ - *A*])
fix J :: *nat set* **and** F'
assume *J*: *finite J* $\bigwedge i.\ i \in J \Longrightarrow F'\ i \in sets\ (M\ i)$

define *n* **where** n = (*if J* = {} *then 0 else Max J*)
define *F* **where** $F\ i$ = (*if* $i \in J$ *then* $F'\ i$ *else space* ($M\ i$)) **for** i
then have *F*[*simp*, *measurable*]: $F\ i \in sets\ (M\ i)$ **for** i
using *J* **by** *auto*
have *emb-eq*: *prod-emb UNIV M J* (Pi_E *J F'*) = *prod-emb UNIV M* {..*n*} (Pi_E {..*n*} *F*)
proof *cases*
assume $J = \{\}$ **then show** *?thesis*
by (*auto simp add*: *n-def F-def*[*abs-def*] *prod-emb-def PiE-def*)
next
assume $J \neq \{\}$ **then show** *?thesis*
by (*auto simp*: *prod-emb-def PiE-iff F-def n-def less-Suc-eq-le* ‹*finite J*› *split*: *if-split-asm*)
qed

show *emeasure P* (*prod-emb UNIV M J* (Pi_E *J F'*)) = *emeasure Q* (*prod-emb UNIV M J* (Pi_E *J F'*))
unfolding *emb-eq* **by** (*rule eq*) *fact*
qed

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*
unfolding *simp-implies-def* **by** (*rule distr-cong*)

6.1 Constructing Discrete-Time Markov Processes

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

lemma *space-K*: $x \in$ *space M* ⟹ *space* ($K\ x$) = *space M*
  using K **unfolding** *prob-algebra-def* **unfolding** *measurable-restrict-space2-iff*
  by (*auto dest*: *subprob-measurableD*)

lemma *sets-K*[*measurable-cong*]: $x \in$ *space M* ⟹ *sets* ($K\ x$) = *sets M*
  using K **unfolding** *prob-algebra-def* **unfolding** *measurable-restrict-space2-iff*
  by (*auto dest*: *subprob-measurableD*)

lemma *prob-space-K*: $x \in$ *space M* ⟹ *prob-space* ($K\ x$)
  using *measurable-space*[*OF K*] **by** (*simp add*: *space-prob-algebra*)

definition K' :: *'a* ⇒ *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a measure*
where
  $K'\ x\ n'\ \omega' = K$ (*case-nat x* ω' n')

lemma *IT-K'*:
  assumes *x*: $x \in$ *space M* **shows** *Ionescu-Tulcea* ($K'\ x$) (λ-. *M*)
  unfolding *Ionescu-Tulcea-def K'-def*[*abs-def*]
proof *safe*
  fix *i* **show** (λω'. K (*case i of 0* ⇒ *x* | *Suc x* ⇒ ω' *x*)) ∈ Pi_M {*0..<i*} (λ-. *M*) $\to_M$ *subprob-algebra M*
    using *x* **by** (*intro measurable-prob-algebraD measurable-compose*[*OF - K*]) *measurable*
next
  fix *i* :: *nat* **and** ω **assume** ω: ω ∈ *space* (Pi_M {*0..<i*} (λ-. *M*))
  with *x* **have** (*case i of 0* ⇒ *x* | *Suc x* ⇒ ω *x*) ∈ *space M*
    by (*auto simp*: *space-PiM split*: *nat.split*)
  then show *prob-space* (K (*case i of 0* ⇒ *x* | *Suc x* ⇒ ω *x*))
    using K **unfolding** *measurable-restrict-space2-iff prob-algebra-def* **by** *auto*
qed

definition *lim-sequence* :: *'a* ⇒ (*nat* ⇒ *'a*) *measure*
where
  lim-sequence x = *projective-family.lim UNIV* (*Ionescu-Tulcea.CI* ($K'\ x$) (λ-. *M*)) (λ-. *M*)

lemma
  assumes *x*: $x \in$ *space M*
  shows *space-lim-sequence*: *space* (*lim-sequence x*) = *space* (Π_M *i*∈*UNIV*. *M*)
    and *sets-lim-sequence*[*measurable-cong*]: *sets* (*lim-sequence x*) = *sets* (Π_M *i*∈*UNIV*. *M*)
    and *emeasure-lim-sequence-emb*: ⋀*J X*. *finite J* ⟹ *X* ∈ *sets* (Π_M *j*∈*J*. *M*) ⟹

```
      emeasure (lim-sequence x) (prod-emb UNIV (λ-. M) J X) =
      emeasure (Ionescu-Tulcea.CI (K' x) (λ-. M) J) X
    and emeasure-lim-sequence-emb-I0o: ⋀n X. X ∈ sets (Π_M i ∈ {0..<n}. M)
⟹
      emeasure (lim-sequence x) (prod-emb UNIV (λ-. M) {0..<n} X) =
      emeasure (Ionescu-Tulcea.C (K' x) (λ-. M) 0 n (λx. undefined)) X
proof −
  interpret Ionescu-Tulcea K' x λ-. M
    using x by (rule IT-K')
  show space (lim-sequence x) = space (Π_M i∈UNIV. M)
    unfolding lim-sequence-def by simp
  show sets (lim-sequence x) = sets (Π_M i∈UNIV. M)
    unfolding lim-sequence-def by simp

  { fix J :: nat set and X assume finite J X ∈ sets (Π_M j∈J. M)
    then show emeasure (lim-sequence x) (PF.emb UNIV J X) = emeasure (CI
J) X
      unfolding lim-sequence-def by (rule lim) }
  note emb = this

  have up-to-I0o[simp]: up-to {0..<n} = n for n
    unfolding up-to-def by (rule Least-equality) auto

  { fix n :: nat and X assume X ∈ sets (Π_M j∈{0..<n}. M)
   then show emeasure (lim-sequence x) (PF.emb UNIV {0..<n} X) = emeasure
(C 0 n (λx. undefined)) X
    by (simp add: space-C emb CI-def space-PiM distr-id2 sets-C cong: distr-cong-simp)
}
qed

lemma lim-sequence[measurable]: lim-sequence ∈ M →_M prob-algebra (Π_M i∈UNIV.
M)
proof (intro measurable-prob-algebra-generated[OF sets-PiM Int-stable-prod-algebra
prod-algebra-sets-into-space])
  fix a assume [simp]: a ∈ space M
  interpret Ionescu-Tulcea K' a λ-. M
    by (rule IT-K') simp
  have sp: space (lim-sequence a) = prod-emb UNIV (λ-. M) {} (Π_E j∈{}. space
M) space (CI {}) = {} →_E space M
    by (auto simp: space-lim-sequence space-PiM prod-emb-def PF.space-P)
  show prob-space (lim-sequence a)
    apply standard
    using PF.prob-space-P[THEN prob-space.emeasure-space-1, of {}]
    apply (simp add: sp emeasure-lim-sequence-emb del: PiE-empty-domain)
    done
  show sets (lim-sequence a) = sets (Pi_M UNIV (λi. M))
    by (simp add: sets-lim-sequence)
next
  fix X :: (nat ⇒ 'a) set assume X ∈ prod-algebra UNIV (λi. M)
```

then obtain J :: *nat set* **and** F **where** J: $J \neq \{\}$ *finite* J $F \in J \to$ *sets* M
and X: $X =$ *prod-emb UNIV* $(\lambda\text{-}.\ M)$ J $(Pi_E\ J\ F)$
unfolding *prod-algebra-def* **by** *auto*
then have *Pi-F*: *finite* J $Pi_E\ J\ F \in$ *sets* $(Pi_M\ J\ (\lambda\text{-}.\ M))$
by (*auto intro*: *sets-PiM-I-finite*)

define n **where** $n = (LEAST\ n.\ \forall\, i{\geq}n.\ i \notin J)$
have *J-le-n*: $J \subseteq \{0..{<}n\}$
unfolding *n-def*
using ‹*finite* J›
apply −
apply (*rule LeastI2*[*of* - *Suc* (*Max* J)])
apply (*auto simp*: *Suc-le-eq not-le*[*symmetric*])
done

have C: $(\lambda x.$ *Ionescu-Tulcea.C* $(K'\ x)$ $(\lambda\text{-}.\ M)$ *0* n $(\lambda x.$ *undefined*$)) \in M \to_M$ *subprob-algebra* $(Pi_M\ \{0..{<}n\}\ (\lambda\text{-}.\ M))$
apply (*induction* n)
apply (*subst measurable-cong*)
apply (*rule Ionescu-Tulcea.C.simps*[*OF IT-K'*])
apply *assumption*
apply (*rule measurable-compose*[*OF* - *return-measurable*])
apply *simp*
apply (*subst measurable-cong*)
apply (*rule Ionescu-Tulcea.C.simps*[*OF IT-K'*])
apply *assumption*
apply (*rule measurable-bind'*)
apply *assumption*
apply (*subst measurable-cong*)
proof −
fix n :: *nat* **and** w **assume** $w \in$ *space* $(M \bigotimes_M Pi_M\ \{0..{<}n\}\ (\lambda\text{-}.\ M))$
then show (*case* w *of* $(x,\ xa) \Rightarrow$ *Ionescu-Tulcea.eP* $(K'\ x)$ $(\lambda\text{-}.\ M)$ $(0\ +\ n)$ $xa) =$
(*case* w *of* $(x,\ xa) \Rightarrow$ *distr* $(K'\ x\ n\ xa)$ $(\Pi_M\ i{\in}\{0..{<}Suc\ n\}.\ M)$ (*fun-upd xa* n))
by (*auto simp*: *space-pair-measure Ionescu-Tulcea.eP-def*[*OF IT-K'*] *split*: *prod.split*)
next
fix n **show** $(\lambda w.$ *case* w *of* $(x,\ xa) \Rightarrow$ *distr* $(K'\ x\ n\ xa)$ $(Pi_M\ \{0..{<}Suc\ n\}\ (\lambda i.\ M))$ (*fun-upd xa* n))
$\in M \bigotimes_M Pi_M\ \{0..{<}n\}\ (\lambda\text{-}.\ M) \to_M$ *subprob-algebra* $(Pi_M\ \{0..{<}Suc\ n\}\ (\lambda\text{-}.\ M))$
unfolding *K'-def*
apply *measurable*
apply (*rule measurable-distr2*[**where** $M{=}M$])
apply (*rule measurable-PiM-single'*)
apply (*simp add*: *split-beta'*)
subgoal for i **by** (*cases* $i = n$) *auto*
subgoal by (*auto simp*: *split-beta' PiE-iff extensional-def Pi-iff space-pair-measure*

space-PiM)
apply (*rule measurable-prob-algebraD*)
apply (*rule measurable-compose*[*OF - K*])
apply *measurable*
done
qed

have (λ*a*. *emeasure* (*lim-sequence a*) *X*) ∈ *borel-measurable M* ⟷
(λ*a*. *emeasure* (*Ionescu-Tulcea.CI* (*K' a*) (λ-. *M*) *J*) (*Pi*$_E$ *J F*)) ∈ *borel-measurable M*
unfolding *X* **using** *J Pi-F* **by** (*intro measurable-cong emeasure-lim-sequence-emb*) *auto*
also have ...
apply (*intro measurable-compose*[*OF - measurable-emeasure-subprob-algebra*[*OF Pi-F*(*2*)]])
apply (*subst measurable-cong*)
apply (*subst Ionescu-Tulcea.CI-def*[*OF IT-K'*])
apply *assumption*
apply (*subst Ionescu-Tulcea.up-to-def*[*OF IT-K'*])
apply *assumption*
unfolding *n-def*[*symmetric*]
apply (*rule refl*)
apply (*rule measurable-compose*[*OF - measurable-distr*[*OF measurable-restrict-subset*[*OF J-le-n*]]])
apply (*rule C*)
done
finally show (λ*a*. *emeasure* (*lim-sequence a*) *X*) ∈ *borel-measurable M* **.**
qed

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 −
interpret *Ionescu-Tulcea K' x* λ-. *M*
using *x* **by** (*rule IT-K'*)

have [*simp*]: *space* (*K x*) ≠ {}
using *space-K*[*OF x*] *x* **by** *auto*

have [*simp*]: ((λ-. *undefined*::'*a*)(*0* := *x*)) = *case-nat x* (λ-. *undefined*) **for** *x*
by (*auto simp*: *fun-eq-iff split*: *nat.split*)

have *C 0 1* (λ-. *undefined*) ⤜ *C 1 n* = *eP 0* (λ-. *undefined*) ⤜ *C 1 n*
using *measurable-eP*[*of 0*] *measurable-C*[*of 1 n*, *measurable del*]
by (*simp add*: *bind-return*[**where** *N*=*Pi*$_M$ {*0*} (λ-. *M*)])
also have ... = *K x* ⤜ (λ*y*. *C 1 n* (*case-nat y* (λ-. *undefined*)))
using *measurable-C*[*of 1 n*, *measurable del*] *x*[*THEN sets-K*]

by (*simp add*: *eP-def K′-def bind-distr cong*: *measurable-cong-sets*)
finally show *C 0 1* (λ-. *undefined*) ⤜ *C 1 n* = *K x* ⤜ (λ*y*. *C 1 n* (*case-nat y* (λ-. *undefined*))) **.**
qed

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 (*rule measure-eqI-PiM-infinite*)
show *sets* (*lim-sequence x*) = *sets* (Π$_M$ *j*∈*UNIV*. *M*)
using *x* **by** (*rule sets-lim-sequence*)
have [*simp*]: *space* (*K x*) ≠ {}
using *space-K*[*OF x*] *x* **by** *auto*
show *sets* (*?B x*) = *sets* (*Pi$_M$ UNIV* (λ*j*. *M*))
using *x* **by** (*subst sets-bind*) *auto*
interpret *lim-sequence*: *prob-space lim-sequence x*
using *lim-sequence x* **by** (*auto simp*: *measurable-restrict-space2-iff prob-algebra-def*)
show *finite-measure* (*lim-sequence x*)
by (*rule lim-sequence.finite-measure*)

interpret *Ionescu-Tulcea K′ x* λ-. *M*
using *x* **by** (*rule IT-K′*)

let *?U* = λ-::*nat*. *undefined* :: *′a*

fix *J* :: *nat set* **and** *F′*
assume *J*: *finite J* ⋀*i*. *i* ∈ *J* ⟹ *F′ i* ∈ *sets M*

define *n* **where** *n* = (*if J* = {} *then 0 else Max J*)
define *F* **where** *F i* = (*if i* ∈ *J then F′ i else space M*) **for** *i*
then have *F*[*simp*, *measurable*]: *F i* ∈ *sets M* **for** *i*
using *J* **by** *auto*
have *emb-eq*: *PF.emb UNIV J* (*Pi$_E$ J F′*) = *PF.emb UNIV* {*0..<Suc n*} (*Pi$_E$* {*0..<Suc n*} *F*)
proof *cases*
assume *J* = {} **then show** *?thesis*
by (*auto simp add*: *n-def F-def*[*abs-def*] *prod-emb-def PiE-def*)
next
assume *J* ≠ {} **then show** *?thesis*
by (*auto simp*: *prod-emb-def PiE-iff F-def n-def less-Suc-eq-le* ‹*finite J*› *split*: *if-split-asm*)
qed

have *emeasure* (*lim-sequence x*) (*PF.emb UNIV J* (*Pi$_E$ J F′*)) = *emeasure* (*C 0* (*Suc n*) *?U*) (*Pi$_E$* {*0..<Suc n*} *F*)
using *x* **unfolding** *emb-eq* **by** (*rule emeasure-lim-sequence-emb-I0o*) (*auto intro*!: *sets-PiM-I-finite*)

also have *C 0 (Suc n) ?U = K x ⤜ (λy. C 1 n (case-nat y ?U))*
using *split-C[of ?U 0 Suc 0 n] step-C[OF x]* **by** *simp*
also have *emeasure (K x ⤜ (λy. C 1 n (case-nat y ?U))) (Pi$_E$ {0..<Suc n} F) =*
(∫$^+$y. C 1 n (case-nat y ?U) (Pi$_E$ {0..<Suc n} F) ∂K x)
using *measurable-C[of 1 n, measurable del] x[THEN sets-K] F x*
by (*intro emeasure-bind[OF - measurable-compose[OF - measurable-C]]*)
(*auto cong*: *measurable-cong-sets intro*!: *measurable-PiM-single′ split*: *nat.split-asm*)
also have *... = (∫$^+$y. distr (lim-sequence y) (Pi$_M$ UNIV (λj. M)) (case-nat y) (PF.emb UNIV J (Pi$_E$ J F′)) ∂K x)*
proof (*intro nn-integral-cong*)
fix *y* **assume** *y ∈ space (K x)*
then have *y*: *y ∈ space M*
using *x* **by** (*simp add*: *space-K*)
then interpret *y*: *Ionescu-Tulcea K′ y λ-. M*
by (*rule IT-K′*)

let *?y = case-nat y*
have [*simp*]: *?y ?U ∈ space (Pi$_M$ {0} (λi. M))*
using *y* **by** (*auto simp*: *space-PiM PiE-iff extensional-def split*: *nat.split*)
have *yM*[*measurable*]: *?y ∈ Pi$_M$ {0..<m} (λ-. M) →$_M$ Pi$_M$ {0..<Suc m} (λi. M)* **for** *m*
using *y* **by** (*intro measurable-PiM-single′*) (*auto simp*: *space-PiM PiE-iff extensional-def split*: *nat.split*)

have *y′*: *?y ?U ∈ space (Pi$_M$ {0..<1} (λi. M))*
by (*simp add*: *space-PiM PiE-def y extensional-def split*: *nat.split*)

have *eq1*: *?y −‘ Pi$_E$ {0..<Suc n} F ∩ space (Pi$_M$ {0..<n} (λ-. M)) =*
(if y ∈ F 0 then Pi$_E$ {0..<n} (F∘Suc) else {})
unfolding *set-eq-iff* **using** *y sets.sets-into-space[OF F]*
by (*auto simp*: *space-PiM PiE-iff extensional-def Ball-def split*: *nat.split nat.split-asm*)

have *eq2*: *?y −‘ PF.emb UNIV {0..<Suc n} (Pi$_E$ {0..<Suc n} F) ∩ space (Pi$_M$ UNIV (λ-. M)) =*
(if y ∈ F 0 then PF.emb UNIV {0..<n} (Pi$_E$ {0..<n} (F∘Suc)) else {})
unfolding *set-eq-iff* **using** *y sets.sets-into-space[OF F]*
by (*auto simp*: *space-PiM PiE-iff prod-emb-def extensional-def Ball-def split*: *nat.split nat.split-asm*)

let *?I = indicator (F 0) y*

have *C 1 n (?y ?U) = distr (y.C 0 n ?U) (Π$_M$ i∈{0..<Suc n}. M) ?y*
proof (*induction n*)
case (*Suc m*)

have *C 1 (Suc m) (?y ?U) = distr (y.C 0 m ?U) (Pi$_M$ {0..<Suc m} (λi. M)) ?y ⤜ eP (Suc m)*

```
    using Suc by simp
  also have ... = y.C 0 m ?U ⤜ (λx. eP (Suc m) (?y x))
    by (intro bind-distr[where K=Pi_M {0..<Suc (Suc m)} (λ-. M)]) (simp-all
add: y y.space-C y.sets-C cong: measurable-cong-sets)
  also have ... = y.C 0 m ?U ⤜ (λx. distr (y.eP m x) (Pi_M {0..<Suc (Suc
m)} (λi. M)) ?y)
  proof (intro bind-cong refl)
    fix ω' assume ω': ω' ∈ space (y.C 0 m ?U)
    moreover have K' x (Suc m) (?y ω') = K' y m ω'
      by (auto simp: K'-def)
    ultimately show eP (Suc m) (?y ω') = distr (y.eP m ω') (Pi_M {0..<Suc
(Suc m)} (λi. M)) ?y
      unfolding eP-def y.eP-def
      by (subst distr-distr)
        (auto simp: y.space-C y.sets-P split: nat.split cong: measurable-cong-sets
            intro!: distr-cong measurable-fun-upd[where J={0..<m}])
  qed
  also have ... = distr (y.C 0 m ?U ⤜ y.eP m) (Pi_M {0..<Suc (Suc m)}
(λi. M)) ?y
    by (intro distr-bind[symmetric, OF - - yM]) (auto simp: y.space-C y.sets-C
cong: measurable-cong-sets)
  finally show ?case
    by simp
qed (use y in ‹simp add: PiM-empty distr-return›)
then have C 1 n (case-nat y ?U) (Pi_E {0..<Suc n} F) =
  (distr (y.C 0 n ?U) (Π_M i∈{0..<Suc n}. M) ?y) (Pi_E {0..<Suc n} F) by
simp
also have ... = ?I * y.C 0 n ?U (Pi_E {0..<n} (F ∘ Suc))
  by (subst emeasure-distr) (auto simp: y.sets-C y.space-C eq1 cong: measur-
able-cong-sets)
also have ... = ?I * lim-sequence y (PF.emb UNIV {0..<n} (Pi_E {0..<n}
(F ∘ Suc)))
  using y by (simp add: emeasure-lim-sequence-emb-I0o sets-PiM-I-finite)
 also have ... = distr (lim-sequence y) (Pi_M UNIV (λj. M)) ?y (PF.emb
UNIV {0..<Suc n} (Pi_E {0..<Suc n} F))
  using y by (subst emeasure-distr) (simp-all add: eq2 space-lim-sequence)
finally show emeasure (C 1 n (case-nat y (λ-. undefined))) (Pi_E {0..<Suc n}
F) =
  emeasure (distr (lim-sequence y) (Pi_M UNIV (λj. M)) (case-nat y)) (PF.emb
UNIV J (Pi_E J F'))
  unfolding emb-eq .
qed
also have ... =
  emeasure (K x ⤜ (λy. distr (lim-sequence y) (Pi_M UNIV (λj. M)) (case-nat
y))) (PF.emb UNIV J (Pi_E J F'))
  using J
  by (subst emeasure-bind[where N=PiM UNIV (λ-. M)])
   (auto simp: sets-K x intro!: measurable-distr2[OF - measurable-prob-algebraD[OF
lim-sequence]] cong: measurable-cong-sets)
```

finally show *emeasure* (*lim-sequence x*) (*PF.emb UNIV J* (*Pi*$_E$ *J F'*)) =
emeasure (*K x* ⤜ (λ*y*. *distr* (*lim-sequence y*) (*Pi*$_M$ *UNIV* (λ*j*. *M*)) (*case-nat y*)))
(*PF.emb UNIV J* (*Pi*$_E$ *J F'*)) .
qed

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*))
apply (*simp add*: *lim-sequence-eq cong del*: *AE-cong*)
apply (*subst AE-bind*)
apply (*rule measurable-prob-algebraD*)
apply *measurable*
apply (*auto intro*!: *AE-cong simp add*: *space-K AE-distr-iff*)
done

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*)
unfolding *lim-stream-def* **by** (*simp add*: *space-stream-space*)

lemma *sets-lim-stream*[*measurable-cong*]: *sets* (*lim-stream x*) = *sets* (*stream-space M*)
unfolding *lim-stream-def* **by** *simp*

lemma *lim-stream*[*measurable*]: *lim-stream* ∈ *M* →$_M$ *prob-algebra* (*stream-space M*)
unfolding *lim-stream-def*[*abs-def*] **by** (*intro measurable-distr-prob-space2*[*OF lim-sequence*]) *auto*

lemma *space-stream-space-M-ne*: *x* ∈ *space M* ⟹ *space* (*stream-space M*) ≠ {}
using *sconst-streams*[*of x space M*] **by** (*auto simp*: *space-stream-space*)

lemma *prob-space-lim-stream*: *x* ∈ *space M* ⟹ *prob-space* (*lim-stream x*)
using *measurable-space*[*OF lim-stream*, *of x*] **by** (*simp add*: *space-prob-algebra*)

lemma *lim-stream-eq*:
assumes *x*: *x* ∈ *space M*
shows *lim-stream x* = *do* { *y* ← *K x*; *ω* ← *lim-stream y*; *return* (*stream-space M*) (*y* ## *ω*) }
unfolding *lim-stream-def*
apply (*subst lim-sequence-eq*[*OF x*])
apply (*subst distr-bind*[*OF - - measurable-to-stream*])
subgoal
by (*auto simp*: *sets-K x cong*: *measurable-cong-sets*

intro!: *measurable-prob-algebraD measurable-distr-prob-space2*[**where** M=Pi_M *UNIV* (λj. *M*)] *lim-sequence*) []
subgoal
using *x* **by** (*auto simp add*: *space-K*)
apply (*intro bind-cong refl*)
apply (*subst distr-distr*)
apply (*auto simp*: *space-K sets-lim-sequence x cong*: *measurable-cong-sets intro*!: *distr-cong*)
apply (*subst bind-return-distr'*)
apply (*auto simp*: *space-stream-space-M-ne*)
apply (*subst distr-distr*)
apply (*auto simp*: *space-K sets-lim-sequence x to-stream-nat-case cong*: *measurable-cong-sets intro*!: *distr-cong*)
done

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))
unfolding *lim-stream-eq*[*OF x*]
by (*simp-all add*: *space-K space-lim-stream space-stream-space AE-return AE-bind*[*OF measurable-prob-algebraD P*] *cong*: *AE-cong-simp*)

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* $\cap$ *space* (*stream-space M*)) ∂*K x*)
apply (*subst lim-stream-eq*, *simp*)
apply (*subst emeasure-bind*[*OF - - A*], *simp add*: *prob-space.not-empty prob-space-K*)
apply (*rule measurable-prob-algebraD*)
apply *measurable*
apply (*intro nn-integral-cong*)
apply (*subst bind-return-distr'*)
apply (*auto intro*!: *prob-space.not-empty prob-space-lim-stream simp*: *space-K emeasure-distr*)
apply (*simp add*: *space-lim-stream space-stream-space*)
done

lemma *lim-stream-eq-coinduct*[*case-names in-space step*]:
fixes *R* :: 'a $\Rightarrow$ 'a *stream measure* $\Rightarrow$ *bool*
assumes *x*: *R x B x* $\in$ *space M*
assumes *R*: $\bigwedge x$ *B*. *R x B* $\Longrightarrow$ $\exists B' \in M \rightarrow_M$ *prob-algebra* (*stream-space M*). (*AE y in K x*. *R y* (*B' y*) $\vee$ *lim-stream y* = *B' y*) $\wedge$ *B* = *do* { *y* $\leftarrow$ *K x*; $\omega \leftarrow$ *B' y*; *return* (*stream-space M*) (*y* ## ω) }
shows *lim-stream x* = *B*
using *x*
proof (*coinduction arbitrary*: *x B rule*: *stream-space-coinduct*[**where** *M*=*M*, *case-names*

step])
case (*step x B*)
from *R*[*OF* ‹*R x B*›] **obtain** *B'* **where** *B'*: *B'* ∈ *M* →$_M$ *prob-algebra* (*stream-space M*)
and *ae*: *AE y in K x*. *R y* (*B' y*) ∨ *lim-stream y* = *B' y*
and *eq*: *B* = *K x* ⤜ (λ*y*. *B' y* ⤜ (λω. *return* (*stream-space M*) (*y* ## ω)))
by *blast*
show *?case*
apply (*rule bexI*[*of* - *K x*], *rule bexI*[*OF* - *lim-stream*], *rule bexI*[*OF* - *B'*])
apply (*intro conjI*)
subgoal
using *ae AE-space* **by** *eventually-elim* (*insert* ‹*x*∈*space M*›, *auto simp*: *space-K*)
subgoal
by (*rule lim-stream-eq*) *fact*
subgoal
by (*rule eq*)
subgoal
using *K* ‹*x* ∈ *space M*› **by** (*rule measurable-space*)
done
qed

lemma *prob-space-lim-sequence*: *x* ∈ *space M* ⟹ *prob-space* (*lim-sequence x*)
using *measurable-space*[*OF lim-sequence*, *of x*] **by** (*simp add*: *space-prob-algebra*)

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*∈{*i*::*nat*. *i* ≤ *n*}. *vimage-algebra* (*streams* (*space M*)) (λω . ω !! *i*) *M*)

lemma *measurable-stream-filtration1*: *enat i* ≤ *n* ⟹ (λω. ω !! *i*) ∈ *stream-filtration M n* →$_M$ *M*
by (*auto intro*!: *measurable-SUP1 measurable-vimage-algebra1 snth-in simp*: *stream-filtration-def*)

lemma *measurable-stream-filtration2*:
f ∈ *space N* → *streams* (*space M*) ⟹ (⋀*i*. *enat i* ≤ *n* ⟹ (λ*x*. *f x* !! *i*) ∈ *N* →$_M$ *M*) ⟹ *f* ∈ *N* →$_M$ *stream-filtration M n*
by (*auto simp*: *stream-filtration-def enat-0*
intro!: *measurable-SUP2 measurable-vimage-algebra2 elim*!: *allE*[*of* - *0*::*nat*])

lemma *space-stream-filtration*: *space* (*stream-filtration M n*) = *space* (*stream-space M*)
by (*auto simp add*: *space-stream-space space-Sup-eq-UN stream-filtration-def enat-0 elim*!: *allE*[*of* - *0*])

lemma *sets-stream-filteration-le-stream-space*: *sets* (*stream-filtration M n*) ⊆ *sets* (*stream-space M*)
unfolding *sets-stream-space-eq stream-filtration-def*
by (*intro SUP-subset-mono le-measureD2*) (*auto simp*: *space-Sup-eq-UN enat-0 elim*!: *allE*[*of - 0*])

interpretation *stream-filtration*: *filtration space* (*stream-space M*) *stream-filtration M*
proof
show *space* (*stream-filtration M i*) = *space* (*stream-space M*) **for** *i*
by (*simp add*: *space-stream-filtration*)
show *sets* (*stream-filtration M i*) ⊆ *sets* (*stream-filtration M j*) **if** *i* ≤ *j* **for** *i j*
proof (*rule le-measureD2*)
show *stream-filtration M i* ≤ *stream-filtration M j*
using ‹*i* ≤ *j*› **unfolding** *stream-filtration-def* **by** (*intro SUP-subset-mono*) *auto*
qed (*simp add*: *space-stream-filtration*)
qed

lemma *measurable-stopping-time-stream*:
stopping-time (*stream-filtration M*) *T* ⟹ *T* ∈ *stream-space M* →$_M$ *count-space UNIV*
using *sets-stream-filteration-le-stream-space*
by (*subst measurable-cong-sets*[*OF refl sets-borel-eq-count-space*[*symmetric*, **where** *'a=enat*]])
(*auto intro*!: *measurable-stopping-time simp*: *space-stream-filtration*)

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 −
have {*ω*∈*streams* (*space M*). *T ω* = *0*} ∈ *vimage-algebra* (*streams* (*space M*)) (λ*ω*. *ω* !! *0*) *M*
using *stopping-timeD*[*OF T*, *of 0*] **by** (*simp add*: *stream-filtration-def pred-def enat-0-iff*)
then obtain *A*
where *A*: *A* ∈ *sets M*
and ∗: {*ω* ∈ *streams* (*space M*). *T ω* = *0*} = (λ*ω*. *ω* !! *0*) −‘ *A* ∩ *streams* (*space M*)
by (*auto simp*: *sets-vimage-algebra2 streams-shd*)
have *A* = {*x*∈*space M*. ∀*ω*∈*streams* (*space M*). *T* (*x* ## *ω*) = *0*}
proof *safe*
fix *x ω* **assume** *x* ∈ *A ω* ∈ *streams* (*space M*)
then have *x* ## *ω* ∈ {*ω* ∈ *streams* (*space M*). *T ω* = *0*}
unfolding ∗ **using** *A*[*THEN sets.sets-into-space*] **by** *auto*
then show *T* (*x* ## *ω*) = *0* **by** *auto*
next
fix *x* **assume** *x* ∈ *space M* ∀*ω*∈*streams* (*space M*). *T* (*x* ## *ω*) = *0*

```
    then have ∀ω∈streams (space M). x ## ω ∈ {ω ∈ streams (space M). T ω
= 0}
      by simp
    with ‹x∈space M› show x ∈ A
      unfolding ∗ by (auto simp: streams-empty-iff)
  qed (use A[THEN sets.sets-into-space] in auto)
  with ‹A ∈ sets M› show ?thesis by auto
qed

lemma stopping-time-0:
  assumes T: stopping-time (stream-filtration M) T
    and x: x ∈ space M and ω: ω ∈ streams (space M) T (x ## ω) > 0
    and ω': ω' ∈ streams (space M)
  shows T (x ## ω') > 0
  unfolding zero-less-iff-neq-zero
proof
  assume T (x ## ω') = 0
  with x ω' have x': x ## ω' ∈ {ω ∈ streams (space M). T ω = 0}
    by auto

  have {ω∈streams (space M). T ω = 0} ∈ vimage-algebra (streams (space M))
(λω. ω !! 0) M
    using stopping-timeD[OF T, of 0] by (simp add: stream-filtration-def pred-def
enat-0-iff)
  then obtain A
    where A: A ∈ sets M
      and ∗: {ω ∈ streams (space M). T ω = 0} = (λω. ω !! 0) −‘ A ∩ streams
(space M)
    by (auto simp: sets-vimage-algebra2 streams-shd)
  with x' have x ∈ A
    by auto
  with ω x have x ## ω ∈ (λω. ω !! 0) −‘ A ∩ streams (space M)
    by auto
  with ω show False
    unfolding ∗[symmetric] by auto
qed

lemma stopping-time-epred-SCons:
  assumes T: stopping-time (stream-filtration M) T
    and x: x ∈ space M and ω: ω ∈ streams (space M) T (x ## ω) > 0
  shows stopping-time (stream-filtration M) (λω. epred (T (x ## ω)))
proof (rule stopping-timeI, rule measurable-cong[THEN iffD2])
  show ω ∈ space (stream-filtration M t) ⟹ (epred (T (x ## ω)) ≤ t) = (T (x
## ω) ≤ eSuc t) for t ω
    by (cases T (x ## ω) rule: enat-coexhaust)
     (auto simp add: space-stream-filtration space-stream-space dest!: stopping-time-0[OF
T x ω])
  show Measurable.pred (stream-filtration M t) (λw. T (x ## w) ≤ eSuc t) for t
  proof (rule measurable-compose[of SCons x])
```

show (##) $x \in$ *stream-filtration M t* $\rightarrow_M$ *stream-filtration M* (*eSuc t*)
proof (*intro measurable-stream-filtration2*)
show *enat i* $\leq$ *eSuc t* $\Longrightarrow$ (λ*xa*. (*x* ## *xa*) !! *i*) $\in$ *stream-filtration M t* $\rightarrow_M$ *M* **for** *i*
using ‹*x*∈*space M*›
by (*cases i*) (*auto simp*: *eSuc-enat*[*symmetric*] *intro*!: *measurable-stream-filtration1*)
qed (*auto simp*: *space-stream-filtration space-stream-space* ‹*x*∈*space M*›)
qed (*rule T*[*THEN stopping-timeD*])
qed

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 $\gg\!=$ ($\lambda\omega$. *case T* ω *of*
enat i $\Rightarrow$ *distr* (*lim-stream* (ω !! *i*)) (*stream-space M*) ($\lambda\omega'$. *stake* (*Suc i*) ω @− ω')
| ∞ $\Rightarrow$ *return* (*stream-space M*) ω)
(**is** - = *?L T x*)
using *assms*
proof (*coinduction arbitrary*: *x T rule*: *lim-stream-eq-coinduct*)
case (*step x T*)
note *T* = ‹*stopping-time* (*stream-filtration M*) *T*›[*THEN measurable-stopping-time-stream*, *measurable*]
define *L* **where** *L T x* = *?L T x* **for** *T x*
have *L*[*measurable* (*raw*)]:
(λ(*x*, ω). *T x* ω) $\in$ *N* $\bigotimes_M$ *stream-space M* $\rightarrow_M$ *count-space UNIV* $\Longrightarrow$
f $\in$ *N* $\rightarrow_M$ *M* $\Longrightarrow$ (λ*x*. *L* (*T x*) (*f x*)) $\in$ *N* $\rightarrow_M$ *prob-algebra* (*stream-space M*)
for *f* :: *'a* $\Rightarrow$ *'a* **and** *N T*
unfolding *L-def*
by (*intro measurable-bind-prob-space2*[*OF measurable-compose*[*OF - lim-stream*]] *measurable-case-enat*
measurable-distr-prob-space2[*OF measurable-compose*[*OF - lim-stream*]]
measurable-return-prob-space measurable-stopping-time-stream)
auto

define *S* **where** *S x* = (*if* $\forall\omega$∈*streams* (*space M*). *T* (*x*##ω) = *0 then lim-stream x else L* ($\lambda\omega$. *epred* (*T* (*x* ## ω))) *x*) **for** *x*
then have *S-eq*: $\forall\omega$∈*streams* (*space M*). *T* (*x*##ω) = *0* $\Longrightarrow$ *S x* = *lim-stream x*
¬ ($\forall\omega$∈*streams* (*space M*). *T* (*x*##ω) = *0*) $\Longrightarrow$ *S x* = *L* ($\lambda\omega$. *epred* (*T* (*x* ## ω))) *x* **for** *x*
by *auto*
have [*measurable*]: *S* $\in$ *M* $\rightarrow_M$ *prob-algebra* (*stream-space M*)
unfolding *S-def*[*abs-def*]
by (*subst measurable-If-restrict-space-iff*, *safe intro*!: *L*)
(*auto intro*!: *measurable-stopping-time-All-eq-0 step measurable-restrict-space1*

```
lim-stream
                measurable-compose[OF - measurable-epred] measurable-compose[OF
- T]
                    measurable-Stream measurable-compose[OF measurable-fst]
            simp: measurable-split-conv)

  show ?case
    unfolding L-def[symmetric]
  proof (intro bexI[of - S] conjI AE-I2)
    fix y assume y ∈ space (K x)
     then show (∃x T. y = x ∧ S y = L T x ∧ x ∈ space M ∧ stopping-time
(stream-filtration M) T) ∨
      lim-stream y = S y
      using ‹x∈space M›
      by (cases ∀ω∈streams (space M). T (y##ω) = 0)
          (auto simp add: S-eq space-K intro!: exI[of - λω. epred (T (y ## ω))]
stopping-time-epred-SCons step)
  next
    note ‹x∈space M›[simp]
    have L T x = K x ⤜
      (λy. lim-stream y ⤜ (λω. case T (y##ω) of
            enat i ⇒ distr (lim-stream ((y##ω) !! i)) (stream-space M) (λω'. stake
(Suc i) (y##ω) @− ω')
          | ∞     ⇒ return (stream-space M) (y##ω))) (is - = K x ⤜ ?L')
      unfolding L-def
      apply (subst lim-stream-eq[OF ‹x∈space M›])
      apply (subst bind-assoc[where N=stream-space M and R=stream-space M,
OF measurable-prob-algebraD measurable-prob-algebraD];
          measurable)
      apply (rule bind-cong[OF refl])
      apply (simp add: space-K)
      apply (subst bind-assoc[where N=stream-space M and R=stream-space M,
OF measurable-prob-algebraD measurable-prob-algebraD];
          measurable)
      apply (rule bind-cong[OF refl])
      apply (simp add: space-lim-stream)
    apply (subst bind-return[where N=stream-space M, OF measurable-prob-algebraD])
       apply (measurable; fail) []
      apply (simp add: space-stream-space)
      apply rule
      done
    also have ... = K x ⤜ (λy. S y ⤜ (λω. return (stream-space M) (y ##
ω)))
    proof (intro bind-cong[of K x] refl)
      fix y assume y ∈ space (K x)
      then have [simp]: y ∈ space M
        by (simp add: space-K)
      show ?L' y = S y ⤜ (λω. return (stream-space M) (y ## ω))
      proof cases
```

assume $\forall\,\omega{\in}streams\ (space\ M).\ T\ (y\#\#\omega) = 0$
with x **show** *?thesis*
by (*auto simp*: *S-eq space-lim-stream shift.simps*[*abs-def*] *streams-empty-iff bind-const'*[*OF - prob-space-imp-subprob-space*] *prob-space-lim-stream prob-space.prob-space-distr*
intro!: *bind-return-distr'*[*symmetric*]
cong: *bind-cong-simp*)
next
assume $*$: $\neg\ (\forall\,\omega{\in}streams\ (space\ M).\ T\ (y\#\#\omega) = 0)$
then have *T-pos*: $\omega \in streams\ (space\ M) \Longrightarrow T\ (y\ \#\#\ \omega) \neq 0$ **for** ω
using *stopping-time-0*[*OF* ‹*stopping-time* (*stream-filtration M*) *T*›, *of y* - ω] **by** *auto*
show *?thesis*
apply (*simp add*: *S-eq*(*2*)[*OF* $*$] *L-def*)
apply (*subst bind-assoc*[**where** *N*=*stream-space M* **and** *R*=*stream-space M*, *OF measurable-prob-algebraD measurable-prob-algebraD*];
measurable)
apply (*intro bind-cong refl*)
apply (*auto simp*: *T-pos enat-0 space-lim-stream shift.simps*[*abs-def*] *diff-Suc space-stream-space*
intro!: *bind-return*[**where** *N*=*stream-space M*, *OF measurable-prob-algebraD*, *symmetric*]
bind-distr-return[*symmetric*]
split: *nat.split enat.split*)
done
qed
qed
finally show $L\ T\ x = K\ x \gg\!= (\lambda y.\ S\ y \gg\!= (\lambda\omega.\ return\ (stream\text{-}space\ M)\ (y\ \#\#\ \omega)))$ **.**
qed *fact*
qed *fact*

