Analyzing Discrete-Time Markov Chains with Countable State Space in Isabelle/HOL

Johannes Hölzl

September 23, 2013

Abstract

Discrete-time Markov chains are an important tool in probabilistic analysis of computer systems. For example they are used to describe the behavior of computer programs with probabilistic choice or the time-dependent distribution of input values. Current formalizations of Markov chains are restricted to a finite state space. We extend this to a countable state space, construct the stochastic process of a Markov chain given a matrix of transition probabilities, and prove the equivalence with the axiomatic definition as stochastic process. Based on this we introduce irreducible, recurrent, and aperiodic classes, generating functions and stationary distributions to analyze Markov chains.

This document is generated from the theory files. The main document can be found in [1].

Contents

1 Introduction

The examples from Fig. 1 (a) and (b) in [1] are formalized in Section 6 and Section 7.

1.1 Constant table

In the paper [1] some of the constants where renamed for presentation purposes.

Paper	**Isabelle/HOL theories**
$\mathbb{B}$	*bool*
$\mathbb{N}$	*nat*
$\mathbb{R}$	*real*
$\overline{\mathbb{R}}$	*ereal*
$\omega \circ_n \omega$	*comb-seq n ω ω*
$s{\cdot}\omega$	*nat-case s ω*
$\chi_A\ x$	*indicator A x*
$f'\ z \xrightarrow{z \to 1^-} y$	$((\lambda z.\ f'\ z) \dashrightarrow y)\ (\textit{at-left}\ 1)$
$X\ n \xrightarrow{n \to \infty} y$	$(\lambda n.\ X\ n) \dashrightarrow y$
count	*count-space*
point	*point-measure*
$\Pi_{i \in I} M\ i$	$Pi_M\ I\ (\lambda i.\ M\ i)$
trace	*path*
$\mathcal{P}_x$	*paths x*
$(x, y) \in \textit{accessible}$	*accessible x y*
essential	*essential-class*
G_g	*gf-G*
M_g	*gf-U'*
U_g	*gf-U*

1.2 List of Definitions and Theorems

- `locale` Discrete-Time-Markov-Chain – Section 3
- `locale` Discrete-Markov-Kernel – Section 2
- Definition of *trace* – *local.path* in Section 2.3
- Definition 1 (Trace space) $\mathcal{P}_x$ – *paths* in Section 2.4
- Lemma 2 (Splitting rules) in Section 2.5
- Lemma 3 (Chapman-Kolmogorov) in Section 2.6

- Theorem 4 (The trace space $\mathcal{P}_x$ is a Markov chain) in Section 4
- Definition 5 (Markov kernel from the stochastic process) in Section 3.1
- Theorem 6 (The Markov kernel K' has the distribution of X) in Section 3.3
- Definition 7 (Enabled states) in Section 2.1
- Theorem 8 in Section 2.1.1
- Definition 9 (Accessible states) in Section 5.1
- Theorem 10. in Section 5.3.1 (In the paper p is unfolded)
- Definition 11 (Communicating states) in Section 5.2
- Definition 12 (Essential class). in Section 5.7
- Definition 13 (Recurrent states) in Section 5.6
- Definition 14 (Hitting time) in Section 5.8
- Definition 15 (Average hitting time) is does not have it's own constant
- Definition 16 (Positive recurrent states) in Section 5.9
- Definition 17 (Unbound quantities) in Section 5.4 and Section sec:recurrent
- Definition 18 (Time-bounded reachability probabilities) in Section 5.3
- Lemma 19 (Unbound quantities as series of time-bounded probabilities) in Section 5.4.1, 5.4.2, and 5.8.1
- Definition 20 (Generating functions for G, M , U, and F) in Section 5.5 and 5.8
- Lemma 21 (Unbound quantities as limits of generating functions) in Section 5.5 and 5.8.2
- Theorem 22 (Relating recurrent and G) in Section 5.6.1 and 5.5.1
- Corollary 23 (recurrent is invariant on irreducible classes) in Section 5.6.2
- Lemma 24 (Relation between recurrent and H and U) in Section 5.6 and 5.6.3
- Theorem 25 (Recurrent classes are essential) in Section 5.7
- Corollary 26 (Finite essential classes are recurrent) in Section 5.7

- Lemma 27 (Relation between M and U) in Section 5.8.2
- Corollary 28 (Positive recurrent is invariant on irreducible classes) in Section 5.9
- Definition 29 (Stationary distribution) in Section 5.10
- Theorem 31 (Stationary distribution implies positive recurrence) in Section 5.10.1
- Definition 32 (Aperiodic classes) in Section 5.11
- Lemma 33 (Aperiodic class implies non-zero p) in Section 5.11.1
  Here instead of the quantifiers $\exists N.\ \forall n \geq N.\ P\ n$ we use *eventually P sequentially*
- Theorem 34 (Stationary distribution is asymptotic distribution) in Section 5.13

theory *Discrete-Markov-Kernel*
  imports *Auxiliary*
begin

2 Discrete Markov Kernel

locale *Discrete-Markov-Kernel* =
  fixes S :: *'s set* **and** K :: *'s* $\Rightarrow$ *'s measure*
  assumes *countable-space*[*simp*]: *countable S*
  assumes *non-empty-space*: $S \neq \{\}$
  assumes *sets-eq-S*[*simp*]: $\bigwedge s.$ *sets* $(K\ s) =$ *Pow S*
  assumes *markov-kernel*: $\bigwedge s.$ *prob-space* $(K\ s)$
begin

abbreviation
  $D \equiv (\Pi_M\ s{\in}S.\ K\ s)$

abbreviation
  D-seq $\equiv (\Pi_M\ n{\in}UNIV$ :: *nat set*. $D)$

sublocale K: *prob-space K s* **for** s **by** (*rule markov-kernel*)

sublocale D: *product-prob-space* $\lambda s.\ K\ s\ S$ **..**

sublocale P: *sequence-space D* **..**

definition *s0* = (*SOME s*. *s* ∈ *S*)

lemma *s0*: *s0* ∈ *S*
using *non-empty-space* **by** (*auto intro*: *someI-ex simp*: *s0-def*)

lemma *space-eq-S*[*simp*]: *space* (*K s*) = *S*
using *sets-eq-imp-space-eq*[*of K s count-space S*] *sets-eq-S* **by** *simp*

lemma *K-measurable1*: *measurable* (*K s*) *M* = *measurable* (*count-space S*) *M*
unfolding *measurable-def* **by** *auto*

lemma *K-measurable2*: *measurable M* (*K s*) = *measurable M* (*count-space S*)
unfolding *measurable-def* **by** *auto*

lemma *K-measurable1-imp*[*measurable* (*raw*)]:
f ∈ *measurable* (*count-space S*) *M* ⟹ *f* ∈ *measurable* (*K s*) *M*
unfolding *measurable-def* **by** *auto*

lemma *K-measurable2-imp*[*measurable* (*raw*)]:
f ∈ *measurable M* (*count-space S*) ⟹ *f* ∈ *measurable M* (*K s*)
unfolding *measurable-def* **by** *auto*

lemma *AE-all-S*: (⋀*s*. *s* ∈ *S* ⟹ *AE x in M*. *P s x*) ⟹ *AE x in M*. ∀ *s*∈*S*. *P s x*
by (*auto intro*!: *AE-impI simp*: *AE-all-countable countable-all*[*OF countable-space*])

lemma *K-eq-density*:
assumes [*simp*]: *s* ∈ *S* **shows** *K s* = *density* (*count-space S*) (λ*x*. *emeasure* (*K s*) {*x*})
proof (*rule measure-eqI-countable*)
show *sets* (*K s*) = *Pow S sets* (*density* (*count-space S*) (λ*x*. *emeasure* (*K s*) {*x*})) = *Pow S*
by *simp-all*
show *countable S* **by** (*rule countable-space*)
fix *t* **assume** [*simp*]: *t* ∈ *S*
then have *emeasure* (*density* (*count-space S*) (λ*x*. *emeasure* (*K s*) {*x*})) {*t*} =
(∫⁺*a*. *emeasure* (*K s*) {*a*} ∗ *indicator* {*t*} *a* ∂*count-space S*)
by (*simp add*: *emeasure-density*)
also have ... = (∫⁺*a*. *emeasure* (*K s*) {*t*} ∗ *indicator* {*t*} *a* ∂*count-space S*)
by (*intro positive-integral-cong*) (*auto split*: *split-indicator*)
also have ... = *emeasure* (*K s*) {*t*}
by (*simp add*: *positive-integral-cmult-indicator emeasure-nonneg one-ereal-def*[*symmetric*])
finally show *emeasure* (*K s*) {*t*} = *emeasure* (*density* (*count-space S*) (λ*x*. *emeasure* (*K s*) {*x*})) {*t*} **..**
qed

2.1 Enabled states

definition *E s* = {*s'* ∈ *S*. *emeasure* (*K s*) {*s'*} ≠ *0*}

lemma *E-subset-S*[*simp*]: $E\ s \subseteq S$
unfolding *E-def* **by** *auto*

lemma *E-in-S*[*dest*]: $s' \in E\ s \Longrightarrow s' \in S$
unfolding *E-def* **by** *auto*

lemma *measurable-E*[*measurable* (*raw*)]:
assumes *f*[*measurable*]: $f \in$ *measurable M* (*count-space S*)
and *g*[*measurable*]: $g \in$ *measurable M* (*count-space S*)
shows $(\lambda x.\ f\ x \in E\ (g\ x)) \in$ *measurable M* (*count-space UNIV*)
proof −
have [*measurable*]: $\bigwedge s.\ E\ s \in$ *sets* (*count-space S*) **by** *simp*

have $\{x \in$ *space M*. $f\ x \in E\ (g\ x)\} = \{x \in$ *space M*. $\exists\, s \in S.\ g\ x = s \wedge f\ x \in E\ s\}$
using *measurable-space*[*OF g*] **by** *auto*
also have $\ldots \in$ *sets M*
by *measurable*
finally show *?thesis* **by** (*simp add*: *pred-def*)
qed

lemma *AE-K-iff*:
assumes [*simp*]: $s \in S$
shows (*AE t in K s*. *P t*) $\longleftrightarrow$ $(\forall\, t \in E\ s.\ P\ t)$
proof −
let *?C* = *to-nat-on S* ‘ *S* **and** *?f* = *from-nat-into S*
have (*AE t in K s*. *P t*) $\longleftrightarrow$ *emeasure* (*K s*) $\{t \in S.\ \neg\ P\ t\} = 0$
by (*rule AE-iff-measurable*) *auto*
also have *emeasure* (*K s*) $\{t \in S.\ \neg\ P\ t\} =$
emeasure (*K s*) $(\bigcup i.$ *if* $i \in$ *?C* $\wedge\ \neg\ P$ (*?f i*) *then* {*?f i*} *else* {})
by (*intro arg-cong2*[**where** *f*=*emeasure*])
(*auto split*: *split-if-asm cong*: *rev-conj-cong simp*: *eq-from-nat-into-iff*)
also have $\ldots =$ $(\sum i.$ *emeasure* (*K s*) (*if* $i \in$ *?C* $\wedge\ \neg\ P$ (*?f i*) *then* {*?f i*} *else* {}))
by (*subst suminf-emeasure*) (*auto simp*: *disjoint-family-on-def*)
also have $(\sum i.$ *emeasure* (*K s*) (*if* $i \in$ *?C* $\wedge\ \neg\ P$ (*?f i*) *then* {*?f i*} *else* {})) = *0* $\longleftrightarrow$
$(\forall\, i.$ *emeasure* (*K s*) (*if* $i \in$ *?C* $\wedge\ \neg\ P$ (*?f i*) *then* {*?f i*} *else* {}) = *0*)
by (*rule suminf-ereal-eq-0*) *auto*
also have $\ldots \longleftrightarrow (\forall\, i.\ i \in$ *?C* $\longrightarrow$ *?f i* $\in E\ s \longrightarrow P$ (*?f i*))
by (*auto simp add*: *E-def*)
also have $\ldots \longleftrightarrow (\forall\, t \in S.\ t \in E\ s \longrightarrow P\ t)$
by (*auto simp*: *bij-betw-def*)
finally show *?thesis* **by** *auto*
qed

lemma *AE-enabled*:
assumes [*simp*]: $s \in S$ **shows** *AE s' in K s*. $s' \in E\ s$
by (*simp add*: *AE-K-iff*)

2.1.1 Almost every path is everywhere enabled

lemma *AE-all-E*:
assumes $*$: $\bigwedge t.\ t \in E\ s \Longrightarrow AE\ x\ in\ M.\ P\ t\ x$
shows $AE\ x\ in\ M.\ \forall\, t{\in}E\ s.\ P\ t\ x$
proof −
have $\bigwedge P\ x.\ (\forall\, t{\in}E\ s.\ P\ t) \longleftrightarrow (\forall\, t{\in}S.\ t \in E\ s \longrightarrow P\ t)$
by (*auto simp*: *E-def*)
with $*$ **show** *?thesis*
by (*auto intro*!: *AE-all-S AE-impI*)
qed

2.2 The set of reachable states

inductive-set *reachable* :: *'s set* ⇒ *'s* ⇒ *'s set* **for** Φ :: *'s set* **and** *s* :: *'s* **where**
start: $t \in E\ s \Longrightarrow t \in reachable\ \Phi\ s$
| *step*: $t \in reachable\ \Phi\ s \Longrightarrow t' \in E\ t \Longrightarrow t \in \Phi \Longrightarrow t' \in reachable\ \Phi\ s$

lemma *reachable-induct-trans*[*consumes 1*, *case-names start step*]:
assumes *t*: $t \in reachable\ \Phi\ s$
assumes *1*: $\bigwedge t\ s.\ t \in E\ s \Longrightarrow P\ t\ s$
assumes *2*: $\bigwedge t\ t'\ s.\ t' \in reachable\ \Phi\ s \Longrightarrow P\ t'\ s \Longrightarrow t' \in \Phi \Longrightarrow t \in reachable\ \Phi\ t' \Longrightarrow P\ t\ t' \Longrightarrow P\ t\ s$
shows $P\ t\ s$
using *t*
by *induct* (*blast intro*: *1 2 reachable.intros*)+

lemma *reachable-trans*:
assumes $t \in reachable\ \Phi\ s\ t' \in reachable\ \Phi\ t\ t \in \Phi$ **shows** $t' \in reachable\ \Phi\ s$
using *assms*(*2*,*1*,*3*) **by** (*induct t'*) (*auto dest*: *reachable.step*)

lemma *reachable-step-rev*:
assumes $t \in reachable\ \Phi\ s\ s \in E\ s'\ s \in \Phi$ **shows** $t \in reachable\ \Phi\ s'$
using *assms reachable-trans*[*of s* Φ *s' t*] **by** (*auto intro*: *reachable.start*)

lemma *reachable-rev*:
assumes *t*: $t \in reachable\ \Phi\ s$
obtains (*start*) $t \in E\ s$ | (*step*) s' **where** $t \in reachable\ \Phi\ s'\ s' \in \Phi\ s' \in E\ s$
using *t* **by** *induct* (*auto intro*: *reachable.step reachable.start*)

lemma *reachable-induct-rev*[*consumes 1*, *case-names start step*]:
assumes *t*: $t \in reachable\ \Phi\ s$
assumes *1*: $\bigwedge s.\ t \in E\ s \Longrightarrow P\ s$
assumes *2*: $\bigwedge t'\ s.\ t' \in E\ s \Longrightarrow t' \in \Phi \Longrightarrow t \in reachable\ \Phi\ t' \Longrightarrow P\ t' \Longrightarrow P\ s$
shows $P\ s$
using *t*
proof *cases*
case (*step s'*)
then have $t \in reachable\ \Phi\ s'\ P\ s'\ s' \in \Phi$

by (*auto intro*: *1 reachable.start*)
with *step*(*1*) **show** *?thesis*
by (*induct rule*: *reachable-induct-trans*) (*auto intro*: *1 2 reachable-trans*)
qed *fact*

lemma *reachable-in-S*[*dest*]:
assumes $t \in$ *reachable* Φ s **shows** $t \in S$
using *assms* **by** (*induct t*) *auto*

lemma *reachable-closed*:
assumes $s \in R$ $s \in \Phi$ $\forall s{\in}R \cap \Phi.$ E $s \subseteq R$
shows *reachable* $(\Phi - \Psi)$ $s \subseteq R$
proof
fix x **assume** $x \in$ *reachable* $(\Phi - \Psi)$ s
then show $x \in R$
by *induct* (*insert assms*, *auto*)
qed

lemma *reachable-closed-rev*:
assumes *t*: $t \in$ *reachable* $(\Phi - \Psi)$ s
and *R*: $t \in R$ $\{s{\in}\Phi{-}\Psi.$ $R \cap E$ $s \neq \{\}\} \subseteq R$
and *s*: $s \in \Phi$ $s \notin \Psi$
shows $s \in R$
using t s **by** (*induct rule*: *reachable-induct-rev*) (*insert R*, *auto*)

lemma *reachableE*:
assumes *t*: $t \in$ *reachable* Φ s
obtains (*path*) ω n **where** $\bigwedge i.$ $i \leq n \Longrightarrow \omega$ $i \in E$ (*nat-case* s ω i) $\bigwedge i.$ $i < n$ $\Longrightarrow \omega$ $i \in \Phi$ ω $n = t$
using t
proof (*induct arbitrary*: *thesis rule*: *reachable-induct-rev*)
case (*start* t')
from *start.hyps start.prems*[*of 0 nat-case* t ($\lambda x.$ *s0*)] **show** *?case* **by** *auto*
next
case (*step* t' s)
from *step*(*4*) **guess** n ω **. note** $\omega = this$
let *?*ω = *nat-case* t' ω
show *?case*
proof (*rule step.prems*)
fix i **show** $i <$ *Suc* $n \Longrightarrow$ *?*ω $i \in \Phi$ *?*ω (*Suc* n) $= t$ $i \leq$ *Suc* $n \Longrightarrow$ *?*ω $i \in E$ (*nat-case* s *?*ω i)
using ω(*1*, *2*)[*of* $i - 1$] ω(*3*) *step.hyps*(*1*,*2*) **by** (*auto split*: *nat.split*)
qed
qed

lemma *reachableI*:
$(\bigwedge i.$ $i \leq n \Longrightarrow \omega$ $i \in E$ (*nat-case* s ω i)) $\Longrightarrow$ $(\bigwedge i.$ $i < n \Longrightarrow \omega$ $i \in \Phi) \Longrightarrow \omega$ n $= t \Longrightarrow$
$t \in$ *reachable* Φ s

proof (*induct n arbitrary*: *t*)
case (*Suc n*)
from *Suc.prems Suc.hyps*[*of* ω *n*]
have ω *n* $\in$ *reachable* Φ *s* **by** *auto*
with *Suc.prems*(*1*)[*of Suc n*] *Suc.prems*(*2*)[*of n*] *Suc.prems*(*3*)
show *?case*
by (*auto intro*: *reachable.intros*)
qed (*auto intro*: *reachable.intros*)

lemma *reachableI2*:
($\bigwedge i.\ i \leq n \Longrightarrow \omega\ i \in E\ (\textit{nat-case}\ s\ \omega\ i)$) $\Longrightarrow$ ($\bigwedge i.\ i < n \Longrightarrow \omega\ i \in \Phi$) $\Longrightarrow$ ω *n* $\in$ *reachable* Φ *s*
by (*rule reachableI*[*of n* ω]) *auto*

2.3 The path generating function

primrec *path* :: $'s \Rightarrow (nat \Rightarrow 's \Rightarrow 's) \Rightarrow (nat \Rightarrow 's)$ **where**
path s x 0 = *x 0* (*if s* $\in$ *S then s else s0*)
| *path s x* (*Suc n*) = *x* (*Suc n*) (*path s x n*)

abbreviation
S-seq $\equiv$ (Π_M *n*$\in$*UNIV*::*nat set*. *count-space S*)

lemma *path-in-S*: *x* $\in$ *space D-seq* $\Longrightarrow$ *path s x n* $\in$ *S*
by (*induct n*) (*auto simp*: *s0 space-PiM PiE-iff*)

lemma *measurable-path*:
path s $\in$ *measurable D-seq S-seq*
proof (*rule measurable-PiM-single'*)
fix *i* **show** ($\lambda\omega$. *path s* ω *i*) $\in$ *measurable D-seq* (*count-space S*)
proof (*induct i*)
case (*Suc i*)
note *measurable-compose-countable*[*OF countable-space this*, *measurable* (*raw*)]
show *?case*
unfolding *path.simps* **by** *measurable*
qed (*simp add*: *s0*)
qed (*simp add*: *path-in-S PiE-iff*)

lemma *measurable-path'*[*measurable* (*raw*)]:
assumes *f*: *f* $\in$ *measurable M* (*count-space S*) **and** *g*: *g* $\in$ *measurable M D-seq*
shows (λx. *path* (*f x*) (*g x*)) $\in$ *measurable M S-seq*
using *measurable-compose-countable*[*OF countable-space f*,
OF measurable-compose[*OF g measurable-path*]] **.**

lemma *path-nat-case*:
s $\in$ *S* $\Longrightarrow$ *s'* $\in$ *S* $\rightarrow$ *S* $\Longrightarrow$ *path s* (*nat-case s'* ω) *i* = *nat-case* (*s' s*) (*path* (*s' s*) ω) *i*
by (*induct i*) (*auto split*: *nat.split*)

lemma *path-comb-seq*:
assumes $s \in S$ **and** ω: $\omega \in$ *space D-seq*
shows *path s* (*comb-seq i* ω ω') = *comb-seq i* (*path s* ω) (*path* (*nat-case s* (*path s* ω) *i*) ω')
proof (*induct i arbitrary*: ω')
case *0* **then show** *?case* **by** (*simp add*: *comb-seq-0*)
next
case (*Suc n*)
moreover have *path* (*nat-case s* (*path s* ω) *n*) (*nat-case* (ω *n*) ω') = *nat-case* (*path s* ω *n*) (*path* (*path s* ω *n*) ω') (**is** *?A* = *?B*)
proof
fix *i* **show** *?A i* = *?B i* **by** (*induct i*) (*simp-all split*: *nat.split add*: ω *path-in-S*)
qed
ultimately show *?case*
by (*simp add*: *comb-seq-Suc*)
qed

lemma *measurable-smallest*[*measurable*]: *Measurable.pred S-seq* ($\lambda\omega$. *smallest* ω *X i*)
unfolding *smallest-def* **by** *simp*

2.4 *paths* is a probability space

definition
paths s = *distr D-seq S-seq* (*path s*)

lemma *sets-paths*[*simp*]: *sets* (*paths s*) = *sets S-seq* **by** (*simp add*: *paths-def*)
lemma *space-paths*[*simp*]: *space* (*paths s*) = *space S-seq* **by** (*simp add*: *paths-def*)
lemma *measurable-paths1*[*simp*]:
measurable (*paths s*) *M* = *measurable S-seq M* **by** (*simp add*: *measurable-def*)
lemma *measurable-paths2*[*simp*]:
measurable M (*paths s*) = *measurable M S-seq* **by** (*simp add*: *measurable-def*)

sublocale *path*: *prob-space paths s* **for** *s*
unfolding *paths-def* **using** *measurable-path* **by** (*rule P.prob-space-distr*)

lemma *borel-measurable-positive-integral-paths*[*measurable* (*raw*)]:
assumes *f*: *f* ∈ *measurable M* (*count-space S*)
assumes *g*: (λ(*x*, *y*). *g x y*) ∈ *borel-measurable* (*M* $\bigotimes_M$ *S-seq*)
shows (λx. *integral*^{P} (*paths* (*f x*)) (*g x*)) ∈ *borel-measurable M*
by (*rule measurable-compose-countable*[*OF countable-space f*])
(*simp add*: *g borel-measurable-positive-integral cong*: *measurable-cong-sets*)

lemma *borel-measurable-lebesgue-integral-paths*[*measurable* (*raw*)]:
assumes *f*: *f* ∈ *measurable M* (*count-space S*)
assumes *g*: (λ(*x*, *y*). *g x y*) ∈ *borel-measurable* (*M* $\bigotimes_M$ *S-seq*)
shows (λx. *integral*^{L} (*paths* (*f x*)) (*g x*)) ∈ *borel-measurable M*
by (*rule measurable-compose-countable*[*OF countable-space f*])
(*simp add*: *g borel-measurable-lebesgue-integral cong*: *measurable-cong-sets*)

2.5 Splitting rules

lemma *positive-integral-split*:
assumes [*simp*]: $s \in S$ **and** f[*measurable*]: $f \in$ *borel-measurable S-seq*
shows $(\int^+\omega.\ f\ \omega\ \partial paths\ s) = (\int^+\omega.\ (\int^+\omega'.\ f\ (comb\text{-}seq\ i\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ i))\ \partial paths\ s)$
proof −
have $(\int^+\omega.\ f\ \omega\ \partial(paths\ s)) = (\int^+\omega.\ f\ (path\ s\ \omega)\ \partial D\text{-}seq)$
unfolding *paths-def positive-integral-distr*[*OF measurable-path f*] **..**
also have $\ldots = (\int^+\omega.\ f\ (path\ s\ \omega)\ \partial distr\ (D\text{-}seq \bigotimes_M D\text{-}seq)\ D\text{-}seq\ (\lambda(\omega, \omega').\ comb\text{-}seq\ i\ \omega\ \omega'))$
unfolding *PiM-comb-seq* **..**
also have $\ldots = (\int^+\omega.\ f\ (path\ s\ ((\lambda(\omega, \omega').\ comb\text{-}seq\ i\ \omega\ \omega')\ \omega))\ \partial(D\text{-}seq \bigotimes_M D\text{-}seq))$
by (*subst positive-integral-distr*) *auto*
also have $\ldots = (\int^+\omega.\ (\int^+\omega'.\ f\ (path\ s\ (comb\text{-}seq\ i\ \omega\ \omega'))\ \partial D\text{-}seq)\ \partial D\text{-}seq)$
by (*subst P.positive-integral-fst-measurable*(*2*)[*symmetric*]) *auto*
also have $\ldots = (\int^+\omega.\ (\int^+\omega'.\ f\ (comb\text{-}seq\ i\ (path\ s\ \omega)\ (path\ (nat\text{-}case\ s\ (path\ s\ \omega)\ i)\ \omega'))\ \partial D\text{-}seq)\ \partial D\text{-}seq)$
by (*intro positive-integral-cong*) (*simp add*: *path-comb-seq*)
also have $\ldots = (\int^+\omega.\ (\int^+\omega'.\ f\ (comb\text{-}seq\ i\ \omega\ (path\ (nat\text{-}case\ s\ \omega\ i)\ \omega'))\ \partial D\text{-}seq)\ \partial paths\ s)$
unfolding *paths-def* **by** (*simp add*: *positive-integral-distr*)
also have $\ldots = (\int^+\omega.\ (\int^+\omega'.\ f\ (comb\text{-}seq\ i\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ i))\ \partial paths\ s)$
unfolding *paths-def* **by** (*intro positive-integral-cong*) (*simp add*: *positive-integral-distr*)
finally show *?thesis* **.**
qed

lemma *integrable-split-AE*:
assumes [*simp*]: $s \in S$ **and** f: *integrable* (*paths s*) f
shows *AE* ω *in paths s*. *integrable* (*paths* (*nat-case s* ω *i*)) ($\lambda\omega'$. f (*comb-seq i* ω ω'))
proof −
have [*measurable*]: $f \in$ *borel-measurable S-seq*
using f[*THEN integrableD*(*1*)] **by** *simp*
from *integrableD*[*OF f*]
positive-integral-split[*symmetric*, *of s* $\lambda x.$ *ereal* ($f\ x$) *i*]
positive-integral-split[*symmetric*, *of s* $\lambda x.$ − *ereal* ($f\ x$) *i*]
have *AE* ω *in paths s*. $(\int^+\omega'.\ f\ (comb\text{-}seq\ i\ \omega\ \omega')\ \partial(paths\ (nat\text{-}case\ s\ \omega\ i))) \neq \infty$
and *AE* ω *in paths s*. $(\int^+\omega'.\ -f\ (comb\text{-}seq\ i\ \omega\ \omega')\ \partial(paths\ (nat\text{-}case\ s\ \omega\ i))) \neq \infty$
by (*auto intro*!: *positive-integral-PInf-AE*)
with *AE-space* **show** *?thesis*
by *eventually-elim* (*auto simp*: *integrable-def*)
qed

lemma *integrable-split*:
assumes [*simp*]: $s \in S$ **and** f: *integrable* (*paths s*) f

shows *integrable* (*paths s*) (λω. *integral*^{L} (*paths* (*nat-case s ω i*)) (λω′. *f* (*comb-seq i ω ω′*)))
proof −
have [*measurable*]: *f* ∈ *borel-measurable S-seq*
using *f*[*THEN integrableD*(*1*)] **by** *simp*
have *f-abs*: *integrable* (*paths s*) (λ*x*. |*f x*|)
using *f* **by** (*rule integrable-abs*)
from *integrableD*[*OF this*]
have (∫$^+$ ω. (∫$^+$ ω′. *ereal* |*f* (*comb-seq i ω ω′*)| ∂*paths* (*nat-case s ω i*)) ∂*paths s*) ≠ ∞
by (*subst positive-integral-split*[*OF* ‹*s*∈*S*›, *symmetric*]) *auto*
also have (∫$^+$ ω. (∫$^+$ ω′. *ereal* |*f* (*comb-seq i ω ω′*)| ∂*paths* (*nat-case s ω i*)) ∂*paths s*) =
(∫$^+$ ω. *ereal* (∫ ω′. |*f* (*comb-seq i ω ω′*)| ∂*paths* (*nat-case s ω i*)) ∂*paths s*)
using *integrable-split-AE*[*OF* ‹*s*∈*S*› *f-abs*, *of i*]
by (*intro positive-integral-cong-AE*) (*auto simp*: *positive-integral-eq-integral*)
finally have
integrable (*paths s*) (λω. *integral*^{L} (*paths* (*nat-case s ω i*)) (λω′. |*f* (*comb-seq i ω ω′*)|))
by (*auto intro*!: *lebesgue-integral-nonneg integrable-nonneg*)
then have *integrable* (*paths s*) (λω. |*integral*^{L} (*paths* (*nat-case s ω i*)) (λω′. *f* (*comb-seq i ω ω′*))|)
apply (*rule integrable-bound*)
using *integrable-split-AE*[*OF* ‹*s*∈*S*› *f*]
apply *eventually-elim*
apply (*auto intro*!: *integral-triangle-inequality*)
done
then show *?thesis*
by (*simp add*: *integrable-abs-iff*)
qed

lemma *integral-split*:
assumes [*simp*]: *s* ∈ *S* **and** *f*: *integrable* (*paths s*) *f*
shows (∫ ω. *f ω* ∂(*paths s*)) = (∫ ω. (∫ ω′. *f* (*comb-seq i ω ω′*) ∂(*paths* (*nat-case s ω i*))) ∂*paths s*)
proof −
have [*measurable*]: *f* ∈ *borel-measurable S-seq*
using *f*[*THEN integrableD*(*1*)] **by** *simp*

{ fix *f* **assume** *f*: *integrable* (*paths s*) *f* **and** *nneg*: ⋀*x*. *0* ≤ *f x*
from *nneg* **have** (∫ ω. *f ω* ∂(*paths s*)) = *real* (∫$^+$ω. *f ω* ∂(*paths s*))
by (*rule integral-eq-positive-integral*)
also have ... = *real* (∫$^+$ω. (∫$^+$ω′. *f* (*comb-seq i ω ω′*) ∂(*paths* (*nat-case s ω i*))) ∂*paths s*)
using *integrableD*(*1*)[*OF f*] **by** (*subst positive-integral-split*) *auto*
also have ... = *real* (∫$^+$ω. (∫ ω′. *f* (*comb-seq i ω ω′*) ∂(*paths* (*nat-case s ω i*))) ∂*paths s*)
using *integrable-split-AE*[*OF* ‹*s*∈*S*› *f*, *of i*] *AE-space*
by (*intro positive-integral-cong-AE arg-cong*[**where** *f*=*real*])

```
      (auto simp: nneg positive-integral-eq-integral)
    also have ... = (∫ ω. (∫ ω'. f (comb-seq i ω ω') ∂(paths (nat-case s ω i)))
∂paths s)
      by (subst positive-integral-eq-integral)
         (auto intro!: integrable-split f lebesgue-integral-nonneg nneg)
    finally have (∫ ω. f ω ∂(paths s)) =
      (∫ ω. (∫ ω'. f (comb-seq i ω ω') ∂(paths (nat-case s ω i))) ∂paths s) . }
  note nonneg = this
  def Pf ≡ λx. max 0 (f x) and Nf ≡ λx. max 0 (− f x)
  then have f-eq: f = (λx. Pf x − Nf x)
    and Pf: integrable (paths s) Pf ⋀x. 0 ≤ Pf x
    and Nf: integrable (paths s) Nf ⋀x. 0 ≤ Nf x
    by (auto intro!: integrable-max f)
  then have (∫ ω. f ω ∂(paths s)) = (∫ ω. Pf ω ∂(paths s)) − (∫ ω. Nf ω ∂(paths
s))
    by simp
  also have ... = (∫ ω. (∫ ω'. Pf (comb-seq i ω ω') ∂(paths (nat-case s ω i)))
∂paths s) −
      (∫ ω. (∫ ω'. Nf (comb-seq i ω ω') ∂(paths (nat-case s ω i))) ∂paths s)
    using nonneg[OF Pf] nonneg[OF Nf] by simp
  also have ... = (∫ ω. (∫ ω'. Pf (comb-seq i ω ω') ∂(paths (nat-case s ω i))) −
      (∫ ω'. Nf (comb-seq i ω ω') ∂(paths (nat-case s ω i))) ∂paths s)
    by (intro integral-diff(2)[symmetric] integrable-split Pf Nf ‹s∈S›)
  also have ... = (∫ ω. (∫ ω'. Pf (comb-seq i ω ω') − Nf (comb-seq i ω ω') ∂(paths
(nat-case s ω i))) ∂paths s)
    using integrable-split-AE[OF ‹s∈S› Pf(1), of i] integrable-split-AE[OF ‹s∈S›
Nf(1), of i]
    by (intro integral-cong-AE) auto
  finally show ?thesis
    by (simp add: f-eq)
qed

lemma emeasure-split:
  assumes [simp]: s ∈ S and A[measurable]: A ∈ sets S-seq
  shows emeasure (paths s) A =
    (∫ +ω. emeasure (paths (nat-case s ω i)) (comb-seq i ω −‘ A ∩ space S-seq)
∂paths s)
proof −
  have emeasure (paths s) A = (∫ + x. indicator A x ∂paths s)
    using A by auto
  also have ... = (∫ +ω. (∫ +ω'. indicator A (comb-seq i ω ω') ∂(paths (nat-case
s ω i))) ∂paths s)
    by (subst positive-integral-split) auto
  also have ... = (∫ +ω. (∫ +ω'. indicator (comb-seq i ω −‘ A ∩ space S-seq) ω'
∂(paths (nat-case s ω i))) ∂paths s)
    by (auto intro!: positive-integral-cong simp: indicator-def)
  also have ... = (∫ +ω. emeasure (paths (nat-case s ω i)) (comb-seq i ω −‘ A ∩
space S-seq) ∂paths s)
    by (auto intro!: positive-integral-cong positive-integral-indicator)
```

finally show *?thesis* **.**
qed

lemma *emeasure-split-Collect*:
assumes $s \in S$ **and** P: $\{x \in space\ S\text{-}seq.\ P\ x\} \in sets\ S\text{-}seq$
shows *emeasure* (*paths s*) $\{x \in space\ (paths\ s).\ P\ x\} =$
$(\int^+ \omega.\ emeasure\ (paths\ (nat\text{-}case\ s\ \omega\ i))\ \{\omega' \in space\ S\text{-}seq.\ P\ (comb\text{-}seq\ i\ \omega\ \omega')\}\ \partial paths\ s)$
using *emeasure-split*[*OF assms*, *of i*]
using *measurable-space*[*OF measurable-comb-seq*, *of - count-space S i*]
by (*auto intro*!: *positive-integral-cong arg-cong2*[**where** f=*emeasure*]
simp: *space-PiM* ‹$s \in S$› *PiE-iff space-pair-measure split*: *nat.split*)