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 $('a \times real)\ stream$ and $real \Rightarrow 'a$

partial-function (*tailrec*) *trace-at* :: $'a \Rightarrow (real \times 'a)\ stream \Rightarrow real \Rightarrow 'a$
where
trace-at s ω *j* = (*case* ω *of* (t', s')##$\omega \Rightarrow$ *if* $t' \leq j$ *then trace-at* s' ω *j else s*)

lemma *trace-at-simp*[*simp*]: *trace-at s* ((t', s')##ω) *j* = (*if* $t' \leq j$ *then trace-at* s'

ω j else s)
by (*subst trace-at.simps*) *simp*

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 (*split enat.split*; *safe*)
assume *sfirst* (λ*x*. *j* < *fst* (*shd x*)) *ω* = ∞
with *sfirst-finite*[*of* λ*x*. *j* < *fst* (*shd x*) *ω*]
have *alw* (λ*x*. *fst* (*shd x*) ≤ *j*) *ω*
by (*simp add*: *not-ev-iff not-less*)
then show *trace-at s ω j* = *undefined*
by (*induction arbitrary*: *s ω rule*: *trace-at.fixp-induct*) (*auto split*: *stream.split*)
next
show *sfirst* (λ*x*. *j* < *fst* (*shd x*)) *ω* = *enat n* ⟹ *trace-at s ω j* = (*s ## smap snd ω*) !! *n* **for** *n*
proof (*induction n arbitrary*: *s ω*)
case *0* **then show** *?case*
by (*subst trace-at.simps*) (*auto simp add*: *enat-0 sfirst-eq-0 split*: *stream.split*)
next
case (*Suc n*) **show** *?case*
using *sfirst.simps*[*of* λ*x*. *j* < *fst* (*shd x*) *ω*] *Suc.prems Suc.IH*[*of stl ω snd* (*shd ω*)]
by (*cases ω*) (*auto simp add*: *eSuc-enat*[*symmetric*] *split*: *stream.split if-split-asm*)
qed
qed

lemma *trace-at-shift*: *trace-at s* (*smap* (λ(*t*, *s*′). (*t* + *t*′, *s*′)) *ω*) *t* = *trace-at s ω* (*t* − *t*′)
by (*induction arbitrary*: *s ω rule*: *trace-at.fixp-induct*) (*auto split*: *stream.split*)

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*′)
by (*cases x*) (*subst merge-at.code*; *simp*)

7.2 Exponential Distribution

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

lemma *space-exponential*: *space* (*exponential l*) = *UNIV*
by (*simp add*: *exponential-def*)

lemma *sets-exponential*[*measurable-cong*]: *sets* (*exponential l*) = *sets borel*
by (*simp add*: *exponential-def*)

lemma *prob-space-exponential*: *0* < *l* ⟹ *prob-space* (*exponential l*)
unfolding *exponential-def* **by** (*intro prob-space-exponential-density*)

lemma *AE-exponential*: *0* < *l* ⟹ *AE x in exponential l*. *0* < *x*
unfolding *exponential-def* **using** *AE-lborel-singleton*[*of 0*] **by** (*auto simp add*: *AE-density exponential-density-def*)

lemma *emeasure-exponential-Ioi-cutoff*:
assumes *0* < *l*
shows *emeasure* (*exponential l*) {*x* <..} = *exp* (− (*max 0 x*) ∗ *l*)
proof −
interpret *prob-space exponential l*
unfolding *exponential-def* **using** ‹*0*<*l*› **by** (*rule prob-space-exponential-density*)
have ∗: *prob* {*xa* ∈ *space* (*exponential l*). *max 0 x* < *xa*} = *exp* (− *max 0 x* ∗ *l*)
apply (*rule exponential-distributedD-gt*[*OF - -* ‹*0*<*l*›])
apply (*auto simp*: *exponential-def distributed-def*)
apply (*subst* (*6*) *distr-id*[*symmetric*])
apply (*subst* (*2*) *distr-cong*)
apply *simp-all*
done
have *emeasure* (*exponential l*) {*x* <..} = *emeasure* (*exponential l*) {*max 0 x* <..}
using *AE-exponential*[*OF* ‹*0*<*l*›] **by** (*intro emeasure-eq-AE*) *auto*
also have ... = *exp* (− (*max 0 x*) ∗ *l*)
using ∗ **unfolding** *emeasure-eq-measure* **by** (*simp add*: *space-exponential greaterThan-def*)
finally show *?thesis* **.**
qed

lemma *emeasure-exponential-Ioi*:
0 < *l* ⟹ *0* ≤ *x* ⟹ *emeasure* (*exponential l*) {*x* <..} = *exp* (− *x* ∗ *l*)
using *emeasure-exponential-Ioi-cutoff*[*of l x*] **by** *simp*

lemma *exponential-eq-stretch*:
assumes *0* < *l*
shows *exponential l* = *distr* (*exponential 1*) *borel* (λ*x*. (*1*/*l*) ∗ *x*)
proof (*intro measure-eqI*)
fix *A* **assume** *A* ∈ *sets* (*exponential l*)
then have [*measurable*]: *A* ∈ *sets borel*
by (*simp add*: *sets-exponential*)
then have [*measurable*]: (λ*x*. *x* / *l*) −‘ *A* ∈ *sets borel*
by (*rule measurable-sets-borel*[*rotated*]) *simp*
have *emeasure* (*exponential l*) *A* =
(∫⁺*x*. *ennreal l* ∗ (*indicator* (((∗) (*1*/*l*) −‘ *A*) ∩ {*0* ..}) (*l* ∗ *x*) ∗ *ennreal* (*exp* (− (*l* ∗ *x*)))) *∂lborel*)
using ‹*0* < *l*›

by (*auto simp*: *ac-simps emeasure-distr exponential-def emeasure-density exponential-density-def*
ennreal-mult zero-le-mult-iff
intro!: *nn-integral-cong split*: *split-indicator*)
also have ... = ($\int^+ x$. *indicator* (((∗) (*1*/*l*) −‘ *A*) ∩ {*0* ..}) *x* ∗ *ennreal* (*exp* (− *x*)) *∂lborel*)
using ‹*0*<*l*›
apply (*subst nn-integral-stretch*)
apply (*auto simp*: *nn-integral-cmult*)
apply (*simp add*: *ennreal-mult*[*symmetric*] *mult.assoc*[*symmetric*])
done
also have ... = *emeasure* (*distr* (*exponential 1*) *borel* (λ*x*. (*1*/*l*) ∗ *x*)) *A*
by (*auto simp add*: *emeasure-distr exponential-def emeasure-density exponential-density-def*
intro!: *nn-integral-cong split*: *split-indicator*)
finally show *emeasure* (*exponential l*) *A* = *emeasure* (*distr* (*exponential 1*) *borel* (λ*x*. (*1*/*l*) ∗ *x*)) *A* **.**
qed (*simp add*: *sets-exponential*)

lemma *uniform-measure-exponential*:
assumes *0* < *l 0* ≤ *t*
shows *uniform-measure* (*exponential l*) {*t* <..} = *distr* (*exponential l*) *borel* ((+) *t*) (**is** *?L* = *?R*)
proof (*rule measure-eqI-lessThan*)
fix *x*
have *0* < *emeasure* (*exponential l*) {*t*<..}
unfolding *emeasure-exponential-Ioi*[*OF assms*] **by** *simp*
with *assms* **show** *?L* {*x*<..} < ∞
by (*simp add*: *ennreal-divide-eq-top-iff less-top*[*symmetric*] *lessThan-Int-lessThan emeasure-exponential-Ioi*)
have ∗: ((+) *t* −‘ {*x*<..} ∩ *space* (*exponential l*)) = {*x* − *t* <..}
by (*auto simp*: *space-exponential*)
show *?L* {*x*<..} = *?R* {*x*<..}
using *assms* **by** (*simp add*: *lessThan-Int-lessThan emeasure-exponential-Ioi divide-ennreal*
emeasure-distr ∗ *emeasure-exponential-Ioi-cutoff exp-diff*[*symmetric*] *field-simps split*: *split-max*)
qed (*auto simp*: *sets-exponential*)

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* ∗ ($\sum$ *i*∈*J*. *R i*))
proof (*subst emeasure-PiM-emb*)
from *assms* **show** ($\prod$ *i*∈*J*. *emeasure* (*exponential* (*R i*)) {*x*<..}) = *ennreal* (*exp* (− *x* ∗ *sum R J*))
by (*subst prod.cong*[*OF refl emeasure-exponential-Ioi*])
(*auto simp add*: *prod-ennreal exp-sum sum-negf*[*symmetric*] *sum-distrib-left*)
qed (*insert assms*, *auto intro*!: *prob-space-exponential*)

lemma *emeasure-PiM-exponential-Ioi-sequence*:
assumes *R*: *summable R* $\bigwedge i.\ 0 < R\ i\ 0 \leq x$
shows *emeasure* $(\Pi_M\ i{\in}UNIV.\ exponential\ (R\ i))\ (\Pi\ i{\in}UNIV.\ \{x{<}..\}) = exp\ (-\ x * suminf\ R)$
proof −
let *?R* = $\lambda i.\ exponential\ (R\ i)$ **let** *?P* = $\Pi_M\ i{\in}UNIV.\ ?R\ i$
let *?N* = $\lambda n{::}nat.\ prod\text{-}emb\ UNIV\ ?R\ \{..{<}n\}\ (\Pi_E\ i{\in}\{..{<}n\}.\ \{x{<}..\})$
interpret *prob-space ?P*
by (*intro prob-space-PiM prob-space-exponential R*)
have $(\Pi_M\ i{\in}UNIV.\ exponential\ (R\ i))\ (\bigcap n.\ ?N\ n) = (INF\ n.\ (\Pi_M\ i{\in}UNIV.\ exponential\ (R\ i))\ (?N\ n))$
by (*intro INF-emeasure-decseq*[*symmetric*] *decseq-emb-PiE*) (*auto simp*: *incseq-def*)
also have ... = $(INF\ n.\ ennreal\ (exp\ (-\ x * (\sum i{<}n.\ R\ i))))$
using *R* **by** (*intro INF-cong emeasure-PiM-exponential-Ioi-finite*) *auto*
also have ... = $ennreal\ (exp\ (-\ x * (SUP\ n.\ (\sum i{<}n.\ R\ i))))$
using *R*
by (*subst continuous-at-Sup-antimono*[**where** $f{=}\lambda r.\ ennreal\ (exp\ (-\ x * r))$])
(*auto intro*!: *bdd-aboveI2*[**where** $M{=}\sum i.\ R\ i$] *sum-le-suminf summable-mult mult-left-mono*
continuous-mult continuous-at-ennreal continuous-within-exp[*THEN continuous-within-compose3*] *continuous-minus*
simp: *less-imp-le antimono-def image-comp*)
also have ... = $ennreal\ (exp\ (-\ x * (\sum i.\ R\ i)))$
using *R* **by** (*subst suminf-eq-SUP-real*) (*auto simp*: *less-imp-le*)
also have $(\bigcap n.\ ?N\ n) = (\Pi\ i{\in}UNIV.\ \{x{<}..\})$
by (*fastforce simp*: *prod-emb-def Pi-iff PiE-iff space-exponential*)
finally show *?thesis*
using *R* **by** *simp*
qed

lemma *emeasure-PiM-exponential-Ioi-countable*:
assumes *R*: $J \subseteq I$ *countable J* $\bigwedge i.\ i \in I \Longrightarrow 0 < R\ i\ 0 \leq x$ **and** *finite*: *integrable* (*count-space J*) *R*
shows *emeasure* $(\Pi_M\ i{\in}I.\ exponential\ (R\ i))\ (prod\text{-}emb\ I\ (\lambda i.\ exponential\ (R\ i))\ J\ (\Pi_E\ j{\in}J.\ \{x{<}..\})) =$
$exp\ (-\ x * (LINT\ i|count\text{-}space\ J.\ R\ i))$
proof *cases*
assume *finite J* **with** *assms* **show** *?thesis*
by (*subst emeasure-PiM-exponential-Ioi-finite*)
(*auto simp*: *lebesgue-integral-count-space-finite*)
next
assume *infinite J*
let *?R* = $\lambda i.\ exponential\ (R\ i)$ **let** *?P* = $\Pi_M\ i{\in}I.\ ?R\ i$
define *f* **where** *f* = *from-nat-into J*
have *J-eq*: *J* = *range f* **and** *f*: *inj f f* ∈ *UNIV* → *I*
using *from-nat-into-inj-infinite*[*of J*] *range-from-nat-into*[*of J*] ‹*countable J*› ‹*infinite J*› ‹$J \subseteq I$›

by (*auto simp*: *inj-on-def f-def simp del*: *range-from-nat-into*)
have *Bf*: *bij-betw f UNIV J*
unfolding *J-eq* **using** *inj-on-imp-bij-betw*[*OF f*(*1*)] **.**

have *summable-R*: *summable* (λ*i*. *R* (*f i*))
using *finite* **unfolding** *integrable-bij-count-space*[*OF Bf*, *symmetric*] *integrable-count-space-nat-iff*
by (*rule summable-norm-cancel*)

have *emeasure* (Π$_M$ *i*∈*UNIV*. *exponential* (*R* (*f i*))) (Π *i*∈*UNIV*. {*x*<..}) = *exp* (− *x* ∗ (∑ *i*. *R* (*f i*)))
using *finite assms* **unfolding** *J-eq* **by** (*intro emeasure-PiM-exponential-Ioi-sequence*[*OF summable-R*]) *auto*
also have (Π$_M$ *i*∈*UNIV*. *exponential* (*R* (*f i*))) = *distr ?P* (Π$_M$ *i*∈*UNIV*. *exponential* (*R* (*f i*))) (λω. λ*i*∈*UNIV*. ω (*f i*))
using *R* **by** (*intro distr-PiM-reindex*[*symmetric*, *OF - f*] *prob-space-exponential*) *auto*
also have ... (Π *i*∈*UNIV*. {*x*<..}) = *?P* ((λω. λ*i*∈*UNIV*. ω (*f i*)) −‘ (Π *i*∈*UNIV*. {*x*<..}) ∩ *space ?P*)
using *f*(*2*) **by** (*intro emeasure-distr infprod-in-sets*) (*auto simp*: *Pi-iff*)
also have (λω. λ*i*∈*UNIV*. ω (*f i*)) −‘ (Π *i*∈*UNIV*. {*x*<..}) ∩ *space ?P* = *prod-emb I ?R J* (Π$_E$ *j*∈*J*. {*x*<..})
by (*auto simp*: *prod-emb-def space-PiM space-exponential Pi-iff J-eq*)
also have (∑ *i*. *R* (*f i*)) = (*LINT i*|*count-space J*. *R i*)
using *finite*
by (*subst integral-count-space-nat*[*symmetric*])
(*auto simp*: *integrable-bij-count-space*[*OF Bf*] *integral-bij-count-space*[*OF Bf*])
finally show *?thesis* **.**
qed

lemma *AE-PiM-exponential-suminf-infty*:
fixes *R* :: *nat* ⇒ *real*
assumes *R*: ⋀*n*. *0* < *R n* **and** *finite*: (∑ *n*. *ennreal* (*1* / *R n*)) = *top*
shows *AE* ω *in* Π$_M$ *n*∈*UNIV*. *exponential* (*R n*). (∑ *n*. *ereal* (ω *n*)) = ∞
proof −
let *?P* = Π$_M$ *n*∈*UNIV*. *exponential* (*R n*)
interpret *prob-space exponential* (*R n*) **for** *n*
by (*intro prob-space-exponential R*)
interpret *product-prob-space* λ*n*. *exponential* (*R n*) *UNIV*
proof qed

have *AE-pos*: *AE* ω *in ?P*. ∀ *i*. *0* < ω *i*
unfolding *AE-all-countable* **by** (*intro AE-PiM-component allI prob-space-exponential R AE-exponential*) *simp*

have *indep*: *indep-vars* (λ*i*. *borel*) (λ*i x*. *x i*) *UNIV*
using *PiM-component*
apply (*subst P.indep-vars-iff-distr-eq-PiM*)
apply (*auto simp*: *restrict-UNIV distr-id2*)

```
    apply (subst distr-id2)
     apply (intro sets-PiM-cong)
      apply (auto simp: sets-exponential cong: distr-cong)
    done

  have [simp]: 0 ≤ x + x * R i ⟷ 0 ≤ x for x i
    using zero-le-mult-iff[of x 1 + R i] R[of i] by (simp add: field-simps)

  have (∫⁺ω. eexp (∑ n. − ereal (ω n)) ∂?P) = (∫⁺ω. (INF n. ∏ i<n. eexp (−
ereal (ω i))) ∂?P)
  proof (intro nn-integral-cong-AE, use AE-pos in eventually-elim)
    fix ω :: nat ⇒ real assume ω: ∀ i. 0 < ω i
    show eexp (∑ n. − ereal (ω n)) = (∏ n. ∏ i<n. eexp (− ereal (ω i)))
    proof (rule LIMSEQ-unique[OF - LIMSEQ-INF])
      show (λi. ∏ i<i. eexp (− ereal (ω i))) ⟶ eexp (∑ n. − ereal (ω n))
        using ω by (intro eexp-suminf summable-minus-ereal summable-ereal-pos)
(auto intro: less-imp-le)
      show decseq (λn. ∏ i<n. eexp (− ereal (ω i)))
        using ω by (auto simp: decseq-def intro!: prod-mono3 intro: less-imp-le)
    qed
  qed
  also have ... = (INF n. (∫⁺ω. (∏ i<n. eexp (− ereal (ω i))) ∂?P))
  proof (intro nn-integral-monotone-convergence-INF-AE')
    show AE ω in ?P. (∏ i<Suc n. eexp (− ereal (ω i))) ≤ (∏ i<n. eexp (− ereal
(ω i))) for n
      using AE-pos
    proof eventually-elim
      case (elim ω)
      show ?case
        by (rule prod-mono3) (auto simp: elim le-less)
    qed
  qed (auto simp: less-top[symmetric])
  also have ... = (INF n. (∏ i<n. (∫⁺ω. eexp (− ereal (ω i)) ∂?P)))
  proof (intro INF-cong refl indep-vars-nn-integral)
    show indep-vars (λ-. borel) (λi ω. eexp (− ereal (ω i))) {..<n} for n
    proof (rule indep-vars-compose2[of - - - λi x. eexp(− ereal x)])
      show indep-vars (λi. borel) (λi x. x i) {..<n}
        by (rule indep-vars-subset[OF indep]) auto
    qed auto
  qed auto
  also have ... = (INF n. (∏ i<n. R i * (∫⁺x. indicator {0 ..} ((1 + R i) * x)
* ennreal (exp (− ((1 + R i) * x))) ∂lborel)))
    by (subst product-nn-integral-component)
     (auto simp: field-simps exponential-def nn-integral-density ennreal-mult'[symmetric]
ennreal-mult''[symmetric]
           exponential-density-def exp-diff exp-minus nn-integral-cmult[symmetric]
          intro!: INF-cong prod.cong nn-integral-cong split: split-indicator)
  also have ... = (INF n. (∏ i<n. ennreal (R i / (1 + R i))))
  proof (intro INF-cong prod.cong refl)
```

```
    show R i * (∫⁺ x. indicator {0..} ((1 + R i) * x) * ennreal (exp (- ((1 + R
i) * x))) ∂lborel) =
      ennreal (R i / (1 + R i)) for i
      using nn-intergal-power-times-exp-Ici[of 0] ‹0 < R i›
      by (subst nn-integral-stretch[where c=1 + R i])
          (auto simp: mult.assoc[symmetric] ennreal-mult''[symmetric] less-imp-le
mult.commute)
  qed
  also have ... = (INF n. ennreal (∏i<n. R i / (1 + R i)))
   using R by (intro INF-cong refl prod-ennreal divide-nonneg-nonneg) (auto simp:
less-imp-le)
  also have ... = (INF n. ennreal (inverse (∏i<n. (1 + R i) / R i)))
    by (subst prod-inversef[symmetric]) simp-all
  also have ... = (INF n. inverse (ennreal (∏i<n. (1 + R i) / R i)))
    using R by (subst inverse-ennreal) (auto intro!: prod-pos divide-pos-pos simp:
add-pos-pos)
  also have ... = inverse (SUP n. ennreal (∏i<n. (1 + R i) / R i))
    by (subst continuous-at-Sup-antimono [where f = inverse])
    (auto simp: antimono-def image-comp intro!: continuous-on-imp-continuous-within[OF
continuous-on-inverse-ennreal ↑])
  also have (SUP n. ennreal (∏i<n. (1 + R i) / R i)) = top
  proof (cases SUP n. ennreal (∏i<n. (1 + R i) / R i))
    case (real r)
    have (λn. ennreal (∏i<n. (1 + R i) / R i)) ⟶ r
      using R unfolding real(2)[symmetric]
        by (intro LIMSEQ-SUP monoI ennreal-leI prod-mono2) (auto intro!: di-
vide-nonneg-nonneg add-nonneg-nonneg intro: less-imp-le)
    then have (λn. (∏i<n. (1 + R i) / R i)) ⟶ r
      by (rule tendsto-ennrealD)
      (use R real in ‹auto intro!: always-eventually prod-nonneg divide-nonneg-nonneg
add-nonneg-nonneg intro: less-imp-le›)
    moreover have (1 + R i) / R i = 1 / R i + 1 for i
      using ‹0 < R i› by (auto simp: field-simps)
    ultimately have convergent (λn. ∏i<n. 1 / R i + 1)
      by (auto simp: convergent-def)
    then have summable (λi. 1 / R i)
      using R by (subst summable-iff-convergent-prod) (auto intro: less-imp-le)
    moreover have 0 ≤ 1 / R i for i
      using R by (auto simp: less-imp-le)
    ultimately show ?thesis
      using finite ennreal-suminf-neq-top[of λi. 1 / R i] by blast
  qed
  finally have (∫⁺ω. eexp (∑ n. - ereal (ω n)) ∂?P) = 0
    by simp
  then have AE ω in ?P. eexp (∑ n. - ereal (ω n)) = 0
    by (subst (asm) nn-integral-0-iff-AE) auto
  then show ?thesis
    using AE-pos
  proof eventually-elim
```

show $(\forall i.\ 0 < \omega\ i) \Longrightarrow eexp\ (\sum n.\ -\ ereal\ (\omega\ n)) = 0 \Longrightarrow (\sum n.\ ereal\ (\omega\ n)) = \infty$ **for** ω
apply (*auto simp del*: *uminus-ereal.simps simp add*: *uminus-ereal.simps*[*symmetric*] *intro*!: *summable-iff-suminf-neq-top intro*: *less-imp-le*)
apply (*subst* (*asm*) *suminf-minus-ereal*)
apply (*auto intro*!: *summable-ereal-pos intro*: *less-imp-le*)
done
qed
qed

7.3 Transition Rates

locale *transition-rates* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow real$
assumes *R-nonneg*[*simp*]: $\bigwedge x\ y.\ 0 \leq R\ x\ y$
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 −
let *?P* = *point-measure UNIV* $(R\ x)$
interpret *finite-measure ?P*
proof
show *emeasure ?P* (*space ?P*) $\neq \infty$
using *finite-weight*
by (*simp add*: *emeasure-density point-measure-def less-top*)
qed
from *countable-support emeasure-point-measure-finite2*[*of* {-} *UNIV R x*]
show *?thesis*
by (*simp add*: *emeasure-eq-measure less-le*)
qed

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

lemma *ennreal-escape-rate*: *ennreal* (*escape-rate* x) $= (\int^+ y.\ R\ x\ y\ \partial count\text{-}space\ UNIV)$
using *finite-weight*[*of x*] **unfolding** *escape-rate-def*

by (*intro nn-integral-eq-integral*[*symmetric*]) (*auto simp*: *integrable-iff-bounded*)

lemma *escape-rate-pos*: *0* < *escape-rate x*
using *positive-weight* **unfolding** *ennreal-escape-rate*[*symmetric*] **by** *simp*

lemma *nonneg-escape-rate*[*simp*]: *0* ≤ *escape-rate x*
using *escape-rate-pos*[*THEN less-imp-le*] **.**

lemma *prob-space-exponential-escape-rate*: *prob-space* (*exponential* (*escape-rate x*))
using *escape-rate-pos* **by** (*rule prob-space-exponential*)

lemma *measurable-escape-rate*[*measurable*]: *escape-rate* ∈ *count-space UNIV* →$_M$ *borel*
by *auto*

lemma *measurable-exponential-escape-rate*[*measurable*]: (λ*x*. *exponential* (*escape-rate x*)) ∈ *count-space UNIV* →$_M$ *prob-algebra borel*
by (*auto simp*: *space-prob-algebra sets-exponential prob-space-exponential-escape-rate*)

interpretation *pmf-as-function* **.**

lift-definition *J* :: *'a* ⇒ *'a pmf* **is** λ*x y*. *R x y* / *escape-rate x*
proof *safe*
show *0* ≤ *R x y* / *escape-rate x* **for** *x y*
by (*auto intro*!: *integral-nonneg-AE divide-nonneg-nonneg R-nonneg simp*: *escape-rate-def*)
show (∫$^+$*y*. *R x y* / *escape-rate x* ∂*count-space UNIV*) = *1* **for** *x*
using *escape-rate-pos*[*of x*]
by (*auto simp add*: *divide-ennreal*[*symmetric*] *nn-integral-divide ennreal-escape-rate*[*symmetric*] *intro*!: *ennreal-divide-self*)
qed

lemma *set-pmf-J*: *set-pmf* (*J x*) = *I x*
using *escape-rate-pos*[*of x*] **by** (*auto simp*: *set-pmf-iff J.rep-eq less-le*)

interpretation *exp-esc*: *pair-prob-space distr* (*exponential* (*escape-rate x*)) *borel* ((+) *t*) *J x* **for** *x*
proof −
interpret *prob-space distr* (*exponential* (*escape-rate x*)) *borel* ((+) *t*)
by (*intro prob-space.prob-space-distr prob-space-exponential-escape-rate*) *simp*
show *pair-prob-space* (*distr* (*exponential* (*escape-rate x*)) *borel* ((+) *t*)) (*measure-pmf* (*J x*))
by *standard*
qed

7.4 Continuous-time Kernel

definition *K* :: (*real* × *'a*) ⇒ (*real* × *'a*) *measure* **where**
K = (λ(*t*, *x*). (*distr* (*exponential* (*escape-rate x*)) *borel* ((+) *t*)) ⨂$_M$ *J x*)

interpretation *K*: *discrete-Markov-process borel* $\bigotimes_M$ *count-space UNIV K*
proof
show *K* ∈ *borel* $\bigotimes_M$ *count-space UNIV* →$_M$ *prob-algebra* (*borel* $\bigotimes_M$ *count-space UNIV*)
unfolding *K-def*
apply *measurable*
apply (*rule measurable-snd*[*THEN measurable-compose*])
apply (*auto simp*: *space-prob-algebra prob-space-measure-pmf*)
done
qed

interpretation *DTMC*: *MC-syntax J* **.**

lemma *in-space-S*[*simp*]: *x* ∈ *space S*
by (*simp add*: *space-pair-measure*)

lemma *in-space-T*[*simp*]: *x* ∈ *space T*
by (*simp add*: *space-pair-measure space-stream-space*)

lemma *in-space-lim-stream*: *ω* ∈ *space* (*K.lim-stream x*)
unfolding *K.space-lim-stream space-stream-space*[*symmetric*] **by** *simp*

lemma *prob-space-K-lim*: *prob-space* (*K.lim-stream x*)
using *K.lim-stream*[*THEN measurable-space*] **by** (*simp add*: *space-prob-algebra*)

definition *select-first* :: *'a* ⇒ (*'a* ⇒ *real*) ⇒ *'a* ⇒ *bool*
where *select-first x p y* = (*y* ∈ *I x* ∧ (∀ *y'*∈*I x* − {*y*}. *p y* < *p y'*))

lemma *select-firstD1*: *select-first x p y* ⟹ *y* ∈ *I x*
by (*simp add*: *select-first-def*)

lemma *select-first-unique*:
assumes *y*: *select-first x p y1 select-first x p y2* **shows** *y1* = *y2*
proof −
have *y1* ≠ *y2* ⟹ *p y1* < *p y2 y1* ≠ *y2* ⟹ *p y2* < *p y1*
using *y* **by** (*auto simp*: *select-first-def*)
then show *y1* = *y2*
by (*rule-tac ccontr*) *auto*
qed

lemma *The-select-first*[*simp*]: *select-first x p y* ⟹ *The* (*select-first x p*) = *y*
by (*intro the-equality select-first-unique*)

lemma *select-first-INF*:
select-first x p y ⟹ (*INF x*∈*I x*. *p x*) = *p y*
by (*intro antisym cINF-greatest cINF-lower bdd-belowI2*[**where** *m*=*p y*])
(*auto simp*: *select-first-def le-less*)

lemma *measurable-select-first*[*measurable*]:
(λ*p*. *select-first x p y*) ∈ (Π$_M$ *y*∈*I x*. *borel*) →$_M$ *count-space UNIV*
using *I-countable* **unfolding** *select-first-def* **by** (*intro measurable-pred-countable pred-intros-conj1*′) *measurable*

lemma *measurable-THE-select-first*[*measurable*]:
(λ*p*. *The* (*select-first x p*)) ∈ (Π$_M$ *y*∈*I x*. *borel*) →$_M$ *count-space UNIV*
by (*rule measurable-THE*) (*auto intro*: *select-first-unique I-countable dest*: *select-firstD1*)

lemma *sets-S-eq*: *sets S* = *sigma-sets UNIV* { {*t* ..} × *A* | *t A*. *A* ⊆ − *I x* ∨ (∃ *s*∈*I x*. *A* = {*s*}) }
proof (*subst sets-pair-eq*)
let *?CI* = λ*a*::*real*. {*a* ..} **let** *?Ea* = *range ?CI*

show *?Ea* ⊆ *Pow* (*space borel*) *sets borel* = *sigma-sets* (*space borel*) *?Ea*
unfolding *borel-Ici* **by** *auto*
show *?CI*‘*Rats* ⊆ *?Ea* (⋃ *i*∈*Rats*. *?CI i*) = *space borel*
using *Rats-dense-in-real*[*of x* − *1 x* **for** *x*] **by** (*auto intro*: *less-imp-le*)

let *?Eb* = *Pow* (− *I x*) ∪ (λ*s*. {*s*}) ‘ *I x*
have *b* ∈ *sigma-sets UNIV* (*Pow* (− *I x*) ∪ (λ*s*. {*s*}) ‘ *I x*) **for** *b*
proof −
have *b* = (*b* − *I x*) ∪ (⋃ *x*∈*b* ∩ *I x*. {*x*})
by *auto*
also have ... ∈ *sigma UNIV* (*Pow* (− *I x*) ∪ (λ*s*. {*s*}) ‘ *I x*)
using *I-countable* **by** (*intro sets.Un sets.countable-UN*′) *auto*
finally show *?thesis*
by *simp*
qed
then show *sets* (*count-space UNIV*) = *sigma-sets* (*space* (*count-space UNIV*)) *?Eb*
by *auto*
show *countable* ({− *I x*} ∪ (⋃ *s*∈*I x*. {{*s*}}))
using *I-countable* **by** *auto*
show *sets* (*sigma* (*space borel* × *space* (*count-space UNIV*)) {*a* × *b* |*a b*. *a* ∈ *?Ea* ∧ *b* ∈ *?Eb*}) =
sigma-sets UNIV {{*t* ..} × *A* |*t A*. *A* ⊆ − *I x* ∨ (∃ *s*∈*I x*. *A* = {*s*})}
apply *simp*
apply (*intro arg-cong*[**where** *f*=*sigma-sets* -])
apply *auto*
done
qed (*auto intro*: *countable-rat*)