lemma *prob-split*:
assumes $s \in S$ **and** A: $A \in sets\ S\text{-}seq$
shows *prob s A* =
$(\int \omega.\ prob\ (nat\text{-}case\ s\ \omega\ i)\ (comb\text{-}seq\ i\ \omega\ -`\ A \cap space\ S\text{-}seq)\ \partial paths\ s)$
unfolding *measure-def emeasure-split*[*OF assms*, *of i*]
by (*subst integral-eq-positive-integral*) (*auto simp*: *emeasure-eq-measure measure-nonneg*)

lemma *prob-split-Collect*:
assumes $s \in S$ **and** P: $\{x \in space\ S\text{-}seq.\ P\ x\} \in sets\ S\text{-}seq$
shows $\mathcal{P}(x\ in\ paths\ s.\ P\ x) = (\int \omega.\ \mathcal{P}(\omega'\ in\ paths\ (nat\text{-}case\ s\ \omega\ i).\ P\ (comb\text{-}seq\ i\ \omega\ \omega'))\ \partial paths\ s)$
using *prob-split*[*OF assms*, *of i*]
using *measurable-space*[*OF measurable-comb-seq*, *of - count-space S i*]
by (*auto intro*!: *integral-cong arg-cong2*[**where** f=*measure*]
simp: *space-PiM* ‹$s \in S$› *PiE-iff space-pair-measure split*: *nat.split*)

lemma *AE-split*:
assumes [*simp*]: $s \in S$ **and** P[*measurable*]: $\{x \in space\ S\text{-}seq.\ P\ x\} \in sets\ S\text{-}seq$
shows $(AE\ \omega\ in\ paths\ s.\ P\ \omega) \longleftrightarrow$
$(AE\ \omega\ in\ paths\ s.\ AE\ \omega'\ in\ paths\ (nat\text{-}case\ s\ \omega\ i).\ P\ (comb\text{-}seq\ i\ \omega\ \omega'))$
proof −
have $(AE\ x\ in\ paths\ s.\ P\ x) \longleftrightarrow integral^P\ (paths\ s)\ (indicator\ \{x.\ \neg\ P\ x\}) = 0$
using P **by** (*subst AE-iff-positive-integral*) *auto*
also have $integral^P\ (paths\ s)\ (indicator\ \{x.\ \neg\ P\ x\}) =$
$(\int^+ \omega.\ (\int^+ \omega'.\ indicator\ \{\omega.\ \neg\ P\ \omega\}\ (comb\text{-}seq\ i\ \omega\ \omega')\ \partial(paths\ (nat\text{-}case\ s\ \omega\ i)))\ \partial paths\ s)$
by (*subst positive-integral-split*) *auto*
also have $\ldots = (\int^+ \omega.\ (\int^+ \omega'.\ indicator\ \{\omega'.\ \neg\ P\ (comb\text{-}seq\ i\ \omega\ \omega')\}\ \omega'\ \partial(paths\ (nat\text{-}case\ s\ \omega\ i)))\ \partial paths\ s)$
by (*auto intro*!: *positive-integral-cong split*: *split-indicator*)
also have $((\int^+ \omega.\ (\int^+ \omega'.\ indicator\ \{\omega'.\ \neg\ P\ (comb\text{-}seq\ i\ \omega\ \omega')\}\ \omega'\ \partial(paths\ (nat\text{-}case\ s\ \omega\ i)))\ \partial paths\ s) = 0) \longleftrightarrow$
$(AE\ \omega\ in\ paths\ s.\ (\int^+ \omega'.\ indicator\ \{\omega'.\ \neg\ P\ (comb\text{-}seq\ i\ \omega\ \omega')\}\ \omega'\ \partial(paths\ (nat\text{-}case\ s\ \omega\ i))) = 0)$
by (*subst positive-integral-0-iff-AE*) (*simp*, *simp add*: *eq-iff positive-integral-positive*)
also have $\ldots \longleftrightarrow (AE\ \omega\ in\ paths\ s.\ AE\ \omega'\ in\ paths\ (nat\text{-}case\ s\ \omega\ i).\ P\ (comb\text{-}seq$

$i\ \omega\ \omega'$))
by (*intro AE-cong*) (*simp add*: *AE-iff-positive-integral*)
finally show *?thesis* **.**
qed

2.6 The Generic Chapman-Kolmogorov theorem

lemma *Chapman-Kolmogorov*:
assumes [*simp*]: $s \in S$
assumes [*measurable*]: *Measurable.pred* (*paths s*) *Q1 Measurable.pred* (*paths s*) *P*
assumes *eq*: $\bigwedge\omega\ \omega'.\ \omega \in UNIV \to S \Longrightarrow \omega' \in UNIV \to S \Longrightarrow P$ (*comb-seq n* $\omega\ \omega'$) $\longleftrightarrow$ *Q1* $\omega \wedge$ *Q2* (*nat-case* (*nat-case s* ω *n*) ω')
shows $\mathcal{P}(\omega$ *in paths s*. *P* $\omega) = (\int t.\ \mathcal{P}(\omega$ *in paths s*. *Q1* $\omega \wedge$ *nat-case s* ω *n* = *t*) $*$ $\mathcal{P}(\omega$ *in paths t*. *Q2* (*nat-case t* ω)) ∂*count-space S*)
proof −
interpret *S*: *sigma-finite-measure count-space S*
using *countable-space* **by** (*rule sigma-finite-measure-count-space*)
interpret *SP*: *pair-sigma-finite count-space S paths x* **for** *x* **..**

have *eq-t*: $\bigwedge\omega.\ \omega \in$ *space S-seq* $\Longrightarrow$ $\{t.\ t =$ *nat-case s* ω *n* $\wedge\ t \in S\} = \{$*nat-case s* ω *n*$\}$
by (*auto simp*: *space-PiM split*: *nat.split*)

have $\mathcal{P}(\omega$ *in paths s*. *P* $\omega) = (\int^+\omega.\ \mathcal{P}(\omega'$ *in paths* (*nat-case s* ω *n*). *P* (*comb-seq n* $\omega\ \omega'$)) ∂*paths s*)
unfolding *emeasure-eq-measure*[*symmetric*] **by** (*subst emeasure-split-Collect*[**where** *i*=*n*]) *simp-all*
also have ... = $(\int^+\omega.\ \mathcal{P}(\omega'$ *in paths* (*nat-case s* ω *n*). *Q1* $\omega \wedge$ *Q2* (*nat-case* (*nat-case s* ω *n*) ω')) ∂*paths s*)
unfolding *emeasure-eq-measure*[*symmetric*]
by (*intro positive-integral-cong arg-cong2*[**where** *f*=*emeasure*] *arg-cong*[**where** *f*=*Collect*] *ext conj-cong refl eq*)
(*simp-all add*: *space-PiM Pi-iff PiE-iff*)
also have ... = $(\int^+\omega.\ (\int^+t.$ (*ereal* $\mathcal{P}(\omega'$ *in paths* (*nat-case s* ω *n*). *Q2* (*nat-case* (*nat-case s* ω *n*) ω')) $*$ *indicator* $\{x \in$ *space S-seq*. *Q1* $\omega\}$ ω) $*$ *indicator* $\{t \in S.\ t =$ *nat-case s* ω *n*$\}$ *t* ∂*count-space S*) ∂*paths s*)
by (*auto simp*: *positive-integral-cmult-indicator zero-ereal-def*[*symmetric*] *measure-nonneg subset-eq eq-t*
split: *split-indicator intro*!: *positive-integral-cong*)
also have ... = $(\int^+\omega.\ (\int^+t.$ (*ereal* $\mathcal{P}(\omega'$ *in paths t*. *Q2* (*nat-case t* ω')) $*$ *indicator* $\{x \in$ *space S-seq*. *Q1 x*$\}$ ω) $*$ *indicator* $\{t \in S.\ t =$ *nat-case s* ω *n*$\}$ *t* ∂*count-space S*) ∂*paths s*)
by (*intro positive-integral-cong*) (*auto split*: *split-indicator*)
also have ... = $(\int^+t.\ (\int^+\omega.$ (*ereal* $\mathcal{P}(\omega'$ *in paths* (*nat-case s* ω *n*). *Q2* (*nat-case* (*nat-case s* ω *n*) ω')) $*$ *indicator* $\{x \in$ *space S-seq*. *Q1 x*$\}$ ω) $*$ *indicator* $\{t \in S.\ t =$ *nat-case s* ω *n*$\}$ *t* ∂*paths s*) ∂*count-space S*)
apply (*subst SP.Fubini'*)
unfolding *measurable-split-conv*

apply (*rule measurable-compose-countable*[**where** S=S **and** g=*fst*])
apply *simp*
apply *simp*
apply (*rule measurable-compose*[**where** f=*snd* **and** N=*paths s*])
apply (*rule measurable-snd*)
apply *simp*
apply (*intro positive-integral-cong*)
apply (*simp split*: *split-indicator*)
done
also have ... = ($\int^+ t$. ($\int^+ \omega$. *ereal* ($\mathcal{P}(\omega'$ *in paths t. Q2* (*nat-case t* ω'))) $*$ *indicator* {x∈*space* (*paths s*). *Q1 x* $\wedge$ t = *nat-case s x n*} ω ∂*paths s*) ∂*count-space S*)
apply (*intro positive-integral-cong*)
apply (*auto split*: *split-indicator*)
done
also have ... = ($\int^+ t$. *ereal* $\mathcal{P}(\omega'$ *in paths t. Q2* (*nat-case t* ω')) $*$ *emeasure* (*paths s*) {$\omega' \in$ *space* (*paths s*). *Q1* $\omega' \wedge$ *nat-case s* ω' n = t} ∂*count-space S*)
apply (*subst eq-commute*)
by (*intro positive-integral-cong positive-integral-cmult-indicator*) (*auto simp*: *measure-nonneg*)
finally show *?thesis*
apply (*subst* (*asm*) *eq-commute*)
apply (*subst integral-eq-positive-integral*)
apply (*simp add*: *measure-nonneg mult-nonneg-nonneg*)
apply (*simp add*: *emeasure-eq-measure ac-simps*)
done
qed

2.7 Iteration rules and splitting at 1

lemma *distr-K*:
assumes [*simp*]: $s \in S$ **shows** *distr* (*paths s*) (*K s*) ($\lambda\omega$. ω *0*) = *K s*
proof −
have *distr* (Pi_M *UNIV* (λn::*nat*. *D*)) (*K s*) (λx. *x 0 s*) =
distr (*distr* (Pi_M *UNIV* (λn::*nat*. *D*)) *D* (λx. *x 0*)) (*K s*) (λx. *x s*)
by (*subst distr-distr*) (*auto simp*: *comp-def*)
also have ... = *K s*
by (*simp add*: *PiM-component D.PiM-component*)
finally show *?thesis*
unfolding *paths-def*
by (*subst distr-distr*) (*auto simp*: *comp-def*)
qed

lemma *positive-integral-paths-0*:
assumes *s*[*simp*]: $s \in S$ **shows** ($\int^+ \omega$. *f* (ω *0*) ∂*paths s*) = ($\int^+ s'$. *f s'* ∂*K s*)
by (*subst distr-K*[*symmetric*, *OF s*]) (*simp add*: *positive-integral-distr*)

lemma *emeasure-paths-0*:
assumes *s*[*simp*]: $s \in S$

shows *emeasure* (*paths s*) {ω∈*space S-seq*. *P* (ω 0)} = *emeasure* (*K s*) {*s'*∈*space* (*count-space S*). *P s'*}
apply (*subst distr-K*[*symmetric*, *OF s*])
apply (*subst emeasure-distr*)
apply (*auto intro*!: *arg-cong2*[**where** *f*=*emeasure*] *simp*: *space-PiM*)
done

lemma *measure-paths-0*:
assumes *s*[*simp*]: *s* ∈ *S*
shows *measure* (*paths s*) {ω∈*space S-seq*. *P* (ω 0)} = *measure* (*K s*) {*s'*∈*space* (*count-space S*). *P s'*}
using *emeasure-paths-0*[*OF assms*, *of P*] **unfolding** *K.emeasure-eq-measure emeasure-eq-measure* **by** *simp*

lemma *prob-paths-0*:
assumes *s*[*simp*]: *s* ∈ *S*
shows $\mathcal{P}$(ω *in paths s*. *P* (ω 0)) = $\mathcal{P}$(*t in K s*. *P t*)
using *emeasure-paths-0*[*OF assms*, *of P*] **unfolding** *K.emeasure-eq-measure emeasure-eq-measure* **by** *simp*

lemma *integrable-paths-0*:
assumes *s*[*simp*]: *s* ∈ *S*
shows *integrable* (*paths s*) (λω. *f* (ω 0)) = *integrable* (*K s*) *f*
apply (*subst distr-K*[*symmetric*, *OF s*])
apply (*rule integrable-distr-eq*[*symmetric*])
apply *measurable*
done

lemma *integral-paths-0*:
assumes *s*[*simp*]: *s* ∈ *S*
shows $integral^L$ (*paths s*) (λω. *f* (ω 0)) = $integral^L$ (*K s*) *f*
apply (*subst distr-K*[*symmetric*, *OF s*])
apply (*rule integral-distr*[*symmetric*])
apply *measurable*
done

lemma *AE-paths-0*:
assumes *s*[*simp*]: *s* ∈ *S*
shows (*AE* ω *in paths s*. *P* (ω 0)) = (*AE s' in K s*. *P s'*)
apply (*subst distr-K*[*symmetric*, *OF s*])
apply (*subst AE-distr-iff*)
apply *measurable*
done

lemma *positive-integral-iterate*:
assumes [*simp*]: *s* ∈ *S* **and** *f*: *f* ∈ *borel-measurable S-seq*
shows ($\int^+$ω. *f* ω ∂(*paths s*)) = ($\int^+ s'$. ($\int^+$ω. *f* (*nat-case s'* ω) ∂(*paths s'*)) ∂*K s*)
by (*subst positive-integral-split*[*OF assms*, **where** *i*=*1*])

(*simp add*: *positive-integral-paths-0*[*symmetric*])

lemma *integrable-iterate-AE*:
assumes *s*[*simp*]: $s \in S$ **and** *f*: *integrable* (*paths s*) *f*
shows *AE s' in K s*. *integrable* (*paths s'*) (λω. *f* (*nat-case s' ω*))
using *integrable-split-AE*[*OF assms*, **where** *i=1*]
apply *simp*
apply (*subst* (*asm*) *AE-paths-0*)
apply *simp-all*
done

lemma *integrable-iterate*:
assumes [*simp*]: $s \in S$ **and** *f*: *integrable* (*paths s*) *f*
shows *integrable* (*K s*) (λ*s'*. *integral*^{L} (*paths s'*) (λω. *f* (*nat-case s' ω*)))
using *integrable-split*[*OF assms*, **where** *i=1*]
by (*simp add*: *integrable-paths-0*[*symmetric*])

lemma *integral-iterate*:
assumes [*simp*]: $s \in S$ **and** *f*: *integrable* (*paths s*) *f*
shows $(\int \omega.\ f\ \omega\ \partial(\text{paths } s)) = (\int s'.\ (\int \omega.\ f\ (\text{nat-case } s'\ \omega)\ \partial(\text{paths } s'))\ \partial K\ s)$
by (*subst integral-split*[*OF assms*, **where** *i=1*])
(*simp add*: *integral-paths-0*[*symmetric*])

lemma *emeasure-iterate*:
assumes [*simp*]: $s \in S$ **and** *A*[*measurable*]: $A \in$ *sets S-seq*
shows *emeasure* (*paths s*) *A* = ($\int^+ s'$. *emeasure* (*paths s'*) (*nat-case s'* −‘ *A* ∩ *space S-seq*) ∂*K s*)
apply (*subst emeasure-split*[*OF assms*, **where** *i=1*])
apply *simp*
apply (*subst positive-integral-paths-0*)
apply *simp-all*
done

lemma *prob-iterate*:
assumes $s \in S$ **and** *A*: $A \in$ *sets S-seq*
shows *prob s A* = ($\int s'$. *prob s'* (*nat-case s'* −‘ *A* ∩ *space S-seq*) ∂*K s*)
unfolding *measure-def emeasure-iterate*[*OF assms*]
by (*subst integral-eq-positive-integral*) (*auto simp*: *emeasure-eq-measure measure-nonneg*)

lemma *prob-iterate-Collect*:
assumes $s \in S$ **and** *P*: {*x*∈*space S-seq*. *P x*} ∈ *sets S-seq*
shows $\mathcal{P}$(*x in paths s*. *P x*) = ($\int s'$. $\mathcal{P}$(*x in paths s'*. *P* (*nat-case s' x*)) ∂*K s*)
using *prob-iterate*[*OF assms*]
by (*auto intro*!: *integral-cong arg-cong2*[**where** *f*=*measure*] *simp*: *space-PiM* ‹*s* ∈ *S*› *PiE-iff*
split: *nat.split*)

lemma *prob-path*:
$s \in S \Longrightarrow (\bigwedge i.\ i < n \Longrightarrow \omega'\ i \in S) \Longrightarrow$