7.5 Kernel equals Parallel Choice

abbreviation *PAR* :: ′*a* ⇒ (′*a* ⇒ *real*) *measure*
where
PAR x ≡ (Π$_M$ *y*∈*I x*. *exponential* (*R x y*))

lemma *PAR-least*:
assumes *y*: $y \in I\ x$
shows $PAR\ x\ \{p{\in}space\ (PAR\ x).\ t \leq p\ y \wedge select\text{-}first\ x\ p\ y\} =$
$emeasure\ (exponential\ (escape\text{-}rate\ x))\ \{t\ ..\} * ennreal\ (pmf\ (J\ x)\ y)$
proof −
let $?E = \lambda y.\ exponential\ (R\ x\ y)$ **let** $?P' = \Pi_M\ y{\in}I\ x - \{y\}.\ ?E\ y$
interpret P': *prob-space* $?P'$
by (*intro prob-space-PiM prob-space-exponential*) *simp*
have $*$: $PAR\ x = (\Pi_M\ y{\in}insert\ y\ (I\ x - \{y\}).\ ?E\ y)$
using *y* **by** (*intro PiM-cong*) *auto*
have $0 < R\ x\ y$
using *y* **by** *simp*
have $**$: $(\lambda(x,\ X).\ X(y := x)) \in exponential\ (R\ x\ y) \bigotimes_M Pi_M\ (I\ x - \{y\})\ (\lambda i.$
$exponential\ (R\ x\ i)) \rightarrow_M PAR\ x$
using *y*
apply (*subst measurable-cong-sets*[*OF sets-pair-measure-cong*[*OF sets-exponential*
sets-PiM-cong[*OF refl sets-exponential*]] *sets-PiM-cong*[*OF refl sets-exponential*]])
apply *measurable*
apply (*rule measurable-fun-upd*[**where** $J{=}I\ x - \{y\}$])
apply *auto*
done
have $PAR\ x\ \{p{\in}space\ (PAR\ x).\ t \leq p\ y \wedge (\forall\, y'{\in}I\ x{-}\{y\}.\ p\ y < p\ y')\} =$
$(\int^+ ty.\ indicator\ \{t..\}\ ty * ?P'\ \{p{\in}space\ ?P'.\ \forall\, y'{\in}I\ x{-}\{y\}.\ ty < p\ y'\}\ \partial ?E\ y)$
unfolding $*$ **using** ‹$y \in I\ x$›
apply (*subst distr-pair-PiM-eq-PiM*[*symmetric*])
apply (*auto intro*!: *prob-space-exponential simp*: *emeasure-distr insert-absorb*)
apply (*subst emeasure-distr*[*OF* $**$])
subgoal
using *I-countable* **by** (*auto simp*: *pred-def*[*symmetric*])
apply (*subst P'.emeasure-pair-measure-alt*)
subgoal
using *I-countable*[*of x*]
apply (*intro measurable-sets*[*OF* $**$])
apply (*auto simp*: *pred-def*[*symmetric*])
done
apply (*auto intro*!: *nn-integral-cong arg-cong2*[**where** $f{=}emeasure$] *split*: *split-indicator*
if-split-asm
simp: *space-exponential space-PiM space-pair-measure PiE-iff extensional-def*)
done
also have $\ldots = (\int^+ ty.\ indicator\ \{t..\}\ ty * ennreal\ (exp\ (-\ ty * (escape\text{-}rate\ x$
$-\ R\ x\ y)))\ \partial ?E\ y)$
apply (*intro nn-integral-cong-AE*)
using *AE-exponential*[*OF* ‹$0 < R\ x\ y$›]
proof *eventually-elim*
fix *ty* :: *real* **assume** $0 < ty$
have *escape-rate x* =
$(\int^+ y'.\ R\ x\ y' * indicator\ \{y\}\ y'\ \partial count\text{-}space\ UNIV) + (\int^+ y'.\ R\ x\ y' *$
$indicator\ (I\ x - \{y\})\ y'\ \partial count\text{-}space\ UNIV)$

unfolding *ennreal-escape-rate* **by** (*subst nn-integral-add*[*symmetric*]) (*auto simp*: *less-le split*: *split-indicator intro*!: *nn-integral-cong*)
also have ... = *R x y* + (∫⁺*y'*. *R x y'* ∂*count-space* (*I x* − {*y*}))
by (*auto simp add*: *nn-integral-count-space-indicator less-le simp del*: *nn-integral-indicator-singleton intro*!: *arg-cong2*[**where** *f*=(+)] *nn-integral-cong split*: *split-indicator*)
finally have (∫⁺*y'*. *R x y'* ∂*count-space* (*I x* − {*y*})) = *escape-rate x* − *R x y* ∧ *R x y* ≤ *escape-rate x*
using *escape-rate-pos*[*THEN less-imp-le*]
by (*cases* (∫⁺*y'*. *R x y'* ∂*count-space* (*I x* − {*y*}))) (*auto simp*: *add-top ennreal-plus*[*symmetric*] *simp del*: *ennreal-plus*)
then have *integrable* (*count-space* (*I x* − {*y*})) (*R x*) (*LINT y'*|*count-space* (*I x* − {*y*}). *R x y'*) = *escape-rate x* − *R x y*
by (*auto simp*: *nn-integral-eq-integrable*)
then have *?P'* (*prod-emb* (*I x*−{*y*}) *?E* (*I x*−{*y*}) (Π$_E$ *j*∈(*I x*−{*y*}). {*ty*<..})) = *exp* (− *ty* ∗ (*escape-rate x* − *R x y*))
using *I-countable* ‹*0* < *ty*› **by** (*subst emeasure-PiM-exponential-Ioi-countable*) *auto*
also have *prod-emb* (*I x*−{*y*}) *?E* (*I x*−{*y*}) (Π$_E$ *j*∈(*I x*−{*y*}). {*ty*<..}) = {*p*∈*space ?P'*. ∀ *y'*∈*I x*−{*y*}. *ty* < *p y'*}
by (*simp add*: *set-eq-iff prod-emb-def space-PiM space-exponential ac-simps Pi-iff*)
finally show *indicator* {*t*..} *ty* ∗ *?P'* {*p*∈*space ?P'*. ∀ *y'*∈*I x*−{*y*}. *ty* < *p y'*} =
indicator {*t*..} *ty* ∗ *ennreal* (*exp* (− *ty* ∗ (*escape-rate x* − *R x y*)))
by *simp*
qed
also have ... = (∫⁺*ty*. *ennreal* (*R x y*) ∗ (*ennreal* (*exp* (− *ty* ∗ *escape-rate x*)) ∗ *indicator* {*max 0 t*..} *ty*) ∂*lborel*)
by (*auto simp add*: *exponential-def exponential-density-def nn-integral-density ennreal-mult*[*symmetric*] *exp-add*[*symmetric*] *field-simps intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = (*R x y* / *escape-rate x*) ∗ *emeasure* (*exponential* (*escape-rate x*)) {*max 0 t*..}
using *escape-rate-pos*[*of x*]
by (*auto simp*: *exponential-def exponential-density-def emeasure-density nn-integral-cmult*[*symmetric*] *ennreal-mult*[*symmetric*] *split*: *split-indicator intro*!: *nn-integral-cong*)
also have ... = *pmf* (*J x*) *y* ∗ *emeasure* (*exponential* (*escape-rate x*)) {*t*..}
using *AE-exponential*[*OF escape-rate-pos*[*of x*]]
by (*intro arg-cong2*[**where** *f*=(∗)] *emeasure-eq-AE*) (*auto simp*: *J.rep-eq*)
finally show *?thesis*
using *assms* **by** (*simp add*: *mult-ac select-first-def*)
qed

lemma *AE-PAR-least*: *AE p in PAR x*. ∃ *y*∈*I x*. *select-first x p y*
proof −
have *D*: *disjoint-family-on* (λ*y*. {*p* ∈ *space* (*PAR x*). *select-first x p y*}) (*I x*)
by (*auto simp*: *disjoint-family-on-def dest*: *select-first-unique*)
have *PAR x* {*p*∈*space* (*PAR x*). ∃ *y*∈*I x*. *select-first x p y*} =

PAR x ($\bigcup y{\in}I\ x.\ \{p{\in}space\ (PAR\ x).\ select\text{-}first\ x\ p\ y\}$)
by (*auto intro*!: *arg-cong2*[**where** *f*=*emeasure*])
also have ... = ($\int^+ y.\ PAR\ x\ \{p{\in}space\ (PAR\ x).\ select\text{-}first\ x\ p\ y\}\ \partial count\text{-}space\ (I\ x)$)
using *I-countable* **by** (*intro emeasure-UN-countable D*) *auto*
also have ... = ($\int^+ y.\ PAR\ x\ \{p{\in}space\ (PAR\ x).\ 0 \le p\ y \wedge select\text{-}first\ x\ p\ y\}\ \partial count\text{-}space\ (I\ x)$)
proof (*intro nn-integral-cong emeasure-eq-AE*, *goal-cases*)
case (*1 y*) **with** *AE-PiM-component*[*of I x* $\lambda y.$ *exponential* (*R x y*) *y* (<) *0*] *AE-exponential*[*of R x y*] **show** *?case*
by (*auto simp*: *prob-space-exponential*)
qed (*insert I-countable*, *auto*)
also have ... = ($\int^+ y.\ emeasure\ (exponential\ (escape\text{-}rate\ x))\ \{0\ ..\} * ennreal\ (pmf\ (J\ x)\ y)\ \partial count\text{-}space\ (I\ x)$)
by (*auto simp add*: *PAR-least intro*!: *nn-integral-cong*)
also have ... = ($\int^+ y.\ emeasure\ (exponential\ (escape\text{-}rate\ x))\ \{0\ ..\}\ \partial J\ x$)
by (*auto simp*: *nn-integral-measure-pmf nn-integral-count-space-indicator ac-simps pmf-eq-0-set-pmf set-pmf-J*
simp del: *nn-integral-const intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = *1*
using *AE-exponential*[*of escape-rate x*]
by (*auto intro*!: *prob-space.emeasure-eq-1-AE prob-space-exponential simp*: *escape-rate-pos less-imp-le*)
finally show *?thesis*
using *I-countable*
by (*subst prob-space.AE-iff-emeasure-eq-1 prob-space-PiM prob-space-exponential*)
(*auto intro*!: *prob-space-PiM prob-space-exponential simp del*: *Set.bex-simps*(*6*))
qed

lemma *K-alt*: *K* (*t*, *x*) = *distr* ($\Pi_M\ y{\in}I\ x.\ exponential\ (R\ x\ y)$) *S* ($\lambda p.\ (t + (INF\ y{\in}I\ x.\ p\ y),\ The\ (select\text{-}first\ x\ p))$) (**is** - = *?R*)
proof (*rule measure-eqI-generator-eq-countable*)
let *?E* = { $\{t\ ..\} \times A \mid (t{::}real)\ A.\ A \subseteq - I\ x \vee (\exists s{\in}I\ x.\ A = \{s\})$ }
show *Int-stable ?E*
apply (*auto simp*: *Int-stable-def*)
subgoal for *t1 A1 t2 A2*
by (*intro exI*[*of - max t1 t2*] *exI*[*of - A1* $\cap$ *A2*]) *auto*
subgoal for *t1 t2 y1 y2*
by (*intro exI*[*of - max t1 t2*] *exI*[*of -* $\{y1\} \cap \{y2\}$]) *auto*
done
show *sets* (*K* (*t*, *x*)) = *sigma-sets UNIV ?E*
unfolding *K.sets-K*[*OF in-space-S*] **by** (*subst sets-S-eq*) *rule*
show *sets ?R* = *sigma-sets UNIV ?E*
using *sets-S-eq* **by** *simp*
show *countable* (($\lambda(t, A).\ \{t\ ..\} \times A$) ` ($\mathbb{Q} \times$ ($\{-I\ x\} \cup (\lambda s.\ \{s\})$ ` *I x*)))
by (*intro countable-image countable-SIGMA countable-rat countable-Un I-countable*) *auto*

have $*$: (+) *t* −` $\{t'..\} \cap$ *space* (*exponential* (*escape-rate x*)) = $\{t' - t..\}$ **for** t'

```
    by (auto simp: space-exponential)
  { fix X assume X ∈ ?E
    then consider
        t' s where s ∈ I x X = {t' ..} × {s}
      | t' A where A ⊆ − I x X = {t' ..} × A
      by auto
    then show K (t, x) X = ?R X
    proof cases
      case 1
      have AE p in PAR x. (t' − t ≤ p s ∧ select-first x p s) =
            (t' ≤ t + (∏x∈I x. p x) ∧ The (select-first x p) = s)
        using AE-PAR-least by eventually-elim (auto dest: select-first-unique simp:
select-first-INF)
      with 1 I-countable show ?thesis
        by (auto simp add: K-def measure-pmf.emeasure-pair-measure-Times emea-
sure-distr emeasure-pmf-single ∗
          PAR-least[symmetric] intro!: emeasure-eq-AE)
    next
      case 2
      moreover
      then have emeasure (measure-pmf (J x)) A = 0
        by (subst AE-iff-measurable[symmetric, where P=λx. x ∉ A])
          (auto simp: AE-measure-pmf-iff set-pmf-J subset-eq)
      moreover
      have PAR x ((λp. (t + ∏(p ‘ (I x)), The (select-first x p))) −‘ ({t'..} × A)
∩ space (PAR x)) = 0
        using ‹A ⊆ − I x› AE-PAR-least[of x] I-countable
        by (subst AE-iff-measurable[symmetric, where P=λp. (t + ∏(p ‘ (I x)),
The (select-first x p)) ∉ {t'..} × A])
          (auto simp del: all-simps(5) simp add: imp-ex imp-conjL subset-eq)
      ultimately show ?thesis
        using I-countable
          by (simp add: K-def measure-pmf.emeasure-pair-measure-Times emea-
sure-distr ∗)
    qed }

  interpret prob-space K ts for ts
    by (rule K.prob-space-K) simp
  show emeasure (K (t, x)) a ≠ ∞ for a
    using emeasure-finite by simp
qed (insert Rats-dense-in-real[of x − 1 x for x], auto, blast intro: less-imp-le)

lemma AE-K: AE y in K x. fst x < fst y ∧ snd y ∈ J (snd x)
  unfolding K-def split-beta
  apply (subst exp-esc.AE-pair-iff[symmetric])
  apply measurable
  apply (simp-all add: AE-distr-iff AE-measure-pmf-iff exponential-def AE-density
exponential-density-def cong del: AE-cong)
  using AE-lborel-singleton[of 0]
```

apply *eventually-elim*
apply *simp*
done

lemma *AE-lim-stream*:
AE ω *in K.lim-stream x.* $\forall i$. *snd* $((x \#\# \omega) !! i) \in DTMC.acc`` \{snd\ x\} \wedge snd\ (\omega !! i) \in J\ (snd\ ((x \#\# \omega) !! i)) \wedge fst\ ((x \#\# \omega) !! i) < fst\ (\omega !! i)$
(**is** *AE* ω *in K.lim-stream x.* $\forall i$. *?P* ω *i*)
unfolding *AE-all-countable*
proof
let *?F* $= \lambda i\ x\ \omega$. *fst* $((x \#\# \omega) !! i)$ **and** *?S* $= \lambda i\ x\ \omega$. *snd* $((x \#\# \omega) !! i)$
fix *i* **show** *AE* ω *in K.lim-stream x. ?P* ω *i*
proof (*induction i arbitrary*: *x*)
case *0* **with** *AE-K*[*of x*] **show** *?case*
by (*subst K.AE-lim-stream*) (*auto simp add*: *space-pair-measure cong del*: *AE-cong*)
next
case (*Suc i*)
show *?case*
proof (*subst K.AE-lim-stream*, *goal-cases*)
case *2* **show** *?case*
using *DTMC.countable-reachable*
by (*intro measurable-compose-countable-restrict*[**where** *f*=*?S* (*Suc i*) *x*])
(*simp-all del*: *Image-singleton-iff*)
next
case *3* **show** *?case*
apply (*simp del*: *AE-conj-iff cong del*: *AE-cong*)
using *AE-K*[*of x*]
apply *eventually-elim*
subgoal premises *K-prems* **for** *y*
using *Suc*
by *eventually-elim* (*insert K-prems*, *auto intro*: *converse-rtrancl-into-rtrancl*)
done
qed (*simp add*: *space-pair-measure*)
qed
qed

lemma *measurable-merge-at*[*measurable*]: $(\lambda(\omega, \omega').$ *merge-at* $\omega\ j\ \omega') \in (T \bigotimes_M T) \to_M T$
proof (*rule measurable-stream-space2*)
define *F* **where** *F x n* = (*case x of* $(\omega$::(*real* $\times$ $'a)$ *stream*, $\omega') \Rightarrow$ *merge-at* $\omega\ j\ \omega')$!! *n* **for** *x n*
fix *n*
have $(\lambda x.$ *F x n*$) \in$ *stream-space S* $\bigotimes_M$ *stream-space S* $\to_M$ *S*
proof (*induction n*)
case *0* **then show** *?case*
by (*simp add*: *F-def split-beta*$'$ *stream.case-eq-if*)
next
case (*Suc n*)

from *Suc*[*measurable*]
have *eq*: *F x* (*Suc n*) = (*case fst x of* (*t*, *s*) ## *ω* ⇒ *if t* ≤ *j then F* (*ω*, *snd x*) *n else snd x* !! *Suc n*) **for** *x*
by (*auto simp*: *F-def split*: *prod.split stream.split*)
show *?case*
unfolding *eq stream.case-eq-if* **by** *measurable*
qed
then show (*λx*. (*case x of* (*ω*, *ω'*) ⇒ *merge-at ω j ω'*) !! *n*) ∈ *stream-space S* ⨂$_M$ *stream-space S* →$_M$ *S*
unfolding *F-def* **by** *auto*
qed

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

lemma *measurable-trace-at'*: (*λ*((*s*, *j*), *ω*). *trace-at s ω j*) ∈ ((*count-space UNIV* ⨂$_M$ *borel*) ⨂$_M$ *T*) →$_M$ *count-space UNIV*
unfolding *trace-at-eq split-beta'* **by** *measurable*

lemma *K-time-split*:
assumes *t* ≤ *j* **and** [*measurable*]: *f* ∈ *S* →$_M$ *borel*
shows (∫$^+$*x*. *f x* ∗ *indicator* {*j* <..} (*fst x*) *∂K* (*t*, *s*)) = (∫$^+$*x*. *f x ∂K* (*j*, *s*)) ∗ *exponential* (*escape-rate s*) {*j* − *t* <..}
proof −
have (∫$^+$ *y*. ∫$^+$ *x*. *f* (*t* + *x*, *y*) ∗ *indicator* {*j*<..} (*t* + *x*) *∂exponential* (*escape-rate s*) *∂J s*) =
(∫$^+$ *y*. ∫$^+$ *x*. *f* (*t* + *x*, *y*) ∗ *indicator* {*j* − *t*<..} *x ∂exponential* (*escape-rate s*) *∂J s*)
by (*intro nn-integral-cong*) (*auto split*: *split-indicator*)
also have ... = (∫$^+$ *y*. ∫$^+$ *x*. *f* (*t* + *x*, *y*) *∂uniform-measure* (*exponential* (*escape-rate s*)) {*j*−*t* <..} *∂J s*) ∗
emeasure (*exponential* (*escape-rate s*)) {*j* − *t* <..}
using ‹*t* ≤ *j*› *escape-rate-pos*
by (*subst nn-integral-uniform-measure*)
(*auto simp*: *nn-integral-divide ennreal-divide-times emeasure-exponential-Ioi*)
also have ... = (∫$^+$ *y*. ∫$^+$ *x*. *f* (*j* + *x*, *y*) *∂exponential* (*escape-rate s*) *∂J s*) ∗
emeasure (*exponential* (*escape-rate s*)) {*j* − *t* <..}
using ‹*t* ≤ *j*› *escape-rate-pos* **by** (*simp add*: *uniform-measure-exponential nn-integral-distr*)
finally show *?thesis*
by (*simp add*: *K-def exp-esc.nn-integral-snd*[*symmetric*] *nn-integral-distr*)
qed

lemma *K-in-space*[*simp*]: *K x* ∈ *space* (*prob-algebra S*)
by (*rule measurable-space* [*OF K.K*]) *simp*

lemma *L-in-space*[*simp*]: *K.lim-stream x* ∈ *space* (*prob-algebra T*)
by (*rule measurable-space* [*OF K.lim-stream*]) *simp*

7.6 Markov Chain Property

lemma *lim-time-split*:
$t \leq j \Longrightarrow K.lim\text{-}stream\ (t,\ s) = do\ \{\ \omega \leftarrow K.lim\text{-}stream\ (t,\ s)\ ;\ \omega' \leftarrow K.lim\text{-}stream\ (j,\ trace\text{-}at\ s\ \omega\ j)\ ;\ return\ T\ (merge\text{-}at\ \omega\ j\ \omega')\}$
(**is** - $\Longrightarrow$ - = *?DO t s*)
proof (*coinduction arbitrary*: *t s rule*: *K.lim-stream-eq-coinduct*)
case *step* **let** *?L* = *K.lim-stream*

note *measurable-compose*[*OF measurable-prob-algebraD measurable-emeasure-subprob-algebra*, *measurable* (*raw*)]

define B' **where** $B' = (\lambda(t',\ s).\ if\ t' \leq j\ then\ ?DO\ t'\ s\ else\ ?L\ (t',\ s))$
show *?case*
proof (*intro bexI conjI AE-I2*)
show [*measurable*]: $B' \in S \rightarrow_M prob\text{-}algebra\ T$
unfolding B'*-def* **by** *measurable*
show $(\exists\, t\ s.\ y = (t,\ s) \wedge B'\ y = ?DO\ t\ s \wedge t \leq j) \vee ?L\ y = B'\ y$ **for** *y*
by (*cases y*; *cases fst y* $\leq j$) (*auto simp*: B'*-def*)
let *?C* = $\lambda x.$ *do* $\{\ \omega \leftarrow ?L\ x;\ \omega' \leftarrow ?L\ (j,\ trace\text{-}at\ s\ (x\#\#\omega)\ j);\ return\ T\ (merge\text{-}at\ (x\#\#\omega)\ j\ \omega')\ \}$
have *?DO t s* = *do* $\{\ x \leftarrow K\ (t,\ s);\ ?C\ x\ \}$
apply (*subst K.lim-stream-eq*[*OF in-space-S*])
apply (*subst bind-assoc*[*OF measurable-prob-algebraD measurable-prob-algebraD*])
apply (*subst measurable-cong-sets*[*OF K.sets-K*[*OF in-space-S*] *refl*])
apply *measurable*
apply (*subst bind-assoc*[*OF measurable-prob-algebraD measurable-prob-algebraD*])
apply *measurable*
apply (*subst bind-cong*[*OF refl bind-cong*[*OF refl bind-return*[*OF measurable-prob-algebraD*]]])
apply *measurable*
done
also have ... = $K\ (t,\ s) \gg\!= (\lambda y.\ B'\ y \gg\!= (\lambda\omega.\ return\ T\ (y\ \#\#\ \omega)))$ (**is** *?DO'* = *?R*)
proof (*rule measure-eqI*)
have *sets ?DO'* = *sets T*
by (*intro sets-bind'*[*OF K-in-space*]) *measurable*
moreover have *sets ?R* = *sets T*
by (*intro sets-bind'*[*OF K-in-space*]) *measurable*
ultimately show *sets ?DO'* = *sets ?R*
by *simp*
fix *A* **assume** $A \in$ *sets ?DO'*
then have *A*[*measurable*]: $A \in T$
unfolding ‹*sets ?DO'* = *sets T*› **.**
have *?DO' A* = $(\int^+ x.\ ?C\ x\ A\ \partial K\ (t,\ s))$
by (*subst emeasure-bind-prob-algebra*[*OF K-in-space*]) *measurable*
also have ... = $(\int^+ x.\ ?C\ x\ A * indicator\ \{..\ j\}\ (fst\ x)\ \partial K\ (t,\ s)) +$
$(\int^+ x.\ ?C\ x\ A * indicator\ \{j <..\}\ (fst\ x)\ \partial K\ (t,\ s))$
by (*subst nn-integral-add*[*symmetric*]) (*auto intro*!: *nn-integral-cong split*: *split-indicator*)

```
    also have (∫⁺x. ?C x A * indicator {.. j} (fst x) ∂K (t, s)) =
      (∫⁺y. emeasure (B' y ⤜ (λω. return T (y ## ω))) A * indicator {.. j}
(fst y) ∂K (t, s))
     proof (intro nn-integral-cong ennreal-mult-right-cong refl arg-cong2[where
f=emeasure])
      fix x :: real × 'a assume indicator {..j} (fst x) ≠ (0::ennreal)
      then have fst x ≤ j
        by (auto split: split-indicator-asm)
      then show ?C x = (B' x ⤜ (λω. return T (x ## ω)))
        apply (cases x)
        apply (simp add: B'-def)
      apply (subst bind-assoc[OF measurable-prob-algebraD measurable-prob-algebraD])
        apply measurable
      apply (subst bind-assoc[OF measurable-prob-algebraD measurable-prob-algebraD])
        apply measurable
        apply (subst bind-return)
        apply measurable
        done
    qed
    also have (∫⁺x. ?C x A * indicator {j <..} (fst x) ∂K (t, s)) =
      (∫⁺y. emeasure (B' y ⤜ (λω. return T (y ## ω))) A * indicator {j <..}
(fst y) ∂K (t, s))
    proof −
      have ∗: (+) t −‘ {j<..} = {j − t <..}
        by auto

      have (∫⁺x. ?C x A * indicator {j <..} (fst x) ∂K (t, s)) =
        (∫⁺x. ?L (j, s) A * indicator {j <..} (fst x) ∂K (t, s))
         by (intro nn-integral-cong ennreal-mult-right-cong refl arg-cong2[where
f=emeasure])
          (auto simp: K.sets-lim-stream bind-return'' bind-const' prob-space-K-lim
prob-space-imp-subprob-space split: split-indicator-asm)
       also have ... = ?L (j, s) A * exponential (escape-rate s) {j − t <..}
      by (subst nn-integral-cmult) (simp-all add: K-def exp-esc.nn-integral-snd[symmetric]
emeasure-distr space-exponential ∗)
       also have ... = (∫⁺x. emeasure (?L x ⤜ (λω. return T (x ## ω))) A
∂K (j, s)) * exponential (escape-rate s) {j − t <..}
         by (subst K.lim-stream-eq) (auto simp: emeasure-bind-prob-algebra[OF
K-in-space - A])
      also have ... = (∫⁺y. emeasure (?L y ⤜ (λω. return T (y ## ω))) A *
indicator {j <..} (fst y) ∂K (t, s))
        using ‹t ≤ j› by (rule K-time-split[symmetric]) measurable
      also have ... = (∫⁺y. emeasure (B' y ⤜ (λω. return T (y ## ω))) A *
indicator {j <..} (fst y) ∂K (t, s))
         by (intro nn-integral-cong ennreal-mult-right-cong refl arg-cong2[where
f=emeasure])
          (auto simp add: B'-def split: split-indicator-asm)
      finally show ?thesis .
    qed
```

also have ($\int^+ y$. *emeasure* (B' *y* $\gg\!=$ ($\lambda\omega$. *return T* (*y* ## ω))) *A* $*$ *indicator* {.. *j*} (*fst y*) ∂K (*t*, *s*)) +
($\int^+ y$. *emeasure* (B' *y* $\gg\!=$ ($\lambda\omega$. *return T* (*y* ## ω))) *A* $*$ *indicator* {*j* <..} (*fst y*) ∂K (*t*, *s*)) =
($\int^+ y$. *emeasure* (B' *y* $\gg\!=$ ($\lambda\omega$. *return T* (*y* ## ω))) *A* ∂K (*t*, *s*))
by (*subst nn-integral-add*[*symmetric*]) (*auto intro*!: *nn-integral-cong split*: *split-indicator*)
also have ... = *emeasure* (*K* (*t*, *s*) $\gg\!=$ (λy. B' *y* $\gg\!=$ ($\lambda\omega$. *return T* (*y* ## ω)))) *A*
by (*rule emeasure-bind-prob-algebra*[*symmetric*, *OF K-in-space - A*]) *auto*
finally show *?DO'* *A* = *emeasure* (*K* (*t*, *s*) $\gg\!=$ (λy. B' *y* $\gg\!=$ ($\lambda\omega$. *return T* (*y* ## ω)))) *A* **.**
qed
finally show *?DO t s* = *K* (*t*, *s*) $\gg\!=$ (λy. B' *y* $\gg\!=$ ($\lambda\omega$. *return T* (*y* ## ω)))
.
qed
qed (*simp add*: *space-pair-measure*)

lemma *K-eq*: *K* (*t*, *s*) = *distr* (*exponential* (*escape-rate s*) $\bigotimes_M$ *J s*) *S* (λ(*t'*, *s*). (*t* + *t'*, *s*))
proof −
have *distr* (*exponential* (*escape-rate s*)) *borel* ((+) *t*) $\bigotimes_M$ *distr* (*J s*) (*J s*) (λx. *x*) =
distr (*exponential* (*escape-rate s*) $\bigotimes_M$ *J s*) (*borel* $\bigotimes_M$ *J s*) (λ(*x*, *y*). (*t* + *x*, *y*))
proof (*intro pair-measure-distr*)
interpret *prob-space distr* (*measure-pmf* (*J s*)) (*measure-pmf* (*J s*)) (λx. *x*)
by (*intro measure-pmf.prob-space-distr*) *simp*
show *sigma-finite-measure* (*distr* (*measure-pmf* (*J s*)) (*measure-pmf* (*J s*)) (λx. *x*))
by *unfold-locales*
qed *auto*
also have ... = *distr* (*exponential* (*escape-rate s*) $\bigotimes_M$ *J s*) *S* (λ(*x*, *y*). (*t* + *x*, *y*))
by (*intro distr-cong refl sets-pair-measure-cong*) *simp*
finally show *?thesis*
by (*simp add*: *K-def*)
qed

lemma *K-shift*: *K* (*t* + *t'*, *s*) = *distr* (*K* (*t*, *s*)) *S* (λ(*t*, *s*). (*t* + *t'*, *s*))
unfolding *K-eq* **by** (*subst distr-distr*) (*auto simp*: *comp-def split-beta' ac-simps*)

lemma *K-not-empty*: *space* (*K x*) ≠ {}
by (*simp add*: *K-def space-pair-measure split*: *prod.split*)

lemma *lim-stream-not-empty*: *space* (*K.lim-stream x*) ≠ {}
by (*simp add*: *K.space-lim-stream space-pair-measure split*: *prod.split*)

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 (*coinduction arbitrary*: *t s rule*: *K.lim-stream-eq-coinduct*)
case *step* **then show** *?case*
proof (*intro bexI*[*of* - λ(*t*, *s*). *?D* (*t* − *t'*) *s*] *conjI*)
show *?D t s* = *K* (*t* + *t'*, *s*) ⤜ (λ*y*. (*case y of* (*t*, *s*) ⇒ *?D* (*t* − *t'*) *s*) ⤜ (λω. *return T* (*y* ## ω)))
apply (*subst K.lim-stream-eq*[*OF in-space-S*])
apply (*subst K-shift*)
apply (*subst distr-bind*[*OF measurable-prob-algebraD K-not-empty*])
apply (*measurable*; *fail*)
apply (*measurable*; *fail*)
apply (*subst bind-distr*[*OF* - *measurable-prob-algebraD K-not-empty*])
apply (*measurable*; *fail*)
apply (*measurable*; *fail*)
apply (*intro bind-cong refl*)
apply (*subst distr-bind*[*OF measurable-prob-algebraD lim-stream-not-empty*])
apply (*measurable*; *fail*)
apply (*measurable*; *fail*)
apply (*simp add*: *distr-return split-beta*)
apply (*subst bind-distr*[*OF* - *measurable-prob-algebraD lim-stream-not-empty*])
apply (*measurable*; *fail*)
apply (*measurable*; *fail*)
apply (*simp add*: *split-beta'*)
done
qed (*auto cong*: *conj-cong intro*!: *exI*[*of* - - − *t'*])
qed *simp*

lemma *lim-0*: *K.lim-stream* (*t*, *s*) = *distr* (*K.lim-stream* (*0*, *s*)) *T* (*smap* (λ(*t'*, *s*). (*t'* + *t*, *s*)))
using *lim-shift*[*of 0 t s*] **by** *simp*

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*)))
using *less-Suc-eq-0-disj* **by** *auto*

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 (*rule measure-eqI-PiM-sequence*)
show *sets ?D* = *sets* (*PiM UNIV* (λ-. *borel*)) *sets ?P* = *sets* (*PiM UNIV* (λ-.

```
borel))
    by (auto intro!: sets-PiM-cong simp: sets-exponential)
  have [measurable]: ts ∈ space S
    by auto
  { interpret prob-space ?D
    by (intro prob-space.prob-space-distr K.prob-space-lim-stream measurable-abs-UNIV)
auto
    show finite-measure ?D
      by unfold-locales }

  interpret E: prob-space exponential 1
    by (rule prob-space-exponential) simp
  interpret P: product-prob-space λ-. exponential 1 UNIV
    by unfold-locales

  let distr - - (?f ts) = ?D

  fix A :: nat ⇒ real set and n :: nat assume A[measurable]: ⋀i. A i ∈ sets borel
  define n' where n' = Suc n
  have emeasure ?D (prod-emb UNIV (λ-. borel) {..n} (Pi_E {..n} A)) =
    emeasure (K.lim-stream ts) {ω∈space (stream-space S). ∀ i<n'. ?f ts ω i ∈ A
i}
    apply (subst emeasure-distr)
      apply (auto intro!: measurable-abs-UNIV arg-cong[where f=emeasure -])
     apply (auto simp: prod-emb-def K.space-lim-stream space-pair-measure n'-def)
    done
  also have ... = (∏ i<n'. emeasure (exponential 1) (A i))
    using A
  proof (induction n' arbitrary: A ts)
    case 0 then show ?case
      using prob-space.emeasure-space-1[OF prob-space-K-lim]
      by (simp add: K.space-lim-stream space-pair-measure)
  next
    case (Suc n A ts)
    from Suc.prems[measurable]
    have [measurable]: ts ∈ space S
      by auto

    have emeasure (K.lim-stream ts) {ω ∈ space (stream-space S). ∀ i<Suc n. ?f
ts ω i ∈ A i} =
      (∫⁺ts'. indicator (A 0) (escape-rate (snd ts) * (fst ts' − fst ts)) *
        emeasure (K.lim-stream ts') {ω ∈ space (stream-space S). ∀ i<n. ?f ts' ω i
∈ A (Suc i)} ∂K ts)
      apply (subst K.emeasure-lim-stream)
      apply simp
       apply measurable
        apply (auto intro!: nn-integral-cong arg-cong2[where f=emeasure] split:
split-indicator
        simp: ball-less-Suc-eq)
```

```
    done
  also have ... = (∫⁺ts'. indicator (A 0) (escape-rate (snd ts) * (fst ts' − fst
ts)) ∂K ts) *
    (∏i<n. emeasure (exponential 1) (A (Suc i)))
    by (subst Suc.IH) (simp-all add: nn-integral-multc)
  also have (∫⁺ts'. indicator (A 0) (escape-rate (snd ts) * (fst ts' − fst ts)) ∂K
ts) =
    (∫⁺t. indicator (A 0) (escape-rate (snd ts) * t) ∂exponential (escape-rate (snd
ts)))
    by (simp add: K-def exp-esc.nn-integral-snd[symmetric] nn-integral-distr split:
prod.split)
  also have ... = emeasure (exponential 1) (A 0)
    using escape-rate-pos[of snd ts]
    by (subst exponential-eq-stretch) (simp-all add: nn-integral-distr)
  also have emeasure (exponential 1) (A 0) * (∏i<n. emeasure (exponential 1)
(A (Suc i))) =
    (∏i<Suc n. emeasure (exponential 1) (A i))
    by (rule prod.lessThan-Suc-shift[symmetric])
  finally show ?case .
 qed
 also have ... = emeasure ?P (prod-emb UNIV (λ-. borel) {..<n'} (Pi_E {..<n'}
A))
  using P.emeasure-PiM-emb[of {..<n'} A] by (simp add: prod-emb-def space-exponential)
 finally show emeasure ?D (prod-emb UNIV (λ-. borel) {..n} (Pi_E {..n} A)) =
   emeasure ?P (prod-emb UNIV (λ-. borel) {..n} (Pi_E {..n} A))
   by (simp add: n'-def lessThan-Suc-atMost)
qed

lemma AE-explosion-infty:
 assumes bdd: bdd-above (range escape-rate)
 shows AE ω in K.lim-stream x. explosion ω = ∞
proof −
 have escape-rate undefined ≤ (SUP x. escape-rate x)
   using bdd by (intro cSUP-upper) auto
 then have SUP-escape-pos: 0 < (SUP x. escape-rate x)
   using escape-rate-pos[of undefined] by simp
 then have SUP-escape-nonneg: 0 ≤ (SUP x. escape-rate x)
   by (rule less-imp-le)

 have [measurable]: x ∈ space S by auto
 have (∑ i. 1::ennreal) = top
  by (rule sums-unique[symmetric]) (auto simp: sums-def of-nat-tendsto-top-ennreal)
 then have AE ω in (PiM UNIV (λ-. exponential 1)). (∑ i. ereal (ω i)) = ∞
   by (intro AE-PiM-exponential-suminf-infty) auto
 then have AE ω in K.lim-stream x.
   (∑ i. ereal (escape-rate (snd ((x##ω) !! i)) * (fst (ω !! i) − fst ((x##ω) !!
i)))) = ∞
   apply (subst (asm) lim-stream-timediff-eq-exponential-1[symmetric, of x])
   apply (subst (asm) AE-distr-iff)
```

```
    apply (auto intro!: measurable-abs-UNIV)
    done
  then show ?thesis
    using AE-lim-stream
  proof eventually-elim
    case (elim ω)
    then have le: fst ((x##ω) !! n) ≤ fst ((x ## ω) !! m) if n ≤ m for n m
      by (intro lift-Suc-mono-le[OF - ‹n ≤ m›, of λi. fst ((x ## ω) !! i)]) (auto
intro: less-imp-le)
    have [simp]: fst x ≤ fst ((x##ω) !! i) fst ((x##ω) !! i) ≤ fst (ω !! i) for i
      using le[of i Suc i] le[of 0 i] by auto

    have (∑ i. ereal (escape-rate (snd ((x ## ω) !! i)) * (fst (ω !! i) − fst ((x ##
ω) !! i)))) =
      (SUP n. ∑ i<n. ereal (escape-rate (snd ((x ## ω) !! i)) * (fst (ω !! i) − fst
((x ## ω) !! i))))
      by (intro suminf-ereal-eq-SUP) (auto intro!: mult-nonneg-nonneg)
    also have ... ≤ (SUP n. (SUP x. escape-rate x) * (ereal (fst ((x ## ω) !! n))
− ereal (fst x)))
    proof (intro SUP-least SUP-upper2)
      fix n
      have (∑ i<n. ereal (escape-rate (snd ((x ## ω) !! i)) * (fst (ω !! i) − fst ((x
## ω) !! i)))) ≤
        (∑ i<n. ereal ((SUP i. escape-rate i) * (fst (ω !! i) − fst ((x ## ω) !! i))))
        using elim bdd by (intro sum-mono) (auto intro!: cSUP-upper)
      also have ... = (SUP i. escape-rate i) * (∑ i<n. fst ((x ## ω) !! Suc i) −
fst ((x ## ω) !! i))
        using elim bdd by (subst sum-ereal) (auto simp: sum-distrib-left)
      also have ... = (SUP i. escape-rate i) * (fst ((x ## ω) !! n) − fst x)
        by (subst sum-lessThan-telescope) simp
      finally show (∑ i<n. ereal (escape-rate (snd ((x ## ω) !! i)) * (fst (ω !! i)
− fst ((x ## ω) !! i))))
        ≤ (SUP x. escape-rate x) * (ereal (fst ((x ## ω) !! n)) − ereal (fst x))
        by simp
    qed simp
    also have ... = (SUP x. escape-rate x) * ((SUP n. ereal (fst ((x ## ω) !! n)))
− ereal (fst x))
      using elim SUP-escape-nonneg by (subst SUP-ereal-mult-left) (auto simp:
SUP-ereal-minus-left[symmetric])
    also have (SUP n. ereal (fst ((x ## ω) !! n))) = explosion ω
      unfolding explosion-def
      apply (intro SUP-eq)
      subgoal for i by (intro bexI[of - i]) auto
      subgoal for i by (intro bexI[of - Suc i]) auto
      done
    finally show explosion ω = ∞
      using elim SUP-escape-pos by (cases explosion ω) (auto split: if-splits)
  qed
qed
```

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*)
  by (*cases x*) (*subst trace-in.simps*; *auto*)

lemma *trace-in-eq-lfp*:
  trace-in ss t = *lfp* (*λF s*. *λ*(*t'*, *s'*)##*ω* ⇒ *if t* < *t' then s* ∈ *ss else F s' ω*)
 unfolding *trace-in-def* **by** (*intro arg-cong*[**where** *f*=*lfp*] *ext*) (*auto split*: *stream.splits*)

lemma *trace-in-shiftD*: *trace-in ss t s ω* ⟹ *trace-in ss* (*t* + *t'*) *s* (*smap* (*λ*(*t*, *s'*). (*t* + *t'*, *s'*)) *ω*)
  by (*induction rule*: *trace-in.induct*) *auto*

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*
  using *trace-in-shiftD*[*of ss t s smap* (*λ*(*t*, *s'*). (*t* + *t'*, *s'*)) *ω* − *t'*]
    trace-in-shiftD[*of ss t* − *t' s ω t'*]
  by (*auto simp add*: *stream.map-comp prod.case-eq-if*)

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 −
  let *?F* = *λF*. *λ*(*t*, *s*, (*t'*, *s'*)##*ω*) ⇒ *if t* < *t' then s* ∈ *ss else F* (*t*, *s'*, *ω*)
  have [*measurable*]: *Measurable.pred* (*count-space UNIV*) (*λx*. *x* ∈ *ss*)
    by *simp*
  have *trace-in ss* = (*λt s ω*. *lfp ?F* (*t*, *s*, *ω*))
    unfolding *trace-in-def*
    apply (*subst lfp-arg*)
    apply (*subst lfp-rolling*[**where** *g*=*λF t s ω*. *F* (*t*, *s*, *ω*)])
    subgoal by (*auto simp*: *mono-def le-fun-def split*: *stream.splits*)
    subgoal by (*auto simp*: *mono-def le-fun-def split*: *stream.splits*)
    subgoal
      by (*intro arg-cong*[**where** *f*=*lfp*])
       (*auto simp*: *mono-def le-fun-def split-beta' not-less fun-eq-iff split*: *stream.splits intro*!: *arg-cong*[**where** *f*=*lfp*])

```
    done
  then have eq: (λ(t, s, ω). trace-in ss t s ω) = lfp ?F
    by simp
  have sup-continuous ?F
    by (auto simp: sup-continuous-def fun-eq-iff split: stream.splits)
  then show ?thesis
    unfolding eq
  proof (rule measurable-lfp)
    fix F assume ?M F then show ?M (?F F)
      by measurable
  qed
qed

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 (λx. trace-in ss (f x) (g x) (h x))
  using measurable-compose[of λx. (f x, g x, h x) M, OF - measurable-trace-in'[of
ss]] by simp

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 −
  have ∗: (SIGMA x:space (count-space UNIV ⨂M borel). {ω ∈ streams (space
S). trace-in {s'} (snd x) (fst x) ω}) =
    {x∈space ((count-space UNIV ⨂M borel) ⨂M T). trace-in {s'} (snd (fst x))
(fst (fst x)) (snd x)}
    by (auto simp: space-pair-measure)

  note measurable-trace-at'[measurable]
  show ?thesis
    unfolding p-def[abs-def] split-beta'
    by (rule measure-measurable-prob-algebra2[where N=T])
      (auto simp: K.space-lim-stream ∗ pred-def[symmetric]
            intro!: pred-count-space-const1 measurable-trace-at'[unfolded split-beta'])
qed

lemma p-nonpos: assumes t ≤ 0 shows p s s' t = of-bool (s = s')
proof −
  have AE ω in K.lim-stream (0, s). trace-in {s'} t s ω = (s = s')
  proof (subst K.AE-lim-stream)
    show AE y in K (0, s). AE ω in K.lim-stream y. trace-in {s'} t s (y ## ω)
= (s = s')
      using AE-K
    proof eventually-elim
      fix y :: real × 'a assume fst (0, s) < fst y ∧ snd y ∈ set-pmf (J (snd (0,
```

```
s)))
      with ‹t≤0› show AE ω in K.lim-stream y. trace-in {s'} t s (y ## ω) = (s
= s')
        by (cases y) auto
    qed
  qed auto
  then have p s s' t = P(ω in K.lim-stream (0, s). s = s')
   unfolding p-def by (intro prob-space.prob-eq-AE K.prob-space-lim-stream) auto
  then show ?thesis
    using prob-space.prob-space[OF K.prob-space-lim-stream] by simp
qed

lemma p-0: p s s' 0 = of-bool (s = s')
  using p-nonpos[of 0] by simp

lemma in-sets-T[measurable (raw)]: Measurable.pred T P ⟹ {ω. P ω} ∈ sets T
  unfolding pred-def by simp

lemma distr-id': sets M = sets N ⟹ distr M N (λx. x) = M
  by (subst distr-cong[of M M N M - λx. x] ) simp-all

lemma p-nonneg[simp]: 0 ≤ p s s' t
  by (simp add: p-def)

lemma p-le-1[simp]: p s s' t ≤ 1
  unfolding p-def by (intro prob-space.prob-le-1 K.prob-space-lim-stream) simp

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 −
  have *: (+) 0 = (λx::real. x)
    by auto
  interpret L: prob-space K.lim-stream x for x
    by (rule K.prob-space-lim-stream) simp
  interpret E: prob-space exponential (escape-rate s) for s
    by (intro escape-rate-pos prob-space-exponential)
  have p s s'' t = emeasure (K.lim-stream (0, s)) {ω∈space T. trace-in {s''} t s
ω}
   by (simp add: p-def L.emeasure-eq-measure K.space-lim-stream space-stream-space
del: in-space-T)
  also have ... = (∫⁺y. emeasure (K.lim-stream y) {ω∈space T. trace-in {s''} t
s (y##ω) } ∂K (0, s))
    apply (subst K.lim-stream-eq[OF in-space-S])
    apply (subst emeasure-bind-prob-algebra[OF K-in-space])
    apply (measurable; fail)
    apply (measurable; fail)
    apply (subst bind-return-distr'[OF lim-stream-not-empty])
```

apply (*measurable*; *fail*)
apply (*simp add*: *emeasure-distr*)
done
also have ... = ($\int^+ y$. *indicator* {t <..} (*fst y*) * *of-bool* ($s = s''$) + *indicator* {0<..t} (*fst y*) * p (*snd y*) s'' (t − *fst y*) ∂K (0, s))
apply (*intro nn-integral-cong-AE*)
using *AE-K*
apply *eventually-elim*
subgoal for y
using *L.emeasure-space-1*
apply (*cases y*)
apply (*auto split*: *split-indicator simp del*: *in-space-T*)
subgoal for t' *s2*
unfolding *p-def L.emeasure-eq-measure*[*symmetric*] *K.space-lim-stream space-stream-space*[*symmetric*]
by (*subst lim-0*) (*simp add*: *emeasure-distr*)
subgoal
by (*auto split*: *split-indicator cong*: *rev-conj-cong simp add*: *K.space-lim-stream space-stream-space simp del*: *in-space-T*)
done
done
also have ... = ($\int^+ u$. $\int^+ s'$. *indicator* {t <..} u * *of-bool* ($s = s''$) + *indicator* {0<..t} u * p s' s'' (t − u) ∂J s ∂*exponential* (*escape-rate s*))
unfolding *K-def*
by (*simp add*: *K-def measure-pmf.nn-integral-fst*[*symmetric*] * *distr-id' sets-exponential*)
also have ... = *ennreal* (*exp* (− t * *escape-rate s*) * *of-bool* ($s = s''$)) + ($\int^+ u$. *indicator* {0<..t} u * $\int^+ s'$. p s' s'' (t − u) ∂J s ∂*exponential* (*escape-rate s*))
using ‹$0 \le t$› **by** (*simp add*: *nn-integral-add nn-integral-cmult ennreal-indicator ennreal-mult emeasure-exponential-Ioi escape-rate-pos*)
also have ($\int^+ u$. *indicator* {0<..t} u * $\int^+ s'$. p s' s'' (t − u) ∂J s ∂*exponential* (*escape-rate s*)) =
($\int^+ u$. *indicator* {0<..t} u *$_R$ (*LINT* s'|J s. p s' s'' (t − u)) ∂*exponential* (*escape-rate s*))
by (*simp add*: *measure-pmf.integrable-const-bound*[*of - 1*] *nn-integral-eq-integral ennreal-mult ennreal-indicator*)
also have ... = (*LINT* u:{0<..t}|*exponential* (*escape-rate s*). (*LINT* s'|J s. p s' s'' (t − u)))
unfolding *set-lebesgue-integral-def*
by (*intro nn-integral-eq-integral E.integrable-const-bound*[*of - 1*] *AE-I2*)
(*auto intro*!: *mult-le-one measure-pmf.integral-le-const measure-pmf.integrable-const-bound*[*of - 1*])
also have ... = (*LINT* u:{0<..t}|*lborel*. *escape-rate s* * *exp* (− *escape-rate s* * u) * (*LINT* s'|J s. p s' s'' (t − u)))
unfolding *exponential-def set-lebesgue-integral-def*
by (*subst integral-density*)
(*auto simp*: *ac-simps exponential-density-def fun-eq-iff split*: *split-indicator simp del*: *integral-mult-right integral-mult-right-zero intro*!: *arg-cong2*[**where** f=*integral*^{L}])

also have ... = (*LINT u:{0..t}|lborel. escape-rate s * exp (− escape-rate s * (t − u)) * (LINT s'|J s. p s' s'' u)*)
using *AE-lborel-singleton[of 0] AE-lborel-singleton[of t]* **unfolding** *set-lebesgue-integral-def*
by (*subst lborel-integral-real-affine*[**where** *t=t* **and** *c=−1*])
(*auto intro*!: *integral-cong-AE split*: *split-indicator*)
also have ... = *exp (− t * escape-rate s) * escape-rate s * (LINT u:{0..t}|lborel. exp (escape-rate s * u) * (LINT s'|J s. p s' s'' u))*
by (*simp add*: *field-simps exp-diff exp-minus*)
finally show *p s s'' t = (of-bool (s = s'') + (LBINT u:{0..t}. escape-rate s * exp (escape-rate s * u) * (LINT s'|J s. p s' s'' u))) / exp (t * escape-rate s)*
unfolding *set-lebesgue-integral-def*
by (*simp del*: *ennreal-plus add*: *ennreal-plus*[*symmetric*] *exp-minus field-simps*)
qed

lemma *continuous-on-p*: *continuous-on A (p s s')*
proof −
interpret *E*: *prob-space exponential (escape-rate s'')* **for** *s''*
by (*intro escape-rate-pos prob-space-exponential*)
have *continuous-on {..0} (p s s')*
by (*simp add*: *p-nonpos continuous-on-const cong*: *continuous-on-cong-simp*)
moreover have *continuous-on {0..} (p s s')*
proof (*subst continuous-on-cong*[*OF refl p-eq*])
let *?I = λt. escape-rate s * exp (escape-rate s * t) * (LINT s''|J s. p s'' s' t)*
show *continuous-on {0..} (λt. (of-bool (s = s') + (LBINT u:{0..t}. ?I u)) / exp (t * escape-rate s))*
proof (*intro continuous-intros continuous-on-LBINT*[*THEN continuous-on-subset*])
fix *t* :: *real* **assume** *t*: *0 ≤ t*
then have *0 ≤ x ⟹ x ≤ t ⟹ exp (x * escape-rate s) * (LINT s''|J s. p s'' s' x) ≤ exp (t * escape-rate s) * 1* **for** *x*
by (*intro mult-mono*) (*auto intro*!: *mult-mono measure-pmf.integral-le-const measure-pmf.integrable-const-bound*[*of - 1*])
with *t* **show** *set-integrable lborel {0..t} ?I*
using *escape-rate-pos*[*of s*] **unfolding** *set-integrable-def*
by (*intro integrableI-bounded-set-indicator*[**where** *B=escape-rate s * exp (escape-rate s * t)*])
(*auto simp*: *field-simps*)
qed *auto*
qed *simp*
ultimately have *continuous-on ({0..} ∪ {..0}) (p s s')*
by (*intro continuous-on-closed-Un*) *auto*
also have *{0..} ∪ {..0::real} = UNIV* **by** *auto*
finally show *?thesis*
by (*rule continuous-on-subset*) *simp*
qed

lemma *p-vector-derivative*: — Backward equation
assumes *0 ≤ 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 −
  let ?I = λt. escape-rate s * exp (escape-rate s * t) * (LINT s''|J s. p s'' s' t)
  let ?p = λt. (of-bool (s = s') + integral {0..t} ?I) * exp (t *R − escape-rate s)

  { fix t :: real assume 0 ≤ t
    have p s s' t = (of-bool (s = s') + (LBINT u:{0..t}. ?I u)) * exp (− t *
escape-rate s)
      using p-eq[OF ‹0 ≤ t›, of s s'] by (simp add: exp-minus field-simps)
    also have (LBINT u:{0..t}. ?I u) = integral {0..t} ?I
    proof (intro set-borel-integral-eq-integral)
      have 0 ≤ x ⟹ x ≤ t ⟹ exp (x * escape-rate s) * (LINT s''|J s. p s'' s' x)
≤ exp (t * escape-rate s) * 1 for x
        by (intro mult-mono) (auto intro!: mult-mono measure-pmf.integral-le-const
measure-pmf.integrable-const-bound[of - 1])
      with ‹0≤t› show set-integrable lborel {0..t} ?I
        using escape-rate-pos[of s] unfolding set-integrable-def
          by (intro integrableI-bounded-set-indicator[where B=escape-rate s * exp
(escape-rate s * t)])
             (auto simp: field-simps)
    qed
    finally have p s s' t = ?p t
      by simp }
  note p-eq = this

  have at-eq: at t within {0..} = at t within {0 .. t + 1}
    by (intro at-within-nhd[where S={..< t+1}]) auto

  have c-I: continuous-on {0..t + 1} ?I
    by (intro continuous-intros continuous-on-LINT-pmf[where B=1] continu-
ous-on-p) simp

  show ?thesis
  proof (subst has-vector-derivative-cong-ev)
    show ∀F u in nhds t. u ∈ {0..} ⟶ p s s' u = ?p u p s s' t = ?p t
      using ‹0≤t› by (simp-all add: p-eq)
    have (?p has-vector-derivative escape-rate s * ((LINT s''|J s. p s'' s' t) − p s
s' t)) (at t within {0..})
      unfolding at-eq
      apply (intro refl derivative-eq-intros)
      apply rule
      apply (rule integral-has-vector-derivative[OF c-I])
      apply (simp add: ‹0 ≤ t›)
      apply rule
      apply (rule exp-scaleR-has-vector-derivative-right)
      apply (simp add: field-simps exp-minus p-eq ‹0≤t› split del: split-of-bool)
      done
    also have escape-rate s * ((LINT s''|J s. p s'' s' t) − p s s' t) =
```

```
      (LINT s''|count-space UNIV. R s s'' * p s'' s' t) - escape-rate s * p s s' t
    using escape-rate-pos[of s]
    by (simp add: measure-pmf-eq-density integral-density J.rep-eq field-simps)
  finally show (?p has-vector-derivative ?A) (at t within {0..}) .
 qed
qed

coinductive wf-times :: real ⇒ (real × 'a) stream ⇒ bool
where
 t < t' ⟹ 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
 by (cases x) (subst wf-times.simps; simp)

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 safe
 assume ?merge from this ω' show ?cases
 proof (induction ω≡merge-at ω t' ω' arbitrary: ω ω')
  case (1 j s' y ω'') then show ?case
   by (cases ω) (auto split: if-splits)
 next
  case (2 j x ω' s' ω ω'') then show ?case
   by (cases ω) (auto split: if-splits)
 qed
next
 assume ?cases then show ?merge
 proof (split if-split-asm)
  assume trace-in ss t x ω t < t' from this ω' show ?thesis
  proof induction
   case 1 then show ?case
    by (cases ω') auto
  qed auto
 next
  assume ∃ y. trace-in {y} t' x ω ∧ trace-in ss t y ω' ¬ t < t'
  then obtain y where trace-in {y} t' x ω trace-in ss t y ω' t' ≤ t
   by auto
  from this ω' show ?thesis
   by induction auto
 qed
qed

lemma AE-lim-wf-times: AE ω in K.lim-stream (t, s). wf-times t ω
 using AE-lim-stream
proof eventually-elim
```

```
  fix ω assume ∗: ∀ i. snd (((t, s) ## ω) !! i) ∈ DTMC.acc `` {snd (t, s)} ∧
            snd (ω !! i) ∈ J (snd (((t, s) ## ω) !! i)) ∧
            fst (((t, s) ## ω) !! i) < fst (ω !! i)
  have (t ## smap fst ω) !! i < fst (ω !! i) for i
    using ∗[THEN spec, of i] by (cases i) auto
  then show wf-times t ω
  proof (coinduction arbitrary: t ω)
     case wf-times from this[THEN spec, of 0] this[THEN spec, of Suc i for i]
show ?case
      by (cases ω) auto
  qed
qed

lemma wf-times-shiftD: wf-times t' (smap (λ(t', y). (t' + t, y)) ω) ⟹ wf-times
(t' − t) ω
  apply (coinduction arbitrary: t' t ω)
  subgoal for t' t ω
    apply (cases ω; cases shd ω)
    apply auto
    subgoal for ω' j x
      by (rule exI[of - j + t]) auto
    done
  done

lemma wf-times-shift[simp]: wf-times t' (smap (λ(t', y). (t' + t, y)) ω) = wf-times
(t' − t) ω
  using wf-times-shiftD[of t' − t −t smap (λ(t', y). (t' + t, y)) ω]
 by (auto simp: stream.map-comp stream.case-eq-if prod.case-eq-if wf-times-shiftD)

lemma trace-in-unique: trace-in {y1} t x ω ⟹ trace-in {y2} t x ω ⟹ y1 = y2
  by (induction rule: trace-in.induct) auto

lemma trace-at-eq: trace-in {z} t x ω ⟹ trace-at x ω t = z
  by (induction rule: trace-in.induct) auto

lemma AE-lim-acc: AE ω in K.lim-stream (t, x). ∀ t z. trace-in {z} t x ω ⟶ (x,
z) ∈ DTMC.acc
  using AE-lim-stream
proof (eventually-elim, safe)
  fix t' z ω assume ∗: ∀ i. snd (((t, x) ## ω) !! i) ∈ DTMC.acc `` {snd (t, x)} ∧
    snd (ω !! i) ∈ J (snd (((t, x) ## ω) !! i)) ∧ fst (((t, x) ## ω) !! i) < fst (ω
!! i)
    and t: trace-in {z} t' x ω
  define X where X = DTMC.acc `` {x}
  have (x ## smap snd ω) !! i ∈ X for i
    using ∗[THEN spec, of i] by (cases i) (auto simp: X-def)
  from t this have z ∈ X
  proof induction
    case (1 j y x ω) with 1.prems[of 0] show ?case
```

```
    by simp
  next
    case (2 j y ω x) with 2.prems[of Suc i for i] show ?case
      by simp
  qed
  then show (x, z) ∈ DTMC.acc
    by (simp add: X-def)
qed

lemma p-add:
  assumes 0 ≤ t 0 ≤ t'
  shows p x y (t + t') = (LINT z|count-space (DTMC.acc‘‘{x}). p x z t * p z y
t')
proof −
  interpret L: prob-space K.lim-stream xy for xy
    by (rule K.prob-space-lim-stream) simp
  interpret A: sigma-finite-measure count-space (DTMC.acc‘‘{x})
   by (intro sigma-finite-measure-count-space-countable DTMC.countable-acc) simp
  interpret LA: pair-sigma-finite count-space (DTMC.acc‘‘{x}) K.lim-stream xy
for xy
    by unfold-locales

  have p x y (t + t') = (∫⁺ ω. ∫⁺ω'. indicator {ω∈space T. trace-in {y} (t + t')
x ω} (merge-at ω t ω')
    ∂K.lim-stream (t, trace-at x ω t) ∂K.lim-stream (0, x))
    unfolding p-def L.emeasure-eq-measure[symmetric]
    apply (subst lim-time-split[OF ‹0 ≤ t›])
   apply (subst emeasure-bind[OF lim-stream-not-empty measurable-prob-algebraD])
    apply (measurable; fail)
    apply (measurable; fail)
    apply (intro nn-integral-cong)
   apply (subst emeasure-bind[OF lim-stream-not-empty measurable-prob-algebraD])
    apply (measurable; fail)
    apply (measurable; fail)
    apply (simp add: in-space-lim-stream)
    done
  also have ... = (∫⁺ ω. ∫⁺ω'. indicator {ω∈space T. trace-in {y} (t + t') x ω}
(merge-at ω t (smap (λ(t'', s). (t'' + t, s)) ω'))
    ∂K.lim-stream (0, trace-at x ω t) ∂K.lim-stream (0, x))
    unfolding lim-0[of t] by (subst nn-integral-distr) (measurable; fail)+
  also have ... = (∫⁺ ω. ∫⁺ω'. of-bool (∃ z∈DTMC.acc‘‘{x}. trace-in {z} t x ω
∧ trace-in {y} t' z ω')
    ∂K.lim-stream (0, trace-at x ω t) ∂K.lim-stream (0, x))
    apply (rule nn-integral-cong-AE)
    using AE-lim-wf-times AE-lim-acc
    apply eventually-elim
    subgoal premises ω for ω
      apply (rule nn-integral-cong-AE)
      using AE-lim-wf-times AE-lim-acc
```

apply *eventually-elim*
using ω *assms*
apply (*auto simp add*: *trace-in-merge-at indicator-eq-1-iff*)
done
done
also have ... = ($\int^+ \omega.\ \int^+ \omega'.\ \int^+ z.$ *of-bool* (*trace-in* {z} t x ω $\wedge$ *trace-in* {y} t' z ω')
∂*count-space* (*DTMC.acc*``{x}) ∂*K.lim-stream* (0, *trace-at* x ω t) ∂*K.lim-stream* (0, x))
by (*intro nn-integral-cong of-bool-Bex-eq-nn-integral*) (*auto dest*: *trace-in-unique*)
also have ... = ($\int^+ \omega.\ \int^+ z.\ \int^+ \omega'.$ *of-bool* (*trace-in* {z} t x ω $\wedge$ *trace-in* {y} t' z ω')
∂*K.lim-stream* (0, *trace-at* x ω t) ∂*count-space* (*DTMC.acc*``{x}) ∂*K.lim-stream* (0, x))
apply (*subst LA.Fubini'*)
apply (*subst measurable-split-conv*)
apply (*rule measurable-compose-countable'*[*OF* - *measurable-fst*])
apply (*auto simp*: *DTMC.countable-acc*)
done
also have ... = ($\int^+ z.\ \int^+ \omega.$ *of-bool* (*trace-in* {z} t x ω) $*$ $\int^+ \omega'.$ *of-bool* (*trace-in* {y} t' z ω')
∂*K.lim-stream* (0, z) ∂*K.lim-stream* (0, x) ∂*count-space* (*DTMC.acc*``{x}))
apply (*subst LA.Fubini'*)
apply (*subst measurable-split-conv*)
apply (*rule measurable-compose-countable'*[*OF* - *measurable-fst*])
apply (*rule nn-integral-measurable-subprob-algebra2*)
apply (*measurable*; *fail*)
apply (*rule measurable-prob-algebraD*)
apply (*auto simp*: *DTMC.countable-acc trace-at-eq intro*!: *nn-integral-cong*)
done
also have ... = ($\int^+ z.$ ($\int^+ \omega.$ *of-bool* (*trace-in* {z} t x ω)∂*K.lim-stream* (0, x)) $*$
($\int^+ \omega'.$ *of-bool* (*trace-in* {y} t' z ω') ∂*K.lim-stream* (0, z)) ∂*count-space* (*DTMC.acc*``{x}))
by (*auto intro*!: *nn-integral-cong simp*: *nn-integral-multc*)
also have ... = ($\int^+ z.$ *ennreal* (p x z t) $*$ *ennreal* (p z y t') ∂*count-space* (*DTMC.acc*``{x}))
unfolding *p-def L.emeasure-eq-measure*[*symmetric*]
by (*auto intro*!: *nn-integral-cong arg-cong2*[**where** f=($*$)]
simp: *nn-integral-indicator*[*symmetric*] *simp del*: *nn-integral-indicator*)
finally have ($\int^+ z.$ p x z t $*$ p z y t' ∂*count-space* (*DTMC.acc*``{x})) = p x y (t + t')
by (*simp add*: *ennreal-mult*)
then show *?thesis*
by (*subst* (*asm*) *nn-integral-eq-integrable*) *auto*
qed

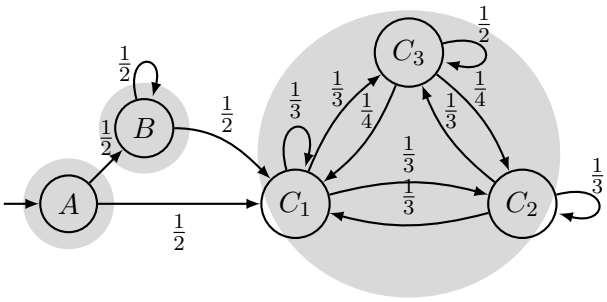
end

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*}
using *state.nchotomy* **by** *auto*

instance *state* :: *finite*
by *standard* (*simp add*: *UNIV-state*)

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*
by (*auto simp*: *tau-def nn-integral-count-space-finite UNIV-state split*: *state.split simp del*: *ennreal-plus*)

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* **.**

8.1 The essential classs $\{C1, C2, C3\}$

context
begin

interpretation *pmf-as-function* **.**

lemma *A-E-eq*:
set-pmf (*K x*) = (*case x of A* ⇒ {*B*, *C1*} | *B* ⇒ {*B*, *C1*} | - ⇒ {*C1*, *C2*, *C3*})
using *state.nchotomy* **by** *transfer* (*auto simp*: *tau-def split*: *prod.split state.split*)

lemma *A-essential*: *A.essential-class* {*C1*, *C2*, *C3*}
by (*rule A.essential-classI2*) (*auto simp*: *A-E-eq*)

lemma *A-aperiodic*: *A.aperiodic* {*C1*, *C2*, *C3*}
unfolding *A.aperiodic-def*
proof *safe*
have *eq*: ⋀*x'*. (*if x' = C1 then 1 else 0*) = *indicator* {*C1*} *x'* **by** *auto*

show {*C1*, *C2*, *C3*} ∈ *UNIV* // *A.communicating*
using *A-essential* **by** (*simp add*: *A.essential-class-def*)
then have *A.period* {*C1*, *C2*, *C3*} = *Gcd* (*A.period-set C1*)
by (*rule A.period-eq*) *simp*
also have ... = *1*
by (*rule Gcd-nat-eq-one*) (*simp add*: *A-E-eq A.period-set-def A.p-Suc' A.p-0 eq measure-pmf-single pmf-positive*)
finally show *A.period* {*C1*, *C2*, *C3*} = *1* **.**
qed

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*
by (*auto simp*: *UNIV-state nn-integral-count-space-finite split*: *state.split*)

lemma *stationary-distribution-N*: *A.stationary-distribution n*
unfolding *A.stationary-distribution-def*
apply (*auto intro*!: *pmf-eqI simp*: *pmf-bind integral-measure-pmf*[*of UNIV*])
apply *transfer*
apply (*auto simp*: *UNIV-state tau-def split*: *state.split*)
done

lemma *exclusive-N*[*simp*]: *set-pmf n* = {*C1*, *C2*, *C3*}
using *state.nchotomy* **by** *transfer* (*auto split*: *state.splits*)

end

lemma *n-is-limit*:
assumes *x*: *x* ∈ {*C1*, *C2*, *C3*} **and** *y*: *y* ∈ {*C1*, *C2*, *C3*}
shows (*A.p x y*) ⟶ *pmf n y*
using *A.stationary-distribution-imp-p-limit*[*OF A-aperiodic A-essential - stationary-distribution-N - x y*]
by *simp*

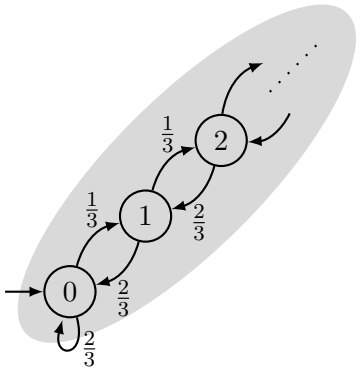
lemma *C-is-pos-recurrent*: *x* ∈ {*C1*, *C2*, *C3*} ⟹ *A.pos-recurrent x*
using *A.stationary-distributionD*(*1*)[*OF A-essential - stationary-distribution-N*]
by *auto*

lemma *C-recurrence-time*:
assumes *x*: *x* ∈ {*C1*, *C2*, *C3*}
shows *A.U' x x* = *1* / *pmf n x*
proof −
from *A.stationary-distributionD*(*2*)[*OF A-essential - stationary-distribution-N -*]
have *A.stat* {*C1*, *C2*, *C3*} = *n* **by** *simp*
with *x* **have** *1* / *pmf n x* = *1* / *emeasure* (*A.stat* {*C1*, *C2*, *C3*}) {*x*}
by (*simp add*: *emeasure-pmf-single pmf-positive divide-ennreal ennreal-1*[*symmetric*] *del*: *ennreal-1*)
also have ... = *A.U' x x*
unfolding *A.stat-def* **using** *x*
by (*subst emeasure-point-measure-finite*) (*simp-all add*: *A.U'-def*)
finally show *?thesis* **..**
qed

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* ⇒ *nat pmf* **where**
K x = *map-pmf* (λ*True* ⇒ *x* + *1* | *False* ⇒ *x* − *1*) (*bernoulli-pmf* (*1*/*3*))

For the special case when *x* = *0* we have *x* − *1* = *0* and hence *tau 0 0* = (*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* **.**

9.1 Enabled, accessible and communicating states

For each step the predecessor and the successor are enabled (in the *0* case, the predecessor is again *0*. 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*}
by (*auto simp*: *set-pmf-bernoulli K-def split*: *bool.split*)

lemma *B-E-Suc*: *Suc x* ∈ *set-pmf* (*K x*) *x* ∈ *set-pmf* (*K* (*Suc x*))
unfolding *B-E-eq* **by** *auto*

lemma *B-accessible*[*intro*]: (*i*, *j*) ∈ *B.acc*
proof (*cases i j rule*: *linorder-le-cases*)
assume *i* ≤ *j* **then show** *?thesis*
by (*induct rule*: *inc-induct*) (*auto intro*: *B-E-Suc converse-rtrancl-into-rtrancl*)
next
assume *j* ≤ *i* **then show** *?thesis*
by (*induct rule*: *dec-induct*) (*auto intro*: *B-E-Suc converse-rtrancl-into-rtrancl*)
qed

lemma *B-communicating*[*intro*]: $(i, j) \in B.communicating$
by (*simp add*: *B.communicating-def B-accessible*)

lemma *B-essential*: *B.essential-class UNIV*
by (*rule B.essential-classI2*) *auto*

9.2 B is aperiodic

lemma *B-aperiodic*: *B.aperiodic UNIV*
unfolding *B.aperiodic-def*
proof *safe*
have *eq*: $\bigwedge x'$. (*if* $x' = 0$ *then 1 else 0*) = *indicator* $\{0\}$ x' **by** *auto*

show $UNIV \in UNIV$ // *B.communicating*
using *B-essential* **by** (*simp add*: *B.essential-class-def*)
then have *B.period UNIV* = *Gcd* (*B.period-set 0*)
by (*rule B.period-eq*) *simp*
also have ... = *1*
by (*rule Gcd-nat-eq-one*) (*simp add*: *B.period-set-def B.p-Suc' B.p-0 eq measure-pmf-single pmf-positive-iff K-def set-pmf-bernoulli UNIV-bool*)
finally show *B.period UNIV* = *1* **.**
qed

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*
unfolding *B.stationary-distribution-def*
proof (*rule pmf-eqI*)
fix *a* **show** *pmf N a* = *pmf* (*bind-pmf N K*) *a*
apply (*simp add*: *pmf-bind K-def map-pmf-def*)
apply (*subst integral-measure-pmf*[*of* {*a* − *1*, *a* + *1*}])
apply (*auto split*: *split-indicator-asm nat.splits simp*: *minus-nat.diff-Suc*)
done
qed

9.4 Limit behavior and recurrence times

lemma *limit*: (*B.p i j*) ⟶ (*1/2*)^*Suc j*
proof −
have *B.p i j* ⟶ *pmf N j*
by (*rule B.stationary-distribution-imp-p-limit*[*OF B-aperiodic B-essential - stationary-distribution-N*])
auto
then show *?thesis*
by (*simp add*: *ac-simps*)
qed

lemma *pos-recurrent*: *B.pos-recurrent i*
using *B.stationary-distributionD*(*1*)[*OF B-essential - stationary-distribution-N -*]
by *auto*

lemma *recurrence-time*: *B.U' i i = 2^Suc i*
proof −
have *B.stat UNIV = N*
using *B.stationary-distributionD*(*2*)[*OF B-essential - stationary-distribution-N -*] **by** *simp*
then have *2^Suc i = 1 / emeasure* (*B.stat UNIV*) {*i*}
apply (*simp add*: *field-simps emeasure-pmf-single pmf-positive*)
apply (*subst divide-ennreal*[*symmetric*])
apply (*auto simp*: *ennreal-mult ennreal-power*[*symmetric*])
done
also have *... = B.U' i i*
unfolding *B.stat-def*
by (*subst emeasure-point-measure-finite2*)
(*simp-all add*: *B.U'-def*)
finally show *?thesis*
by *simp*
qed

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* ⇒ *'s pmf* **and** *S* :: *'s set* **and** *ϱ* :: *'s* ⇒ *real* **and** *ι* :: *'s* ⇒ *'s* ⇒ *real*
assumes *ι-nonneg*[*simp*]: ⋀*s t. 0* ≤ *ι s t* **and** *ϱ-nonneg*[*simp*]: ⋀*s. 0* ≤ *ϱ s*
assumes *measurable-ι*: (*λ*(*a*, *b*). *ι a b*) ∈ *borel-measurable* (*count-space UNIV* ⨂*M count-space UNIV*)
assumes *finite-S*[*simp*]: *finite S* **and** *S-not-empty*: *S* ≠ {}
assumes *E-closed*: (⋃*s*∈*S*. *set-pmf* (*K s*)) ⊆ *S*
begin

lemma *measurable-ι'*[*measurable* (*raw*)]:
f ∈ *measurable M* (*count-space UNIV*) ⟹ *g* ∈ *measurable M* (*count-space UNIV*) ⟹
(*λx. ι* (*f x*) (*g x*)) ∈ *borel-measurable M*
using *measurable-compose*[*OF - measurable-ι*, *of λx.* (*f x*, *g x*) *M*] **by** *simp*

lemma *measurable-ϱ*[*measurable*]: $\varrho \in$ *borel-measurable* (*count-space UNIV*)
by *simp*

sublocale *R?*: *MC-with-rewards K ι ϱ*
by *standard* (*auto intro*: *ι-nonneg ϱ-nonneg*)

lemma *single-l*:
fixes *s* **and** *x* :: *real* **assumes** $s \in S$
shows $(\sum s' \in S.\ (\textit{if } s' = s \textit{ then 1 else 0}) * l\ s') = x \longleftrightarrow l\ s = x$
by (*simp add*: *assms if-distrib* [*of* $\lambda x.\ x * a$ **for** *a*] *cong*: *if-cong*)

definition *order* :: *nat* ⇒ *'s*
where *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*]: $\bigwedge i.\ i < \textit{card } S \Longrightarrow \textit{order } i \in S$
proof −
from *finite-same-card-bij*[*OF - finite-S*] **show** *bij-betw order* {..< *card S*} *S*
unfolding *order-def* **by** (*rule someI-ex*) *auto*
then show *inj-on order* {..<*card S*} *order* ‘ {..<*card S*} = *S*
unfolding *bij-betw-def* **by** *auto*
then show $\bigwedge i.\ i < \textit{card } S \Longrightarrow \textit{order } i \in S$
by *auto*
qed

lemma *order-Ex*:
assumes $s \in S$ **obtains** *i* **where** $i < \textit{card } S\ s = \textit{order } i$
proof −
from ‹$s \in S$› **have** $s \in$ *order* ‘ {..<*card S*}
by *simp*
with *that* **show** *thesis*
by (*auto simp del*: *image-order*)
qed

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

lemma *bij-iorder*: *bij-betw iorder S* {..<*card S*}
unfolding *iorder-def* **by** (*rule bij-betw-the-inv-into bij-order*)+

lemma *iorder-image-eq*: *iorder* ‘ *S* = {..<*card S*}
and *inj-iorder*: *inj-on iorder S*
using *bij-iorder* **unfolding** *bij-betw-def* **by** *auto*

lemma *order-iorder*: $\bigwedge s.\ s \in S \Longrightarrow \textit{order } (\textit{iorder } s) = s$
unfolding *iorder-def* **using** *bij-order*
by (*intro f-the-inv-into-f*) (*auto simp*: *bij-betw-def*)

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 −
note ‹*gauss-jordan' M a* = *Some f*›
moreover define *M'* **where** *M'* = (λ*i j*. *if j* = *card S then*
a (*order i*) *else M* (*order i*) (*order j*))
ultimately obtain *sol* **where** *sol*: *gauss-jordan M'* (*card S*) = *Some sol*
and *f*: *f* = (λ*i*. *sol* (*iorder i*) (*card S*))
by (*auto simp*: *gauss-jordan'-def Let-def split*: *bind-split-asm*)

from *gauss-jordan-correct*[*OF sol*]
have ∀ *i*∈{..<*card S*}. (∑ *j*<*card S*. *M* (*order i*) (*order j*) ∗ *sol j* (*card S*)) = *a* (*order i*)
unfolding *solution-def M'-def* **by** (*simp add*: *atLeast0LessThan*)
then show *?thesis*
unfolding *iorder-image-eq*[*symmetric*] *f* **using** *inj-iorder*
by (*subst* (*asm*) *sum.reindex*) (*auto simp*: *order-iorder*)
qed

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 −
define *M'* **where** *M'* = (λ*i j*. *if j* = *card S then*
a (*order i*) *else M* (*order i*) (*order j*))

{ fix *x*
have *iorder-neq-card-S*: ⋀*s*. *s* ∈ *S* ⟹ *iorder s* ≠ *card S*
using *iorder-image-eq* **by** (*auto simp*: *set-eq-iff less-le*)
have *solution2 M'* (*card S*) (*card S*) *x* ⟷
(∀ *s*∈{..<*card S*}. (∑ *s'*∈{..<*card S*}. *M' s s'* ∗ *x s'*) = *M' s* (*card S*))
unfolding *solution2-def* **by** (*auto simp*: *atLeast0LessThan*)
also have ... ⟷ (∀ *s*∈*S*. (∑ *s'*∈*S*. *M s s'* ∗ *x* (*iorder s'*)) = *a s*)
unfolding *iorder-image-eq*[*symmetric*] *M'-def*
using *inj-iorder iorder-neq-card-S*
by (*simp add*: *sum.reindex order-iorder*)
finally have *solution2 M'* (*card S*) (*card S*) *x* ⟷
(∀ *s*∈*S*. (∑ *s'*∈*S*. *M s s'* ∗ *x* (*iorder s'*)) = *a s*) **. }**

note *sol2-eq* = *this*
have *usolution M′* (*card S*) (*card S*) (λ*i*. *x* (*order i*))
unfolding *usolution-def*
proof *safe*
from *exists* **show** *solution2 M′* (*card S*) (*card S*) (λ*i*. *x* (*order i*))
by (*simp add*: *sol2-eq order-iorder*)
next
fix *y j* **assume** *y*: *solution2 M′* (*card S*) (*card S*) *y* **and** *j* < *card S*
then have ∀ *s*∈*S*. (∑ *s′*∈*S*. *M s s′* ∗ *y* (*iorder s′*)) = *a s*
by (*simp add*: *sol2-eq*)
from *unique*[*OF this*]
have ∀ *i*∈{..<*card S*}. *y i* = *x* (*order i*)
unfolding *iorder-image-eq*[*symmetric*]
by (*simp add*: *order-iorder*)
with ‹*j* < *card S*› **show** *y j* = *x* (*order j*) **by** *simp*
qed
from *gauss-jordan-complete*[*OF - this*]
show *?thesis*
by (*auto simp*: *gauss-jordan′-def simp*: *M′-def*)
qed

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)
by (*induct n*) *simp-all*

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* ∈ *S*. *s* ∈ *L*} |
svalid (*Neg F*) = *S* − *svalid F* |
svalid (*And F1 F2*) = *svalid F1* ∩ *svalid F2* |
svalid (*Prob rel r F*) = {*s* ∈ *S*. *inrealrel rel r* $\mathcal{P}(\omega$ *in T s*. *pvalid F* (*s* ## $\omega))$ } |
svalid (*Exp rel r F*) = {*s* ∈ *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*
by (*induct F*) *auto*

lemma *finite-svalid*[*simp*, *intro*]: *finite* (*svalid F*)
using *svalid-subset-S finite-S* **by** (*blast intro*: *finite-subset*)

lemma *svalid-sets*[*measurable*]: *svalid F* ∈ *sets* (*count-space S*)
using *svalid-subset-S* **by** *auto*

lemma *pvalid-sets*[*measurable*]: *Measurable.pred R.S* (*pvalid F*)
by (*cases F*) (*auto intro*!: *svalid-sets*)

lemma *reward-measurable*[*measurable*]: *reward F* ∈ *borel-measurable R.S*
by (*cases F*) *auto*

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*
unfolding *Prob0-def* **by** *auto*

lemma *Prob0-iff-reachable*:
assumes Φ ⊆ *S* Ψ ⊆ *S*
shows *Prob0* Φ Ψ = {*s* ∈ *S*. ((*SIGMA x*:Φ. *E x*)* `` {*s*}) ∩ Ψ = {}} (**is** - = *?U*)
unfolding *Prob0-def*
proof (*intro while-rule*[**where** *Q*=λ*R*. *S* − *R* = *?U* **and** *P*=λ*R*. Ψ ⊆ *R* ∧ *R* ⊆ *S* − *?U*] *conjI*)
show *wf* {(*B*, *A*). *A* ⊂ *B* ∧ *B* ⊆ *S*}
by (*rule wf-bounded-set*[**where** *ub*=λ-. *S* **and** *f*=λ*x*. *x*]) *auto*
show Ψ ⊆ *S* − *?U*
using *assms* **by** *auto*

let *?*Δ = λ*R*. {*s*∈Φ. *R* ∩ *E s* ≠ {}}
{ fix *R* **assume** *R*: Ψ ⊆ *R* ∧ *R* ⊆ *S* − *?U* **and** ∃ *s*∈Φ. *R* ∩ *E s* ≠ {} ∧ *s* ∉ *R*
with *assms* **show** (*R* ∪ *?*Δ *R*, *R*) ∈ {(*B*, *A*). *A* ⊂ *B* ∧ *B* ⊆ *S*} Ψ ⊆ *R* ∪ *?*Δ *R*
by *auto*

{ fix *s s*′ **assume** *s*: *s* ∈ Φ *s*′ ∈ *R s*′ ∈ *E s* **and** *r*: (*Sigma* Φ *E*)* `` {*s*} ∩ Ψ = {}
with *R* **have** (*s*, *s*′) ∈ (*Sigma* Φ *E*)* *s*′ ∈ Φ − Ψ
by (*auto elim*: *converse-rtranclE*)
moreover with ‹*s*′ ∈ *R*› *R* **obtain** *s*′′ **where** (*s*′, *s*′′) ∈ (*Sigma* Φ *E*)* *s*′′ ∈ Ψ
by *auto*
ultimately have (*s*, *s*′′) ∈ (*Sigma* Φ *E*)* *s*′′ ∈ Ψ
by *auto*
with *r* **have** *False*
by *auto* **}**
with ‹Φ ⊆ *S*› *R* **show** *R* ∪ *?*Δ *R* ⊆ *S* − *?U* **by** *auto* **}**

{ fix *R* **assume** *R*: Ψ ⊆ *R* ∧ *R* ⊆ *S* − *?U* **and** *dR*: ¬ (∃ *s*∈Φ. *R* ∩ *E s* ≠ {} ∧ *s* ∉ *R*)
{ fix *s t* **assume** *s*: *s* ∈ *S* − *R*
assume *s-t*: (*s*, *t*) ∈ (*Sigma* Φ *E*)* **then have** *t* ∈ *S* − *R*
proof *induct*
case (*step t u*) **with** *R dR E-closed* **show** *?case*