$\mathcal{P}(\omega$ *in paths s.* $\forall i{<}n.\ \omega\ i = \omega'\ i) = (\prod i{<}n.$ *measure* $(K$ *(nat-case s* $\omega'\ i))\ \{\omega'$ $i\})$
proof (*induct n arbitrary*: $s\ \omega'$)
case *0* **with** *prob-space*[*of s*] **show** *?case* **by** *simp*
next
case (*Suc n s* ω')
have *Suc-eq*: $\{..<$ *Suc n*$\} =$ *insert 0* (*Suc* ‘ $\{..<$ *n*$\})$ *0* $\notin$ *Suc* ‘ $\{..<$ *n*$\}$
by (*auto simp*: *image-iff Ball-def*) *arith*
have $\mathcal{P}(\omega$ *in paths s.* $\forall i{<}$*Suc n.* $\omega\ i = \omega'\ i) =$
$(\int t.\ \mathcal{P}(\omega$ *in paths t.* $\forall i{<}$*Suc n.* (*case i of 0* $\Rightarrow t \mid$ *Suc i* $\Rightarrow \omega\ i) = \omega'\ i)\ \partial K\ s)$
by (*subst prob-iterate-Collect*[*OF* ‹$s \in S$›]) *simp-all*
also have $\ldots = (\int t.\ \mathcal{P}(\omega$ *in paths* $(\omega'\ 0).\ \forall i{<}n.\ \omega\ i = \omega'$ (*Suc i*)) $*$ *indicator* $\{\omega'\ 0\}\ t\ \partial K\ s)$
using *Suc*(*2*,*3*) **by** (*intro integral-cong*) (*auto simp*: *indicator-def all-less-Suc-split*)
also have $\ldots = \mathcal{P}(\omega$ *in paths* $(\omega'\ 0).\ \forall i{<}n.\ \omega\ i = \omega'$ (*Suc i*)) $*$ $(\int t.$ *indicator* $\{\omega'\ 0\}\ t\ \partial K\ s)$
using *Suc*(*2*,*3*) **by** (*simp add*: *lebesgue-integral-cmult*)
also have $\ldots = (\prod i{<}n.$ *measure* $(K\ (\omega'\ i))\ \{\omega'$ (*Suc i*)$\}) *$ *measure* $(K\ s)\ \{\omega'$ *0*$\}$
using *Suc*(*2*,*3*) **by** (*subst Suc*) (*simp-all add*: *K.emeasure-eq-measure nat-case-idem*)
also have $\ldots = (\prod i{\in}$*Suc* ‘ $\{..<n\}.$ *measure* $(K$ *(nat-case s* $\omega'\ i))\ \{\omega'\ i\}) *$ *measure* $(K$ *(nat-case s* $\omega'\ 0))\ \{\omega'\ 0\}$
by (*simp add*: *setprod.reindex*)
also have $\ldots = (\prod i{<}$*Suc n. measure* $(K$ *(nat-case s* $\omega'\ i))\ \{\omega'\ i\})$
by (*simp add*: *Suc-eq*)
finally show *?case* **.**
qed

lemma *emeasure-path*:
$s \in S \Longrightarrow (\bigwedge i.\ i < n \Longrightarrow \omega'\ i \in S) \Longrightarrow$
emeasure (*paths s*) $\{\omega{\in}$*space* (*paths s*)$.\ \forall i{<}n.\ \omega\ i = \omega'\ i\} = (\prod i{<}n.$ *emeasure* $(K$ *(nat-case s* $\omega'\ i))\ \{\omega'\ i\})$
using *prob-path*[*of s n* ω'] **by** (*simp add*: *emeasure-eq-measure K.emeasure-eq-measure setprod-ereal*)

lemma *AE-iterate*:
assumes [*simp*]: $s \in S$ **and** *P*[*measurable*]: $\{x{\in}$*space S-seq. P x*$\} \in$ *sets S-seq*
shows (*AE x in paths s. P x*) $\longleftrightarrow$ (*AE* s' *in K s. AE x in paths* s'*. P* (*nat-case* $s'\ x$))
apply (*subst AE-split*[*OF assms*, **where** *i=1*])
apply (*simp cong del*: *AE-cong*)
apply (*subst AE-paths-0*)
apply *simp-all*
done

lemma *AE-all-enabled*:
assumes *s*[*simp*]: $s \in S$ **shows** *AE* ω *in paths s.* $\forall i.\ \omega\ i \in E$ (*nat-case s* ω *i*)
unfolding *AE-all-countable*
proof

fix i **from** s **show** $AE\ \omega$ *in paths* $s.\ \omega\ i \in E\ (nat\text{-}case\ s\ \omega\ i)$
proof (*induct i arbitrary*: s)
case *0* **with** *AE-enabled*[*of* s] **show** *?case* **by** (*subst AE-iterate*) *auto*
next
case (*Suc i*) **then show** *?case* **by** (*subst AE-iterate*) *auto*
qed
qed

2.8 Fairness

The fairness proof is similar to Theorem 8.1.5 in Baier 1998 (Habilitation thesis). The differences are

- we only prove it for s-fairness (only one transition)
- our prove works for systems with arbitrary size, i.e. also countable infinite systems

definition $fair\ s\ t\ \omega \longleftrightarrow finite\ \{i.\ \omega\ i = s \wedge \omega\ (Suc\ i) = t\} \longrightarrow finite\ \{i.\ \omega\ i = s\}$

lemma *fairI*:
(*finite* $\{i.\ nat\text{-}case\ s'\ \omega\ i = s \wedge \omega\ i = t\} \Longrightarrow finite\ \{i.\ nat\text{-}case\ s'\ \omega\ i = s\}) \Longrightarrow$
$fair\ s\ t\ (nat\text{-}case\ s'\ \omega)$
unfolding *fair-def* **by** *auto*

lemma *measurable-fair*[*measurable*]: $\{\omega \in space\ S\text{-}seq.\ fair\ s\ t\ \omega\} \in sets\ S\text{-}seq$
unfolding *fair-def* **by** *simp*

lemma *positive-integral-prefixes*:
assumes *s*[*simp*]: $s \in S$
assumes [*measurable*]: $i \in measurable\ S\text{-}seq\ (count\text{-}space\ UNIV)$
and [*measurable*]: $f \in borel\text{-}measurable\ S\text{-}seq$
and f: $AE\ x\ in\ paths\ s.\ 0 \leq f\ x$
and *inv-i*: $\bigwedge \omega\ \omega'.\ 0 < i\ \omega \Longrightarrow (\bigwedge j.\ j < i\ \omega \Longrightarrow \omega\ j = \omega'\ j) \Longrightarrow i\ \omega = i\ \omega'$
and *inv-f*: $\bigwedge \omega.\ i\ \omega = 0 \Longrightarrow f\ \omega = 0$
shows $(\int^+ \omega.\ f\ \omega\ \partial paths\ s) = (\int^+ \omega.\ indicator\ \{\omega.\ i\ \omega \neq 0\}\ \omega *$
$(\int^+ \omega'.\ f\ (comb\text{-}seq\ (i\ \omega)\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ (i\ \omega)))\ \partial paths\ s)$
proof −
let *?A* $= \lambda n.\ \{\omega.\ i\ \omega = n \wedge i\ \omega \neq 0\}$
have *inv-i-eq*: $\bigwedge \omega\ \omega'.\ 0 < i\ \omega \Longrightarrow 0 < i\ (comb\text{-}seq\ (i\ \omega)\ \omega\ \omega') \Longrightarrow i\ (comb\text{-}seq\ (i\ \omega)\ \omega\ \omega') = i\ \omega$
by (*rule inv-i*[*symmetric*]) (*simp-all add*: *comb-seq-def*)

{ fix $\omega\ \omega'\ n$ **assume** $0 < n$ **and** i: $i\ (comb\text{-}seq\ n\ \omega\ \omega') = n$
with *inv-i*[*of comb-seq* $n\ \omega\ \omega'\ \omega$] **have** $i\ \omega = n$
by (*auto simp*: *comb-seq-def*) **}**
note *inv-i'* = *this*

{ fix ω **have** $f\ \omega = (\sum n.\ indicator\ (?A\ n)\ \omega * f\ \omega)$
by (*subst suminf-finite*[**where** $N=\{i\ \omega\}$]) (*auto split*: *split-indicator simp*: *inv-f*) **}**
then have $(\int^+ \omega.\ f\ \omega\ \partial paths\ s) = (\int^+ \omega.\ (\sum n.\ indicator\ (?A\ n)\ \omega * f\ \omega)\ \partial paths\ s)$
by (*rule positive-integral-cong*)
also have $\ldots = (\sum n.\ \int^+ \omega.\ indicator\ (?A\ n)\ \omega * f\ \omega\ \partial paths\ s)$
using *f* **by** (*intro positive-integral-suminf*) *auto*
also have $\ldots = (\sum n.\ \int^+ \omega.\ \int^+ \omega'.\ indicator\ (?A\ n)\ (comb\text{-}seq\ n\ \omega\ \omega') *$
$f\ (comb\text{-}seq\ n\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ n)\ \partial paths\ s)$
by (*subst positive-integral-split*[*symmetric*]) *simp-all*
also have $\ldots = (\sum n.\ \int^+ \omega.\ \int^+ \omega'.\ indicator\ (?A\ n)\ \omega *$
$f\ (comb\text{-}seq\ n\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ n)\ \partial paths\ s)$
by (*intro arg-cong*[**where** *f*=*suminf*] *ext positive-integral-cong*)
(*auto split*: *split-indicator intro*: *inv-f inv-i' inv-i-eq*)
also have $\ldots = (\sum n.\ \int^+ \omega.\ indicator\ (?A\ n)\ \omega * \int^+ \omega'.$
$f\ (comb\text{-}seq\ n\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ n)\ \partial paths\ s)$
by (*intro arg-cong*[**where** *f*=*suminf*] *ext positive-integral-cong*)
(*simp add*: *positive-integral-cmult*)
also have $\ldots = (\sum n.\ \int^+ \omega.\ indicator\ (?A\ n)\ \omega * \int^+ \omega'.$
$f\ (comb\text{-}seq\ (i\ \omega)\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ (i\ \omega))\ \partial paths\ s)$
by (*intro arg-cong*[**where** *f*=*suminf*] *ext positive-integral-cong*) (*simp split*: *split-indicator*)
also have $\ldots = (\int^+ \omega.\ (\sum n.\ indicator\ (?A\ n)\ \omega * \int^+ \omega'.$
$f\ (comb\text{-}seq\ (i\ \omega)\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ (i\ \omega)))\ \partial paths\ s)$
by (*simp add*: *positive-integral-positive positive-integral-suminf*)
also have $\ldots = (\int^+ \omega.\ indicator\ \{\omega.\ i\ \omega \neq 0\}\ \omega * \int^+ \omega'.$
$f\ (comb\text{-}seq\ (i\ \omega)\ \omega\ \omega')\ \partial paths\ (nat\text{-}case\ s\ \omega\ (i\ \omega))\ \partial paths\ s)$ (**is** $integral^P$ - *?f* = $integral^P$ - *?g*)
proof (*intro positive-integral-cong*)
fix ω **show** *?f* ω = *?g* ω
by (*subst suminf-finite*[**where** $N=\{i\ \omega\}$]) (*auto split*: *split-indicator*)
qed
finally show *?thesis* **.**
qed

lemma *AE-fair*:
assumes *s*[*simp*]: $s' \in S$ **and** *s*[*simp*]: $s \in S$ **and** *t*[*simp*]: $t \in E\ s$
shows $AE\ \omega\ in\ paths\ s'.\ fair\ s\ t\ (nat\text{-}case\ s'\ \omega)$
proof −
def $unfair \equiv \lambda s'.\ \{\omega \in space\ S\text{-}seq.\ infinite\ \{i.\ nat\text{-}case\ s'\ \omega\ i = s\} \wedge$
$(\forall i.\ nat\text{-}case\ s'\ \omega\ i = s \longrightarrow \omega\ i \neq t)\}$
then have *measurable-unfair*[*measurable*]: $\bigwedge s.\ unfair\ s \in sets\ S\text{-}seq$ **by** *simp*

def $i \equiv \lambda\omega.\ if\ \exists i.\ \omega\ i = s\ then\ Suc\ (LEAST\ i.\ \omega\ i = s)\ else\ 0$
then have *measurable-i*[*measurable*]: $i \in measurable\ S\text{-}seq\ (count\text{-}space\ UNIV)$
by *simp*

{ fix ω s' **assume** $0 < i\ \omega$ **then have** $nat\text{-}case\ s'\ \omega\ (i\ \omega) = s$

```
    by (auto simp add: i-def intro: LeastI split: split-if-asm) }
  note i-eq = this

  { fix i have (∀ n. i ≠ Suc n) ⟷ i = 0 by (cases i) auto }
  note this[simp]

  { fix s' assume [simp]: s' ∈ S
    have emeasure (paths s') (unfair s') = (∫⁺ω. indicator (unfair s') ω ∂paths
s')
      by simp
    also have ... = (∫⁺ω. indicator {ω. i ω ≠ 0} ω *
        (∫⁺ω'. indicator (unfair s') (comb-seq (i ω) ω ω') ∂paths (nat-case s' ω (i
ω))) ∂paths s')
    proof (rule positive-integral-prefixes)
      fix ω ω' assume i: 0 < i ω and le: ⋀j. j < i ω ⟹ ω j = ω' j
      def j ≡ i ω - 1
      from i have ∃ i. ω i = s
        unfolding i-def by (auto split: split-if-asm)
      then have j < i ω ω j = s ∀ k<j. ω k ≠ s
      by (auto simp: i-def j-def intro: LeastI-ex dest: not-less-Least split: split-if-asm)
      with le have i ω' = Suc j
        by (subst i-def) (auto simp add: not-less[symmetric] intro!: Least-equality)
      then show i ω = i ω'
        using ‹0 < i ω› by (simp add: j-def)
    next
      fix ω assume i ω = 0
      then have ⋀k. ω k ≠ s by (auto simp: i-def split: split-if-asm)
      then show indicator (unfair s') ω = (0::ereal)
        by (auto simp add: unfair-def nat.split[of λx. x = s] split: split-indicator)
    qed simp-all
    also have ... = (∫⁺ω. indicator {ω. i ω ≠ 0} ω *
      (∫⁺ω'. indicator {ω. s' = s ⟶ ω 0 ≠ t} ω * indicator (unfair s) ω' ∂paths
s) ∂paths s')
    proof (intro positive-integral-cong ereal-left-mult-cong arg-cong2[where f=integral^P]
ext)
      fix ω ω' assume ω: ω : space (paths s') and indicator {ω. i ω ≠ 0} ω ≠ 0
      then have 0 < i ω
        by (simp add: indicator-def split: split-if-asm)
      with i-eq[of ω] show paths (nat-case s' ω (i ω)) = paths s
        by simp

      from ‹0 < i ω› have less-i: ⋀n. n < i ω ⟹ ω n = s ⟷ Suc n = i ω
        by (auto simp: i-def less-Suc-eq LeastI split: split-if-asm
              intro: LeastI dest: not-less-Least)

      from ‹0 < i ω› have *: {j. nat-case s' (comb-seq (i ω) ω ω') j = s} =
          (if s' = s then {0} else {}) ∪ (op + (i ω) ` {j. nat-case s ω' j = s})
        by (intro set-eqI)
           (auto simp add: less-i split: split-comb-seq nat.split)
```

```
    have infinite {j. nat-case s' (comb-seq (i ω) ω ω') j = s} ⟷
        infinite {j. nat-case s ω' j = s}
      unfolding ∗ by (auto dest!: finite-imageD simp: inj-on-def)
    moreover
    have (∀j. nat-case s' (comb-seq (i ω) ω ω') j = s ⟶ comb-seq (i ω) ω ω' j
≠ t) ⟷
        (s = s' ⟶ ω 0 ≠ t) ∧ (∀j. nat-case s ω' j = s ⟶ ω' j ≠ t)
      (is (∀j. ?P j) ⟷ ?t ∧ (∀j. ?Q j))
    proof (intro iffI allI conjI)
      fix j assume ∗: ∀j. ?P j
      show ?Q j
        using ∗[THEN spec, where x=j + i ω] ‹0 < i ω› less-i
        by (auto split: nat.splits simp: nat-case-comb-seq comb-seq-add)
      show ?t
        using ∗[THEN spec, of 0] ‹0 < i ω› by (simp add: comb-seq-less)
    next
      fix j assume ∗: ?t ∧ (∀j. ?Q j)
      show ?P j
      proof (cases j i ω rule: nat-boundary-cases)
        case less with ∗ show ?thesis
          by (simp add: comb-seq-less less-i split: nat.splits)
      next
        case (add k) with ‹0 < i ω› conjunct2[OF ∗, THEN spec, of k] show
?thesis
          by (simp add: nat-case-comb-seq comb-seq-add split: nat.split-asm)
      qed
    qed
    moreover
    have comb-seq (i ω) ω ω' ∈ space S-seq ⟷ ω' ∈ space S-seq
      using ω by (auto simp: space-PiM PiE-iff split: split-comb-seq)
    ultimately
    have comb-seq (i ω) ω ω' ∈ unfair s' ⟷ ?t ∧ ω' ∈ unfair s
      by (auto simp: unfair-def)
    then show indicator (unfair s') (comb-seq (i ω) ω ω') =
        (indicator {ω. s' = s ⟶ ω 0 ≠ t} ω ∗ indicator (unfair s) ω' :: ereal)
      by (simp add: indicator-def)
  qed
  also have ... = (∫⁺ω. indicator {ω. i ω ≠ 0 ∧ (s' = s ⟶ ω 0 ≠ t)} ω ∗
      (∫⁺ω'. indicator (unfair s) ω' ∂paths s) ∂paths s')
    by (auto simp: indicator-def intro!: positive-integral-cong)
  also have ... = (∫⁺ω. indicator {ω. i ω ≠ 0 ∧ (s' = s ⟶ ω 0 ≠ t)} ω
∂paths s') ∗
      emeasure (paths s) (unfair s)
    by (simp add: positive-integral-multc emeasure-nonneg)
  finally have emeasure (paths s') (unfair s') = ... . }
 note split-unfair = this

 have integral^P (paths s) (indicator {ω. i ω ≠ 0 ∧ (s = s ⟶ ω 0 ≠ t)}) ≤
```

```
    (∫⁺ ω. indicator {s. s ≠ t} (ω 0) ∂paths s)
   by (auto split: split-indicator intro!: positive-integral-mono)
  also have ... = emeasure (K s) (space (K s) - {t})
   by (auto simp add: positive-integral-paths-0 positive-integral-indicator'
         intro!: arg-cong2[where f=emeasure])
  also have ... = 1 - emeasure (K s) {t}
   using K.emeasure-space-1[of s] emeasure-Diff[of K s {t} S] t unfolding E-def
by auto
  also have ... < 1
   using t measure-nonneg[of K s {t}] unfolding E-def by (simp add: K.emeasure-eq-measure
one-ereal-def)
  finally have integral^P (paths s) (indicator {ω. i ω ≠ 0 ∧ (s = s ⟶ ω 0 ≠ t)})
< 1 .
  with split-unfair[of s] have emeasure (paths s) (unfair s) = 0
   by (cases integral^P (paths s) (indicator {ω. i ω ≠ 0 ∧ (s = s ⟶ ω 0 ≠ t)}))
     (auto simp: emeasure-eq-measure one-ereal-def)

  let ?P = λs' n ω. (∀ i. nat-case s' ω (i + n) = s ⟶ ω (i + n) ≠ t) ⟶
   finite {i. nat-case s' ω (i + n) = s}
  { fix s' assume [simp]: s' ∈ S
   have emeasure (paths s') {ω∈space (paths s'). ¬ ?P s' 0 ω} = emeasure (paths
s') (unfair s')
     by (auto simp: unfair-def intro!: arg-cong2[where f=emeasure])
   also have ... = 0
     using split-unfair[of s'] ‹emeasure (paths s) (unfair s) = 0› by simp
   finally have AE ω in paths s'. ?P s' 0 ω
     by (rule AE-I[OF subset-refl]) auto }
  note AE-never = this

  have AE ω in paths s'. ∀ n. ?P s' n ω
   unfolding AE-all-countable
  proof (rule, subst AE-split)
    fix n show AE ω in paths s'. AE ω' in paths (nat-case s' ω n). ?P s' n
(comb-seq n ω ω')
     by (rule AE-I2)
       (insert AE-never, simp add: comb-seq-add nat-case-comb-seq space-PiM
PiE-iff split: nat.split)
  qed simp-all
  then show ?thesis
  proof (eventually-elim, intro fairI)
   fix ω assume P: ∀ n. ?P s' n ω and fin: finite {i. nat-case s' ω i = s ∧ ω i
= t}
   from fin[THEN finite-nat-bounded] obtain n where
     ⋀i. nat-case s' ω i = s ⟹ ω i = t ⟹ i < n
    by auto
   with P[THEN spec, of n]
   have finite ({n + i | i. nat-case s' ω (n + i) = s} ∪ {..n})
    by (auto simp: ac-simps)
   then show finite {i. nat-case s' ω i = s}
```