```
      by auto
    qed fact
    then have t ∉ Ψ
      using R by auto }
  with R show S − R = ?U
    by auto }
qed rule

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)
  unfolding Prob0-iff-reachable[OF assms]
proof (intro Collect-cong conj-cong refl iffI)
  fix s assume s: s ∈ S (Sigma Φ E)* `` {s} ∩ Ψ = {}
  { fix ω assume (HLD Φ suntil HLD Ψ) ω enabled (shd ω) (stl ω) (Sigma Φ E)*
`` {shd ω} ∩ Ψ = {}
    from this have False
    proof induction
      case (step ω)
      moreover
      then have (shd ω, shd (stl ω)) ∈ (Sigma Φ E)*
        by (auto simp: enabled.simps[of - stl ω] HLD-iff)
      then have (Sigma Φ E)* `` {shd (stl ω)} ⊆ (Sigma Φ E)* `` {shd ω}
        by auto
      ultimately show ?case
        by (auto simp add: enabled.simps[of - stl ω])
    qed (auto simp: HLD-iff) }
  from s this[of s ## ω for ω] show AE ω in T s. ¬ (HLD Φ suntil HLD Ψ) (s
## ω)
    using AE-T-enabled[of s] by auto
next
  fix s assume s: AE ω in T s. ¬ (HLD Φ suntil HLD Ψ) (s ## ω)
  { fix t assume (s, t) ∈ (Sigma Φ E)* from this s have t ∉ Ψ
    proof (induction rule: converse-rtrancl-induct)
      case (step s u) then show ?case
        by (simp add: AE-T-iff[where x=s] suntil-Stream[of - - s])
    qed (simp add: suntil-Stream) }
  then show (Sigma Φ E)* `` {s} ∩ Ψ = {}
    by auto
qed

lemma E-rtrancl-closed:
  assumes s ∈ S (s, t) ∈ (SIGMA x:A. B x)* ⋀x. x ∈ A ⟹ B x ⊆ E x shows t
∈ S
  using assms(2,3,1) E-closed by induction force+
```

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 −
note *P0* = *Prob0-iff-reachable*[*OF assms*]
have ∗: Φ − Ψ ⊆ *S ?P0* ⊆ *S*
using *P0 assms* **by** *auto*

have *P0-subset*: *S* − Φ − Ψ ⊆ *?P0*
unfolding *P0* **by** (*auto elim*: *converse-rtranclE*)

have *Prob1 ?P0* Φ Ψ = {*s* ∈ *S*. (*Sigma* (Φ − Ψ) *E*)* `` {*s*} ∩ *?P0* = {}}
unfolding *Prob0-iff-reachable*[*OF* ∗] *Prob1-def* **..**
also have ... = {*s*∈*S*. *AE* ω *in T s*. (*HLD* Φ *suntil HLD* Ψ) (*s* ## ω)}
proof (*intro Collect-cong conj-cong refl iffI*)
fix *s* **assume** *s*: *s* ∈ *S* (*Sigma* (Φ − Ψ) *E*)* `` {*s*} ∩ *?P0* = {}
then have *s* ∉ *?P0*
by *auto*
then have *s* ∈ Φ − Ψ ∨ *s* ∈ Ψ
using *P0-subset* ‹*s* ∈ *S*› **by** *auto*
moreover
{ assume *s* ∈ Φ − Ψ
have *AE* ω *in T s*. *ev* (*HLD* (Ψ ∪ *?P0*)) ω
proof (*rule AE-T-ev-HLD*)
fix *t* **assume** *s-t*: (*s*, *t*) ∈ *acc-on* (− (Ψ ∪ *?P0*))
from ‹*s* ∈ *S*› *s-t* **have** *t* ∈ *S*
by (*rule E-rtrancl-closed*) *auto*

show ∃ *t′*∈Ψ ∪ *?P0*. (*t*, *t′*) ∈ *acc*
proof *cases*
assume *t* ∈ *?P0* **then show** *?thesis* **by** *auto*
next
assume *t* ∉ *?P0*
with ‹*t*∈*S*› **obtain** *s* **where** *t-s*: (*t*, *s*) ∈ (*SIGMA x*:Φ. *E x*)* **and** *s* ∈ Ψ
unfolding *P0* **by** *auto*
from *t-s* **have** (*t*, *s*) ∈ *acc*
by (*rule rev-subsetD*) (*intro rtrancl-mono Sigma-mono*, *auto*)
with ‹*s* ∈ Ψ› **show** *?thesis* **by** *auto*
qed
next
have *acc-on* (− (Ψ ∪ *?P0*)) `` {*s*} ⊆ *S*
using ‹*s* ∈ *S*› **by** (*auto intro*: *E-rtrancl-closed*)
then show *finite* (*acc-on* (− (Ψ ∪ *?P0*)) `` {*s*})

```
      using finite-S by (auto dest: finite-subset)
    qed
    then have AE ω in T s. (HLD Φ suntil HLD Ψ) ω
      using AE-T-enabled
    proof eventually-elim
      fix ω assume ev (HLD (Ψ ∪ ?P0)) ω enabled s ω
      from this s ‹s ∈ Φ − Ψ› show (HLD Φ suntil HLD Ψ) ω
      proof (induction arbitrary: s)
        case (base ω) then show ?case
          by (auto simp: HLD-iff enabled.simps[of s] intro: suntil.intros)
      next
        case (step ω)
        then have (s, shd ω) ∈ (Sigma (Φ − Ψ) E)
          by (auto simp: enabled.simps[of s])
        then have ∗: (Sigma (Φ − Ψ) E)* `` {shd ω} ∩ ?P0 = {}
          using step.prems by (auto intro: converse-rtrancl-into-rtrancl)
        then have shd ω ∈ Φ − Ψ ∨ shd ω ∈ Ψ shd ω ∈ S
            using P0-subset step.prems(1,2) E-closed by (auto simp add: en-
abled.simps[of s])
        then show ?case
          using step.prems(1) step.IH[OF - - ∗] ‹shd ω ∈ S›
        by (auto simp add: suntil.simps[of - - ω] HLD-iff[abs-def] enabled.simps[of
s ω])
      qed
    qed }
  ultimately show AE ω in T s. (HLD Φ suntil HLD Ψ) (s ## ω)
    by (cases s ∈ Φ − Ψ) (auto simp add: suntil-Stream)
 next
  fix s assume s: s ∈ S AE ω in T s. (HLD Φ suntil HLD Ψ) (s ## ω)
  { fix t assume (s, t) ∈ (SIGMA s:Φ−Ψ. E s)*
    from this ‹s ∈ S› have (AE ω in T t. (HLD Φ suntil HLD Ψ) (t ## ω)) ∧
t ∈ S
    proof induction
      case (step t u) with E-closed show ?case
        by (auto simp add: AE-T-iff[of - t] suntil-Stream)
    qed (insert s, auto)
    then have t ∉ ?P0
      unfolding Prob0-iff[OF assms] by (auto dest: T.AE-contr) }
  then show (Sigma (Φ − Ψ) E)* `` {s} ∩ Prob0 Φ Ψ = {}
    by auto
 qed
 finally show ?thesis .
qed
```

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

```
abbreviation τ s t ≡ pmf (K s) t

fun ProbU :: 's ⇒ nat ⇒ 's set ⇒ 's set ⇒ real where
```

$ProbU\ q\ 0\ S1\ S2 \quad = (if\ q \in S2\ then\ 1\ else\ 0)\ |$
$ProbU\ q\ (Suc\ k)\ S1\ S2 =$
$(if\ q \in S1 - S2\ then\ (\sum q' \in S.\ \tau\ q\ q' * ProbU\ q'\ k\ S1\ S2)$
$\quad else\ if\ q \in S2\ then\ 1\ else\ 0)$

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

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

11.3.4 *LES*

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

11.3.5 *ProbUinfty*, compute unbounded until

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

11.3.6 *ExpFuture*, compute unbounded reward

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

11.3.7 *Sat*

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

$Sat\ (Prob\ rel\ r\ (X\ F)) \quad = do\ \{\ F \leftarrow Sat\ F\ ;\ Some\ \{q \in S.\ inrealrel\ rel\ r$
$(\sum q' \in F.\ \tau\ q\ q')\}\ \}\ |$
$Sat\ (Prob\ rel\ r\ (U\ k\ F1\ F2)) \ = do\ \{\ F1 \leftarrow Sat\ F1\ ;\ F2 \leftarrow Sat\ F2\ ;\ Some\ \{q \in$
$S.\ inrealrel\ rel\ r\ (ProbU\ q\ k\ F1\ F2)\ \}\ \}\ |$

Sat (Prob rel r (U^∞ 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 ⟹ 𝒫(ω in T s. P ω) = (∑ t∈S. τ s t * 𝒫(ω in T t. P (t ## ω)))*
unfolding *prob-T* **using** *E-closed* **by** (*subst integral-measure-pmf*[*OF finite-S*]) (*auto simp*: *mult.commute*)

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))*
unfolding *nn-integral-T* **using** *E-closed*
by (*subst nn-integral-measure-pmf-support*[*OF finite-S*])
(*auto simp*: *mult.commute*)

lemma *T-space*[*simp*]: *measure (T s) (space R.S) = 1*
using *T.prob-space* **by** *simp*

lemma *emeasure-T-space*[*simp*]: *emeasure (T s) (space R.S) = 1*
using *T.emeasure-space-1* **by** *simp*

lemma *τ-distr*[*simp*]: *s ∈ S ⟹ (∑ t∈S. τ s t) = 1*
using *prob-sum*[*of s λ-. True*] **by** *simp*

lemma *ProbU*:
q ∈ S ⟹ ProbU q k (svalid F1) (svalid F2) = 𝒫(ω in T q. pvalid (U k F1 F2) (q ## ω))
proof (*induct k arbitrary*: *q*)
case *0* **with** *T.prob-space* **show** *?case* **by** *simp*
next
case (*Suc k*)

have *𝒫(ω in T q. pvalid (U (Suc k) F1 F2) (q ## ω)) =*
(if q ∈ svalid F2 then 1 else if q ∈ svalid F1 then
*∑ t∈S. τ q t * 𝒫(ω in T t. pvalid (U k F1 F2) (t ## ω)) else 0)*
using ‹*q ∈ S*› **by** (*subst prob-sum*) *simp-all*
also have *... = ProbU q (Suc k) (svalid F1) (svalid F2)*
using *Suc* **by** *simp*
finally show *?case* **..**
qed

lemma *Prob0-imp-not-Psi*:

assumes $\Phi \subseteq S$ $\Psi \subseteq S$ $s \in Prob0\ \Phi\ \Psi$ **shows** $s \notin \Psi$
proof −
have $s \in S$ **using** ‹$s \in Prob0\ \Phi\ \Psi$› *Prob0-subset-S* **by** *auto*
with *assms* **show** *?thesis* **by** (*auto simp add*: *Prob0-iff suntil-Stream*)
qed

lemma *Psi-imp-not-Prob0*:
assumes $\Phi \subseteq S$ $\Psi \subseteq S$ **shows** $s \in \Psi \Longrightarrow s \notin Prob0\ \Phi\ \Psi$
using *Prob0-imp-not-Psi*[*OF assms*] **by** *metis*

11.3.8 Finite expected reward

abbreviation $s0 \equiv SOME\ s.\ s \in S$

lemma *s0-in-S*: $s0 \in S$
using *S-not-empty* **by** (*auto intro*!: *someI-ex*[*of* $\lambda x.\ x \in S$])

lemma *nn-integral-reward-finite*:
assumes $s \in S$
assumes *until*: *AE* ω *in T s.* (*HLD S suntil HLD* (*svalid F*)) (s ## ω)
shows $(\int^+ \omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) \neq \infty$
proof −
have $(\int^+ \omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) = (\int^+ \omega.\ reward\text{-}until\ (svalid\ F)\ s\ \omega\ \partial T\ s)$
using *until* **by** (*auto intro*!: *nn-integral-cong-AE ev-suntil*)
also have $\ldots \neq \infty$
proof *cases*
assume $s \notin svalid\ F$
show *?thesis*
proof (*rule nn-integral-reward-until-finite*)
have *acc* ‘‘ $\{s\} \subseteq S$
using *E-rtrancl-closed*[*of s - - E*] ‹$s \in S$› **by** *auto*
then show *finite* (*acc* ‘‘ $\{s\}$)
using *finite-S* **by** (*auto dest*: *finite-subset*)
show *AE* ω *in T s.* (*ev* (*HLD* (*svalid F*))) ω
using *until* **by** (*auto simp add*: *suntil-Stream* ‹$s \notin svalid\ F$› *intro*: *ev-suntil*)
qed *auto*
qed *simp*
finally show *?thesis* **.**
qed

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
fix s **assume** $s \in S$

show *l1 s = l2 s*
proof *cases*
assume *s ∈ N* **then show** *?thesis*
by (*rule eq*)
next
assume *s ∉ N*
show *?thesis*
proof (*rule unique-les*[*of - S − N K N*])
show *finite* ((λ*x*. *l1 x − l2 x*) ‘ (*S − N* ∪ *N*)) (⋃*x*∈*S − N*. *E x*) ⊆ *S − N* ∪ *N*
using *E-closed finite-S* ‹*N* ⊆ *S*› **by** (*auto dest*: *finite-subset*)
show ⋀*s*. *s ∈ N* ⟹ *l1 s = l2 s* **by** *fact*
{ fix *s* **assume** *s ∈ S − N* **with** *E-closed finite-S* **show** *integrable* (*K s*) *l1* *integrable* (*K s*) *l2*
by (*auto intro*!: *integrable-measure-pmf-finite dest*: *finite-subset*)
obtain *t* **where** (*t ∈* Ψ ∧ (*s*, *t*) ∈ (*Sigma* Φ *E*)*) ∨ *s ∈ N*
using ‹*s ∈ S − N*› *in-S*(*4*) **unfolding** *Prob0-iff-reachable*[*OF in-S*(*1*,*2*)] **by** *auto*
moreover have (*Sigma* Φ *E*)* ⊆ *acc*
by (*intro rtrancl-mono Sigma-mono*) *auto*
ultimately show ∃ *t*∈*N*. (*s*, *t*) ∈ *acc*
using ‹Ψ ⊆ *N*› **by** *auto*
show *l1 s = integral*^{L} (*K s*) *l1* + *c s*
using *E-closed l1* ‹*s ∈ S − N*›
by (*subst integral-measure-pmf*[*OF finite-S*]) (*auto simp*: *subset-eq field-simps*)
show *l2 s = integral*^{L} (*K s*) *l2* + *c s*
using *E-closed l2* ‹*s ∈ S − N*›
by (*subst integral-measure-pmf*[*OF finite-S*]) (*auto simp*: *subset-eq field-simps*)
}
qed (*insert* ‹*s ∉ N*› ‹*s ∈ S*›, *auto*)
qed
qed

lemma *uniqueness-of-ProbU*:
assumes *sol*:
∀ *s*∈*S*. (∑ *s*′∈*S*. *LES* (*Prob0* (*svalid F1*) (*svalid F2*) ∪ *svalid F2*) *s s*′ * *l s*′) =
(*if s ∈ svalid F2 then 1 else 0*)
shows ∀ *s*∈*S*. *l s* = *P*(ω *in T s*. *pvalid* (*U*$^\infty$ *F1 F2*) (*s ##* ω))
proof (*rule unique*)
show *svalid F1* ⊆ *S svalid F2* ⊆ *S*
Prob0 (*svalid F1*) (*svalid F2*) ⊆ *Prob0* (*svalid F1*) (*svalid F2*) ∪ *svalid F2*
svalid F2 ⊆ *Prob0* (*svalid F1*) (*svalid F2*) ∪ *svalid F2*
Prob0 (*svalid F1*) (*svalid F2*) ∪ *svalid F2* ⊆ *S*
using *svalid-subset-S* **by** (*auto simp*: *Prob0-def*)
next
fix *s* **assume** *s*: *s ∈ S s ∉ Prob0* (*svalid F1*) (*svalid F2*) ∪ *svalid F2*
have (∑ *s*′∈*S*. (*if s*′ = *s then* τ *s s*′ − *1 else* τ *s s*′) * *l s*′) =
(∑ *s*′∈*S*. τ *s s*′ * *l s*′ − (*if s*′ = *s then 1 else 0*) * *l s*′)
by (*auto intro*!: *sum.cong simp*: *field-simps*)

```
  also have ... = (∑ s'∈S. τ s s' * l s') - l s
    using ‹s ∈ S› by (simp add: sum-subtractf single-l)
  finally show l s - 0 = (∑ s'∈S. τ s s' * l s')
    using sol[THEN bspec, of s] s by (simp add: LES-def)
next
  fix s assume s: s ∈ S s ∉ Prob0 (svalid F1) (svalid F2) ∪ svalid F2
  then show P(ω in T s. pvalid (U^∞ F1 F2) (s ## ω)) - 0 =
    (∑ t∈S. τ s t * P(ω in T t. pvalid (U^∞ F1 F2) (t ## ω)))
    unfolding Prob0-iff[OF svalid-subset-S svalid-subset-S]
    by (subst prob-sum) (auto simp add: suntil-Stream)
next
  fix s assume s ∈ Prob0 (svalid F1) (svalid F2) ∪ svalid F2
  then show l s = P(ω in T s. pvalid (U^∞ F1 F2) (s ## ω))
  proof
    assume P0: s ∈ Prob0 (svalid F1) (svalid F2)
    then have s ∈ S AE ω in T s. ¬ (HLD (svalid F1) suntil HLD (svalid F2))
(s ## ω)
      unfolding Prob0-iff[OF svalid-subset-S svalid-subset-S] by auto
    then have P(ω in T s. pvalid (U^∞ F1 F2) (s ## ω)) = 0
      by (intro T.prob-eq-0-AE) simp
    moreover have l s = 0
      using ‹s ∈ S› P0 sol[THEN bspec, of s] Prob0-subset-S
        Prob0-imp-not-Psi[OF svalid-subset-S svalid-subset-S P0]
      by (auto simp: LES-def single-l split: if-split-asm)
    ultimately show l s = P(ω in T s. pvalid (U^∞ F1 F2) (s ## ω)) by simp
  next
    assume s: s ∈ svalid F2
    moreover with svalid-subset-S have s ∈ S by auto
    moreover note Psi-imp-not-Prob0[OF svalid-subset-S svalid-subset-S s]
    ultimately have l s = 1
      using sol[THEN bspec, of s]
      by (auto simp: LES-def single-l dest: Psi-imp-not-Prob0[OF svalid-subset-S
svalid-subset-S])
    then show l s = P(ω in T s. pvalid (U^∞ F1 F2) (s ## ω))
      using s by (simp add: suntil-Stream)
  qed
qed

lemma infinite-reward:
  fixes s F
  defines N ≡ Prob0 S (svalid F) (is - ≡ Prob0 S ?F)
  defines Y ≡ Prob1 N S (svalid F)
  assumes s: s ∈ S s ∉ Y
  shows (∫⁺ω. reward (Future F) (s ## ω) ∂T s) = ∞
proof -
  { assume (AE ω in T s. ev (HLD ?F) ω)
    with AE-T-enabled have (AE ω in T s. (HLD S suntil HLD ?F) ω)
    proof eventually-elim
      fix ω assume ev (HLD ?F) ω enabled s ω
```

```
      from this ‹s ∈ S› show (HLD S suntil HLD ?F) ω
      proof (induction arbitrary: s)
        case (step ω) show ?case
          using E-closed step.IH[of shd ω] step.prems
          by (auto simp: subset-eq enabled.simps[of s] suntil.simps[of - - ω] HLD-iff)
      qed (auto intro: suntil.intros)
    qed }
  moreover have ¬ (AE ω in T s. (HLD S suntil HLD ?F) (s ## ω))
    using s svalid-subset-S unfolding N-def Y-def by (simp add: Prob1-iff)
  ultimately have *: ¬ (AE ω in T s. ev (HLD ?F) (s ## ω))
    using ‹s ∈ S› by (cases s ∈ ?F) (auto simp add: suntil-Stream ev-Stream)

  show ?thesis
  proof (rule ccontr)
    assume ¬ ?thesis
    from nn-integral-PInf-AE[OF - this] ‹s∈S›
    have AE ω in T s. ev (HLD ?F) (s ## ω)
      by (simp split: if-split-asm)
    with * show False ..
  qed
qed
```

11.3.9 The expected reward implies a unique LES

```
lemma existence-of-ExpFuture:
  fixes s F
  assumes N-def: N ≡ Prob0 S (svalid F) (is - ≡ Prob0 S ?F)
  assumes Y-def: Y ≡ Prob1 N S (svalid F)
  assumes s: s ∈ S s ∉ S − (Y − ?F)
  shows enn2real (∫⁺ω. reward (Future F) (s ## ω) ∂T s) − (ϱ s + (∑ s'∈S. τ
s s' * ι s s')) =
    (∑ s'∈S. τ s s' * enn2real (∫⁺ω. reward (Future F) (s' ## ω) ∂T s'))
proof −
  let ?R = reward (Future F)

  from s have s ∈ Prob1 (Prob0 S ?F) S ?F
    unfolding Y-def N-def by auto
  then have AE-until: AE ω in T s. (HLD S suntil HLD (svalid F)) (s ## ω)
    using Prob1-iff[of S ?F] svalid-subset-S by auto

  from s have s ∉ ?F by auto

  let ?E = λs'. ∫⁺ ω. reward (Future F) (s' ## ω) ∂T s'
  have *: (∑ s'∈S. τ s s' * ?E s') = (∑ s'∈S. ennreal (τ s s' * enn2real (?E s')))
  proof (rule sum.cong)
    fix s' assume s' ∈ S
    show τ s s' * ?E s' = ennreal (τ s s' * enn2real (?E s'))
    proof cases
      assume τ s s' ≠ 0
```

```
      with ‹s ∈ S› ‹s' ∈ S› have s' ∈ E s by (simp add: set-pmf-iff)
      from ‹s ∉ ?F› AE-until have AE ω in T s. (HLD S suntil HLD ?F) (s ##
ω)
        using svalid-subset-S ‹s ∈ S› by simp
      with nn-integral-reward-finite[OF ‹s' ∈ S›, of F] ‹s ∈ S› ‹s' ∈ E s› ‹s ∉ ?F›
      have ?E s' ≠ ∞
        by (simp add: AE-T-iff[of - s] AE-measure-pmf-iff suntil-Stream
              del: reward.simps)
      then show ?thesis by (cases ?E s') (auto simp: ennreal-mult)
    qed simp
  qed simp

  have AE ω in T s. ?R (s ## ω) = ϱ s + ι s (shd ω) + ?R ω
    using ‹s ∉ svalid F› by (auto simp: ev-Stream )
  then have (∫⁺ω. ?R (s ## ω) ∂T s) = (∫⁺ω. (ϱ s + ι s (shd ω)) + ?R ω ∂T
s)
    by (rule nn-integral-cong-AE)
  also have ... = (∫⁺ω. ϱ s + ι s (shd ω) ∂T s) +
    (∫⁺ω. ?R ω ∂T s)
    using ‹s ∈ S›
    by (subst nn-integral-add)
       (auto simp add: space-PiM PiE-iff simp del: reward.simps)
  also have ... = ennreal (ϱ s + (∑ s'∈S. τ s s' * ι s s')) + (∫⁺ω. ?R ω ∂T s)
    using ‹s ∈ S›
    by (subst nn-integral-eq-sum)
     (auto simp: field-simps sum.distrib sum-distrib-left[symmetric] ennreal-mult[symmetric]
sum-nonneg)
  finally show ?thesis
    apply (simp del: reward.simps)
    apply (subst nn-integral-eq-sum[OF ‹s ∈ S› reward-measurable])
     apply (simp del: reward.simps ennreal-plus add: * ennreal-plus[symmetric]
sum-nonneg)
    done
qed

lemma uniqueness-of-ExpFuture:
  fixes F
  assumes N-def: N ≡ Prob0 S (svalid F) (is - ≡ Prob0 S ?F)
  assumes Y-def: Y ≡ Prob1 N S (svalid F)
  assumes const-def: const ≡ λs. if s ∈ Y ∧ s ∉ svalid F then − ϱ s − (∑ s'∈S.
τ s s' * ι s s') else 0
  assumes sol: ⋀s. s∈S ⟹ (∑ s'∈S. LES (S − Y ∪ ?F) s s' * l s') = const s
  shows ∀ s∈S. l s = enn2real (∫⁺ω. reward (Future F) (s ## ω) ∂T s)
    (is ∀ s∈S. l s = enn2real (∫⁺ω. ?R (s ## ω) ∂T s))
proof (rule unique)
  show S ⊆ S ?F ⊆ S using svalid-subset-S by auto
  show S − (Y − ?F) ⊆ S Prob0 S ?F ⊆ S − (Y − ?F) ?F ⊆ S − (Y − ?F)
    using svalid-subset-S
    by (auto simp add: Y-def N-def Prob1-iff)
```

(*auto simp add*: *Prob0-iff dest*!: *T.AE-contr*)
next
fix *s* **assume** $s \in S\ s \notin S - (Y - ?F)$
then show $enn2real\ (\int^+\omega.\ ?R\ (s\ \#\#\ \omega)\ \partial T\ s) - (\varrho\ s + (\sum s'\in S.\ \tau\ s\ s' * \iota\ s\ s')) =$
$(\sum s'\in S.\ \tau\ s\ s' * enn2real\ (\int^+\omega.\ ?R\ (s'\ \#\#\ \omega)\ \partial T\ s'))$
by (*rule existence-of-ExpFuture*[*OF N-def Y-def*])
next
fix *s* **assume** $s \in S\ s \notin S - (Y - ?F)$
then have $s \in Y\ s \notin ?F$ **by** *auto*
have $(\sum s'\in S.\ (if\ s' = s\ then\ \tau\ s\ s' - 1\ else\ \tau\ s\ s') * l\ s') =$
$(\sum s'\in S.\ \tau\ s\ s' * l\ s' - (if\ s' = s\ then\ 1\ else\ 0) * l\ s')$
by (*auto intro*!: *sum.cong simp*: *field-simps*)
also have $\ldots = (\sum s'\in S.\ \tau\ s\ s' * l\ s') - l\ s$
using ‹$s \in S$› **by** (*simp add*: *sum-subtractf single-l*)
finally have $l\ s = (\sum s'\in S.\ \tau\ s\ s' * l\ s') - (\sum s'\in S.\ (if\ s' = s\ then\ \tau\ s\ s' - 1\ else\ \tau\ s\ s') * l\ s')$
by (*simp add*: *field-simps*)
then show $l\ s - (\varrho\ s + (\sum s'\in S.\ \tau\ s\ s' * \iota\ s\ s')) = (\sum s'\in S.\ \tau\ s\ s' * l\ s')$
using *sol*[*OF* ‹$s \in S$›] ‹$s \in Y$› ‹$s \notin ?F$› **by** (*simp add*: *const-def LES-def*)
next
fix *s* **assume** *s*: $s \in S - (Y - ?F)$
with *sol*[*of s*] **have** $l\ s = 0$
by (*cases* $s \in ?F$) (*simp-all add*: *const-def LES-def single-l*)
also have $0 = enn2real\ (\int^+\omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s)$
proof *cases*
assume $s \in ?F$ **then show** *?thesis*
by (*simp add*: *HLD-iff ev-Stream*)
next
assume $s \notin ?F$
with *s* **have** $s \in S - Y$ **by** *auto*
with *infinite-reward*[*of s F*] **show** *?thesis*
by (*simp add*: *Y-def N-def del*: *reward.simps*)
qed
finally show $l\ s = enn2real\ (\int^+\omega.\ ?R\ (s\ \#\#\ \omega)\ \partial T\ s)$ **.**
qed

11.4 Soundness of *Sat*

theorem *Sat-sound*:
$Sat\ F \neq None \Longrightarrow Sat\ F = Some\ (svalid\ F)$
proof (*induct F rule*: *Sat.induct*)
case (*5 rel r F*)
{ fix *q* **assume** $q \in S$
with *svalid-subset-S* **have** $sum\ (\tau\ q)\ (svalid\ F) = \mathcal{P}(\omega\ in\ T\ q.\ HLD\ (svalid\ F)\ \omega)$
by (*subst prob-sum*[*OF* ‹$q \in S$›]) (*auto intro*!: *sum.mono-neutral-cong-left*) **}**
with *5* **show** *?case*
by (*auto split*: *bind-split-asm*)

next
case (*6 rel r k F1 F2*)
then show *?case*
by (*simp add*: *ProbU cong*: *conj-cong split*: *bind-split-asm*)

next
case (*7 rel r F1 F2*)
moreover
define *constants* :: $'s \Rightarrow$ *real* **where** *constants* = (λs. *if* $s \in$ (*svalid F2*) *then 1 else 0*)
moreover define *distr* **where** *distr* = *LES* (*Prob0* (*svalid F1*) (*svalid F2*) $\cup$ *svalid F2*)
ultimately obtain *l* **where** *eq*: *Sat F1* = *Some* (*svalid F1*) *Sat F2* = *Some* (*svalid F2*)
and *l*: *gauss-jordan'* *distr constants* = *Some l*
by *atomize-elim* (*simp add*: *ProbUinfty-def split*: *bind-split-asm*)

from *l* **have** *P*: *ProbUinfty* (*svalid F1*) (*svalid F2*) = *Some l*
unfolding *ProbUinfty-def constants-def distr-def* **by** *simp*

have $\forall s{\in}S.\ l\ s = \mathcal{P}(\omega\ in\ T\ s.\ pvalid\ (U^{\infty}\ F1\ F2)\ (s\ \#\#\ \omega))$
proof (*rule uniqueness-of-ProbU*)
show $\forall s{\in}S.\ (\sum s'{\in}S.\ LES\ (Prob0\ (svalid\ F1)\ (svalid\ F2) \cup svalid\ F2)\ s\ s' * l\ s') =$
(*if s* $\in$ *svalid F2 then 1 else 0*)
using *gauss-jordan'-correct*[*OF l*]
unfolding *distr-def constants-def* **by** *simp*
qed
then show *?case*
by (*auto simp add*: *eq P*)
next
case (*8 rel r k*)
{ fix *s* **assume** $s \in S$
then have *ExpCumm s k* = $(\int^+ x.\ ennreal\ (\sum i{<}k.\ \varrho\ ((s\ \#\#\ x)\ !!\ i) + \iota\ ((s\ \#\#\ x)\ !!\ i)\ (x\ !!\ i))\ \partial T\ s)$
proof (*induct k arbitrary*: *s*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc k*)
have $(\int^+\omega.\ ennreal\ (\sum i{<}Suc\ k.\ \varrho\ ((s\ \#\#\ \omega)\ !!\ i) + \iota\ ((s\ \#\#\ \omega)\ !!\ i)\ (\omega\ !!\ i))\ \partial T\ s)$
$= (\int^+\omega.\ ennreal\ (\varrho\ s + \iota\ s\ (\omega\ !!\ 0)) + ennreal\ (\sum i{<}k.\ \varrho\ (\omega\ !!\ i) + \iota\ (\omega\ !!\ i)\ (\omega\ !!\ (Suc\ i)))\ \partial T\ s)$
by (*auto intro*!: *nn-integral-cong*
simp del: *ennreal-plus*
simp: *ennreal-plus*[*symmetric*] *sum-nonneg sum.reindex lessThan-Suc-eq-insert-0 zero-notin-Suc-image*)
also have ... = $(\int^+\omega.\ \varrho\ s + \iota\ s\ (\omega\ !!\ 0)\ \partial T\ s) +$

```
        (∫⁺ω. (∑ i<k. ϱ (ω !! i) + ι (ω !! i) (ω !! (Suc i))) ∂T s)
       using ‹s ∈ S›
       by (intro nn-integral-add AE-I2) (auto simp: sum-nonneg)
     also have ... = (∑ s'∈S. τ s s' * (ϱ s + ι s s')) +
       (∫⁺ω. (∑ i<k. ϱ (ω !! i) + ι (ω !! i) (ω !! (Suc i))) ∂T s)
       using ‹s ∈ S› by (subst nn-integral-eq-sum)
      (auto simp del: ennreal-plus simp: ennreal-plus[symmetric] ennreal-mult[symmetric]
sum-nonneg)
     also have ... = (∑ s'∈S. τ s s' * (ϱ s + ι s s')) +
       (∑ s'∈S. τ s s' * ExpCumm s' k)
       using ‹s ∈ S› by (subst nn-integral-eq-sum) (auto simp: Suc)
     also have ... = ExpCumm s (Suc k)
       using ‹s ∈ S›
     by (simp add: field-simps sum.distrib sum-distrib-left[symmetric] ennreal-mult[symmetric]
           ennreal-plus[symmetric] sum-nonneg del: ennreal-plus)
     finally show ?case by simp
   qed }
  then show ?case by auto

next
  case (9 rel r k)
  { fix s assume s ∈ S
    then have ExpState s k = (∫⁺ x. ennreal (ϱ ((s ## x) !! k)) ∂T s)
    proof (induct k arbitrary: s)
      case (Suc k) then show ?case by (simp add: nn-integral-eq-sum[of s])
    qed simp }
  then show ?case by auto

next
  case (10 rel r F)
  moreover
  let ?F = svalid F
  define N where N ≡ Prob0 S ?F
  moreover define Y where Y ≡ Prob1 N S ?F
  moreover define const where const ≡ (λs. if s ∈ Y ∧ s ∉ ?F then − ϱ s −
(∑ s'∈S. τ s s' * ι s s') else 0)
  ultimately obtain l
    where l: gauss-jordan' (LES (S − Y ∪ ?F)) const = Some l
    and F: Sat F = Some ?F
    by (auto simp: ExpFuture-def Let-def split: bind-split-asm)

  from l have EF: ExpFuture ?F =
    Some (λs. if s ∈ Y then ennreal (l s) else ∞)
    unfolding ExpFuture-def N-def Y-def const-def by auto

  let ?R = reward (Future F)
  have l-eq: ∀ s∈S. l s = enn2real (∫⁺ω. ?R (s ## ω) ∂T s)
  proof (rule uniqueness-of-ExpFuture[OF N-def Y-def const-def])
    fix s assume s ∈ S
```

show $\bigwedge s.\ s{\in}S \Longrightarrow (\sum s'{\in}S.\ LES\ (S - Y \cup ?F)\ s\ s' * l\ s') = const\ s$
using *gauss-jordan'-correct*[*OF l*] **by** *auto*
qed
{ fix *s* **assume** [*simp*]: $s \in S\ s \in Y$
then have $s \in Prob1\ (Prob0\ S\ ?F)\ S\ ?F$
unfolding *Y-def N-def* **by** *auto*
then have $AE\ \omega\ in\ T\ s.\ (HLD\ S\ suntil\ HLD\ ?F)\ (s\ \#\#\ \omega)$
using *svalid-subset-S* **by** (*auto simp add*: *Prob1-iff*)
from *nn-integral-reward-finite*[*OF* ‹$s \in S$›] *this*
have $(\int^+\omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) \neq \infty$
by *simp*
with *l-eq* ‹$s \in S$› **have** $(\int^+\omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) = ennreal\ (l\ s)$
by (*auto simp*: *less-top*) **}**
moreover
{ fix *s* **assume** $s \in S\ s \notin Y$
with *infinite-reward*[*of s F*]
have $(\int^+\omega.\ reward\ (Future\ F)\ (s\ \#\#\ \omega)\ \partial T\ s) = \infty$
by (*simp add*: *Y-def N-def*) **}**
ultimately show *?case*
apply (*auto simp add*: *EF F simp del*: *reward.simps*)
apply (*case-tac* $x \in Y$)
apply *auto*
done
qed (*auto split*: *bind-split-asm*)

11.5 Completeness of *Sat*

theorem *Sat-complete*:
$Sat\ F \neq None$
proof (*induct F rule*: *Sat.induct*)
case ($?\ r\ rel\ \Phi\ \Psi$)
then have *F*: $Sat\ \Phi = Some\ (svalid\ \Phi)\ Sat\ \Psi = Some\ (svalid\ \Psi)$
by (*auto intro*!: *Sat-sound*)

define *constants* :: $'s \Rightarrow real$ **where** $constants = (\lambda s.\ if\ s \in svalid\ \Psi\ then\ 1\ else\ 0)$
define *distr* **where** $distr = LES\ (Prob0\ (svalid\ \Phi)\ (svalid\ \Psi) \cup svalid\ \Psi)$
have $\exists l.\ gauss\text{-}jordan'\ distr\ constants = Some\ l$
proof (*rule gauss-jordan'-complete*[*OF - uniqueness-of-ProbU*])
show $\forall s{\in}S.\ (\sum s'{\in}S.\ distr\ s\ s' * \mathcal{P}(\omega\ in\ T\ s'.\ pvalid\ (U^\infty\ \Phi\ \Psi)\ (s'\ \#\#\ \omega))) = constants\ s$
apply (*simp add*: *distr-def constants-def LES-def del*: *pvalid.simps space-T*)
proof *safe*
fix *s* **assume** $s \in svalid\ \Psi\ s \in S$
then show $(\sum s'{\in}S.\ (if\ s' = s\ then\ 1\ else\ 0) * \mathcal{P}(\omega\ in\ T\ s'.\ pvalid\ (U^\infty\ \Phi\ \Psi)\ (s'\ \#\#\ \omega))) = 1$
by (*simp add*: *single-l suntil-Stream*)
next

fix s **assume** s: $s \notin$ *svalid* Ψ $s \in S$
let *?x* = $\lambda s'$. $\mathcal{P}(\omega$ *in* T s'. *pvalid* $(U^\infty\ \Phi\ \Psi)$ $(s' \#\#\ \omega))$
show $(\sum s' \in S.$ (*if* $s \in$ *Prob0* (*svalid* Φ) (*svalid* Ψ) *then if* $s' = s$ *then 1 else 0 else if* $s' = s$ *then* $\tau\ s\ s' - 1$ *else* $\tau\ s\ s'$) $*$ *?x* $s') = 0$
proof *cases*
assume $s \in$ *Prob0* (*svalid* Φ) (*svalid* Ψ)
with s **show** *?thesis*
by (*simp add*: *single-l Prob0-iff svalid-subset-S T.prob-eq-0-AE del*: *space-T*)
next
assume *s-not-0*: $s \notin$ *Prob0* (*svalid* Φ) (*svalid* Ψ)
with s **have** $*$:$\bigwedge s'\ \omega.\ s' \in S \Longrightarrow$ *pvalid* $(U^\infty\ \Phi\ \Psi)$ $(s \#\#\ s' \#\#\ \omega) =$ *pvalid* $(U^\infty\ \Phi\ \Psi)$ $(s' \#\#\ \omega)$
by (*auto simp*: *suntil-Stream Prob0-iff svalid-subset-S*)

have $(\sum s' \in S.$ (*if* $s' = s$ *then* $\tau\ s\ s' - 1$ *else* $\tau\ s\ s'$) $*$ *?x* $s') =$
$(\sum s' \in S.\ \tau\ s\ s' *$ *?x* $s' - ($*if* $s' = s$ *then 1 else 0*$) *$ *?x* $s')$
by (*auto intro*!: *sum.cong simp*: *field-simps*)
also have $\ldots = (\sum s' \in S.\ \tau\ s\ s' *$ *?x* $s') -$ *?x* s
using s **by** (*simp add*: *single-l sum-subtractf*)
finally show *?thesis*
using $*$ *prob-sum*[*OF* ‹$s \in S$›] *s-not-0* **by** (*simp del*: *pvalid.simps*)
qed
qed
qed (*simp add*: *distr-def constants-def*)
then have *P*: $\exists\, l.$ *ProbUinfty* (*svalid* Φ) (*svalid* Ψ) = *Some* l
unfolding *ProbUinfty-def constants-def distr-def* **by** *simp*
with F **show** *?case*
by *auto*
next
case (*10 rel r* Φ)
then have F: *Sat* Φ = *Some* (*svalid* Φ)
by (*auto intro*!: *Sat-sound*)

let *?F* = *svalid* Φ
define N **where** $N \equiv$ *Prob0 S ?F*
define Y **where** $Y \equiv$ *Prob1 N S ?F*
define *const* **where** *const* $\equiv (\lambda s.$ *if* $s \in Y \wedge s \notin$ *?F then* $- \varrho\ s - (\sum s' \in S.\ \tau\ s\ s' * \iota\ s\ s')$ *else 0*)
let *?E* = $\lambda s'.\ \int^+ \omega.$ *reward* (*Future* Φ) $(s' \#\#\ \omega)\ \partial T\ s'$
have $\exists\, l.$ *gauss-jordan'* (*LES* $(S - Y \cup$ *?F*)) *const* = *Some* l
proof (*rule gauss-jordan'-complete*[*OF - uniqueness-of-ExpFuture*[*OF N-def Y-def const-def*]])
show $\forall s \in S.$ $(\sum s' \in S.$ *LES* $(S - Y \cup$ *svalid* $\Phi)$ $s\ s' *$ *enn2real* (*?E* $s'))$ = *const* s
proof
fix s **assume** $s \in S$
show $(\sum s' \in S.$ *LES* $(S - Y \cup$ *svalid* $\Phi)$ $s\ s' *$ *enn2real* (*?E* $s')) =$ *const* s
proof *cases*
assume s: $s \in S - (Y -$ *svalid* $\Phi)$

show *?thesis*
proof *cases*
assume $s \in Y$
with ‹$s \in S$› s ‹$s \in Y$› **show** *?thesis*
by (*simp add*: *LES-def const-def single-l ev-Stream*)
next
assume $s \notin Y$
with *infinite-reward*[*of s* Φ] *Y-def N-def s* ‹$s \in S$›
show *?thesis* **by** (*simp add*: *const-def LES-def single-l del*: *reward.simps*)
qed
next
assume *s*: $s \notin S - (Y - svalid\ \Phi)$

have $(\sum s' \in S.\ (if\ s' = s\ then\ \tau\ s\ s' - 1\ else\ \tau\ s\ s') * enn2real\ (?E\ s')) =$
$(\sum s' \in S.\ \tau\ s\ s' * enn2real\ (?E\ s') - (if\ s' = s\ then\ 1\ else\ 0) * enn2real\ (?E\ s'))$
by (*auto intro*!: *sum.cong simp*: *field-simps*)
also have $\ldots = (\sum s' \in S.\ \tau\ s\ s' * enn2real\ (?E\ s')) - enn2real\ (?E\ s)$
using ‹$s \in S$› **by** (*simp add*: *sum-subtractf single-l*)
finally show *?thesis*
using s ‹$s \in S$› *existence-of-ExpFuture*[*OF N-def Y-def* ‹$s \in S$› s]
by (*simp add*: *LES-def const-def del*: *reward.simps*)
qed
qed
qed *simp*
then have *P*: $\exists l.\ ExpFuture\ (svalid\ \Phi) = Some\ l$
unfolding *ExpFuture-def const-def N-def Y-def* **by** *auto*
with *F* **show** *?case*
by *auto*
qed (*force split*: *bind-split*)+

11.6 Completeness and Soundness *Sat*

corollary *Sat*: $Sat\ \Phi = Some\ (svalid\ \Phi)$
using *Sat-sound Sat-complete* **by** *auto*

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 c₁ c₂) f   = wp c₁ (wp c₂ f)
| wp (If b c₁ c₂) f  = (λs. if b s then wp c₁ f s else wp c₂ f s)
| wp (Par c₁ c₂) f   = wp c₁ f ⊓ wp c₂ f
| wp (Prob p c₁ c₂) f = (λs. pmf p True * wp c₁ f s + pmf p False * wp c₂ f s)
| wp (While b c) f   = lfp (λX s. if b s then wp c X s else f s)

lemma wp-mono: mono (wp c)
  by (induction c)
     (auto simp: monotone-def le-fun-def intro: order-trans le-infI1 le-infI2
           intro!: add-mono mult-left-mono lfp-mono[THEN le-funD])

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 c₁ c₂, s)   = (map-pmf (λ(p1', s'). (if p1' = Skip then c₂ else Seq p1'
c₂, s'))) ‘ step (c₁, s)
| step (If b c₁ c₂, s)  = (if b s then ≪c₁, s≫ else ≪c₂, s≫)
| step (Par c₁ c₂, s)   = ≪c₁, s≫ ∪ ≪c₂, s≫
| step (Prob p c₁ c₂, s) = {map-pmf (λb. if b then (c₁, s) else (c₂, 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)
  by (induction x rule: step.induct) simp-all

lemma step-non-empty: step x ≠ {}
  by (induction x rule: step.induct) simp-all

interpretation step: Markov-Decision-Process step
  proof qed (rule step-non-empty)
```

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*)
unfolding *rF-def*[*abs-def*]
by (*auto simp*: *sup-continuous-def fun-eq-iff SUP-sup-distrib* [*symmetric*] *image-comp*
split: *prod.splits pgcl.splits*)

lemma *mono-rF*: *mono* (*rF f*)
using *continuous-rF* **by** (*rule sup-continuous-mono*)

lemma *r-unfold*: *r f ω* = (*if fst* (*shd ω*) = *Skip then f* (*snd* (*shd ω*)) *else r f* (*stl ω*))
by (*subst lfp-unfold*[*OF mono-rF*]) (*simp add*: *rF-def*)

lemma *mono-r*: *F* ≤ *G* ⟹ *r F ω* ≤ *r G ω*
by (*rule le-funD*[*of - - ω*], *rule lfp-mono*)
(*auto intro*!: *lfp-mono simp*: *rF-def le-fun-def max.coboundedI2*)

lemma *measurable-rF*:
assumes *F*[*measurable*]: *F* ∈ *borel-measurable step.St*
shows *rF f F* ∈ *borel-measurable step.St*
unfolding *rF-def*[*abs-def*]
apply *measurable*
apply (*rule measurable-compose*[*OF measurable-shd*])
apply *measurable* []
apply (*rule measurable-compose*[*OF measurable-stl*])
apply *measurable* []
apply (*rule predE*)
apply (*rule measurable-compose*[*OF measurable-shd*])
apply *measurable*
done

lemma *measurable-r*[*measurable*]: *r f* ∈ *borel-measurable step.St*
using *continuous-rF measurable-rF* **by** (*rule borel-measurable-lfp*)

lemma *mono-r'*: *mono* (λ*F s*. ∏ *D*∈*step s*. ∫⁺ *t*. (*if fst t* = *Skip then f* (*snd t*) *else F t*) ∂*measure-pmf D*)
by (*auto intro*!: *monoI le-funI INF-mono*[*OF bexI*] *nn-integral-mono simp*: *le-fun-def*)

lemma *E-inf-r*:
step.E-inf s (*r f*) =
lfp (λ*F s*. ∏ *D*∈*step s*. ∫⁺ *t*. (*if fst t* = *Skip then f* (*snd t*) *else F t*) ∂*measure-pmf*

D) *s*
proof −
have *step.E-inf s* (*r f*) =
lfp (λ*F s*. ∏ *D*∈*step s*. ∫ + *t*. (*if fst t* = *Skip then f* (*snd t*) *else F t*) ∂*measure-pmf D*) *s*
unfolding *rF-def*[*abs-def*]
proof (*rule step.E-inf-lfp*[*THEN fun-cong*])
let *?F* = λ*t x*. (*if fst t* = *Skip then f* (*snd t*) *else x*)
show (λ(*s*, *x*). *?F s x*) ∈ *borel-measurable* (*count-space UNIV* ⨂ *M borel*)
apply (*simp add*: *measurable-split-conv split-beta*′)
apply (*intro borel-measurable-max borel-measurable-const measurable-If predE measurable-compose*[*OF measurable-snd*] *measurable-compose*[*OF measurable-fst*])
apply *measurable*
done
show ⋀*s*. *sup-continuous* (*?F s*)
by (*auto simp*: *sup-continuous-def SUP-sup-distrib*[*symmetric*] *split*: *prod.split pgcl.split*)
show ⋀*F cfg*. (∫ + ω. *?F* (*state cfg*) (*F* ω) ∂*step.T cfg*) =
?F (*state cfg*) (*nn-integral* (*step.T cfg*) *F*)
by (*auto simp*: *split*: *pgcl.split prod.split*)
qed (*rule step-finite*)
then show *?thesis*
by *simp*
qed

lemma *E-inf-r-unfold*:
step.E-inf s (*r f*) = (∏ *D*∈*step s*. ∫ + *t*. (*if fst t* = *Skip then f* (*snd t*) *else step.E-inf t* (*r f*)) ∂*measure-pmf D*)
unfolding *E-inf-r* **by** (*simp add*: *lfp-unfold*[*OF mono-r*′])

lemma *E-inf-r-induct*[*consumes 1*, *case-names step*]:
assumes *P s y*
assumes ∗: ⋀*F s y*. *P s y* ⟹
(⋀*s y*. *P s y* ⟹ *F s* ≤ *y*) ⟹ (⋀*s*. *F s* ≤ *step.E-inf s* (*r f*)) ⟹
(∏ *D*∈*step s*. ∫ + *t*. (*if fst t* = *Skip then f* (*snd t*) *else F t*) ∂*measure-pmf D*) ≤ *y*
shows *step.E-inf s* (*r f*) ≤ *y*
using ‹*P s y*›
unfolding *E-inf-r*
proof (*induction arbitrary*: *s y rule*: *lfp-ordinal-induct*[*OF mono-r*′[**where** *f*=*f*]])
case (*1 F*) **with** ∗[*of s y F*] **show** *?case*
unfolding *le-fun-def E-inf-r*[**where** *f*=*f*, *symmetric*] **by** *simp*
qed (*auto intro*: *SUP-least*)

lemma *E-inf-Skip*: *step.E-inf* (*Skip*, *s*) (*r f*) = *f s*
by (*subst E-inf-r-unfold*) *simp*

lemma *E-inf-Seq*:

assumes [*simp*]: $\bigwedge x.\ 0 \le f\ x$
shows *step.E-inf* (*Seq a b*, *s*) (*r f*) = *step.E-inf* (*a*, *s*) (*r* (λ*s*. *step.E-inf* (*b*, *s*) (*r f*)))
proof (*rule antisym*)
show *step.E-inf* (*Seq a b*, *s*) (*r f*) ≤ *step.E-inf* (*a*, *s*) (*r* (λ*s*. *step.E-inf* (*b*, *s*) (*r f*)))
proof (*coinduction arbitrary*: *a s rule*: *E-inf-r-induct*)
case *step* **then show** *?case*
by (*rewrite* **in** - ≤ ⌑ *E-inf-r-unfold*)
(*force intro*!: *INF-mono*[*OF bexI*] *nn-integral-mono intro*: *le-infI2*
simp: *E-inf-Skip image-comp*)
qed
show *step.E-inf* (*a*, *s*) (*r* (λ*s*. *step.E-inf* (*b*, *s*) (*r f*))) ≤ *step.E-inf* (*Seq a b*, *s*) (*r f*)
proof (*coinduction arbitrary*: *a s rule*: *E-inf-r-induct*)
case *step* **then show** *?case*
by (*rewrite* **in** - ≤ ⌑ *E-inf-r-unfold*)
(*force intro*!: *INF-mono*[*OF bexI*] *nn-integral-mono intro*: *le-infI2*
simp: *E-inf-Skip image-comp*)
qed
qed

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 (*rule antisym*)
have *E-inf-While-step*: *step.E-inf* (*While g c*, *s*) (*r f*) =
(*if g s then step.E-inf* (*c*, *s*) (*r* (λ*s*. *step.E-inf* (*While g c*, *s*) (*r f*))) *else f s*)
for *f s*
by (*rewrite E-inf-r-unfold*) (*simp add*: *min-absorb1 E-inf-Seq*)

have *mono* (λ*F s*. *if g s then step.E-inf* (*c*, *s*) (*r F*) *else f s*) (**is** *mono ?F*)
by (*auto intro*!: *mono-r step.E-inf-mono simp*: *mono-def le-fun-def max.coboundedI2*)
then show *lfp ?F s* ≤ *step.E-inf* (*While g c*, *s*) (*r f*)
proof (*induction arbitrary*: *s rule*: *lfp-ordinal-induct*[*consumes 1*])
case *mono* **then show** *?case*
by (*rewrite E-inf-While-step*) (*auto intro*!: *step.E-inf-mono mono-r le-funI*)
qed (*auto intro*: *SUP-least*)

define *w* **where** *w F s* = (∏ *D*∈*step s*. ∫⁺ *t*. (*if fst t* = *Skip then if g* (*snd t*) *then F* (*c*, *snd t*) *else f* (*snd t*) *else F t*) ∂*measure-pmf D*)
for *F s*
have *mono w*
by (*auto simp*: *w-def mono-def le-fun-def intro*!: *INF-mono*[*OF bexI*] *nn-integral-mono*)
□

define *d* **where** *d* = *c*
define *t* **where** *t* = *Seq d* (*While g c*)
then have (*t* = *While g c* ∧ *d* = *c* ∧ *g s*) ∨ *t* = *Seq d* (*While g c*)

```
    by auto
  then have step.E-inf (t, s) (r f) ≤ lfp w (d, s)
  proof (coinduction arbitrary: t d s rule: E-inf-r-induct)
    case (step F t d s)
    from step(1)
    show ?case
    proof (elim conjE disjE)
      { fix s have ¬ g s ⟹ F (While g c, s) ≤ f s
          using step(3)[of (While g c, s)] by (simp add: E-inf-While-step) }
      note [simp] = this
      assume t = Seq d (While g c) then show ?thesis
        by (rewrite lfp-unfold[OF ‹mono w›])
         (auto simp: max.absorb2 w-def intro!: INF-mono[OF bexI] nn-integral-mono
step)
    qed (auto intro!: step)
  qed
  also have lfp w = lfp (λF s. step.E-inf s (r (λs. if g s then F (c, s) else f s)))
    unfolding E-inf-r w-def
    by (rule lfp-lfp[symmetric]) (auto simp: le-fun-def intro!: INF-mono[OF bexI]
nn-integral-mono)
  finally have step.E-inf (While g c, s) (r f) ≤ (if g s then ... (c, s) else f s)
    unfolding t-def d-def by (rewrite E-inf-r-unfold) simp
  also have ... = lfp ?F s
    by (rewrite lfp-rolling[symmetric, of λF s. if g s then F (c, s) else f s  λF s.
step.E-inf s (r F)])
     (auto simp: mono-def le-fun-def sup-apply[abs-def] if-distrib[of max 0] max.coboundedI2
max.absorb2
               intro!: step.E-inf-mono mono-r cong del: if-weak-cong)
  finally show step.E-inf (While g c, s) (r f) ≤ ...
    .
qed
```

12.4 Equate Both Semantics

```
lemma E-inf-r-eq-wp: step.E-inf (c, s) (r f) = wp c f s
proof (induction c arbitrary: f s)
  case Skip then show ?case
    by (simp add: E-inf-Skip)
next
  case Abort then show ?case
  proof (intro antisym)
    have lfp (λF s. ∏D∈step s. ∫⁺ t. (if fst t = Skip then f (snd t) else F t)
∂measure-pmf D) ≤
      (λs. if ∃t. s = (Abort, t) then 0 else ⊤)
      by (intro lfp-lowerbound) (auto simp: le-fun-def)
    then show step.E-inf (Abort, s) (r f) ≤ wp Abort f s
      by (auto simp: E-inf-r le-fun-def split: if-split-asm)
  qed simp
next
```

case *Assign* **then show** *?case*
by (*rewrite E-inf-r-unfold*) (*simp add*: *min-absorb1*)
next
case (*If b c1 c2*) **then show** *?case*
by (*rewrite E-inf-r-unfold*) *auto*
next
case (*Prob p c1 c2*) **then show** *?case*
apply (*rewrite E-inf-r-unfold*)
apply *auto*
apply (*rewrite nn-integral-measure-pmf-support*[*of UNIV*::*bool set*])
apply (*auto simp*: *UNIV-bool ac-simps*)
done
next
case (*Par c1 c2*) **then show** *?case*
by (*rewrite E-inf-r-unfold*) (*auto intro*: *inf.commute*)
next
case (*Seq c1 c2*) **then show** *?case*
by (*simp add*: *E-inf-Seq*)
next
case (*While g c*) **then show** *?case*
apply (*simp add*: *E-inf-While*)
apply (*rewrite While*)
apply *auto*
done
qed

end

13 Formalization of the Crowds-Protocol

theory *Crowds-Protocol*
imports *../Discrete-Time-Markov-Chain*
begin

lemma *cond-prob-nonneg*[*simp*]: $0 \leq$ *cond-prob M A B*
by (*auto simp*: *cond-prob-def*)

lemma (**in** *MC-syntax*) *emeasure-suntil-geometric*:
assumes [*measurable*]: *Measurable.pred S P*
assumes $s \in X$ **and** $*$[*simp*]: $0 \leq p$ $0 \leq r$
assumes *r*: $\bigwedge s.\ s \in X \Longrightarrow$ *emeasure* (*T s*) {$\omega \in$*space* (*T s*). *P* ω} = *ennreal r*
assumes *p*: $\bigwedge s.\ s \in X \Longrightarrow$ *emeasure* (*K s*) *X* = *ennreal p* $p < 1$
assumes $\bigwedge t.$ *AE* ω *in T t.* $\neg$ (*P* $\sqcap$ (*HLD X* $\sqcap$ *nxt* (*HLD X suntil P*))) ω
shows *emeasure* (*T s*) {$\omega \in$*space* (*T s*). (*HLD X suntil P*) ω} = *r* / (1 − *p*)
proof (*subst emeasure-suntil-disj*)
let *?F* = $\lambda F\ s.$ *emeasure* (*T s*) {$\omega \in$ *space* (*T s*). *P* ω} + $\int^+ t.$ *F t* $*$ *indicator X t* ∂K *s*
let *?f* = $\lambda x.$ *ennreal r* + *ennreal p* $*$ *x*

```
  have mono ?F mono ?f
    by (auto intro!: monoI max.mono add-mono nn-integral-mono mult-left-mono
mult-right-mono simp: le-fun-def)

  have 1: lfp ?f ≤ lfp ?F s
    using ‹s ∈ X›
  proof (induction arbitrary: s rule: lfp-ordinal-induct[OF ‹mono ?f›])
    case step: (1 x)
    then have ?f x ≤ ?F (λ-. x) s
      by (auto simp: p r[simplified] nn-integral-cmult mult.commute[of - x]
            intro!: add-mono mult-right-mono)
    also have ?F (λ-. x) ≤ ?F (lfp ?F)
      using step
      by (intro le-funI add-mono order-refl nn-integral-mono) (auto simp: split:
split-indicator)
    finally show ?case
      by (subst lfp-unfold[OF ‹mono ?F›]) (auto simp: le-fun-def)
  qed (auto intro!: Sup-least)
  also have 2: lfp ?F s ≤ r / (1 − p)
    using ‹s ∈ X›
  proof (induction arbitrary: s rule: lfp-ordinal-induct[OF ‹mono ?F›])
    case (1 S)
    with r have ?F S s ≤ ennreal r + (∫⁺x. ennreal (r / (1 − p)) ∗ indicator X
x ∂K s)
      by (intro add-mono nn-integral-mono) (auto split: split-indicator)
    also have ... ≤ ennreal r + ennreal (r ∗ p / (1 − p))
    using ‹s ∈ X› by (simp add: nn-integral-cmult-indicator p ennreal-mult''[symmetric])
    also have ... = ennreal (r / (1 − p))
      using ‹p < 1› by (simp add: field-simps ennreal-plus[symmetric] del: en-
nreal-plus)
    finally show ?case .
  qed (auto intro!: SUP-least)
  finally obtain x where x: lfp ?f = ennreal x and [simp]: 0 ≤ x
    by (cases lfp ?f) (auto simp: top-unique)
  from ‹p < 1› have ⋀x. x = r + p ∗ x ⟹ x = r / (1 − p)
    by (auto simp: field-simps)
  with lfp-unfold[OF ‹mono ?f›] ‹p < 1› have lfp ?f = r / (1 − p)
   unfolding x by (auto simp add: ennreal-plus[symmetric] ennreal-mult[symmetric]
simp del: ennreal-plus)
  with 1 2 show lfp ?F s = ennreal (r / (1 − p))
    by auto
qed fact+
```

13.1 Definition of the Crowds-Protocol

```
datatype 'a state = Start | Init 'a | Mix 'a | End

lemma inj-Mix[simp]: inj-on Mix A
  by (auto intro: inj-onI)
```

lemma *inj-Init*[*simp*]: *inj-on Init A*
by (*auto intro*: *inj-onI*)

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*
by *auto*

lemma *Init-cut-Mix*[*simp*]:
Init ‘ *H* ∩ *Mix* ‘ *J* = {}
by *auto*

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*
using *p-f* **by** *simp-all*

lemma *p-j-nonneg*[*simp*]: *0* ≤ *p-j*
by (*simp add*: *p-j-def*)

definition *p-H* = *card H* / *card J*

lemma *p-H-nonneg*[*simp*]: *0* ≤ *p-H p-H* ≤ *1*
by (*auto simp*: *p-H-def divide-le-eq-1 card-gt-0-iff intro*!: *card-mono*)

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 $N\ s = \mathit{embed\text{-}pmf}\ (\mathit{next\text{-}prob}\ s)$

interpretation $\mathit{MC\text{-}syntax}\ N$ **.**

abbreviation $\mathfrak{P} \equiv T\ \mathit{Start}$

abbreviation $E\ s \equiv \mathit{set\text{-}pmf}\ (N\ s)$

lemma $\mathit{finite\text{-}C}[\mathit{simp}]$: $\mathit{finite}\ C$
using $\mathit{C\text{-}smaller}\ \mathit{finite\text{-}J}$ **by** $(\mathit{blast}\ \mathit{intro}\text{: }\mathit{finite\text{-}subset})$

lemma $\mathit{sum\text{-}p\text{-}i\text{-}C}[\mathit{simp}]$: $\mathit{sum}\ \mathit{p\text{-}i}\ C = 0$
by $(\mathit{auto}\ \mathit{intro}\text{: }\mathit{sum.neutral}\ \mathit{p\text{-}i\text{-}C})$

lemma $\mathit{sum\text{-}p\text{-}i\text{-}H}[\mathit{simp}]$: $\mathit{sum}\ \mathit{p\text{-}i}\ H = 1$
using $\mathit{C\text{-}smaller}$ **by** $(\mathit{simp}\ \mathit{add}\text{: }\mathit{sum\text{-}diff}\ \mathit{p\text{-}i\text{-}distr})$

lemma $\mathit{possible\text{-}jondo}$:
obtains j **where** $j \in J\ j \notin C\ \mathit{p\text{-}i}\ j \neq 0$
proof $(\mathit{atomize\text{-}elim},\ \mathit{rule}\ \mathit{ccontr})$
assume $\neg\ (\exists j.\ j \in J \wedge j \notin C \wedge \mathit{p\text{-}i}\ j \neq 0)$
with $\mathit{p\text{-}i\text{-}C}$ **have** $\forall j{\in}J.\ \mathit{p\text{-}i}\ j = 0$
by auto
with $\mathit{p\text{-}i\text{-}distr}$ **show** False
by simp
qed

lemma $\mathit{C\text{-}le\text{-}J}[\mathit{simp}]$: $\mathit{card}\ C < \mathit{card}\ J$
using $\mathit{C\text{-}smaller}$
by $(\mathit{intro}\ \mathit{psubset\text{-}card\text{-}mono})\ \mathit{auto}$

lemma $\mathit{p\text{-}H}$: $0 < \mathit{p\text{-}H}\ \mathit{p\text{-}H} < 1$
using $\mathit{J\text{-}not\text{-}empty}\ \mathit{C\text{-}smaller}\ \mathit{C\text{-}non\text{-}empty}$
by $(\mathit{simp\text{-}all}\ \mathit{add}\text{: }\mathit{p\text{-}H\text{-}def}\ \mathit{card\text{-}Diff\text{-}subset}\ \mathit{card\text{-}mono}\ \mathit{field\text{-}simps}\ \mathit{zero\text{-}less\text{-}divide\text{-}iff}\ \mathit{card\text{-}gt\text{-}0\text{-}iff})$

lemma $\mathit{p\text{-}H\text{-}p\text{-}f\text{-}pos}$: $0 < \mathit{p\text{-}H} * \mathit{p\text{-}f}$
using $\mathit{p\text{-}f}\ \mathit{p\text{-}H}$ **by** $(\mathit{simp}\ \mathit{add}\text{: }\mathit{zero\text{-}less\text{-}mult\text{-}iff})$

lemma $\mathit{p\text{-}H\text{-}p\text{-}f\text{-}less\text{-}1}$: $\mathit{p\text{-}H} * \mathit{p\text{-}f} < 1$
proof $-$
have $\mathit{p\text{-}H} * \mathit{p\text{-}f} < 1 * 1$
using $\mathit{p\text{-}H}\ \mathit{p\text{-}f}$ **by** $(\mathit{intro}\ \mathit{mult\text{-}strict\text{-}mono})\ \mathit{auto}$
then show $\mathit{p\text{-}H} * \mathit{p\text{-}f} < 1$ **by** simp
qed

lemma $\mathit{p\text{-}j\text{-}pos}$: $0 < \mathit{p\text{-}j}$
unfolding $\mathit{p\text{-}j\text{-}def}$ **using** $\mathit{J\text{-}not\text{-}empty}$ **by** auto

lemma *H-compl*: *1 − p-H = real* (*card C*) */ real* (*card J*)
using *C-non-empty J-not-empty C-smaller*
by (*simp add*: *p-H-def card-Diff-subset card-mono of-nat-diff divide-eq-eq field-simps*)

lemma *H-compl2*: *1 − p-H = card C ∗ p-j*
unfolding *H-compl p-j-def* **by** *simp*

lemma *H-eq2*: *card H ∗ p-j = p-H*
unfolding *p-j-def p-H-def* **by** *simp*

lemma *pmf-next-pmf*[*simp*]: *pmf* (*N s*) *t = next-prob s t*
unfolding *N-def*
proof (*rule pmf-embed-pmf*)
show ⋀*x*. *0 ≤ next-prob s x*
using *p-j-pos p-f* **by** (*auto simp*: *next-prob-def intro*: *p-i-nonneg split*: *state.split*)
show (∫ $^+$ *x*. *ennreal* (*next-prob s x*) *∂count-space UNIV*) *= 1*
using *p-f J-not-empty*
by (*subst nn-integral-count-space′*[**where** *A=Init‘H ∪ Mix‘J ∪ {End}*])
(*auto simp*: *next-prob-def sum.reindex sum.union-disjoint p-i-distr p-j-def*
split: *state.split*)
qed

lemma *next-prob-Start*[*simp*]: *next-prob Start* (*Init j*) = (*if j ∈ H then p-i j else 0*)
by (*auto simp*: *next-prob-def*)

lemma *next-prob-to-Init*[*simp*]: *j ∈ H ⟹ next-prob s* (*Init j*) =
(*case s of Start ⇒ p-i j* | *- ⇒ 0*)
by (*cases s*) (*auto simp*: *next-prob-def*)

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*)
by (*cases s*) (*auto simp*: *next-prob-def*)

lemma *next-prob-to-End*[*simp*]: *next-prob s End* =
(*case s of Mix j ⇒ 1 − p-f* | *End ⇒ 1* | *- ⇒ 0*)
by (*cases s*) (*auto simp*: *next-prob-def*)

lemma *next-prob-from-End*[*simp*]: *next-prob End s = 0 ⟷ s ≠ End*
by (*cases s*) (*auto simp*: *next-prob-def*)

lemma *next-prob-Mix-MixI*: ∃ *j*. *s = Mix j ⟹* ∃ *j*∈*J*. *s′ = Mix j ⟹ next-prob s s′ = p-f ∗ p-j*
by (*cases s*) *auto*

lemma *E-Start*: *E Start* = {*Init j* | *j*. *j ∈ H ∧ p-i j ≠ 0* }
using *p-i-C* **by** (*auto simp*: *set-pmf-iff next-prob-def split*: *state.splits if-split-asm*)

lemma *E-Init*: *E* (*Init j*) = {*Mix j* | *j*. *j* ∈ *J* }
using *p-j-pos C-smaller* **by** (*auto simp*: *set-pmf-iff next-prob-def split*: *state.splits if-split-asm*)

lemma *E-Mix*: *E* (*Mix j*) = {*Mix j* | *j*. *j* ∈ *J* } ∪ {*End*}
using *p-j-pos p-f* **by** (*auto simp*: *set-pmf-iff next-prob-def split*: *state.splits if-split-asm*)

lemma *E-End*: *E End* = {*End*}
by (*auto simp*: *set-pmf-iff next-prob-def split*: *state.splits if-split-asm*)

lemma *enabled-End*:
enabled End ω ⟷ *ω* = *sconst End*
proof *safe*
assume *enabled End ω* **then show** *ω* = *sconst End*
proof (*coinduction arbitrary*: *ω*)
case *Eq-stream* **then show** *?case*
by (*auto simp*: *enabled.simps*[*of* - *ω*] *E-End*)
qed
next
show *enabled End* (*sconst End*)
by *coinduction* (*simp add*: *E-End*)
qed

lemma *AE-End*: (*AE ω in T End*. *P ω*) ⟷ *P* (*sconst End*)
proof −
have (*AE ω in T End*. *P ω*) ⟷ (*AE ω in T End*. *P ω* ∧ *ω* = *sconst End*)
using *AE-T-enabled*[*of End*] **by** (*simp add*: *enabled-End*)
also have ... = (*AE ω in T End*. *P* (*sconst End*) ∧ *ω* = *sconst End*)
by (*simp add*: *enabled-End del*: *AE-conj-iff cong*: *rev-conj-cong*)
also have ... = (*AE ω in T End*. *P* (*sconst End*))
using *AE-T-enabled*[*of End*] **by** (*simp add*: *enabled-End*)
finally show *?thesis*
by *simp*
qed