by (*rule rev-finite-subset*) (*auto cong*: *conj-cong simp*: *not-le intro*!: *le-Suc-ex*)
qed
qed

lemma *AE-all-fair*:
$s' \in S \Longrightarrow AE\ \omega\ in\ paths\ s'.\ \forall\, s{\in}S.\ \forall\, t{\in}E\ s.\ fair\ s\ t\ (nat\text{-}case\ s'\ \omega)$
by (*intro AE-all-E AE-all-S AE-fair*)

lemma *fair-eq*: *fair sx tx* $\omega \longleftrightarrow$
$(\exists j.\ \forall\, i{\geq}j.\ \omega\ i \neq sx) \vee (\forall\, i.\ \exists j{\geq}i.\ \omega\ j = sx \wedge \omega\ (Suc\ j) = tx)$
unfolding *fair-def finite-nat-set-iff-bounded disj-not2*[*symmetric*]
by (*simp add*: *not-le not-less*) *blast*

2.9 *until*

definition (**in** −) *until* :: *'s set* ⇒ *'s set* ⇒ (*nat* ⇒ *'s*) *set* **where**
$until\ \Phi\ \Psi = \{\omega.\ \exists\, n.\ (\forall\, i{<}n.\ \omega\ i \in \Phi) \wedge \omega\ n \in \Psi\}$

lemma (**in** −) *measurable-until*[*measurable*]: {$\omega{\in}space\ (\Pi_M\ n{\in}UNIV{::}nat\ set.\ count\text{-}space\ S).\ \omega \in until\ \Phi\ \Psi\} \in sets\ (\Pi_M\ n{\in}UNIV{::}nat\ set.\ count\text{-}space\ S)$
unfolding *until-def* **by** *auto*

lemma *untilI*:
$(\bigwedge i.\ i < n \Longrightarrow \omega\ i \in \Phi) \Longrightarrow \omega\ n \in \Psi \Longrightarrow \omega \in until\ \Phi\ \Psi$
by (*auto simp*: *until-def*)

lemma *untilE*:
assumes ω: $\omega \in until\ \Phi\ \Psi$
obtains (*until*) *n* **where** $\bigwedge i.\ i < n \Longrightarrow \omega\ i \in \Phi - \Psi\ \omega\ n \in \Psi$
proof −
from ω **obtain** *n* **where** *n*: $\bigwedge i.\ i < n \Longrightarrow \omega\ i \in \Phi\ \omega\ n \in \Psi$
by (*auto simp*: *until-def*)
from *smallest*[**where** $P{=}\lambda n.\ \omega\ n \in \Psi$, *OF n*(*2*)] **guess** n' **.**
with *n*(*1*) *that*[*of n'*] **show** *thesis* **by** *auto*
qed

lemma *until-iff*:
$\omega \in until\ \Phi\ \Psi \longleftrightarrow (\exists\, n.\ (\forall\, i < n.\ \omega\ i \in \Phi - \Psi) \wedge \omega\ n \in \Psi)$
by (*auto elim*!: *untilE intro*: *untilI*)

lemma *nat-case-until-iff*[*simp*]:
$nat\text{-}case\ s\ \omega \in until\ \Phi\ \Psi \longleftrightarrow (s \in \Psi \vee (s \in \Phi \wedge \omega \in until\ \Phi\ \Psi))$
by (*auto simp add*: *all-conj-distrib set-eq-iff until-def gr0-conv-Suc split*: *nat.split*) *blast*

lemma *comb-seq-until*:
assumes ω: $\bigwedge j.\ j < i \Longrightarrow \omega\ j \in \Phi - \Psi$
shows $comb\text{-}seq\ i\ \omega\ \omega' \in until\ \Phi\ \Psi \longleftrightarrow \omega' \in until\ \Phi\ \Psi$
proof

assume $\omega' \in$ *until* Φ Ψ
from *untilE*[*OF this*] **guess** n **.**
with ω **show** *comb-seq i* ω $\omega' \in$ *until* Φ Ψ
by (*intro untilI*[*of i* + *n*]) (*auto split*: *split-comb-seq*)
next
assume *comb-seq i* ω $\omega' \in$ *until* Φ Ψ
from *untilE*[*OF this*] **guess** n **. note** n = *this*
moreover
from ω[*of n*] *n*(*2*) **have** $i \leq n$
by (*auto simp*: *not-less*[*symmetric*] *split*: *split-comb-seq-asm*)
moreover
{ fix j **assume** $j < n - i$
with *n*(*1*)[*of i* + *j*] **have** $\omega' j \in \Phi - \Psi$
by (*auto simp*: *less-diff-conv split*: *split-comb-seq-asm*) **}**
ultimately show $\omega' \in$ *until* Φ Ψ
by (*intro untilI*[*of n* − *i*]) (*auto split*: *split-comb-seq-asm*)
qed

lemma *single-K-measure-le-integral*:
assumes [*simp*]: $s \in S$ **and** $t \in E\ s$ **and** *nneg*: *AE t in K s*. $0 \leq f\ t$
and *int*[*measurable*]: *integrable* (*K s*) *f*
shows *measure* (*K s*) {*t*} ∗ *f t* ≤ (∫ *t*. *f t* ∂*K s*)
proof −
from ‹$t \in E\ s$› **have** [*simp*]: $t \in S$ **by** *auto*
have *measure* (*K s*) {*t*} ∗ *f t* = (∫ *t'*. *f t* ∗ *indicator* {*t*} *t'* ∂*K s*)
by (*simp add*: *K.emeasure-eq-measure integral-cmul-indicator*)
also have ... ≤ (∫ *t'*. *f t'* ∂*K s*)
using *nneg* **by** (*intro integral-mono-AE*) (*auto intro*: *int split*: *split-indicator*)
finally show *?thesis* **.**
qed

lemma *AE-paths-iff*:
s ∈ *S* ⟹ {ω∈*space S-seq*. *P* ω} ∈ *sets S-seq* ⟹
(*AE* ω *in paths s*. *P* ω) ⟷
(∀ω. (∀*j*<*i*. ω *j* ∈ *E* (*nat-case s* ω *j*)) ⟶ (*AE* ω′ *in paths* (*nat-case s* ω *i*).
P (*comb-seq i* ω ω′)))
(**is** - ⟹ - ⟹ - ⟷ (∀ω. *?S i s P* ω))
proof (*induct i arbitrary*: *P s*)
case *0* **then show** *?case*
by (*simp add*: *comb-seq-0 cong del*: *AE-cong*)
next
case (*Suc i*)
note *Suc.prems*(*1*)[*simp*] **and** *Suc.prems*(*2*)[*measurable*]
have (*AE* ω *in paths s*. *P* ω) ⟷ (∀ *t*∈*E s*. *AE* ω *in paths t*. *P* (*nat-case t* ω))
by (*subst AE-iterate*) (*auto simp*: *AE-K-iff cong del*: *AE-cong*)
also have ... ⟷ (∀ *t*∈*E s*. *All* (*?S i t* (λω. *P* (*nat-case t* ω))))
by (*intro ball-cong*[*OF refl*] *Suc*) *auto*
also have ... ⟷ *All* (*?S* (*Suc i*) *s P*)
proof (*intro iffI allI ballI*)

fix ω :: *nat* $\Rightarrow$ *'s* **assume** ($\forall$ *t*$\in$*E s. All* (*?S i t* ($\lambda\omega$. *P* (*nat-case t* ω))))
from *this*[*THEN bspec, of* ω *0, THEN spec, of* $\lambda i.$ ω (*Suc i*)]
show *?S* (*Suc i*) *s P* ω
by (*auto simp*: *nat-case-comb-seq' nat-case-idem all-less-Suc-split*
cong del: *AE-cong*)
next
fix *t* :: *'s* **and** ω :: *nat* $\Rightarrow$ *'s* **assume** $*$: *All* (*?S* (*Suc i*) *s P*) **and** *t*: *t* $\in$ *E s*
from $*$[*THEN spec, of nat-case t* ω] *t*
show *?S i t* ($\lambda\omega$. *P* (*nat-case t* ω)) ω
by (*simp add*: *nat-case-comb-seq' all-less-Suc-split cong del*: *AE-cong*)
qed
finally show *?case* **.**
qed

lemma *AE-nuntil-iff-not-reachable*:
assumes *s*[*simp*]: *s* $\in$ *S*
shows (*AE* ω *in paths s. nat-case s* ω $\notin$ *until* Φ Ψ) $\longleftrightarrow$
s $\notin$ Ψ $\wedge$ (*s* $\in$ Φ $\longrightarrow$ *reachable* (Φ $-$ Ψ) *s* $\cap$ Ψ = {})
proof $-$
have $\bigwedge\omega$. *nat-case s* ω $\notin$ *until* Φ Ψ $\longleftrightarrow$ ($\forall$ *n*. ($\forall$ *i*<*n. nat-case s* ω *i* $\in$ Φ $-$ Ψ) $\longrightarrow$ *nat-case s* ω *n* $\notin$ Ψ)
by (*auto simp*: *until-iff*)

let *?*ω = *nat-case s*
{ fix *x* **assume** *x*: *x* $\in$ *reachable* (Φ $-$ Ψ) *s* **and** *x* $\in$ Ψ *s* $\in$ Φ
from *x*[*THEN reachableE*] **guess** ω *n* **. note** ω = *this*
assume (*AE* ω *in paths s. ?*ω ω $\notin$ *until* Φ Ψ)
with ω ‹*s*$\in$$\Phi$› **have** *AE* ω' *in paths* (*?*ω ω (*Suc n*)). *?*ω (*comb-seq* (*Suc n*) ω ω') $\notin$ *until* Φ Ψ
by (*subst* (*asm*) *AE-paths-iff*[**where** *i*=*Suc n*]) (*auto elim*!: *allE*[*of* - ω])
moreover have $\bigwedge\omega'$. *?*ω (*comb-seq* (*Suc n*) ω ω') $\in$ *until* Φ Ψ
using ω ‹*x* $\in$ Ψ› ‹*s* $\in$ Φ›
by (*auto simp*: *until-def intro*!: *exI*[*of* - *Suc n*] *split*: *nat.split split-comb-seq*)
ultimately have *False*
by (*auto simp del*: *nat-case-until-iff simp add*: *AE-False*) **}**
moreover
{ assume *s*: *s* $\notin$ Ψ *reachable* (Φ $-$ Ψ) *s* $\cap$ Ψ = {}
have *AE* ω *in paths s. nat-case s* ω $\notin$ *until* Φ Ψ
using *AE-all-enabled*[*OF* ‹*s* $\in$ *S*›]
proof *eventually-elim*
fix ω **assume** *E*: $\forall$ *i*. ω *i* $\in$ *E* (*nat-case s* ω *i*)
then have $\bigwedge$*n*. ($\bigwedge$*i*. *i* < *n* $\Longrightarrow$ ω *i* $\in$ Φ $-$ Ψ) $\Longrightarrow$ ω *n* $\in$ *reachable* (Φ $-$ Ψ) *s*
by (*intro reachableI2*) *auto*
with *s* **have** ω $\notin$ *until* Φ Ψ
by (*auto elim*!: *untilE*)
then show *nat-case s* ω $\notin$ *until* Φ Ψ
using ‹*s* $\notin$ Ψ› **by** *auto*
qed }

ultimately show *?thesis*
by (*auto simp add*: *AE-False*)
qed

lemma *AE-until*:
assumes *s*: $s \in S$ $s \in \Phi$ **and** Φ: *finite* $(\Phi - \Psi)$ **and** *closed*: *reachable* $(\Phi - \Psi)$ $s \subseteq \Phi \cup \Psi$
assumes *enabled*: $\forall t \in$ *reachable* $(\Phi - \Psi)$ $s \cup \{s\} - \Psi$. *reachable* $(\Phi - \Psi)$ $t \cap \Psi \neq \{\}$
shows *AE* ω *in paths s. nat-case s* $\omega \in$ *until* Φ Ψ
using *AE-all-fair*[*OF* ‹$s \in S$›] *AE-all-enabled*[*OF* ‹$s \in S$›]
proof *eventually-elim*
fix ω **let** *?*ω = *nat-case s* ω **and** *?R* = *reachable* $(\Phi - \Psi)$ $s \cup \{s\} - \Psi$
assume *fair*: $\forall s \in S$. $\forall t \in E$ *s. fair s t ?*ω **and** E: $\forall i$. ω $i \in E$ (*?*ω i)

show *?*$\omega \in$ *until* Φ Ψ
proof (*rule ccontr*)
assume *?*$\omega \notin$ *until* Φ Ψ
then have *nuntil*: $\bigwedge n$. $(\bigwedge i$. $i < n \Longrightarrow$ *?*ω $i \in \Phi - \Psi) \Longrightarrow$ *?*ω $n \notin \Psi$
unfolding *until-iff* **by** *auto*
{ fix *n* **have** *?*ω $n \in \Phi - \Psi$
proof (*induct n rule*: *nat-less-induct*)
case (*1 n*)
with *nuntil*[*of n*] **have** *?*ω $n \notin \Psi$ **by** *auto*
moreover
{ fix *i* **have** $i < n - 1 \Longrightarrow \omega$ $i \in \Phi - \Psi$
using *1*[*THEN spec*, *of Suc i*] **by** *auto* **}**
with *E* **have** ω $(n - 1) \in$ *reachable* $(\Phi - \Psi)$ *s*
by (*intro reachableI2*) *auto*
with *closed* ‹$s \in \Phi$› **have** *?*ω $n \in \Phi \cup \Psi$
by (*cases n*) *auto*
ultimately show *?case* **by** *simp*
qed }
moreover
{ fix *n*
have ω $n \in$ *reachable* $(\Phi - \Psi)$ *s*
proof (*intro reachableI2*)
fix *i* **show** $i < n \Longrightarrow \omega$ $i \in \Phi - \Psi$
using ‹*?*ω (*Suc i*) $\in \Phi - \Psi$› **by** *auto*
qed (*insert E*, *auto*)
then have ω $n \in$ *reachable* $(\Phi - \Psi)$ $s \cup \{s\} - \Psi$
using ‹*?*ω (*Suc n*) $\in \Phi - \Psi$› **by** *auto* **}**
with *nuntil*[*of 0*] **have** $\bigwedge n$. *?*ω $n \in$ *?R*
by (*auto split*: *nat.split*)
moreover have *finite ?R*
using *finite.insertI*[*OF* Φ, *of s*] **by** (*rule rev-finite-subset*) (*insert closed*, *auto*)
moreover note *pigeonhole-infinite-rel*[*of UNIV ?R* λi *s.* *?*ω $i = s$] Φ

```
    ultimately obtain t where t ∈ ?R and t: infinite {i. ?ω i = t}
      by auto
    with enabled obtain t' where t': t' ∈ reachable (Φ − Ψ) t and t' ∈ Ψ
      by auto
    from ‹t ∈ ?R› have t ∈ S
      using closed s by auto

    from t' ‹t ∈ S› t have infinite {i. ?ω i = t'}
    proof (induct rule: reachable-induct-trans)
      case (start t s)
      with fair have infinite {i. ?ω i = s ∧ ω i = t}
        by (auto simp: fair-def)
      then have infinite (Suc ‘ {i. ω i = t})
        by (auto dest!: finite-imageD)
      moreover have Suc ‘ {i. ω i = t} ⊆ {i. ?ω i = t}
        by auto
      ultimately show ?case
        by (auto dest!: finite-subset)
    qed auto
    then have {i. ?ω i = t'} ≠ {}
      by (intro notI) simp
    then obtain i where ?ω i = t' by auto
    with ‹t' ∈ Ψ› ‹?ω i ∈ Φ − Ψ› show False by auto
  qed
qed

lemma AE-until-iff-reachable:
  assumes [simp]: s ∈ S finite (Φ − Ψ)
  shows (AE ω in paths s. nat-case s ω ∈ until Φ Ψ) ⟷
    (s ∈ Φ ∧ reachable (Φ − Ψ) s ⊆ Φ ∪ Ψ ∧
      (∀ t∈reachable (Φ − Ψ) s ∪ {s} − Ψ. reachable (Φ − Ψ) t ∩ Ψ ≠ {})) ∨ s ∈
Ψ
proof (intro iffI disjCI conjI)
  let ?ω = nat-case s
  assume until: AE ω in paths s. ?ω ω ∈ until Φ Ψ s ∉ Ψ
  then show s ∈ Φ by simp

  { fix t assume t: t ∈ reachable (Φ − Ψ) s
    from t[THEN reachableE] guess ω n . note ω = this
    with until ‹s∈Φ› have AE ω' in paths (?ω ω (Suc n)). comb-seq (Suc n) ω ω'
∈ until Φ Ψ
      by (subst (asm) AE-paths-iff[where i=Suc n]) (auto elim!: allE[of - ω])
    moreover have ⋀ω'. comb-seq (Suc n) ω ω' ∈ until Φ Ψ ⟷ nat-case t ω' ∈
until Φ Ψ
      using ω unfolding nat-case-comb-seq'
    by (subst comb-seq-Suc) (auto simp del: nat-case-until-iff intro!: comb-seq-until)
    moreover have ?ω ω (Suc n) = t
      using ω by simp
    ultimately have AE ω' in paths t. nat-case t ω' ∈ until Φ Ψ
```

by (*simp cong del*: *AE-cong*) **}**
note *shift-until* = *this*

{ fix *s* **assume** *reachable* (Φ − Ψ) *s* ∩ Ψ = {} *s* ∈ *S s* ∉ Ψ *s* ∈ Φ
with *AE-nuntil-iff-not-reachable*[*of s* Φ Ψ] ‹*s* ∉ Ψ› ‹*s* ∈ Φ›
have *AE* ω *in paths s. nat-case s* ω ∉ *until* Φ Ψ **by** *auto*
moreover assume *AE* ω *in paths s. nat-case s* ω ∈ *until* Φ Ψ
ultimately have *AE* ω *in paths s. False* **by** *eventually-elim auto*
then have *False* **by** *simp* **}**
note *not-reachable* = *this*