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 −
have ∗: {*Mix j* | *j*. *j* ∈ *J* } = *Mix* ‘ *J*
by *auto*
show *?thesis*
using *emeasure-eq-0-AE*[*OF AE-End*] *p-f*
apply (*subst* (*1 2*) *emeasure-Collect-T*)
apply *simp*
apply (*subst* (*1 2*) *nn-integral-measure-pmf-finite*)
apply (*auto simp*: *E-Mix E-Init* ∗ *sum.reindex sum-distrib-right*[*symmetric*]

divide-ennreal
ennreal-times-divide[*symmetric*])
done
qed

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* ≠ {}
by (*auto simp*: *visit-def HLD-iff*)

lemma *visit-unique2*:
assumes *visit I1 L1 ω visit I2 L2 ω*
shows *L1* ∩ *L2* ≠ {}
proof −
let *?U* = λ*L ω*. (*HLD* (*Mix*‘*J*) *suntil* ((*Mix*‘(*L*∩*J*)) · *HLD* {*End*})) *ω*
have *?U L1* (*stl ω*) *?U L2* (*stl ω*)
using *assms* **by** (*auto simp*: *visit-def*)
then show *L1* ∩ *L2* ≠ {}
proof (*induction stl ω arbitrary*: *ω rule*: *suntil-induct-strong*)
case *base* **then show** *?case*
by (*auto simp add*: *suntil.simps*[*of* - - *stl* (*stl ω*)] *suntil.simps*[*of* - - *stl ω*]
HLD-iff)
next
case *step*
show *?case*
proof *cases*
assume ((*Mix*‘(*L2*∩*J*)) · *HLD* {*End*}) (*stl ω*)
with *step.hyps* **show** *?thesis*
by (*auto simp*: *inj-Mix HLD-iff elim*: *suntil.cases*)
next
assume ¬ ((*Mix*‘(*L2*∩*J*)) · *HLD* {*End*}) (*stl ω*)
with *step.prems* **have** *?U L2* (*stl* (*stl ω*))
by (*auto elim*: *suntil.cases*)
then show *?thesis*
by (*rule step.hyps*(*4*)[*OF refl*])
qed
qed
qed

lemma *visit-imp-in-H*: *visit* {*i*} *J ω* ⟹ *i* ∈ *H*
by (*auto simp*: *visit-def HLD-iff*)

lemma *emeasure-visit*:
assumes *I*: *I* ⊆ *H* **and** *L*: *L* ⊆ *J*
shows *emeasure* 𝔓 {*ω*∈*space* 𝔓. *visit I L ω*} = (∑ *i*∈*I*. *p-i i*) ∗ (*card L* ∗ *p-j*)

```
proof −
  let ?J = HLD (Mix‘J) and ?E = (Mix‘L) · HLD {End}
  let ?φ = ?J aand not ?E
  let ?P = λx P. emeasure (T x) {ω∈space (T x). P ω}

  have [intro]: finite L
    using finite-J ‹L ⊆ J› by (blast intro: finite-subset)
  have [simp, intro]: finite I
    using finite-J ‹I ⊆ H› by (blast intro: finite-subset)

  { fix j assume j: j ∈ H
    have ?P (Mix j) (?J suntil ?E) = (p-f * p-j * (1 − p-f) * card L) / (1 − p-f)
    proof (rule emeasure-suntil-geometric)
      fix s assume s: s ∈ Mix ‘ J
      then have ?P s ?E = (∫⁺x. ennreal (1 − p-f) * indicator (Mix‘L) x ∂N s)
        by (auto simp add: emeasure-HLD-nxt emeasure-HLD AE-measure-pmf-iff
emeasure-pmf-single
                 split: state.split split-indicator simp del: space-T nxt.simps
                 intro!: nn-integral-cong-AE)
      also have ... = ennreal (1 − p-f) * emeasure (N s) (Mix‘L)
        using p-f by (intro nn-integral-cmult-indicator) auto
      also have ... = ennreal ((1 − p-f) * card L * p-j * p-f)
        using s assms
        by (subst emeasure-measure-pmf-finite)
           (auto simp: sum.reindex subset-eq ennreal-mult mult-ac)
      finally show ?P s ?E = p-f * p-j * (1 − p-f) * card L
        by simp
    next
      show ⋀t. AE ω in T  t. ¬ (?E ⊓ (?J ⊓ nxt (?J suntil ?E))) ω
        by (intro AE-I2) (auto simp: HLD-iff elim: suntil.cases)
    qed (insert p-f j, auto simp: emeasure-measure-pmf-finite sum.reindex p-j-def)
    then have ?P (Init j) (?J suntil ?E) = (p-f * p-j * (1 − p-f) * card L) / (1
− p-f) / p-f
      by (subst emeasure-Init-eq-Mix) (simp-all add:  suntil.simps[of - - x ## s for
x s] divide-ennreal p-f)
    then have ?P (Init j) (?J suntil ?E) = p-j * card L
      using p-f by simp }
  note J-suntil-E = this

  have ?P Start (visit I L) = (∫⁺x. ?P x (?J suntil ?E) * indicator (Init‘I) x ∂N
Start)
      unfolding visit-def using I L by (subst emeasure-HLD-nxt) (auto simp:
Int-absorb2)
  also have ... = (∫⁺x. ennreal (p-j * card L) * indicator (Init‘I) x ∂N Start)
    using I J-suntil-E
    by (intro nn-integral-cong ennreal-mult-right-cong)
       (auto split: split-indicator-asm)
  also have ... = ennreal ((∑ i∈I. p-i i) * card L * p-j)
    using p-j-pos assms
```

by (*subst nn-integral-cmult-indicator*)
(*auto simp*: *emeasure-measure-pmf-finite sum.reindex subset-eq ennreal-mult*[*symmetric*] *sum-nonneg*)
finally show *?thesis* **by** (*simp add*: *ac-simps*)
qed

lemma *measurable-visit*[*measurable*]: *Measurable.pred S* (*visit I L*)
by (*simp add*: *visit-def*)

lemma *AE-visit*: *AE* ω *in* $\mathfrak{P}$. *visit H J* ω
proof (*rule T.AE-I-eq-1*)
show *emeasure* $\mathfrak{P}$ {ω∈*space* $\mathfrak{P}$. *visit H J* ω} = *1*
using *J-not-empty* **by** (*subst emeasure-visit*) (*simp-all add*: *p-j-def*)
qed *simp*

13.2 Server gets no information

lemma *server-view1*: *j* ∈ *J* ⟹ $\mathcal{P}$(ω *in* $\mathfrak{P}$. *visit H* {*j*} ω) = *p-j*
unfolding *measure-def* **by** (*subst emeasure-visit*) *simp-all*

lemma *server-view-indep*:
L ⊆ *J* ⟹ *I* ⊆ *H* ⟹ $\mathcal{P}$(ω *in* $\mathfrak{P}$. *visit I L* ω) = $\mathcal{P}$(ω *in* $\mathfrak{P}$. *visit H L* ω) ∗ $\mathcal{P}$(ω *in* $\mathfrak{P}$. *visit I J* ω)
unfolding *measure-def*
by (*subst* (*1 2 3*) *emeasure-visit*) (*auto simp*: *p-j-def sum-nonneg subset-eq*)

lemma *server-view*: $\mathcal{P}$(ω *in* $\mathfrak{P}$. ∃*j*∈*H*. *visit* {*j*} {*j*} ω) = *p-j*
using *finite-J*
proof (*subst T.prob-sum*[**where** *I*=*H* **and** *P*=λ*j*. *visit* {*j*} {*j*}])
show (∑*j*∈*H*. $\mathcal{P}$(ω *in* $\mathfrak{P}$. *visit* {*j*} {*j*} ω)) = *p-j*
by (*auto simp*: *measure-def emeasure-visit sum-distrib-right*[*symmetric*] *simp del*: *space-T sets-T*)
show *AE x in* $\mathfrak{P}$. (∀ *n*∈*H*. *visit* {*n*} {*n*} *x* ⟶ (∃*j*∈*H*. *visit* {*j*} {*j*} *x*)) ∧
((∃*j*∈*H*. *visit* {*j*} {*j*} *x*) ⟶ (∃!*n*. *n* ∈ *H* ∧ *visit* {*n*} {*n*} *x*))
by (*auto dest*: *visit-unique1*)
qed *simp-all*

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*
by (*simp add*: *hit-C-def*)

lemma *measurable-before-C*[*measurable*]: *Measurable.pred S* (*before-C B*)
by (*simp add*: *before-C-def*)

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 −
{ fix ω *s* **assume** ((*HLD* (*Jondo H*)) *suntil* (*Jondo* (*B* ∩ *H*) · *HLD* (*Mix* ‘ *C*))) ω
enabled s ω *s* ∈ *Jondo H*
then have (*HLD* (*Mix* ‘ *H*) *suntil* (*Mix* ‘ (*B* ∩ *H*) · (*HLD* (*Mix* ‘ *C*)))) ω
proof (*induction arbitrary*: *s*)
case (*base* ω) **then show** *?case*
by (*auto simp*: *HLD-iff enabled.simps*[*of* - ω] *E-Init E-Mix intro*!: *suntil.intros*(*1*))
next
case (*step* ω) **from** *step.prems step.hyps step.IH*[*of shd* ω] **show** *?case*
by (*auto simp*: *HLD-iff enabled.simps*[*of* - ω] *E-Init E-Mix*
suntil.simps[*of* - - ω] *enabled-End suntil-sconst*)
qed }
note *this*[*of stl* ω *shd* ω]
moreover
{ fix ω *s* **assume** (*HLD* (*Mix* ‘ *H*) *suntil* (*Mix* ‘ (*B* ∩ *H*) · (*HLD* (*Mix* ‘ *C*)))) ω
enabled s ω *s* ∈ *Jondo H*
then have ((*HLD* (*Jondo H*)) *suntil* ((*Jondo* (*B* ∩ *H*)) · *HLD* (*Mix* ‘ *C*))) ω
proof (*induction arbitrary*: *s*)
case (*step* ω) **from** *step.prems step.hyps step.IH*[*of shd* ω] **show** *?case*
by (*auto simp*: *HLD-iff enabled.simps*[*of* - ω] *E-Init E-Mix*
suntil.simps[*of* - - ω] *enabled-End suntil-sconst*)
qed (*auto intro*: *suntil.intros simp*: *HLD-iff*) **}**
note *this*[*of stl* ω *shd* ω]
ultimately show *?thesis*
using *assms*
using ‹*enabled Start* ω›
unfolding *before-C-def suntil.simps*[*of* - - ω] *enabled.simps*[*of* - ω]
by (*auto simp*: *E-Start HLD-iff*)
qed

lemma *before-C-unique*:
assumes ω: *before-C I1* ω *before-C I2* ω **shows** *I1* ∩ *I2* ≠ {}
using ω **unfolding** *before-C-def*
proof *induction*
case (*base* ω) **then show** *?case*
by (*auto simp add*: *suntil.simps*[*of* - - ω] *suntil.simps*[*of* - - *stl* ω] *HLD-iff*)
next
case (*step* ω) **then show** *?case*
by (*auto simp add*: *suntil.simps*[*of* - - ω] *suntil.simps*[*of* - - *stl* ω] *HLD-iff*)
qed

lemma *hit-C-imp-before-C*:
assumes *enabled Start* ω *hit-C* ω **shows** *before-C H* ω

proof −
let *?X* = *Init'H* ∪ *Mix'H*
{ fix *ω s* **assume** *ev* (*HLD* (*Mix'C*)) *ω s*∈*?X enabled s ω*
then have ((*HLD* (*Jondo H*)) *suntil* (*?X* · *HLD* (*Mix ' C*))) (*s ##* *ω*)
proof (*induction arbitrary*: *s rule*: *ev-induct-strong*)
case (*step ω s*) **from** *step.IH*[*of shd ω*] *step.prems step.hyps* **show** *?case*
by (*auto simp*: *enabled.simps*[*of* - *ω*] *suntil-Stream E-Init E-Mix HLD-iff enabled-End ev-sconst*)
qed (*auto simp*: *suntil-Stream*) **}**
from *this*[*of stl ω shd ω*] *assms* **show** *?thesis*
by (*auto simp*: *before-C-def hit-C-def enabled.simps*[*of* - *ω*] *E-Start*)
qed

lemma *before-C-single*:
assumes *before-C I ω* **shows** ∃ *i*∈*I* ∩ *H*. *before-C* {*i*} *ω*
using *assms* **unfolding** *before-C-def* **by** *induction* (*auto simp*: *HLD-iff intro*: *suntil.intros*)

lemma *before-C-imp-in-H*: *before-C* {*i*} *ω* ⟹ *i* ∈ *H*
by (*auto dest*: *before-C-single*)

13.4 The probability that the sender hits a collaborator

lemma *Pr-hit-C*: *P*(*ω in* 𝔓. *hit-C ω*) = (*1* − *p-H*) / (*1* − *p-H* ∗ *p-f*)
proof −
let *?P* = λ*x P*. *emeasure* (*T x*) {*ω*∈*space* (*T x*). *P ω*}
let *?M* = *HLD* (*Mix ' C*) **and** *?I* = *Init'H* **and** *?J* = *Mix'H*
let *?φ* = (*HLD ?J*) *aand not ?M*

{ fix *s* **assume** *s*: *s* ∈ *Jondo J*
have *AE ω in T s*. *ev ?M ω* ⟷ (*HLD ?J suntil ?M*) *ω*
using *AE-T-enabled*
proof *eventually-elim*
fix *ω* **assume** *ω*: *enabled s ω*
show *ev ?M ω* ⟷ (*HLD ?J suntil ?M*) *ω*
proof
assume *ev ?M ω*
from *this ω s* **show** (*HLD ?J suntil ?M*) *ω*
proof (*induct arbitrary*: *s rule*: *ev-induct-strong*)
case (*step ω*) **then show** *?case*
by (*auto simp*: *HLD-iff enabled.simps*[*of* - *ω*] *suntil.simps*[*of* - - *ω*] *E-End E-Init E-Mix enabled-End ev-sconst*)
qed (*auto simp*: *HLD-iff E-Init intro*: *suntil.intros*)
qed (*rule ev-suntil*)
qed }
note *ev-eq-suntil* = *this*

have *?P Start hit-C* = (∫ ⁺*x*. *?P x* (*ev ?M*) ∗ *indicator ?I x* ∂*N Start*)

```
    unfolding hit-C-def by (rule emeasure-HLD-nxt) measurable
  also have ... = (∫⁺x. ennreal ((1 − p-H) / (1 − p-f * p-H)) * indicator ?I x
∂N Start)
  proof (intro nn-integral-cong ennreal-mult-right-cong refl)
    fix x assume indicator (Init ‘ H) x ≠ 0
    then have x ∈ ?I
      by (auto split: split-indicator-asm)
    { fix j assume j: j ∈ H
      with ev-eq-suntil[of Mix j] have ?P (Mix j) (ev ?M) = ?P (Mix j) ((HLD
?J) suntil ?M)
        by (intro emeasure-eq-AE) auto
      also have ... = (((1 − p-H) * p-f)) / (1 − p-H * p-f)
      proof (rule emeasure-suntil-geometric)
        fix s assume s: s ∈ Mix ‘ H
        from s C-smaller show ?P s ?M = ennreal ((1 − p-H) * p-f)
          by (subst emeasure-HLD)
           (auto simp add: emeasure-measure-pmf-finite sum.reindex subset-eq p-j-def
H-compl)
        from s show emeasure (N s) (Mix‘H) = p-H * p-f
          by (auto simp: emeasure-measure-pmf-finite sum.reindex p-H-def p-j-def)
      qed (insert j, auto simp: HLD-iff p-H-p-f-less-1)
      finally have ?P (Init j) (ev ?M) = (1 − p-H) / (1 − p-H * p-f)
        using p-f
        by (subst emeasure-Init-eq-Mix)
        (auto simp: ev-Stream AE-End ev-sconst HLD-iff mult-le-one divide-ennreal)
}
    then show ?P x (ev ?M) = (1 − p-H) / (1 − p-f * p-H)
      using ‹x ∈ ?I› by (auto simp: mult-ac)
  qed
  also have ... = ennreal ((1 − p-H) / (1 − p-H * p-f))
    using p-j-pos p-H p-H-p-f-less-1
    by (subst nn-integral-cmult-indicator)
       (auto simp: emeasure-measure-pmf-finite sum.reindex subset-eq mult-ac
            intro!: divide-nonneg-nonneg)
  finally show ?thesis
    by (simp add: measure-def mult-le-one)
qed

lemma before-C-imp-hit-C:
  assumes enabled Start ω before-C B ω
  shows hit-C ω
proof −
  { fix ω j assume ((HLD (Jondo H)) suntil (Jondo (B ∩ H) · HLD (Mix ‘ C)))
ω
      j ∈ H enabled (Mix j) ω
    then have ev (HLD (Mix‘C)) ω
    proof (induction arbitrary: j rule: suntil-induct-strong)
      case (step ω) then show ?case
      by (auto simp: enabled.simps[of - ω] E-Mix enabled-End ev-sconst suntil-sconst
```

```
HLD-iff)
    qed auto }
  from this[of stl (stl ω)] assms show hit-C ω
    by (force simp: before-C-def hit-C-def E-Start HLD-iff E-Init
      enabled.simps[of - ω] ev.simps[of - ω] suntil.simps[of - - ω]
      enabled.simps[of - stl ω] ev.simps[of - stl ω] suntil.simps[of - - stl ω])
qed

lemma negE: ¬ P ⟹ P ⟹ False
  by blast

lemma Pr-visit-before-C:
  assumes L: L ⊆ H and I: I ⊆ H
  shows 𝒫(ω in 𝔓. visit I J ω ∧ before-C L ω | hit-C ω ) =
    (∑ i∈I. p-i i) * card L * p-j * p-f + (∑ i∈I ∩ L. p-i i) * (1 − p-H * p-f)
proof −
  let ?M = Mix'H
  let ?P = λx P. emeasure (T x) {ω∈space (T x). P ω}
  let ?V = (visit I J aand before-C L) aand hit-C
  let ?U = HLD ?M suntil (Mix'L · HLD (Mix'C))
  let ?L = HLD (Mix'C)

  have IJ: x ∈ I ⟹ x ∈ J for x
    using I by auto

  have [simp, intro]: finite I finite L
    using L I by (auto dest: finite-subset)

  have ?P Start ?V = ?P Start ((Init'I · ?U) or (Init'(I ∩ L) · ?L))
  proof (rule emeasure-Collect-eq-AE)
    show AE ω in 𝔓. ?V ω ⟷ ((Init'I · ?U) or (Init'(I ∩ L) · ?L)) ω
      using AE-T-enabled AE-visit
    proof eventually-elim
      case (elim ω)
      then show ?case
        using before-C-imp-hit-C[of ω L]  before-C[of ω L] I L
        by (auto simp: visit-def HLD-iff Int-absorb2)
    qed
    show Measurable.pred 𝔓 ((Init'I · ?U) or (Init'(I ∩ L) · ?L))
      by measurable
  qed measurable
  also have ... = ?P Start (Init'I · ?U) + ?P Start (Init'(I ∩ L) · ?L)
    using L I
    apply (subst plus-emeasure)
    apply (auto intro!: arg-cong2[where f=emeasure])
    apply (subst (asm) suntil.simps)
    apply (auto simp add: HLD-iff[abs-def] elim: suntil.cases)
    done
  also have ?P Start (Init'(I ∩ L) · ?L) = (∑ i∈I∩L. p-i i * (1 − p-H))
```

```
    using L I C-smaller p-j-pos
    apply (subst emeasure-HLD-nxt emeasure-HLD, simp)+
    apply (subst nn-integral-indicator-finite)
   apply (auto simp: emeasure-measure-pmf-finite sum.reindex next-prob-def sum.If-cases
                     Int-absorb2 H-compl2 ennreal-mult[symmetric] sum-nonneg
                     sum-distrib-left[symmetric] sum-distrib-right[symmetric]
                intro!: sum.cong sum-nonneg)
    apply (subst (asm) ennreal-inj)
     apply (auto intro!: mult-nonneg-nonneg sum-nonneg sum.mono-neutral-left
elim!: negE)
    done
  also have ?P Start (Init‘I · ?U) = (∑ i∈I. ?P (Init i) ?U * p-i i)
    using I
    by (subst emeasure-HLD-nxt, simp)
     (auto simp: nn-integral-indicator-finite sum.reindex emeasure-measure-pmf-finite
           intro!: sum.cong[OF refl])
  also have ... = (∑ i∈I. ennreal (p-f * (1 − p-H) * p-j * card L / (1 − p-H *
p-f)) * p-i i)
  proof (intro sum.cong refl arg-cong2[where f=(*)])
    fix i assume i ∈ I
    with I have i: i ∈ H
     by auto
    have ?P (Mix i) ?U = (p-f * p-f * (1 − p-H) * p-j * card L / (1 − p-H *
p-f))
     unfolding before-C-def
    proof (rule emeasure-suntil-geometric[where X=?M])
     show Mix i ∈ ?M
       using i by auto
    next
     fix s assume s ∈ ?M
     with p-f p-j-pos L C-smaller[THEN less-imp-le]
     show ?P s (Mix‘L · (HLD (Mix ‘ C))) = ennreal (p-f * p-f * (1 − p-H) *
p-j * card L)
      apply (simp add: emeasure-HLD emeasure-HLD-nxt del: nxt.simps space-T)
       apply (subst nn-integral-measure-pmf-support[of Mix‘L])
        apply (auto simp add: subset-eq emeasure-measure-pmf-finite sum.reindex
H-compl p-j-def
          ennreal-mult[symmetric] ennreal-of-nat-eq-real-of-nat)
       done
    next
     fix s assume s ∈ ?M then show emeasure (N s) ?M = ennreal (p-H * p-f)
       by (auto simp add: emeasure-measure-pmf-finite sum.reindex H-eq2)
    next
     show AE ω in T t. ¬ ((Mix ‘ L · ?L) ⊓ (HLD (Mix ‘ H) ⊓ nxt ?U)) ω for t
       using L
       apply (simp add: AE-T-iff[of - t])
       apply (subst AE-T-iff; simp)
       apply (auto simp: HLD-iff suntil-Stream)
       done
```

```
    qed (insert L, auto simp: p-H-p-f-less-1 E-Mix)
    then show ?P (Init i) ?U = p-f * (1 - p-H) * p-j * card L / (1 - p-H *
p-f)
      by (subst emeasure-Init-eq-Mix)
        (auto simp: AE-End suntil-Stream divide-ennreal mult-le-one p-f)
  qed
  finally have *: P(ω in T Start. ?V ω) =
      (p-f * (1 - p-H) * p-j * (card L) / (1 - p-H * p-f)) * (∑ i∈I. p-i i) +
      (∑ i∈I ∩ L. p-i i) * (1 - p-H)
    using sum-nonneg [of I ∩ L p-i]  sum-nonneg [of I p-i]
   by (simp add: mult-ac measure-def sum-distrib-right[symmetric] sum-distrib-left[symmetric]
                 sum-divide-distrib[symmetric] IJ ennreal-mult[symmetric]
                 mult-le-one ennreal-plus[symmetric]
           del: ennreal-plus)
  show ?thesis
    unfolding cond-prob-def Pr-hit-C *
    using *
     using p-f p-H p-j-pos p-H-p-f-less-1 by (simp add: divide-simps) (simp add:
field-simps)
qed

lemma Pr-visit-eq-before-C:
  P(ω in P. ∃ j∈H. visit {j} J ω ∧ before-C {j} ω | hit-C ω ) = 1 - (p-H - p-j)
* p-f
proof -
  let ?V = λj. visit {j} J aand before-C {j} and ?H = hit-C
  let ?J = H
  have P(ω in P. (∃ j∈?J. ?V j ω) ∧ ?H ω) = (∑ j∈?J. P(ω in P. (?V j aand
?H) ω))
  proof (rule T.prob-sum)
    show AE ω in P. (∀ j∈?J. (?V j aand ?H) ω ⟶ ((∃ j∈?J. ?V j ω) ∧ ?H ω))
∧
      (((∃ j∈?J. ?V j ω) ∧ ?H ω) ⟶ (∃!j. j∈?J ∧ (?V j aand ?H) ω))
      by (auto intro!: AE-I2 dest: visit-unique1)
  qed auto
  then have P(ω in P. (∃ j∈?J. ?V j ω) | ?H ω) = (∑ j∈?J. P(ω in P. ?V j ω
| ?H ω))
    by (simp add: cond-prob-def sum-divide-distrib)
  also have ... = p-j * p-f + (1 - p-H * p-f)
    by (simp add: Pr-visit-before-C sum-distrib-right[symmetric] sum.distrib)
  finally show ?thesis
    by (simp add: field-simps)
qed

lemma probably-innocent:
  assumes approx: 1 / (2 * (p-H - p-j)) ≤ p-f and p-H ≠ p-j
  shows P(ω in P. ∃ j∈H. visit {j} J ω ∧ before-C {j} ω | hit-C ω ) ≤ 1 / 2
  unfolding Pr-visit-eq-before-C
proof -
```

have [*simp*]: $\bigwedge n$ :: *nat*. $1 \leq$ *real* $n \longleftrightarrow 1 \leq n$ **by** *auto*
have $0 \leq$ *p-j* **unfolding** *p-j-def* **by** *auto*
then have $1 *$ *p-j* $\leq$ *p-H*
unfolding *H-eq2*[*symmetric*] **using** *C-smaller*
by (*intro mult-mono*) (*auto simp*: *Suc-le-eq card-Diff-subset not-le*)
with ‹*p-H* $\neq$ *p-j*› **have** *p-j* $<$ *p-H* **by** *auto*
with *approx* **show** $1 - (p\text{-}H - p\text{-}j) * p\text{-}f \leq 1 / 2$
by (*auto simp add*: *field-simps divide-le-eq split*: *if-split-asm*)
qed

lemma *Pr-before-C*:
assumes *L*: $L \subseteq H$
shows $\mathcal{P}(\omega$ *in* $\mathfrak{P}$. *before-C L* ω | *hit-C* ω) =
card L $*$ *p-j* $*$ *p-f* + $(\sum l{\in}L.$ *p-i l*) $*$ $(1 -$ *p-H* $*$ *p-f*)
proof −
have $\mathcal{P}(\omega$ *in* $\mathfrak{P}$. *before-C L* ω | *hit-C* ω) =
$\mathcal{P}(\omega$ *in* $\mathfrak{P}$. *visit H J* $\omega \wedge$ *before-C L* ω | *hit-C* ω)
using *AE-visit* **by** (*auto intro*!: *T.cond-prob-eq-AE*)
also have ... = *card L* $*$ *p-j* $*$ *p-f* + $(\sum i{\in}L.$ *p-i i*) $*$ $(1 -$ *p-H* $*$ *p-f*)
using *L* **by** (*subst Pr-visit-before-C*[*OF L order-refl*]) (*auto simp*: *Int-absorb1*)
finally show *?thesis* **.**
qed

lemma *P-visit*:
assumes *I*: $I \subseteq H$
shows $\mathcal{P}(\omega$ *in* $\mathfrak{P}$. *visit I J* ω | *hit-C* ω) = $(\sum i{\in}I.$ *p-i i*)
proof −
have $\mathcal{P}(\omega$ *in* $\mathfrak{P}$. *visit I J* ω | *hit-C* ω) =
$\mathcal{P}(\omega$ *in* $\mathfrak{P}$. *visit I J* $\omega \wedge$ *before-C H* ω | *hit-C* ω)
proof (*rule T.cond-prob-eq-AE*)
show *AE x in* $\mathfrak{P}$. *hit-C x* $\longrightarrow$
visit I J x = (*visit I J x* $\wedge$ *before-C H x*)
using *AE-T-enabled* **by** *eventually-elim* (*auto intro*: *hit-C-imp-before-C*)
qed *auto*
also have ... = *sum p-i I*
using *I* **by** (*subst Pr-visit-before-C*[*OF order-refl*]) (*auto simp*: *Int-absorb2 field-simps p-H-def p-j-def*)
finally show *?thesis* **.**
qed

13.5 Probability space of hitting a collaborator

definition *hC* = *uniform-measure* $\mathfrak{P}$ $\{\omega{\in}$*space* $\mathfrak{P}$. *hit-C* $\omega\}$

lemma *emeasure-hit-C-not-0*: *emeasure* $\mathfrak{P}$ $\{\omega \in$ *space* $\mathfrak{P}$. *hit-C* $\omega\} \neq 0$
using *p-H p-H-p-f-less-1* **unfolding** *Pr-hit-C T.emeasure-eq-measure* **by** *auto*

lemma *measurable-hC*[*measurable* (*raw*)]:
$A \in$ *sets S* $\Longrightarrow A \in$ *sets hC*

$f \in$ *measurable M S* $\Longrightarrow f \in$ *measurable M hC*
$g \in$ *measurable S M* $\Longrightarrow g \in$ *measurable hC M*
$A \cap$ *space S* $\in$ *sets S* $\Longrightarrow A \cap$ *space hC* $\in$ *sets S*
unfolding *hC-def uniform-measure-def*
by *simp-all*

lemma *vimage-Int-space-C*[*simp*]:
$f -` \{x\} \cap$ *space hC* $= \{\omega \in$ *space S*. $f\ \omega = x\}$
by (*auto simp*: *hC-def*)

sublocale *hC*: *information-space hC 2*
proof −
interpret *hC*: *prob-space hC*
unfolding *hC-def*
using *emeasure-hit-C-not-0*
by (*intro prob-space-uniform-measure*) *auto*
show *information-space hC 2*
by *standard simp*
qed

abbreviation
mutual-information-Pow-CP (‹$\mathcal{I}'$(- ; -')›) **where**
$\mathcal{I}(X\ ;\ Y) \equiv$ *hC.mutual-information 2* (*count-space* (*X*‘*space hC*)) (*count-space* (*Y*‘*space hC*)) *X Y*

lemma *simple-functionI*:
assumes *finite* (*range f*)
assumes [*measurable*]: $\bigwedge x.\ \{\omega \in$ *space S*. $f\ \omega = x\} \in$ *sets S*
shows *simple-function hC f*
using *assms* **unfolding** *simple-function-def hC-def*
by (*simp add*: *vimage-def space-stream-space*)

13.6 Estimate the information to the collaborators

lemma *measure-hC*[*simp*]:
assumes *A*[*measurable*]: $A \in$ *sets S*
shows *measure hC A* $= \mathcal{P}(\omega$ *in* $\mathfrak{P}.\ \omega \in A \mid$ *hit-C* $\omega\ )$
unfolding *hC-def cond-prob-def*
using *emeasure-hit-C-not-0 A*
by (*subst measure-uniform-measure*) (*simp-all add*: *T.emeasure-eq-measure Int-def conj-ac*)

13.6.1 Setup random variables for mutual information

definition *first-J* ω = (*THE i*. *visit* $\{i\}$ *J* ω)

lemma *first-J-eq*:
visit $\{i\}$ *J* $\omega \Longrightarrow$ *first-J* $\omega = i$
unfolding *first-J-def* **by** (*intro the-equality*) (*auto dest*: *visit-unique1*)

lemma *AE-first-J*:
AE ω *in* $\mathfrak{P}$. *visit* $\{i\}$ J $\omega \longleftrightarrow$ *first-J* $\omega = i$
using *AE-visit*
proof *eventually-elim*
fix ω **assume** *visit* H J ω
then obtain j **where** *visit* $\{j\}$ J ω $j \in H$
by (*auto simp*: *visit-def HLD-iff*)
then show *visit* $\{i\}$ J $\omega \longleftrightarrow$ *first-J* $\omega = i$
by (*auto dest*: *visit-unique1 first-J-eq*)
qed

lemma *measurbale-first-J*[*measurable*]: *first-J* $\in$ *measurable S* (*count-space UNIV*)
unfolding *first-J-def*[*abs-def*]
by (*intro measurable-THE*[**where** *I*=*H*])
(*auto dest*: *visit-imp-in-H visit-unique1 intro*: *countable-finite*)

definition *last-H* ω = (*THE i*. *before-C* $\{i\}$ ω)

lemma *measurbale-last-H*[*measurable*]: *last-H* $\in$ *measurable S* (*count-space UNIV*)
unfolding *last-H-def*[*abs-def*]
by (*intro measurable-THE*[**where** *I*=*H*])
(*auto dest*: *before-C-single before-C-unique intro*: *countable-finite*)

lemma *last-H-eq*:
before-C $\{i\}$ $\omega \Longrightarrow$ *last-H* $\omega = i$
unfolding *last-H-def* **by** (*intro the-equality*) (*auto dest*: *before-C-unique*)

lemma *last-H*:
assumes *enabled Start* ω *hit-C* ω
shows *before-C* $\{$*last-H* $\omega\}$ ω *last-H* $\omega \in H$
by (*metis before-C-single hit-C-imp-before-C last-H-eq Int-iff assms*)+

lemma *AE-last-H*:
AE ω *in* $\mathfrak{P}$. *hit-C* $\omega \longrightarrow$ *before-C* $\{i\}$ $\omega \longleftrightarrow$ *last-H* $\omega = i$
using *AE-T-enabled*
proof *eventually-elim*
fix ω **assume** *enabled Start* ω **then show** *hit-C* $\omega \longrightarrow$ *before-C* $\{i\}$ ω = (*last-H* $\omega = i$)
by (*auto dest*: *last-H last-H-eq*)
qed

lemma *information-flow*:
defines $h \equiv$ *real* (*card H*)
assumes *init-uniform*: $\bigwedge i.\ i \in H \Longrightarrow$ *p-i i* = *1* / *h*
shows $\mathcal{I}$(*first-J* ; *last-H*) $\leq$ (*1* − (*h* − *1*) ∗ *p-j* ∗ *p-f*) ∗ *log 2 h*
proof −
let *?il* = $\lambda i\ l.\ \mathcal{P}(\omega$ *in* $\mathfrak{P}$. *visit* $\{i\}$ J $\omega \wedge$ *before-C* $\{l\}$ ω | *hit-C* ω)
let *?i* = $\lambda i.\ \mathcal{P}(\omega$ *in* $\mathfrak{P}$. *visit* $\{i\}$ J ω | *hit-C* ω)
let *?l* = $\lambda l.\ \mathcal{P}(\omega$ *in* $\mathfrak{P}$. *before-C* $\{l\}$ ω | *hit-C* ω)

from *init-uniform* **have** *init-H*: $\bigwedge i.\ i \in H \Longrightarrow$ *p-i i* = *p-j* / *p-H*
by (*simp add*: *p-j-def p-H-def h-def*)

from *h-def* **have** *1/h* = *p-j*/*p-H h* = *p-H* / *p-j p-H* = *h* ∗ *p-j*
by (*auto simp*: *p-H-def p-j-def field-simps*)
from *C-smaller* **have** *h-pos*: *0* < *h*
by (*auto simp add*: *card-gt-0-iff h-def*)

let *?s* = (*h* − *1*) ∗ *p-j*
let *?f* = *?s* ∗ *p-f*

from *psubset-card-mono*[*OF - C-smaller*]
have *1* ≤ *card J* − *card C*
by (*simp del*: *C-le-J*)
then have *1* ≤ *h*
using *C-smaller*
by (*simp add*: *h-def card-Diff-subset card-mono field-simps del*: *C-le-J*)

have *log-le-0*: *?f* ∗ *log 2* (*p-H* ∗ *p-f*) ≤ *?f* ∗ *log 2 1*
using *p-H-p-f-less-1 p-H-p-f-pos p-j-pos p-f* ‹*1* ≤ *h*›
by (*intro mult-left-mono log-mono mult-nonneg-nonneg*) *auto*

have (*h* − *1*) ∗ *p-j* < *1*
using ‹*1* ≤ *h*› *C-smaller*
by (*auto simp*: *h-def p-j-def divide-less-eq card-Diff-subset card-mono*)
then have *1*: (*h* − *1*) ∗ *p-j* ∗ *p-f* < *1* ∗ *1*
using *p-f* **by** (*intro mult-strict-mono*) *auto*

{ fix *ω* **have** *first-J ω* ∈ *H* ∨ *first-J ω* = (*THE x. False*)
apply (*cases* ∀ *i*. ¬ *visit* {*i*} *J ω*)
apply (*simp add*: *first-J-def*)
apply (*auto dest*: *visit-imp-in-H first-J-eq*)
done **}**
then have *range-fj*: *range first-J* ⊆ *H* ∪ {*THE x. False*}
by *auto*

have *sf-fj*: *simple-function hC first-J*
by (*rule simple-functionI*) (*auto intro*: *finite-subset*[*OF range-fj*])

have *sd-fj*: *simple-distributed hC first-J ?i*
apply (*rule hC.simple-distributedI*[*OF sf-fj*])
apply (*auto intro*!: *T.cond-prob-eq-AE*)
apply (*auto simp*: *space-stream-space*)
using *AE-first-J*
apply *eventually-elim*
apply *auto*
done

```
{ fix ω have last-H ω ∈ H ∨ last-H ω = (THE x. False)
    apply (cases ∀ i. ¬ before-C {i} ω)
    apply (simp add: last-H-def)
    apply (auto dest: before-C-imp-in-H last-H-eq)
    done }
then have range-lnc: range last-H ⊆ H ∪ {THE x. False}
  by auto

have sf-lnc: simple-function hC last-H
  by (rule simple-functionI) (auto intro: finite-subset[OF range-lnc])

have sd-lnc: simple-distributed hC last-H ?l
  apply (rule hC.simple-distributedI[OF sf-lnc])
  apply (auto intro!: T.cond-prob-eq-AE)
  apply (auto simp: space-stream-space)
  using AE-last-H
  apply eventually-elim
  apply auto
  done

have sd-fj-lnc: simple-distributed hC (λω. (first-J ω, last-H ω)) (λ(i, l). ?il i l)
  apply (rule hC.simple-distributedI)
  apply (rule simple-function-Pair[OF sf-fj sf-lnc])
  apply (auto intro!: T.cond-prob-eq-AE)
  apply (auto simp: space-stream-space)
  using AE-last-H AE-first-J
  apply eventually-elim
  apply auto
  done

define c where c = (SOME j. j ∈ C)
have c: c ∈ C
 using C-non-empty unfolding ex-in-conv[symmetric] c-def by (rule someI-ex)

let ?inner = λi. ∑ l∈H. ?il i l * log 2 (?il i l / (?i i * ?l l))
{ fix i assume i: i ∈ H
  with h-pos have card-idx: real-of-nat (card (H - {i})) = p-H / p-j - 1
    by (auto simp add: p-j-def p-H-def h-def)

  have neq0: p-j ≠ 0 p-H ≠ 0
    unfolding p-j-def p-H-def
    using C-smaller i by auto

  from i have ?inner i =
    (∑ l∈H - {i}. ?il i l * log 2 (?il i l / (?i i * ?l l))) +
    ?il i i * log 2 (?il i i / (?i i * ?l i))
    by (simp add: sum-diff)
  also have ... =
    (∑ l∈H - {i}. p-j/p-H * p-j * p-f * log 2 (p-j * p-f / (p-j * p-f + p-j/p-H
```

```
* (1 − p-H * p-f)))) +
      p-j/p-H * (p-j * p-f + (1 − p-H * p-f)) * log 2 ((p-j * p-f + (1 − p-H *
p-f)) / (p-j * p-f + p-j/p-H * (1 − p-H * p-f)))
      using i p-f p-j-pos p-H
      apply (simp add: Pr-visit-before-C P-visit init-H Pr-before-C
            del: sum-constant)
      apply (simp add: divide-simps distrib-left)
      apply (intro arg-cong2[where f=(*)] refl arg-cong2[where f=log])
      apply (auto simp: field-simps)
      done
   also have ... = (?f * log 2 (h * p-j * p-f) + (1 − ?f) * log 2 ((1 − ?f) * h))
/ h
      using neq0 p-f by (simp add: card-idx field-simps ‹p-H = h * p-j›)
    finally have ?inner i = (?f * log 2 (h * p-j * p-f) + (1 − ?f) * log 2 ((1 −
?f) * h)) / h . }
  then have (∑ i∈H. ?inner i) = ?f * log 2 (h * p-j * p-f) + (1 − ?f) * log 2
((1 − ?f) * h)
    using h-pos by (simp add: h-def[symmetric])
  also have ... = ?f * log 2 (p-H * p-f) + (1 − ?f) * log 2 ((1 − ?f) * h)
    by (simp add: ‹h = p-H / p-j›)
  also have ... ≤ (1 − ?f) * log 2 ((1 − ?f) * h)
    using log-le-0 by simp
  also have ... ≤ (1 − ?f) * log 2 h
    using h-pos ‹1 ≤ h› 1 p-j-pos p-f
    by (intro mult-left-mono log-mono mult-pos-pos mult-nonneg-nonneg) auto
  finally have (∑ i∈H. ?inner i) ≤ (1 − ?f) * log 2 h .
  also have (∑ i∈H. ?inner i) =
      (∑ (i, l)∈(first-J‘space S) × (last-H‘space S). ?il i l * log 2 (?il i l / (?i i *
?l l)))
    unfolding sum.cartesian-product
  proof (safe intro!: sum.mono-neutral-cong-left del: DiffE DiffI)
    show finite ((first-J ‘ space S) × (last-H ‘ space S))
      using sf-fj sf-lnc by (auto simp add: hC-def dest!: simple-functionD(1))
  next
    fix i assume i ∈ H
    then have visit {i} J (Init i ## Mix i ## sconst End)
      before-C {i} (Init i ## Mix c ## sconst End)
      by (auto simp: before-C-def visit-def suntil-Stream HLD-iff c)
    then show i ∈ first-J ‘ space S i ∈ last-H ‘ space S
        by (auto simp: space-stream-space image-iff eq-commute dest!: first-J-eq
last-H-eq)
  next
    fix i l assume (i, l) ∈ first-J ‘ space S × last-H ‘ space S − H × H
    then have H: i ∉ H ∨ l ∉ H
      by auto
    have P(ω in P. (visit {i} J ω ∧ before-C {l} ω) ∧ hit-C ω) = 0
    using H by (intro T.prob-eq-0-AE) (auto dest: visit-imp-in-H before-C-imp-in-H)
    then show ?il i l * log 2 (?il i l / (?i i * ?l l)) = 0
      by (simp add: cond-prob-def)
```

```
  qed
  also have ... = ℐ(first-J ; last-H)
    unfolding sum.cartesian-product
   apply (subst hC.mutual-information-simple-distributed[OF sd-fj sd-lnc sd-fj-lnc])
    apply (simp add: hC-def)
  proof (safe intro!: sum.mono-neutral-right imageI)
    show finite ((first-J ‘ space S) × (last-H ‘ space S))
      using sf-fj sf-lnc by (auto simp add: hC-def dest!: simple-functionD(1))
  next
    fix i l assume (first-J i, last-H l) ∉ (λx. (first-J x, last-H x)) ‘ space S
    moreover
    { fix i l assume i ∈ H l ∈ H
      then have visit {i} J (Init i ## Mix l ## Mix c ## sconst End)
        before-C {l} (Init i ## Mix l ## Mix c ## sconst End)
      using c C-smaller by (auto simp: before-C-def visit-def HLD-iff suntil-Stream)
      then have first-J (Init i ## Mix l ## Mix c ## sconst End) = i
        last-H (Init i ## Mix l ## Mix c ## sconst End) = l
        by (auto intro!: first-J-eq last-H-eq) }
    note this[of first-J i last-H l]
    ultimately have (first-J i, last-H l) ∉ H×H
      by (auto simp: space-stream-space image-iff eq-commute) metis
    then have 𝒫(ω in ℘. (visit {first-J i} J ω ∧ before-C {last-H l} ω) ∧ hit-C
ω) = 0
      by (intro T.prob-eq-0-AE) (auto dest: visit-imp-in-H before-C-imp-in-H)
    then show ?il (first-J i) (last-H l) *
      log 2 (?il (first-J i) (last-H l) / (?i (first-J i) * ?l (last-H l))) = 0
      by (simp add: cond-prob-def)
  qed
  finally show ?thesis by simp
qed

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
  by (auto simp: inj-on-def)
```