{ fix *t* **have** *t* ∈ *reachable* (Φ − Ψ) *s* ⟹ *t* ∉ Φ ∪ Ψ ⟹ *False*
using *shift-until*[*of t*] **by** *auto* **}**
then show *reachable* (Φ − Ψ) *s* ⊆ Φ ∪ Ψ **by** *auto*

show ∀ *t*∈*reachable* (Φ − Ψ) *s* ∪ {*s*} − Ψ. *reachable* (Φ − Ψ) *t* ∩ Ψ ≠ {}
proof *safe*
fix *t* **assume** *t* ∈ *reachable* (Φ − Ψ) *s t* ∉ Ψ
moreover then have *AE* ω′ *in paths t. nat-case t* ω′ ∈ *until* Φ Ψ
using *shift-until*[*of t*] **by** *simp*
moreover assume *reachable* (Φ − Ψ) *t* ∩ Ψ = {}
ultimately show *False* **using** *not-reachable*[*of t*] **by** *auto*
qed (*insert not-reachable*[*of s*] ‹*s* ∈ *S*› ‹*s* ∈ Φ› *until*, *auto*)
qed (*insert AE-until*[*of s* Φ Ψ], *auto*)

2.10 Hitting time (as natural number)

definition *hitting-time* :: *'s set* ⇒ (*nat* ⇒ *'s*) ⇒ *nat* **where**
hitting-time Φ ω = (*LEAST i.* ω *i* ∈ Φ)

lemma *measurable-hitting-time*[*measurable*]:
hitting-time Φ ∈ *measurable S-seq* (*count-space UNIV*)
unfolding *hitting-time-def*[*abs-def*] **by** *measurable*

lemma *hitting-time-eq*:
ω *n* ∈ Φ ⟹ (⋀*i. i* < *n* ⟹ ω *i* ∉ Φ) ⟹ *hitting-time* Φ ω = *n*
unfolding *hitting-time-def*
by (*rule Least-equality*) (*auto simp*: *not-less*[*symmetric*])

lemma *hitting-time-least*: *i* < *hitting-time* Φ ω ⟹ ω *i* ∉ Φ
unfolding *hitting-time-def* **by** (*auto dest*!: *not-less-Least*)

lemma
assumes *until*: ω ∈ *until S* Φ
shows *hitting-time-in*[*intro*]: ω (*hitting-time* Φ ω) ∈ Φ
proof −
from *untilE*[*OF until*] **guess** *n* **. note** *n* = *this*
moreover then have *n-eq*: *hitting-time* Φ ω = *n*
by (*intro hitting-time-eq*) *auto*

```
  ultimately show ω (hitting-time Φ ω) ∈ Φ
    by auto
qed

lemma hitting-time-nat-case-Suc:
  assumes ω ∈ until S Φ s ∉ Φ
  shows hitting-time Φ (nat-case s ω) = Suc (hitting-time Φ ω)
proof −
  have (LEAST i. (nat-case s ω) i ∈ Φ) = Suc (LEAST i. nat-case s ω (Suc i) ∈
Φ)
    using assms by (intro Least-Suc[of - Suc (hitting-time Φ ω)]) auto
  then show ?thesis unfolding hitting-time-def by simp
qed

lemma hitting-time-nat-case-0:
  s ∈ Φ ⟹ hitting-time Φ (nat-case s ω) = 0
  unfolding hitting-time-def by (auto intro!: Least-equality)

lemma positive-integral-hitting-time-finite:
  assumes [simp]: s ∈ S and Φ: finite (S − Φ)
  assumes until: AE ω in paths s. nat-case s ω ∈ until S Φ
  shows (∫⁺ ω. real (hitting-time Φ (nat-case s ω)) ∂paths s) ≠ ∞
proof cases
 assume s: s ∈ Φ with ‹s ∈ S› show ?thesis by (simp add: hitting-time-nat-case-0)
next
  assume s: s ∉ Φ

  let ?R = reachable (S − Φ) s ∪ {s} − Φ
  let ?P = λn t. P(ω in paths t. ∀ i<n. ω i ∈ S − Φ)
  have ∀ t∈?R. ∃ n. ?P n t < 1
  proof
    fix t assume t: t ∈ ?R
    then have t ∈ S by auto
    from t AE-until-iff-reachable[of s S Φ] ‹finite (S − Φ)› until ‹s ∉ Φ›
    obtain t′ where t′ ∈ reachable (S − Φ) t t′ ∈ Φ by auto
    from this ‹t ∈ S›
    have ∃ n. ¬ (AE ω in paths t. ∀ i<n. ω i ∈ S − Φ)
    proof (induct rule: reachable-induct-rev)
      case (start t)
      then show ?case
        using start ‹t ∈ S›
        by (auto simp add: AE-paths-0[of t λs. s ∉ Φ] AE-K-iff intro!: exI[of - Suc
0])
    next
      case (step t s)
      then obtain n where n: ¬ (AE ω in paths t. ∀ i<n. ω i ∈ S − Φ) by auto
      with step show ?case
        apply (intro exI[of - Suc n])
        apply (simp add: all-less-Suc-split del: AE-conj-iff)
```

```
      apply (auto simp add: AE-K-iff AE-iterate[OF ‹s∈S›])
      done
  qed
  then guess n ..
  then show ∃ n. ?P n t < 1
    using AE-in-set-eq-1[of {ω∈space (paths t). ∀ i<n. ω i ∈ S − Φ} t]
    by (auto simp add: less-le)
qed
from bchoice[OF this] guess N .. note N = this

have ?R ⊆ S − Φ by auto
with Φ have finite ?R by (auto dest: finite-subset)
have s ∈ ?R using ‹s ∉ Φ› by auto

def n ≡ Max (N ‘ ?R)
{ fix s assume s: s ∈ ?R
  with ‹finite ?R› have M: N s ≤ n by (auto intro!: Max-ge simp: n-def)
  then have ?P n s ≤ ?P (N s) s
    by (intro finite-measure-mono) auto
  also have ... < 1
    using N s by auto
  finally have ?P n s < 1 . }
note less-1 = this
def d ≡ Max (?P n‘?R)
with ‹finite ?R› ‹s∈?R› less-1 have d < 1 by (auto simp: Max-less-iff[THEN
iffD1])

have 0 < N s
proof (rule ccontr)
  assume ¬ 0 < N s
  then have N s = 0 by simp
  with N[THEN bspec, OF ‹s ∈ ?R›] prob-space show False by simp
qed
with ‹s ∈ ?R› ‹finite ?R› have 0 < n
  by (auto simp add: n-def Max-gr-iff intro!: bexI[of - s])

have d: ⋀s. s∈?R ⟹ ?P n s ≤ d
  using ‹finite ?R› ‹s∈?R› unfolding d-def by (auto intro!: Max-ge)

have 0 ≤ d
  using d[OF ‹s ∈ ?R›] by (auto intro: order-trans measure-nonneg)

{ fix ω t
  assume t: t < hitting-time Φ ω div n
  have ⋀a i j. j < n ⟹ a < i ⟹ a * n + 0 < n * i + j
    using ‹0 < n› by (intro add-less-le-mono) auto
  then have t < hitting-time Φ ω div n ⟶ t * n < hitting-time Φ ω
    using ‹0 < n› by (simp split: split-div)
  with t have ∀ i≤t*n. ω i ∉ Φ
```

```
    by (simp add: hitting-time-least) }
  note less-hitting-time = this

  { fix t
    def R ≡ ?R
    then have s ∈ R using s ‹s ∈ S› by auto
    then have emeasure (paths s) {ω∈space (paths s). ∀ i≤t * n. ω i ∉ Φ} ≤ d ^ t
    proof (induct t arbitrary: s)
      case (Suc t s')
      then have s: s' ∈ S unfolding R-def by auto
      let ?M = λω. emeasure (paths (nat-case s' ω n))
      let ?A = λω. {ω' ∈ space S-seq. (∀ i≤Suc t * n. comb-seq n ω ω' i ∉ Φ)}
      from ‹s'∈S› have emeasure (paths s') {ω ∈ space (paths s'). (∀ i≤Suc t * n.
ω i ∉ Φ)} =
          (∫⁺ω. ?M ω (?A ω) ∂paths s')
        by (intro emeasure-split-Collect) auto
      also have ... ≤ (∫⁺ω. ereal (d^t) * indicator {ω∈space (paths s'). ∀ i<n. ω
i ∉ Φ} ω ∂paths s')
        apply (rule positive-integral-mono-AE)
        using AE-space AE-all-enabled[OF ‹s'∈S›]
      proof eventually-elim
        fix ω assume ω: ω ∈ space (paths s') and E: ∀ i. ω i ∈ E (nat-case s' ω i)

        { assume ω: ⋀i. i < n ⟹ ω i ∈ S - Φ
          with E have ω (n - 1) ∈ reachable (S - Φ) s'
            by (intro reachableI2) (auto split: nat.split)
          with ω Suc(2) have nat-case s' ω n ∈ R
            unfolding R-def by (auto intro: reachable-trans split: nat.split) }
        note in-R = this

        have ?A ω ⊆ (if ∀ i<n. ω i ∉ Φ then {ω ∈ space (paths s). (∀ i≤t * n. ω i
∉ Φ)} else {})
          (is - ⊆ ?B)
          by (simp add: space-PiM all-plus-split comb-seq-add comb-seq-less)
        then have ?M ω (?A ω) ≤ ?M ω ?B
          by (intro emeasure-mono) auto
        also have ... ≤ ereal (d^t) * indicator {ω∈space (paths s'). ∀ i<n. ω i ∉
Φ} ω
          using Suc(1)[OF in-R] s ω
        by (auto simp: less-Suc-eq-le space-PiM PiE-iff split: split-indicator nat.split)
        finally show ?M ω (?A ω) ≤ ereal (d^t) * indicator {ω∈space (paths s').
∀ i<n. ω i ∉ Φ} ω .
      qed
      also have ... = d^t * emeasure (paths s') {ω∈space (paths s'). ∀ i<n. ω i ∉
Φ}
        using ‹0 ≤ d› by (intro positive-integral-cmult-indicator) auto
      also have ... ≤ ereal (d^t) * d
        using d[of s'] ‹0 ≤ d› Suc(2)
      by (intro ereal-mult-left-mono) (auto simp add: emeasure-eq-measure space-PiM
```

```
PiE-iff R-def cong: conj-cong)
      finally show ?case
        by (simp add: ac-simps)
    qed (insert measure-le-1, simp add: one-ereal-def) }
  note upper-limit = this

  have (∫⁺ ω. ereal (of-nat (hitting-time Φ (nat-case s ω))) ∂paths s) ≤
      (∫⁺ ω. ereal (of-nat (hitting-time Φ ω div n)) * n + Suc n ∂paths s)
    apply (rule positive-integral-mono-AE)
    using until
  proof eventually-elim
    fix ω assume nat-case s ω ∈ until S Φ
    then have ereal (of-nat (hitting-time Φ (nat-case s ω))) =
        ereal (of-nat (hitting-time Φ ω + 1))
      using hitting-time-nat-case-Suc[of ω Φ s] ‹s ∉ Φ› by auto
    also have ... ≤ ereal (of-nat ((hitting-time Φ ω div n + 1) * n + 1))
      using ‹0 < n›[THEN mod-le-divisor, of hitting-time Φ ω]
      by (simp del: of-nat-add add: div-mod-equality')
    also have ... = ereal (of-nat (hitting-time Φ ω div n)) * n + n + 1
      by (simp add: field-simps real-eq-of-nat of-nat-mult)
    finally show ereal (of-nat (hitting-time Φ (nat-case s ω))) ≤
        ereal (of-nat (hitting-time Φ ω div n)) * n + Suc n by simp
  qed
  also have ... = (∫⁺ ω. ereal (of-nat (hitting-time Φ ω div n)) ∂paths s) * n +
Suc n
    using emeasure-space-1[of s]
    by (simp add: positive-integral-add positive-integral-multc mult-nonneg-nonneg
          del: plus-ereal.simps times-ereal.simps)
  also have ... = (∑ t. emeasure (paths s) {ω∈space (paths s). t < hitting-time
Φ ω div n}) * n + Suc n
    by (simp add: positive-integral-nat-function)
  also have ... ≤ (∑ t. emeasure (paths s) {ω∈space (paths s). ∀ i≤t * n. ω i ∉
Φ}) * n + Suc n
    by (intro ereal-add-mono ereal-mult-right-mono suminf-le-pos emeasure-mono-AE)
       (auto simp: less-hitting-time)
  also have ... ≤ (∑ t. ereal (d^t)) * n + Suc n
    by (intro ereal-add-mono ereal-mult-right-mono suminf-le-pos upper-limit) auto
  also have ... < ∞
  proof −
    have ∃ r. (op ^ d) sums r
     using ‹0 ≤ d› ‹d < 1› summable-geometric[of d] by (auto simp: summable-def)
    then guess r ..
    then have ereal r = (∑ t. ereal (d^t))
      by (intro sums-unique sums-ereal[THEN iffD2])
    then show ?thesis
      by auto
  qed
  finally show ?thesis
    by (simp add: real-eq-of-nat)
```

qed

end

lemma *measurable-component-singleton-const*[*measurable-app*]:
assumes *f*: $f \in$ *measurable N* (Pi_M *I* ($\lambda i.$ *M*))
assumes *i*: $i \in I$
shows ($\lambda x.$ (f *x*) *i*) $\in$ *measurable N M*
using *measurable-compose*[*OF f measurable-component-singleton*, *OF i*] **.**

lemma *measurable-abs-UNIV*[*measurable*]:
($\bigwedge n.$ ($\lambda\omega.$ *f n* ω) $\in$ *measurable M* (*N n*)) $\Longrightarrow$ ($\lambda\omega$ *n*. *f n* ω) $\in$ *measurable M* (*PiM UNIV N*)
by (*intro measurable-PiM-single*) (*auto dest*: *measurable-space*)

lemma *sets-UNIV* [*measurable* (*raw*)]: *A* $\in$ *sets* (*count-space UNIV*)
by *simp*

end

theory *Constructing-Markov-Chain*
imports *Discrete-Markov-Kernel*
begin

3 Markov Chain as Stochastic Process

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

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

definition S' = *Some* ' *S* $\cup$ {*None*}

lemma *None-in-S'*[*simp*]: *None* ∈ *S'*
by (*simp add*: *S'-def*)

lemma *S-not-empty*: *S* ≠ {}
using *X*[*THEN measurable-space*] *M.not-empty* **by** *auto*

lemma *S'-not-empty*: *S'* ≠ {}
by (*auto simp*: *S'-def*)

lemma *measurable-Some*[*measurable*]: *Some* ∈ *measurable* (*count-space S*) (*count-space S'*)
by (*auto simp*: *S'-def*)

lemma *countable-S'*: *countable S'*
by (*auto simp*: *S S'-def*)

definition *with P f d* = (*if* ∃ *x*. *P x then f* (*SOME x*. *P x*) *else d*)

lemma *withI*[*case-names default exists*]:
((⋀*x*. ¬ *P x*) ⟹ *Q d*) ⟹ (⋀*x*. *P x* ⟹ *Q* (*f x*)) ⟹ *Q* (*with P f d*)
unfolding *with-def* **by** (*auto intro*: *someI2*)

3.1 Construct K'

primrec *K'* :: *'s option* ⇒ *'s option measure* **where**
K' None = *distr M* (*count-space S'*) (λω. *Some* (*X 0* ω))
| *K'* (*Some s*) =
with (λ*n*. *0* < *P*(ω *in M*. *X n* ω = *s*))
(λ*n*. *distr* (*uniform-measure M* {ω∈*space M*. *X n* ω = *s*}) (*count-space S'*) (λω. *Some* (*X* (*Suc n*) ω)))
(*point-measure S'* (λ*x*. *if x* = *Some* (*SOME s*. *s* ∈ *S*) *then 1 else 0*))

lemma *sets-K*[*simp*]: *sets* (*K' s*) = *Pow S'*
proof (*cases s*)
case (*Some s'*) **then show** *?thesis*
apply *simp*
apply (*rule withI*)
apply (*simp-all add*: *sets-point-measure*)
done
qed *simp*

lemma *space-K*[*simp*]: *space* (*K' s*) = *S'*
using *sets-eq-imp-space-eq*[*of K' s count-space S'*] **by** *simp*

3.2 Equations for K'

lemma *emeasure-K-S*[*simp*]:
assumes *0* < *P*(ω *in M*. *X n* ω = *s*)
shows *emeasure* (*K'* (*Some s*)) {*Some t*} = *P*(ω *in M*. *X* (*Suc n*) ω = *t* | *X n* ω = *s*)

```
  unfolding K'.simps
proof (induct rule: withI)
  case default with assms show ?case by simp
next
  case (exists m)
  let ?M = distr (uniform-measure M {ω ∈ space M. X m ω = s}) (count-space
S') (λω. Some (X (Suc m) ω))
  show ?case
  proof cases
    assume t ∈ S
    then have t: Some t ∈ S'
      by (simp add: S'-def)
    have emeasure ?M {Some t} = P(ω in M. X (Suc m) ω = t | X m ω = s)
      using exists
      by (auto simp add: emeasure-distr t cond-prob-def emeasure-eq-measure
            intro!: arg-cong[where f=prob])
    also have ... = P(ω in M. X (Suc n) ω = t | X n ω = s)
      using assms exists by (auto intro!: TH)
    finally show ?thesis .
  next
    assume t ∉ S
    then have emeasure ?M {Some t} = 0
      by (intro emeasure-notin-sets) (auto simp: S'-def)
    moreover
    have *: {ω ∈ space M. X (Suc n) ω = t ∧ X n ω = s} = {}
      using ‹t ∉ S› measurable-space[OF X] by auto
    have 0 = P(ω in M. X (Suc n) ω = t | X n ω = s)
      by (simp add: cond-prob-def *)
    ultimately show ?thesis
      by simp
  qed
qed

lemma emeasure-K-None:
  emeasure (K' (Some s)) {None} = 0
 unfolding K'.simps by (rule withI) (auto simp: emeasure-point-measure emeasure-distr
vimage-def)

lemma emeasure-K-init[simp]: emeasure (K' None) {t} = P(ω in M. Some (X 0
ω) = t)
proof cases
  assume t ∉ S'
  then have *: {ω∈space M. Some (X 0 ω) = t} = {}
    using measurable-space[OF X] by (auto simp: S'-def)
  from ‹t ∉ S'› have emeasure (distr M (count-space S') (λω. Some (X 0 ω)))
{t} = 0
    by (intro emeasure-notin-sets) (auto simp: S'-def)
  then show ?thesis
    by (simp add: *)
```

qed (*auto simp add*: *emeasure-distr emeasure-eq-measure intro*!: *arg-cong*[**where** *f*=*prob*])

sublocale *K*!: *prob-space K' s* **for** *s*
proof (*cases s*)
  case (*Some s'*)
  show *prob-space* (*K' s*)
    unfolding *Some K'.simps*
  proof (*induct rule*: *withI*)
    case (*exists n*) **then show** *?case*
      by (*intro prob-space.prob-space-distr prob-space-uniform-measure*)
        (*auto simp*: *emeasure-eq-measure*)
  qed (*default*, *simp add*: *space-point-measure emeasure-point-measure someI-in S'-def S-not-empty*)
qed (*auto intro*!: *prob-space-distr*)

sublocale *MC*: *Discrete-Markov-Kernel S' K'*
  by *default* (*auto simp add*: *S'-def S*)

declare *K'.simps*[*simp del*]

3.3 Markov Kernel's Path Space equals Stochastic Process

lemma *paths-None-eq*:
  paths None = *distr M S-seq* (λω *n*. *Some* (*X n* ω)) (**is** *?L* = *?R*)
proof (*rule nat-eq-cyl*[*OF countable-S'*])
  interpret *R*!: *prob-space ?R*
    by (*intro M.prob-space-distr*) *simp*

  show *sets ?L* = *sets MC.S-seq sets ?R* = *sets MC.S-seq emeasure ?R* (*space MC.S-seq*) ≠ ∞
    by *simp-all*

  fix ω' :: *nat* ⇒ *'s option* **and** *n* :: *nat* **assume** ω': ⋀*i*. ω' *i* ∈ *S'*
  then have *emeasure ?L* {ω ∈ *space ?L*. ∀ *i*<*Suc n*. ω *i* = ω' *i*} =
    P(ω *in M*. *Some* (*X 0* ω) = ω' *0*) ∗ (∏ *i*<*n*. *emeasure* (*K'* (ω' *i*)) {ω' (*Suc i*)})
    by (*subst emeasure-path*) (*auto simp*: *lessThan-Suc-eq-insert-0 zero-notin-Suc-image setprod-reindex*)
  also have *P*(ω *in M*. *Some* (*X 0* ω) = ω' *0*) ∗ (∏ *i*<*n*. *emeasure* (*K'* (ω' *i*)) {ω' (*Suc i*)})
    = *emeasure M* {ω ∈ *space M*. ∀ *i*≤*n*. *Some* (*X i* ω) = ω' *i*}
  proof *cases*
    assume ∗: ∃ *i*≤*n*. ω' *i* = *None*
    with *smallest*[*of* λ*i*. ω' *i* = *None*]
    obtain *i* **where** *i*: ω' *i* = *None* ⋀*j*. *j* < *i* ⟹ ω' *j* ≠ *None i* ≤ *n*
      by (*metis order-trans*)

    then have {ω ∈ *space M*. ∀ *i*≤*n*. *Some* (*X i* ω) = ω' *i*} = {}

by *auto*
then have *emeasure M* {$\omega \in$ *space M*. $\forall i{\leq}n$. *Some* (*X i* ω) = ω' *i*} = *0*
by (*simp del*: *Collect-empty-eq*)
also have $\mathcal{P}$(ω *in M*. *Some* (*X 0* ω) = ω' *0*) * ($\prod i{<}n$. *emeasure* (*K'* (ω' *i*)) {ω' (*Suc i*)}) = *0*
proof (*cases i*)
case *0* **with** *i* **show** *?thesis*
by *simp*
next
case (*Suc j*)
with *i* **have** ω' *j* $\neq$ *None* **by** *auto*
with *Suc i*(*1*) **have** *emeasure* (*K'* (ω' *j*)) {ω' (*Suc j*)} = *0*
by (*auto simp add*: *emeasure-K-None*)
with *Suc i*(*3*) **show** *?thesis*
by (*auto simp*: *setprod-ereal-0*)
qed
finally show *?thesis* **.**
next
assume ¬ ($\exists i{\leq}n$. ω' *i* = *None*)
then obtain ω **where** ω: $\bigwedge i$. *i* $\leq$ *n* $\Longrightarrow$ ω' *i* = *Some* (ω *i*)
by *atomize-elim* (*rule choice*, *auto*)

have $\mathcal{P}$(ω' *in M*. *X 0* ω' = ω *0*) * ($\prod i{<}n$. *emeasure* (*K'* (*Some* (ω *i*))) {*Some* (ω (*Suc i*))}) =
emeasure M {$\omega' \in$ *space M*. $\forall i{\leq}n$. *X i* ω' = ω *i*}
proof (*induct n*)
case *0* **then show** *?case*
by (*simp add*: *M.emeasure-eq-measure*)
next
case (*Suc n*)
have *le*: $\mathcal{P}$(ω' *in M*. *X* (*Suc n*) ω' = ω (*Suc n*) $\wedge$ ($\forall i{\leq}n$. *X i* ω' = ω *i*)) $\leq$ $\mathcal{P}$(ω' *in M*. $\forall i{\leq}n$. *X i* ω' = ω *i*)
by (*auto intro*!: *M.finite-measure-mono*)

have $\mathcal{P}$(ω' *in M*. *X 0* ω' = ω *0*) * ($\prod i{<}$*Suc n*. *emeasure* (*K'* (*Some* (ω *i*))) {*Some* (ω (*Suc i*))}) =
emeasure M {$\omega' \in$ *space M*. $\forall i{\leq}n$. *X i* ω' = ω *i*} * *emeasure* (*K'* (*Some* (ω *n*))) {*Some* (ω (*Suc n*))}
unfolding *Suc*[*symmetric*] **by** (*simp add*: *lessThan-Suc zero-notin-Suc-image setprod-reindex ac-simps*)
also have ... = *emeasure M* {$\omega' \in$ *space M*. $\forall i{\leq}n$. *X i* ω' = ω *i*} *
$\mathcal{P}$(ω' *in M*. *X* (*Suc n*) ω' = ω (*Suc n*) | $\forall i{\leq}n$. *X i* ω' = ω *i*)
proof (*intro ereal-left-mult-cong*)
assume *emeasure M* {$\omega' \in$ *space M*. $\forall i{\leq}n$. *X i* ω' = ω *i*} $\neq$ *0*
then have $**$: *0* < $\mathcal{P}$(ω' *in M*. $\forall i{\leq}n$. *X i* ω' = ω *i*)
by (*simp add*: *Ball-def M.emeasure-eq-measure measure-not-0*)
also have ... $\leq$ $\mathcal{P}$(ω' *in M*. *X n* ω' = ω *n*)
by (*auto intro*!: *M.finite-measure-mono*)
finally have $*$: *0* < $\mathcal{P}$(ω' *in M*. *X n* ω' = ω *n*)

by *simp*

have *emeasure* $(K'\ (Some\ (\omega\ n)))\ \{Some\ (\omega\ (Suc\ n))\} = \mathcal{P}(\omega'\ in\ M.\ X\ (Suc\ n)\ \omega' = \omega\ (Suc\ n)\ |\ X\ n\ \omega' = \omega\ n\ )$
using $*$ **by** (*rule emeasure-K-S*)
also have $\ldots = \mathcal{P}(\omega'\ in\ M.\ X\ (Suc\ n)\ \omega' = \omega\ (Suc\ n)\ |\ \forall i{\leq}n.\ X\ i\ \omega' = \omega\ i\ )$
using $**$ **by** (*auto intro*!: *MC*[*symmetric*])
finally show *emeasure* $(K'\ (Some\ (\omega\ n)))\ \{Some\ (\omega\ (Suc\ n))\} = \mathcal{P}(\omega'\ in\ M.\ X\ (Suc\ n)\ \omega' = \omega\ (Suc\ n)\ |\ \forall i{\leq}n.\ X\ i\ \omega' = \omega\ i\ )$
using ω **by** (*simp add*: *Ball-def*)
qed
also have $\ldots =$ *emeasure* $M\ \{\omega' \in space\ M.\ \forall i{\leq}Suc\ n.\ X\ i\ \omega' = \omega\ i\}$
using *le measure-nonneg*[*of* $M\ \{\omega' \in space\ M.\ X\ (Suc\ n)\ \omega' = \omega\ (Suc\ n) \wedge (\forall i{\leq}n.\ X\ i\ \omega' = \omega\ i)\}$]
by (*simp add*: *M.emeasure-eq-measure emeasure-distr-Collect cond-prob-def Ball-def le-Suc-eq all-conj-distrib conj-commute*)
finally show *?case* **.**
qed
with ω **show** *?thesis*
by *simp*
qed
finally show *emeasure ?L* $\{\omega \in space\ ?L.\ \forall i{\leq}n.\ \omega\ i = \omega'\ i\} =$ *emeasure ?R* $\{\omega \in space\ ?R.\ \forall i{\leq}n.\ \omega\ i = \omega'\ i\}$
by (*subst emeasure-distr-Collect'*) (*auto simp*: *less-Suc-eq-le*)
qed

end

4 Discrete Markov Kernel Constructs a DTMC

context *Discrete-Markov-Kernel*
begin

lemma *split-at*:
assumes [*simp*]: $s \in S$ **and** *2*: $\mathcal{P}(\omega'\ in\ paths\ s.\ \omega'\ n = \omega\ n) \neq 0$
shows $\mathcal{P}(\omega'\ in\ paths\ s.\ \omega'\ (Suc\ n) = s'\ |\ \omega'\ n = \omega\ n\ ) = \mathcal{P}(\omega'\ in\ paths\ (\omega\ n).\ \omega'\ 0 = s')$
proof −
have $\mathcal{P}(\omega'\ in\ paths\ s.\ \omega'\ (Suc\ n) = s' \wedge \omega'\ n = \omega\ n) =$
$(\int \omega'.\ \mathcal{P}(\omega'\ in\ paths\ (\omega\ n).\ \omega'\ 0 = s') * indicator\ \{\omega' \in space\ (paths\ s).\ \omega'\ n = \omega\ n\}\ \omega'\ \partial paths\ s)$
by (*subst prob-split-Collect*[**where** *i*=*Suc n*])
(*auto intro*!: *integral-cong split*: *split-indicator*
simp: *comb-seq-less less-Suc-eq-le comb-seq-id*
simp del: *integral-cmult*)
also have $\ldots = \mathcal{P}(\omega'\ in\ paths\ (\omega\ n).\ \omega'\ 0 = s') * \mathcal{P}(\omega'\ in\ paths\ s.\ \omega'\ n = \omega\ n)$
by (*simp add*: *emeasure-eq-measure*)
finally show *?thesis*

using *2* **unfolding** *cond-prob-def* **by** *simp*
qed

lemma *is-DTMC*:
assumes [*simp*]: $s \in S$
shows *Discrete-Time-Markov-Chain* (*paths s*) *S* ($\lambda n\ \omega.\ \omega\ n$)
proof
show *countable S*
by (*rule countable-space*)
show $\bigwedge t.\ (\lambda\omega.\ \omega\ t) \in$ *measurable* (*paths s*) (*count-space S*)
by *simp*
next
fix *n* :: *nat* **and** $\omega\ s'$
assume *1*: $\mathcal{P}(\omega'$ *in paths s*. $\forall\, t{\leq}n.\ \omega'\ t = \omega\ t\ ) \neq 0$
then have *2*: $\mathcal{P}(\omega'$ *in paths s*. $\omega'\ n = \omega\ n\ ) \neq 0$
by (*rule finite-measure-not-0-imp-not-0*) *auto*
have $\mathcal{P}(\omega'$ *in paths s*. ω' (*Suc n*) $= s' \wedge (\forall\, t{\leq}n.\ \omega'\ t = \omega\ t)) =$
$(\int \omega'.\ \mathcal{P}(\omega'$ *in paths* (ω *n*). $\omega'\ 0 = s')$ $*$ *indicator* $\{\omega'{\in}$*space* (*paths s*). $\forall\, t{\leq}n.\ \omega'\ t = \omega\ t\}$ ω' ∂*paths s*)
by (*subst prob-split-Collect*[**where** *i*=*Suc n*])
(*auto intro*!: *integral-cong split*: *split-indicator*
simp: *comb-seq-less less-Suc-eq-le comb-seq-id prob-eq-0*
simp del: *integral-cmult*)
also have ... $= \mathcal{P}(\omega'$ *in paths* (ω *n*). $\omega'\ 0 = s')$ $*$ $\mathcal{P}(\omega'$ *in paths s*. $\forall\, t{\leq}n.\ \omega'\ t = \omega\ t)$
by (*simp add*: *emeasure-eq-measure*)
finally have $\mathcal{P}(\omega'$ *in paths s*. ω' (*Suc n*) $= s' \mid (\forall\, t{\leq}n.\ \omega'\ t = \omega\ t)\ ) = \mathcal{P}(\omega'$ *in paths* (ω *n*). $\omega'\ 0 = s')$
using *1* **unfolding** *cond-prob-def* **by** *simp*
then show $\mathcal{P}(\omega'$ *in paths s*. ω' (*Suc n*) $= s' \mid (\forall\, t{\leq}n.\ \omega'\ t = \omega\ t)\ ) = \mathcal{P}(\omega'$ *in paths s*. ω' (*Suc n*) $= s' \mid \omega'\ n = \omega\ n\ )$
using *2 split-at*[*OF* ‹*s*∈*S*›, *of n* ω s'] **by** *simp*
next
fix *n m a b*
assume *1*: $\mathcal{P}(\omega'$ *in paths s*. $\omega'\ m = a\ ) \neq 0$
assume *2*: $\mathcal{P}(\omega'$ *in paths s*. $\omega'\ n = a\ ) \neq 0$
show $\mathcal{P}(\omega'$ *in paths s*. ω' (*Suc n*) $= b \mid \omega'\ n = a\ ) = \mathcal{P}(\omega'$ *in paths s*. ω' (*Suc m*) $= b \mid \omega'\ m = a\ )$
using *split-at*[*OF* ‹*s*∈*S*› *1*] *split-at*[*OF* ‹*s*∈*S*› *2*] **by** *simp*
qed

end

end

theory *Classifying-Markov-Chains*
imports *Discrete-Markov-Kernel Constructing-Markov-Chain*
begin

lemma *int-cases'*: $(\bigwedge n.\ x = int\ n \Longrightarrow P) \Longrightarrow (\bigwedge n.\ x = -\ int\ n \Longrightarrow P) \Longrightarrow P$
by (*metis int-cases*)

lemma *nat-abs-int-diff*: $nat\ |int\ a - int\ b| = (if\ a \leq b\ then\ b - a\ else\ a - b)$
by *auto*

lemma *nat-int-add*: $nat\ (int\ a + int\ b) = a + b$
by *auto*

context *Discrete-Markov-Kernel*
begin

5 Analyzing Markov Chains

5.1 Accessible

inductive *accessible* **for** s **where**
start: $s \in S \Longrightarrow accessible\ s\ s$
| *step*: $accessible\ s\ t \Longrightarrow t' \in E\ t \Longrightarrow accessible\ s\ t'$

lemma *accessible-in-S1*: **assumes** *accessible s t* **shows** $s \in S$
using *assms* **by** *induct auto*

lemma *accessible-in-S2*: **assumes** *accessible s t* **shows** $t \in S$
using *assms* **by** *induct auto*

lemma *accessible-trans*:
assumes *accessible s t accessible t t'* **shows** *accessible s t'*
using *assms*(*2*,*1*) **by** *induct* (*auto intro*: *accessible.step*)