Countability of *zc-state* simplifies measurability of functions on *zc-state*.

```
instance zc-state :: countable
proof
  have countable ({start, ok, error} ∪ probe`UNIV)
    by auto
  also have {start, ok, error} ∪ probe`UNIV = UNIV
    using zc-state.nchotomy by auto
  finally show ∃f::zc-state ⇒ nat. inj f
    using inj-on-to-nat-on[of UNIV :: zc-state set] by auto
qed

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
  using p by auto

lemma q-bounds[simp]: 0 ≤ q q ≤ 1
  using q by auto

abbreviation states where
  states ≡ probe ` {.. N} ∪ {start, ok, error}

primrec τ :: zc-state ⇒ zc-state pmf where
  τ start     = map-pmf (λTrue ⇒ probe 0 | False ⇒ ok) (bernoulli-pmf q)
| τ (probe n) = map-pmf (λTrue ⇒ (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
  using r e by (cases s) auto
```

sublocale *MC-with-rewards* τ ϱ $\lambda s.\ 0$
proof qed (*simp-all add*: *pair-measure-countable*)

14.2 The allocation run is a rewarded DTMC

abbreviation $E\ s \equiv$ *set-pmf* $(\tau\ s)$

lemma *enabled-ok*: *enabled ok* $\omega \longleftrightarrow \omega =$ *sconst ok*
by (*simp add*: *enabled-iff-sconst*)

lemma *finite-E*[*intro*, *simp*]: *finite* $(E\ s)$
by (*cases s*) *auto*

lemma *E-closed*: $s \in$ *states* $\Longrightarrow E\ s \subseteq$ *states*
using *p q* **by** (*cases s*) (*auto split*: *bool.splits*)

lemma *enabled-error*: *enabled error* $\omega \longleftrightarrow \omega =$ *sconst error*
by (*simp add*: *enabled-iff-sconst*)

lemma *pos-neg-q-pn*: $0 < 1 - q * (1 - p \hat{} Suc\ N)$
proof −
have $p \hat{} Suc\ N \leq 1 \hat{} Suc\ N$
using *p* **by** (*intro power-mono*) *auto*
with *p q* **have** $q * (1 - p \hat{} Suc\ N) < 1 * 1$
by (*intro mult-strict-mono*) (*auto simp*: *field-simps simp del*: *power-Suc*)
then show *?thesis* **by** *simp*
qed

lemma *to-error*: **assumes** $n \leq N$ **shows** $(probe\ n, error) \in acc$
using $\langle n \leq N \rangle$
proof (*induction rule*: *inc-induct*)
case (*step* n') **with** *p* **show** *?case*
by (*intro rtrancl-trans*[*OF r-into-rtrancl step.IH*]) *auto*
qed (*insert p*, *auto*)

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* $\Rightarrow$ *0* | *error* $\Rightarrow$ *1* | *probe n* $\Rightarrow$ *p-probe n* | *start* $\Rightarrow$ *p-start*)
(**is** ... $=$ *?E s*)
using *s*
proof (*rule unique-les*)
have [*arith*]: $0 \leq p * (q * p \hat{} N)$
using *p q* **by** *simp*

```
  have p-eq: p-start = p-probe 0 * q
    ⋀n. n < N ⟹ p-probe n = p-probe (Suc n) * p + p-start * (1 - p)
    p-probe N = p + p-start * (1 - p)
    using p q
     by (auto simp: p-probe-def p-start-def power-Suc[symmetric] Suc-diff-Suc di-
vide-simps
           simp del: power-Suc)
       (auto simp: field-simps)
  fix s assume s: s ∈ states - {ok, error}
  then show ?E s = (∫ t. ?E t ∂τ s) + 0
    using p q by (auto intro: p-eq)
  show ∃ t∈{ok, error}. (s, t) ∈ acc
    using s q to-error by auto
  from s show P-err s = integral^L (measure-pmf (τ s)) P-err + 0
     unfolding P-err-def[abs-def] by (subst prob-T) (auto simp: ev-Stream simp
del: UNIV-bool)
next
  fix s assume s ∈ {ok, error} then show P-err s = ?E s
    by (auto intro!: T.prob-eq-0-AE T.prob-Collect-eq-1[THEN iffD2]
          simp: P-err-def AE-sconst ev-sconst HLD-iff ev-Stream T.prob-space
          simp del: space-T sets-T )
qed (insert p q, auto intro!: integrable-measure-pmf-finite split: if-split-asm)

lemma P-err-start: P-err start = (q * p ^ Suc N) / (1 - q * (1 - p ^ Suc N))
  by (simp add: P-err)
```

14.4 An allocation run terminates almost surely

```
lemma states-closed:
  assumes s ∈ states
  assumes (s, t) ∈ acc-on (- {error, ok})
  shows t ∈ states
  using assms(2,1) p q by induction (auto split: if-split-asm)

lemma finite-reached:
  assumes s: s ∈ states shows finite (acc-on (- {error, ok}) `` {s})
  using states-closed[OF s]
  by (rule-tac finite-subset[of - states]) auto

lemma AE-reaches-error-or-ok:
  assumes s: s ∈ states
  shows AE ω in T s. ev (HLD {error, ok}) ω
proof (rule AE-T-ev-HLD)
  { fix t assume t: (s, t) ∈ acc-on (- {error, ok})
    with states-closed[OF s t] to-error p q show ∃ t'∈{error, ok}. (t, t') ∈ acc
      by auto }
qed (rule finite-reached[OF s])
```

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*: $s \neq \textit{error} \Longrightarrow s \neq \textit{ok} \Longrightarrow R\ s = (\int^+ t.\ \textit{ennreal}\ (\varrho\ s\ t) + R\ t\ \partial\tau\ s)$
  unfolding *R-def* **using** *T.emeasure-space-1*
  by (*subst nn-integral-T*)
    (*auto simp del*: *τ.simps ϱ.simps simp add*: *AE-measure-pmf-iff nn-integral-add intro*!: *nn-integral-cong-AE*)

lemma *R-finite*:
  assumes *s*: $s \in \textit{states}$
  shows $R\ s \neq \infty$
  unfolding *R-def*
proof (*rule nn-integral-reward-until-finite*)
  { fix *t* **assume** $(s,\ t) \in \textit{acc}$ **from** *this s p q* **have** $t \in \textit{states}$
    by *induction* (*auto split*: *if-split-asm*) **}**
  then have $\textit{acc}\ ``\ \{s\} \subseteq \textit{states}$
    by *auto*
  then show *finite* $(\textit{acc}\ ``\ \{s\})$
    by (*auto dest*: *finite-subset*)
qed (*auto simp*: *AE-reaches-error-or-ok*[*OF s*])

lemma *R-less-top*: $s \in \textit{states} \Longrightarrow R\ s < \textit{top}$
  using *R-finite*[*of s*] **by** (*subst less-top*[*symmetric*]) *simp*

lemma *R′-iter*: **assumes** *s*: $s \in \textit{states}\ s \neq \textit{error}\ s \neq \textit{ok}$ **shows** $R'\ s = (\int t.\ \varrho\ s\ t + R'\ t\ \partial\tau\ s)$
  unfolding *R′-def R-iter*[*OF s*(*2*,*3*)]
proof (*rule enn2real-nn-integral-eq-integral*)
  have $t \in E\ s \Longrightarrow R\ t < \textit{top}$ **for** *t*
    using ‹*s*∈*states*› *E-closed*[*of s*] **by** (*intro R-less-top*) *auto*
  then show $\textit{AE}\ t\ \textit{in}\ \tau.\ \textit{ennreal}\ (\varrho\ s\ t) + R\ t = \textit{ennreal}\ (\varrho\ s\ t + \textit{enn2real}\ (R\ t))$
    by (*auto simp*: *AE-measure-pmf-iff intro*!: *ennreal-enn2real*[*symmetric*])
qed *auto*

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 −
  have *ok-error*: $R'\ \textit{ok} = 0 \wedge R'\ \textit{error} = 0$
    unfolding *R′-def R-def* **by** (*subst* (*1 2*) *reward-until-unfold*[*abs-def*]) *simp*

  then have *R-start*: $R'\ \textit{start} = q * (r + R'\ (\textit{probe}\ 0)) + (1 - q) * (r * (N +$

```
1))
    using q r by (subst R'-iter) (simp-all add: field-simps)

  have R-probe: ⋀n. n < N ⟹ R' (probe n) = p * R' (probe (Suc n)) + p * r +
(1 - p) * R' start
    using p r by (subst R'-iter) (simp-all add: field-simps distrib-right)

  have R-N: R' (probe N) = p * e + (1 - p) * R' start
    using p e ok-error by (subst R'-iter) (auto simp: mult.commute )

  { fix n
    assume n ≤ N
    then have R' (probe (N - n)) =
      p ^ Suc n * e + (1 - p^n) * r * p / (1 - p) + (1 - p^Suc n) * R' start
    proof (induct n)
      case 0 with R-N show ?case by simp
    next
      case (Suc n)
      moreover then have Suc (N - Suc n) = N - n by simp
      ultimately show ?case
        using R-probe[of N - Suc n] p by (simp-all add: field-simps Suc)
    qed }
  from this[of N]
  have [simp]: R' (probe 0) = p ^ Suc N * e + (1 - p^N) * r * p / (1 - p) +
(1 - p^Suc N) * R' start
    by simp
  have R' start - q * (1 - p^Suc N) * R' start =
    q * (r + p^Suc N * e + (1 - p^N) * r * p / (1 - p)) + (1 - q) * (r * (N
+ 1))
    by (subst R-start) (simp-all add: field-simps)
  then have R' start = (q * (r + p^Suc N * e + (1 - p^N) * r * p / (1 - p))
+ (1 - q) * (r * Suc N)) /
    (1 - q * (1 - p^Suc N))
    using pos-neg-q-pn by (simp-all add: field-simps)
  then show ?thesis
    by (simp add: field-simps)
qed

end

interpretation ZC: Zeroconf-Analysis 2 16 / 65024 :: real 0.01 0.002 3600
  by standard auto

lemma ZC.P-err start ≤ 1 / 10^12
  unfolding ZC.P-err-start by (simp add: power-divide power-one-over[symmetric])

lemma ZC.R' start ≤ 0.007
  unfolding ZC.cost-from-start by (simp add: power-divide power-one-over[symmetric])
```

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* $(\lambda(x,f).\ f(i := x))\ (M \times (\Pi_E\ i{\in}I.\ A\ i))$
unfolding *PiE-def*
proof (*safe intro*!: *inj-onI ext*)
fix $f\ g :: 'a \Rightarrow 'b$ **and** $x\ y :: 'b$
assume $*$: $f(i := x) = g(i := y)$ $f \in$ *extensional I* $g \in$ *extensional I*
then show $x = y$ **by** (*auto simp*: *fun-eq-iff split*: *if-split-asm*)
fix i' **from** $* \langle i \notin I\rangle$ **show** $f\ i' = g\ i'$
by (*cases* $i' = i$) (*auto simp*: *fun-eq-iff extensional-def split*: *if-split-asm*)
qed

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 $(\sum x{\in}Pi_E\ I\ S.\ \prod i{\in}I.\ f\ (x\ i)\ i) = (\prod i{\in}I.\ \sum s{\in}S\ i.\ f\ s\ i)$
using *assms* **proof** (*induct I*)
case *empty* **then show** *?case* **by** *simp*
next
case (*insert i I*)
have $*$: $Pi_E\ (insert\ i\ I)\ S = (\lambda(x,\ f).\ f(i := x))\ `\ (S\ i \times Pi_E\ I\ S)$
by (*auto simp*: *PiE-def intro*!: *image-eqI ext dest*: *extensional-arb*)
have $(\sum x{\in}Pi_E\ (insert\ i\ I)\ S.\ \prod i{\in}insert\ i\ I.\ f\ (x\ i)\ i) =$
sum $((\lambda x.\ \prod i{\in}insert\ i\ I.\ f\ (x\ i)\ i) \circ ((\lambda(x,\ f).\ f(i := x))))\ (S\ i \times Pi_E\ I\ S)$
unfolding $*$ **using** *insert* **by** (*intro sum.reindex*) (*auto intro*!: *inj-on-upd-PiE*)
also have $\ldots = (\sum (a,\ x){\in}(S\ i \times Pi_E\ I\ S).\ f\ a\ i * (\prod i{\in}I.\ f\ (x\ i)\ i))$
using *insert* **by** (*force intro*!: *sum.cong prod.cong arg-cong2*[**where** $f{=}(*)$])
also have $\ldots = (\sum a{\in}S\ i.\ f\ a\ i * (\sum x{\in}Pi_E\ I\ S.\ \prod i{\in}I.\ f\ (x\ i)\ i))$
by (*simp add*: *sum.cartesian-product sum-distrib-left*)
finally show *?case*
using *insert* **by** (*simp add*: *sum-distrib-right*)
qed

15.1 Definition of the Gossip-Broadcast

datatype *state* = *listening* | *sending* | *sleeping*

type-synonym *sys-state* = $(nat \times nat) \Rightarrow state$

lemma *state-UNIV*: $UNIV = \{listening,\ sending,\ sleeping\}$
by (*auto intro*: *state.exhaust*)

locale *gossip-broadcast* =
  fixes *size* :: *nat* **and** *p* :: *real*
  assumes *size*: *0* < *size*
  assumes *p*: *0* < *p p* < *1*
begin

interpretation *pmf-as-function* **.**

definition *states* :: *sys-state set* **where**
  states = ({..< *size*} × {..< *size*}) →$_E$ {*listening*, *sending*, *sleeping*}

definition *start* :: *sys-state* **where**
  start = (λ*x*∈{..< *size*}×{..< *size*}. *listening*)((*0*, *0*) := *sending*)

definition *neighbour-sending* **where**
  neighbour-sending s = (λ(*x*,*y*).
    (*x* > *0* ∧ *s* (*x* − *1*, *y*) = *sending*) ∨
    (*x* < *size* ∧ *s* (*x* + *1*, *y*) = *sending*) ∨
    (*y* > *0* ∧ *s* (*x*, *y* − *1*) = *sending*) ∨
    (*y* < *size* ∧ *s* (*x*, *y* + *1*) = *sending*))

definition *node-trans* :: *sys-state* ⇒ (*nat* × *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*
  by (*simp add*: *node-trans-def split*: *state.split*)

lemma *node-trans-nonneg*[*simp*]: *0* ≤ *node-trans s x i j*
  using *p* **by** (*auto simp*: *node-trans-def split*: *state.split*)

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
  let *?f* = λ*s s′*. *if s′* ∈ *states then* (∏ *x*∈{..< *size*}×{..< *size*}. *node-trans s x* (*s x*) (*s′ x*)) *else 0*
  fix *s* **show** ∀ *t*. *0* ≤ *?f s t*
    using *p* **by** (*auto intro*!: *prod-nonneg simp*: *node-trans-def split*: *state.split*)
  show (∫$^+$*t*. *?f s t* ∂*count-space UNIV*) = *1*
    apply (*subst nn-integral-count-space′*[*of states*])
    apply (*simp-all add*: *prod-nonneg*)
  proof −

show ($\sum x{\in}states.\ \prod xa{\in}\{..<size\} \times \{..<size\}.$ *node-trans s xa* (*s xa*) (*x xa*)) = *1*
unfolding *states-def* **by** (*subst sum-folded-product*) *simp-all*
show *finite states*
by (*auto simp*: *states-def intro*!: *finite-PiE*)
qed
qed

end

15.2 The Gossip-Broadcast forms a DTMC

sublocale *gossip-broadcast* ⊆ *MC-syntax proto-trans* **.**

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*: $s \in S$ $\bigwedge s\ D.\ s \in S1 \Longrightarrow D \in K\ s \Longrightarrow (\sum t{\in}S.\ pmf\ D\ t * x\ t) \leq x\ s$
assumes *2*: $\bigwedge s.\ s \in S1 \Longrightarrow x\ s \neq 0 \Longrightarrow (\exists\, t{\in}S2.\ (s, t) \in (SIGMA\ s{:}S1.\ \bigcup D{\in}K\ s.\ set\text{-}pmf\ D)^*)$
assumes *3*: $\bigwedge s.\ s \in S - S1 - S2 \Longrightarrow x\ s = 0$
assumes *4*: $\bigwedge s.\ s \in S2 \Longrightarrow x\ s = 1$
shows *enn2real* (*p s*) ≤ *x s*
proof (*rule p-ub*[*OF 1 - 4*])
fix *s* **assume** $s \in S$ *p s* = *0* **with** *2*[*of s*] *p-pos*[*of s*] *p-S2*[*of s*] *3*[*of s*] **show** *x s* = *0*
by (*cases x s* = *0*) *auto*
qed

lemma *n-lb′*:
fixes *x*
assumes *wf R*
assumes *1*: $s \in S$ $\bigwedge s\ D.\ s \in S1 \Longrightarrow D \in K\ s \Longrightarrow x\ s \leq (\sum t{\in}S.\ pmf\ D\ t * x\ t)$
assumes *2*: $\bigwedge s\ D.\ s \in S1 \Longrightarrow D \in K\ s \Longrightarrow x\ s \neq 0 \Longrightarrow \exists\, t{\in}D.\ ((t, s) \in R \wedge t \in S1 \wedge x\ t \neq 0) \vee t \in S2$
assumes *3*: $\bigwedge s.\ s \in S - S1 - S2 \Longrightarrow x\ s = 0$

assumes *4*: $\bigwedge s.\ s \in S2 \Longrightarrow x\ s = 1$
shows $x\ s \leq$ *enn2real* $(n\ s)$
proof (*rule n-lb*[*OF 1 - 4*])
fix s **assume** $*$: $s \in S\ n\ s = 0$
show $x\ s = 0$
proof (*rule ccontr*)
assume $x\ s \neq 0$
with $*$ *n-S2*[*of s*] *n-nS12*[*of s*] *3*[*of s*] **have** $s \in S1$
by (*metis DiffI zero-neq-one*)
have $0 < n\ s$
by (*intro n-pos*[*of* $\lambda s.\ x\ s \neq 0$, *OF* ‹$x\ s \neq 0$› ‹$s \in S1$› ‹*wf R*›])
(*metis zero-less-one n-S2 2*)
with ‹$n\ s = 0$› **show** *False* **by** *auto*
qed
qed

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*)
by (*induct xs*) *simp-all*

lemma *lookup-in-set*:
distinct (*map fst xs*) ⟹ *x* ∈ *set xs* ⟹ *lookup d xs* (*fst x*) = *snd x*
unfolding *lookup-eq-map-of* **by** (*subst map-of-is-SomeI*[**where** *y*=*snd x*]) *simp-all*

lemma *lookup-not-in-set*:
x ∉ *fst* ‘ *set xs* ⟹ *lookup d xs x* = *d*
unfolding *lookup-eq-map-of*
by (*subst map-of-eq-None-iff*[*of xs x*, *THEN iffD2*]) *auto*

lemma *lookup-nonneg*:
(⋀*x v*. (*x*, *v*) ∈ *set xs* ⟹ 0 ≤ *v*) ⟹ (0::′*a*::*ordered-comm-monoid-add*) ≤ *lookup 0 xs x*
apply (*induction xs*)
apply *simp*
apply *force*
done

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* = (∑ *i*<*M*. *lookup 0 xs i* ∗ *y* !! *i*)
proof −
from ‹*distinct* (*map fst xs*)› **have** *distinct xs inj-on fst* (*set xs*)
by (*simp-all add*: *distinct-map*)
then have *sparse-mult xs y* = (∑ *x*∈*set xs*. *snd x* ∗ *y* !! *fst x*)
by (*auto intro*!: *sum.cong simp add*: *sparse-mult-def sum-list-distinct-conv-sum-set*)
also have ... = (∑ *x*∈*set xs*. *lookup 0 xs* (*fst x*) ∗ *y* !! *fst x*)
by (*intro sum.cong refl arg-cong2*[**where** *f*=(∗)]) (*simp add*: *lookup-in-set assms*)
also have ... = (∑ *x*∈*fst* ‘ *set xs*. *lookup 0 xs x* ∗ *y* !! *x*)
using ‹*inj-on fst* (*set xs*)› **by** (*simp add*: *sum.reindex*)
also have ... = (∑ *x*<*M*. *lookup 0 xs x* ∗ *y* !! *x*)
using *assms*(*1*)
by (*intro sum.mono-neutral-cong-left*)
(*auto simp*: *list-all-iff lookup-eq-map-of map-of-eq-None-iff*[*THEN iffD2*])
finally show *?thesis* **.**
qed

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*) = (∑ *i*<*M*. *lookup 0 xs i*)
proof −
from ‹*distinct* (*map fst xs*)› **have** *distinct xs inj-on fst* (*set xs*)
by (*simp-all add*: *distinct-map*)
then have *sum-list* (*map snd xs*) = (∑ *x*∈*set xs*. *snd x*)
by (*auto intro*!: *sum.cong simp add*: *sparse-mult-def sum-list-distinct-conv-sum-set*)
also have ... = (∑ *x*∈*set xs*. *lookup 0 xs* (*fst x*))
by (*intro sum.cong refl arg-cong2*[**where** *f*=(∗)]) (*simp add*: *lookup-in-set assms*)
also have ... = (∑ *x*∈*fst* ‘ *set xs*. *lookup 0 xs x*)
using ‹*inj-on fst* (*set xs*)› **by** (*simp add*: *sum.reindex*)

also have ... = ($\sum x{<}M.$ *lookup 0 xs x*)
using *assms*(*1*)
by (*intro sum.mono-neutral-cong-left*)
(*auto simp*: *list-all-iff lookup-eq-map-of map-of-eq-None-iff*[*THEN iffD2*])
finally show *?thesis* **.**
qed

definition
valid-mdp-rp mdp $\longleftrightarrow$
$0 <$ *state-count mdp* $\wedge$
IArray.length (*distrs mdp*) = *state-count mdp* $\wedge$
IArray.length (*states1 mdp*) = *state-count mdp* $\wedge$
IArray.length (*states2 mdp*) = *state-count mdp* $\wedge$
($\forall i{<}$*state-count mdp*. $\neg$ (*states1 mdp* !! *i* $\wedge$ *states2 mdp* !! *i*) $\wedge$
list-all (λ*ds*. *distinct* (*map fst ds*) $\wedge$ *list-all* (λ(*n*, *x*). $0 \leq x \wedge n <$ *state-count mdp*) *ds* $\wedge$
sum-list (*map snd ds*) = *1*) (*distrs mdp* !! *i*) $\wedge$
$\neg$ *List.null* (*distrs mdp* !! *i*))

definition
valid-sub-cert mdp c ord check $\longleftrightarrow$
IArray.length (*witness c*) = *state-count mdp* $\wedge$
IArray.length (*solution c*) = *state-count mdp* $\wedge$
($\forall i{<}$*state-count mdp*.
if states2 mdp !! *i then solution c* !! *i* = *1*
else if states1 mdp !! *i then* $0 \leq$ *solution c* !! *i* $\wedge$
(*list-all* (λ*ds*. *ord* (*sparse-mult ds* (*solution c*)) (*solution c* !! *i*)) (*distrs mdp* !! *i*)) $\wedge$
($0 <$ *solution c* !! *i* $\longrightarrow$ *check* (*distrs mdp* !! *i*) (*witness c* !! *i*))
else solution c !! *i* = *0*)

definition
valid-pos-cert mdp c $\longleftrightarrow$
valid-sub-cert mdp c ($\leq$)
(λ*D* ((*j*, *a*), *n*). *j* < *state-count mdp* $\wedge$ *snd* (*witness c* !! *j*) < *n* $\wedge$ *0* < *solution c* !! *j* $\wedge$
a < *length D* $\wedge$ *lookup 0* (*D* ! *a*) *j* $\neq$ *0*)

definition
valid-neg-cert mdp c $\longleftrightarrow$
valid-sub-cert mdp c ($\geq$)
(λ*D* (*J*, *n*). *list-all2* (λ*j d*. *j* < *state-count mdp* $\wedge$ *snd* (*witness c* !! *j*) < *n* $\wedge$
lookup 0 d j $\neq$ *0* $\wedge$ *0* < *solution c* !! *j*) *J D*)

definition
valid-cert mdp c $\longleftrightarrow$ *valid-pos-cert mdp* (*pos-cert c*) $\wedge$ *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
using *assms* **by** (*auto simp*: *valid-mdp-rp-def*)

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 ≠ []*
using *assms* **by** (*auto simp*: *valid-mdp-rp-def list-all-iff elim!*: *allE*[*of - i*])

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)*
using *valid-mdp-rpD*(*2*,*5*)[*OF assms*] **by** (*intro sparse-mult-eq-sum-lookup*) (*auto simp*: *list-all-iff*)

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)*
using *assms*(*2*,*3*) *valid-mdp-rpD*(*1*)[*OF assms*(*1*,*3*)]
by (*auto simp add*: *valid-sub-cert-def list-all-iff*)

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
using *valid-sub-certD*(*5*)[*OF assms*(*1*) *assms*(*2*)[*unfolded valid-pos-cert-def*] *assms*(*3*,*4*)] *assms*(*5*−) **by** *auto*

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* (λ*j ds*. *j* < *state-count mdp* ∧ *snd* (*witness c* !! *j*) < *n* ∧ *lookup 0 ds j* ≠ *0* ∧ *0* < *solution c* !! *j*) *js* (*distrs mdp* !! *i*)
using *valid-sub-certD*(*5*)[*OF assms*(*1*) *assms*(*2*)[*unfolded valid-neg-cert-def*] *assms*(*3*)] *assms*(*4*−) **by** *auto*

context
fixes *mdp c*
assumes *rp*: *valid-mdp-rp mdp*
assumes *cert*: *valid-cert mdp c*
begin

interpretation *pmf-as-function* **.**

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**
λ*i*. *if i* < *state-count mdp then*
{ (λ*j*. *of-rat* (*lookup 0 D j*) :: *real*) | *D*. *D* ∈ *set* (*distrs mdp* !! *i*) }
else { *indicator* {*0*} }
proof (*auto split*: *if-split-asm simp del*: *IArray.sub-def*)
fix *n D* **assume** *n*: *n* < *state-count mdp* **and** *D*: *D* ∈ *set* (*distrs mdp* !! *n*)
from *valid-mdp-rpD*(*3*)[*OF rp this*] **show** *nn*: ⋀*i*. *0* ≤ *lookup 0 D i*
by (*auto simp add*: *lookup-eq-map-of split*: *option.split dest*: *map-of-SomeD*)
show (∫⁺ *x*. *ennreal* (*real-of-rat* (*lookup 0 D x*)) ∂*count-space UNIV*) = *1*
using *valid-mdp-rpD*(*2*,*3*,*4*,*5*)[*OF rp n D*]
apply (*subst nn-integral-count-space′*[*of* {..< *state-count mdp*}])
apply (*auto intro*: *nn lookup-not-in-set simp*: *of-rat-sum*[*symmetric*] *lookup-nonneg*)
apply (*subst sum-list-eq-sum-lookup*[*symmetric*])
apply (*auto simp*: *list-all-iff lookup-eq-map-of split*: *option.split*)
done
next
show (∫⁺ *x*. *ennreal* (*indicator* {*0*} *x*) ∂*count-space UNIV*) = *1*
by (*subst nn-integral-count-space′*[*of* {*0*}]) *auto*
qed

interpretation *MDP*: *Reachability-Problem K S S1 S2*
proof
show *S1* ∩ *S2* = {} *S1* ⊆ *S S2* ⊆ *S*
using *valid-mdp-rpD*(*1*)[*OF rp*] **by** *auto*
show *finite S S* ≠ {}
using ‹*valid-mdp-rp mdp*› **by** (*auto simp add*: *valid-mdp-rp-def*)
show ⋀*s*. *K s* ≠ {}
using *valid-mdp-rpD*(*6*)[*OF rp*] **by** *transfer simp*
show ⋀*s*. *finite* (*K s*)
by *transfer simp*

fix *s* **assume** *s* ∈ *S* **then show** (⋃ *D*∈*K s*. *set-pmf D*) ⊆ *S*

```
    using valid-mdp-rpD(2)[OF rp]
   by transfer (auto simp: lookup-eq-map-of split: option.splits dest!: map-of-SomeD)
qed

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 −
  have valid-pos-cert mdp (pos-cert c) valid-neg-cert mdp (neg-cert c)
    using ‹valid-cert mdp c› by (auto simp: valid-cert-def)
 note pos = this(1)[unfolded valid-pos-cert-def] and neg = this(2)[unfolded valid-neg-cert-def]

  let ?x = λs. real-of-rat (solution (pos-cert c) !! s)
  have enn2real (MDP.p i) ≤ ?x i
  proof (rule MDP.p-ub′)
    show i ∈ S using assms by simp
  next
    fix s D assume s ∈ S1 D ∈ K s
    then obtain j where j: j < length (distrs mdp !! s)
      ⋀i. i < state-count mdp ⟹ pmf D i = real-of-rat (lookup 0 (distrs mdp !! s
! j) i)
      by transfer (auto simp: in-set-conv-nth)
    with valid-sub-certD(4)[OF ‹valid-mdp-rp mdp› pos, of s distrs mdp !! s ! j] ‹s
∈ S1›
         valid-mdp-rp-sparse-mult[OF ‹valid-mdp-rp mdp›, of s distrs mdp !! s ! j
solution (pos-cert c)]
    show (∑ t∈S. pmf D t * ?x t) ≤ ?x s
      by (simp add: of-rat-mult[symmetric] of-rat-sum[symmetric] of-rat-less-eq j)
  next
    fix s a assume s ∈ S2 then show ?x s = 1
      using valid-sub-certD[OF ‹valid-mdp-rp mdp› pos] by simp
  next
    fix s define X where X = (SIGMA s:S1. ⋃ D∈K s. set-pmf D)
    assume s ∈ S1 ?x s ≠ 0
    with valid-sub-certD(3)[OF rp pos, of s]
    have 0 < ?x s
      by simp
    with ‹s∈S1› show ∃ t∈S2. (s, t) ∈ X*
   proof (induction n≡snd (witness (pos-cert c) !! s) arbitrary: s rule: less-induct)
      case (less s)
      obtain t a n where eq: witness (pos-cert c) !! s = ((t, a), n)
        by (metis prod.exhaust)
        from valid-pos-certD[OF rp ‹valid-pos-cert mdp (pos-cert c)› - - - this]
less.prems
      have ord: snd (witness (pos-cert c) !! t) < snd (witness (pos-cert c) !! s)
```

```
      and t: lookup 0 (distrs mdp !! s ! a) t ≠ 0 0 < ?x t t∈S a < length (distrs
mdp !! s)
      unfolding eq by auto
    with ‹s∈S1› have X: (s, t) ∈ X
      unfolding X-def
      by (transfer fixing: s t a c)
        (auto simp: X-def in-set-conv-nth
            intro!: exI[of - λj. real-of-rat (lookup 0 (distrs mdp !! s ! a) j)]
                exI[of - distrs mdp !! s ! a] exI[of - a])
    show ?case
    proof cases
      assume t ∈ S1
      with less.hyps[OF ord - ‹0 < ?x t›] X show ?thesis
        by auto
    next
      assume t ∉ S1
      with valid-sub-certD[OF ‹valid-mdp-rp mdp› pos, of t] ‹0 < ?x t› ‹t∈S›
      have t ∈ S2
        by auto
      with X show ?thesis
        by auto
    qed
  qed
 next
  fix s assume s ∈ S − S1 − S2 then show ?x s = 0
    using valid-sub-certD(1)[OF ‹valid-mdp-rp mdp› pos, of s] by simp
 qed
 then show ?max
  by (simp add: P-max-def)

 let ?x = λs. real-of-rat (solution (neg-cert c) !! s)
 have ?x i ≤ enn2real (MDP.n i)
 proof (rule MDP.n-lb′)
  show i ∈ S using assms by simp
 next
  fix s D assume s ∈ S1 D ∈ K s
  then obtain j where j: j < length (distrs mdp !! s)
    ⋀i. i < state-count mdp ⟹ pmf D i = real-of-rat (lookup 0 (distrs mdp !! s
! j) i)
    by transfer (auto simp: in-set-conv-nth)
  with valid-sub-certD(4)[OF ‹valid-mdp-rp mdp› neg, of s distrs mdp !! s ! j] ‹s
∈ S1›
      valid-mdp-rp-sparse-mult[OF ‹valid-mdp-rp mdp›, of s distrs mdp !! s ! j
solution (neg-cert c)]
  show ?x s ≤ (∑ t∈S. pmf D t * ?x t)
    by (simp add: of-rat-mult[symmetric] of-rat-sum[symmetric] of-rat-less-eq j)
 next
  fix s a assume s ∈ S2 then show ?x s = 1
    using valid-sub-certD[OF ‹valid-mdp-rp mdp› neg] by simp
```

```
  next
    show wf ((S × S ∩ {(s, t). snd (witness (neg-cert c) !! t) < snd (witness
(neg-cert c) !! s)})⁻¹) (is wf ?F)
      using MDP.S-finite
      by (intro finite-acyclic-wf-converse acyclicI-order[where f=λs. snd (witness
(neg-cert c) !! s)]) auto

    fix s D assume 2: s ∈ S1 D ∈ K s and ?x s ≠ 0
    then have 0 < ?x s
      using valid-sub-certD(3)[OF ‹valid-mdp-rp mdp› neg, of s] by auto

    from 2 obtain a where a: a < length (distrs mdp !! s)
      ⋀i. i < state-count mdp ⟹ pmf D i = real-of-rat (lookup 0 (distrs mdp !! s
! a) i)
      by transfer (auto simp: in-set-conv-nth)

    obtain js n where eq: witness (neg-cert c) !! s = (js, n)
      by (metis prod.exhaust)
    from valid-neg-certD[OF ‹valid-mdp-rp mdp› ‹valid-neg-cert mdp (neg-cert c)›
- - - eq] a ‹s ∈ S1› ‹0 < ?x s›
    have ∗: length js = length (distrs mdp !! s) js ! a ∈ S
      snd (witness (neg-cert c) !! (js ! a)) < snd (witness (neg-cert c) !! s)
      lookup 0 (distrs mdp !! s ! a) (js ! a) ≠ 0
      0 < ?x (js ! a)
      unfolding eq by (auto dest: list-all2-nthD2 list-all2-lengthD)
    with a ‹s ∈ S1› have js-a: js ! a ∈ D (js ! a, s) ∈ ?F
      by (auto simp: set-pmf-iff)

    show ∃ t∈D. (t, s) ∈ ?F ∧ t ∈ S1 ∧ ?x t ≠ 0 ∨ t ∈ S2
    proof cases
      assume js ! a ∈ S1 with js-a ‹0 < ?x (js ! a)› show ?thesis by auto
    next
      assume js ! a ∉ S1
      with ‹0 < ?x (js ! a)› ‹js!a ∈ S› valid-sub-certD[OF rp neg, of js ! a]
      have js ! a ∈ S2
        by (auto simp: less-le)
      with ‹js ! a ∈ D› show ?thesis
        by auto
    qed
  next
    fix s assume s ∈ S − S1 − S2 then show ?x s = 0
      using valid-sub-certD(1)[OF ‹valid-mdp-rp mdp› neg, of s] by simp
  qed
  then show ?min
    by (simp add: P-min-def)
qed

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.