lemma *accessible-single*:
$s \in S \Longrightarrow t \in E\ s \Longrightarrow accessible\ s\ t$
by (*blast intro*: *accessible.intros*)

lemma *accessible-induct-trans*[*consumes 1*, *case-names start step trans*]:
assumes *t*: *accessible s t*
assumes *1*: $\bigwedge s.\ s \in S \Longrightarrow P\ s\ s$
assumes *2*: $\bigwedge s\ t.\ s \in S \Longrightarrow t \in E\ s \Longrightarrow P\ s\ t$
assumes *3*: $\bigwedge t\ t'\ s.\ accessible\ s\ t' \Longrightarrow P\ s\ t' \Longrightarrow accessible\ t'\ t \Longrightarrow P\ t'\ t \Longrightarrow P\ s\ t$
shows $P\ s\ t$
using *t* **by** *induct* (*blast intro*: *1 2 3 accessible.intros accessible-in-S2*)+

lemma *accessible-step-rev*:
assumes *accessible s t s* $\in E\ s'\ s' \in S$ **shows** *accessible s' t*
using *assms accessible-trans*[*of s s' t*] **by** (*auto intro*: *accessible-single accessible-trans*)

lemma *accessible-rev*:

assumes t: *accessible s t*
obtains (*start*) $t = s \;\; s \in S \mid$ (*step*) s' **where** *accessible s' t s'* $\in E\ s\ s \in S$
using t **by** *induct* (*auto intro*: *accessible.intros*)

lemma *accessible-induct-rev*[*consumes 1*, *case-names start step*]:
assumes t: *accessible s t*
assumes *1*: $P\ t$
assumes *2*: $\bigwedge t'\ s.\ t' \in E\ s \Longrightarrow$ *accessible t' t* $\Longrightarrow P\ t' \Longrightarrow P\ s$
shows $P\ s$
using t
proof *cases*
case (*step s'*)
then have *accessible s' t P s'*
by (*auto intro*: *accessible-single accessible-in-S2 2 1 accessible.intros*)
with *step*(*1*) **show** *?thesis*
by (*induct rule*: *accessible-induct-trans*) (*auto intro*: *1 2 accessible-trans*)
qed (*insert 1*, *simp*)

lemma *AE-accessible*:
assumes x: $x \in S$
shows *AE* ω *in paths* $x.\ \forall m.$ *accessible* x (*nat-case* $x\ \omega\ m$)
using *AE-all-enabled*[*OF x*] *AE-space*
proof (*eventually-elim*, *safe*)
fix $\omega\ m$ **assume** ω: $\forall i.\ \omega\ i \in E$ (*nat-case x* ω *i*) $\omega \in$ *space* (*paths x*)
show *accessible x* (*nat-case x* ω m)
proof (*induct m*)
case *0* **then show** *?case*
using $\omega(2)$ **by** (*auto intro*!: *accessible.start split*: *nat.split simp*: *space-PiM* ‹$x \in S$›)
next
case (*Suc m'*) **with** $\omega(1)$[*THEN spec*, *of m'*] **show** *?case*
by (*auto intro*: *accessible.step*)
qed
qed

5.2 Communicating

definition *communicating* :: ('*s* × '*s*) *set* **where**
communicating = {(*s*, *t*). *accessible s t* ∧ *accessible t s*}

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

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

lemma *communicating-in-S*:

assumes $(x, y) \in communicating$ **shows** $x \in S\ y \in S$
using *assms* **by** (*auto simp add*: *communicating-def dest*: *accessible-in-S1 accessible-in-S2*)

lemma *C-subset-S*: $C \in S\ //\ communicating \Longrightarrow C \subseteq S$
by (*auto elim*!: *quotientE simp*: *communicating-in-S*)

lemma *equiv-communicating*: *equiv S communicating*
by (*auto simp*: *equiv-def sym-def communicating-def refl-on-def accessible-in-S1 accessible-in-S2 accessible.start trans-def intro*: *accessible-trans*)

lemma *communicatingD1*:
$C \in S\ //\ communicating \Longrightarrow (a, b) \in communicating \Longrightarrow a \in C \Longrightarrow b \in C$
by (*auto elim*!: *quotientE*) (*auto simp add*: *communicating-def intro*: *accessible-trans*)

lemma *communicatingD2*:
$C \in S\ //\ communicating \Longrightarrow (a, b) \in communicating \Longrightarrow b \in C \Longrightarrow a \in C$
by (*auto elim*!: *quotientE*) (*auto simp add*: *communicating-def intro*: *accessible-trans*)

lemma *irreducible-MC*: $(\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow accessible\ x\ y) \Longrightarrow S \in S$ // *communicating*
by (*auto simp*: *quotient-def Image-def communicating-def accessible-in-S1 intro*!: *bexI*[*of - s0*] *s0*)

lemma *not-empty-irreducible*:
$C \in S\ //\ communicating \Longrightarrow C \neq \{\}$
apply (*auto simp*: *quotient-def Image-def*)
apply (*erule-tac x*=*x* **in** *allE*)
apply (*simp add*: *communicating-def accessible.start*)
done

5.3 Time-Bounded Quantities

definition $p\ x\ y\ n = \mathcal{P}(\omega\ in\ paths\ x.\ nat\text{-}case\ x\ \omega\ n = y)$

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

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 prob-space del*: *space-paths*)

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

lemma *p-Suc*:
assumes $x \in S$

shows $p\ x\ y\ (Suc\ n) = (\int x'.\ p\ x'\ y\ n\ \partial K\ x)$
unfolding *p-def* **by** (*subst prob-iterate-Collect*[*OF* ‹$x \in S$›]) *simp-all*

lemma *accessibleI-pos*:
$x \in S \Longrightarrow y \in S \Longrightarrow 0 < p\ x\ y\ n \Longrightarrow$ *accessible* $x\ y$
proof (*induct n arbitrary*: x)
case *0* **then show** *?case* **by** (*simp add*: *p-0 accessible.start split*: *split-if-asm*)
next
case (*Suc n*)
then have *less*: $0 < (\int x'.\ p\ x'\ y\ n\ \partial K\ x)$
by (*simp add*: *p-Suc*)
have $\exists x' \in E\ x.\ 0 < p\ x'\ y\ n$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *AE* x' *in K* $x.\ p\ x'\ y\ n = 0$
using ‹$x \in S$› **by** (*simp add*: *AE-K-iff p-nonneg less-le*)
then have $(\int x'.\ p\ x'\ y\ n\ \partial K\ x) = (\int x'.\ 0\ \partial K\ x)$
using ‹$x \in S$› **by** (*intro integral-cong-AE*) *simp*
with *less* **show** *False* **by** *simp*
qed
with *Suc* **show** *?case*
by (*auto intro*: *accessible-step-rev*)
qed

lemma *prob-reachable-le*:
assumes [*simp*]: $x \in S\ y \in S\ w \in S$ **and** $m \leq n$
shows $p\ x\ y\ m * p\ y\ w\ (n - m) \leq p\ x\ w\ n$
proof −
have *emeasure* (*paths x*) {$\omega \in$*space* (*paths x*). *nat-case x* $\omega\ n = w$} =
$(\int^+\omega.$ *emeasure* (*paths* (*nat-case x* $\omega\ m$)) {$\omega' \in$*space* (*paths* (*nat-case x* ω m)). *comb-seq* (*Suc m*) (*nat-case x* ω) $\omega'\ n = w$} ∂*paths x*)
by (*subst emeasure-split-Collect*[**where** i=m]) (*simp-all add*: *nat-case-comb-seq'*)
also have $\ldots \geq (\int^+\omega.$ *emeasure* (*paths y*) {$\omega' \in$*space* (*paths y*). *nat-case y* ω' $(n - m) = w$} $*$
indicator {$\omega \in$*space* (*paths x*). *nat-case x* $\omega\ m = y$ } ω ∂*paths x*) (**is** - $\geq$ *?rhs*)
using ‹$m \leq n$›
apply (*auto intro*!: *positive-integral-mono emeasure-mono simp del*: *space-paths split*: *split-indicator*)
apply (*simp add*: *comb-seq-def less-Suc-eq-le minus-nat.diff-Suc split*: *nat.split split-indicator*)
unfolding *space-paths*
apply (*intro predE pred-intros-logic pred-eq-const1*)
apply (*rule measurable-component-singleton'*[**where** I=*UNIV* **and** M=λx. *count-space UNIV*])
apply (*rule measurable-comb-seq'*)
apply *simp-all*
done
also (*xtrans*) **have** *?rhs* = *emeasure* (*paths y*) {$\omega' \in$*space* (*paths y*). *nat-case y* ω' $(n - m) = w$} $*$

emeasure (*paths x*) {ω∈*space* (*paths x*). *nat-case x ω m* = *y* }
by (*subst positive-integral-cmult-indicator*) *auto*
finally show *?thesis*
by (*simp add*: *emeasure-eq-measure ac-simps p-def*)
qed

lemma *accessibleD-pos*:
assumes *accessible x y*
shows ∃ *n*. *0* < *p x y n*
using *assms* **proof** *induction*
case *start* **with** *prob-space*[*of x*] **show** *?case*
by (*auto intro*!: *exI*[*of* - *0*])
next
case (*step w y*)
then obtain *n* **where** *0* < *p x w n*
and *0* < *measure* (*K w*) {*y*}
and [*simp*]: *x* ∈ *S y* ∈ *S w* ∈ *S*
by (*auto simp*: *E-def K*.*emeasure-eq-measure measure-nonneg less-le accessible-in-S1 accessible-in-S2*)
then have *0* < *p x w n* ∗ *measure* (*K w*) {*y*}
by (*intro mult-pos-pos*)
also have ... = *p x w n* ∗ *p w y* (*Suc 0*)
using *prob-paths-0*[*of w* λ*s*. *s* = *y*] **by** (*simp add*: *p-def cong*: *conj-cong*)
also have ... ≤ *p x y* (*Suc n*)
using *prob-reachable-le*[*of x w y n Suc n*] **by** *simp*
finally show *?case* **..**
qed

5.3.1 Accessibility as probability on traces

lemma *accessible-iff*: *accessible x y* ⟷ *x* ∈ *S* ∧ *y* ∈ *S* ∧ (∃ *n*. *0* < *p x y n*)
by (*blast intro*: *accessibleD-pos accessibleI-pos accessible-in-S1 accessible-in-S2*)

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

definition *u x y n* = *P*(ω *in paths x*. (∀ *i*<*n*. ω *i* ≠ *y*) ∧ ω *n* = *y*)

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

lemma *u-le-p*: *u x y n* ≤ *p x y* (*Suc n*)
unfolding *u-def p-def* **by** (*auto intro*!: *finite-measure-mono*)

lemma *p-eq-setsum-p-u*:
assumes *x*[*simp*]: *x* ∈ *S* **shows** *p x y* (*Suc n*) = (∑ *i*≤*n*. *p y y* (*n* − *i*) ∗ *u x y i*)
proof −

let *?S* = $\lambda i.$ $\{\omega \in$ *space* (*paths x*). $((\forall j{<}i.\ \omega\ j \neq y) \wedge \omega\ i = y) \wedge \omega\ n = y\}$
have *p x y* (*Suc n*) = *measure* (*paths x*) ($\bigcup i{\leq}n.$ *?S i*)
apply (*simp add*: *p-def*)
apply (*rule arg-cong2*[**where** *f*=*measure*])
apply *auto*
apply (*rule-tac x*=*LEAST i. x i* = *x n* **in** *bexI*)
apply (*auto dest*: *not-less-Least intro*: *Least-le LeastI*)
done
also have ... = ($\sum i{\leq}n.$ *measure* (*paths x*) (*?S i*))
by (*rule finite-measure-finite-Union*) (*auto simp*: *disjoint-family-on-def neq-iff*)
also have ... = ($\sum i{\leq}n.$ *p y y* (*n* − *i*) ∗ *u x y i*)
proof (*intro setsum-cong refl*)
fix *i* **assume** *i*: *i* ∈ {.. *n*}
have *measure* (*paths x*) (*?S i*) =
($\int \omega.$ $\mathcal{P}(\omega'$ *in paths* (*nat-case x* ω (*Suc i*)).
$((\forall j{<}i.$ *comb-seq* (*Suc i*) ω ω' *j* ≠ *y*) ∧ *comb-seq* (*Suc i*) ω ω' *i* = *y*) ∧ *comb-seq* (*Suc i*) ω ω' *n* = *y*) *∂paths x*)
by (*rule prob-split-Collect*) *simp-all*
also have ... = ($\int \omega.$ *p y y* (*n* − *i*) ∗ *indicator* $\{\omega{\in}$*space* (*paths x*). $(\forall j{<}i.$ ω *j* ≠ *y*) ∧ ω *i* = *y*$\}$ ω *∂paths x*)
using *i*
by (*intro integral-cong*)
(*auto simp*: *comb-seq-less comb-seq-Suc2 prob-eq-0 p-def split*: *split-indicator nat.split*)
also have ... = *p y y* (*n* − *i*) ∗ *u x y i*
by (*simp add*: *emeasure-eq-measure u-def*)
finally show *measure* (*paths x*) (*?S i*) = *p y y* (*n* − *i*) ∗ *u x y i* **.**
qed
finally show *?thesis* **.**
qed

definition *f x y n* = $\mathcal{P}(\omega$ *in paths x*. $(\forall i{<}n.$ *nat-case x* ω *i* ≠ *y*) ∧ *nat-case x* ω *n* = *y*)

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

lemma *f-le-p*: *f x y n* ≤ *p x y n*
unfolding *f-def p-def* **by** (*auto intro*!: *finite-measure-mono*)

lemma *f-Suc*: *x* ≠ *y* ⟹ *f x y* (*Suc n*) = *u x y n*
by (*simp add*: *u-def f-def all-less-Suc-split*)

lemma *f-Suc-eq*: *f x x* (*Suc n*) = *0*
by (*simp add*: *f-def all-less-Suc-split*)

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

lemma *atMost-Suc-insert-0*: $\{..\ Suc\ n\} = insert\ 0\ (Suc\ `\ \{..\ n\})$
apply (*auto simp*: *image-iff Ball-def*)
apply (*case-tac x*)
apply *auto*
done

lemma *p-eq-setsum-p-f*:
assumes *x*[*simp*]: $x \in S$ **shows** $p\ x\ y\ n = (\sum i{\leq}n.\ p\ y\ y\ (n - i) * f\ x\ y\ i)$
proof *cases*
assume $x = y$ **then show** *?thesis*
unfolding *lessThan-Suc-atMost*[*symmetric*] *lessThan-Suc-eq-insert-0*
by (*simp add*: *f-Suc-eq zero-notin-Suc-image setsum-reindex f-0*)
next
assume $x \neq y$ **then show** *?thesis*
apply (*cases n*)
apply (*simp-all add*: *p-0 f-0 del*: *setsum-atMost-Suc*)
apply (*subst p-eq-setsum-p-u*[*OF x*])
apply (*subst atMost-Suc-insert-0*)
apply (*simp add*: *zero-notin-Suc-image setsum-reindex f-0 f-Suc*)
done
qed

5.4 Quantities G, U, and F

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

definition $G\ s\ t = (\int^+\omega.\ (\sum i.\ indicator\ \{t\}\ (nat\text{-}case\ s\ \omega\ i))\ \partial paths\ s)$

definition $U\ s\ t = \mathcal{P}(\omega\ in\ paths\ s.\ \exists n.\ \omega\ n = t)$

definition $F\ s\ t = \mathcal{P}(\omega\ in\ paths\ s.\ \exists n.\ nat\text{-}case\ s\ \omega\ n = t)$

lemma *G-eq*: $G\ s\ t = (\int^+\omega.\ emeasure\ (count\text{-}space\ UNIV)\ \{i.\ nat\text{-}case\ s\ \omega\ i = t\}\ \partial paths\ s)$
proof −
have $*$: $\bigwedge i\ \omega.\ indicator\ \{t\}\ (nat\text{-}case\ s\ \omega\ i) = indicator\ \{i.\ nat\text{-}case\ s\ \omega\ i = t\}\ i$
by (*auto split*: *split-indicator*)
show *?thesis*
unfolding *G-def*
by (*rule positive-integral-cong*)
(*simp add*: $*$ *positive-intergal-count-space-nat*[*symmetric*])
qed

5.4.1 G equals infinite sum of p

lemma *G-eq-suminf*: $G\ x\ y = (\sum i.\ ereal\ (p\ x\ y\ i))$
proof −
have $G\ x\ y = (\int^+\omega.\ (\sum i.\ indicator\ \{\omega{\in}space\ (paths\ x).\ nat\text{-}case\ x\ \omega\ i = y\}\ \omega)\ \partial paths\ x)$

unfolding *G-def* **by** (*auto split*: *split-indicator intro*!: *positive-integral-cong arg-cong*[**where** *f*=*suminf*])
also have ... = ($\sum i$. *ereal* (*p x y i*))
by (*simp add*: *emeasure-eq-measure*[*symmetric*] *p-def emeasure-nonneg positive-integral-suminf*)
finally show *?thesis* **.**
qed

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

lemma *convergence-norm-G*:
convergence-G x y z $\Longrightarrow$ *summable* (λn. *p x y n* * *norm z* ^ *n*)
unfolding *convergence-G-def* **.**

lemma *convergence-G*:
convergence-G x y (*z*::'*a*::{*banach*, *real-normed-div-algebra*}) $\Longrightarrow$ *summable* (λn. *p x y n* *$_R$ *z* ^ *n*)
unfolding *convergence-G-def*
by (*rule summable-norm-cancel*) (*simp add*: *abs-mult p-nonneg 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* ^ *n*) $\leq$ *1* * *norm* (*z* ^ *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* ^ *n*) $\leq$ *norm z* ^ *n*
by (*simp add*: *p-nonneg mult-nonneg-nonneg norm-power*)
qed (*simp add*: *z summable-geometric*)

lemma *convergence-norm-U*:
fixes *z* :: - :: *real-normed-div-algebra*
assumes *z*: *convergence-G x y z*
shows *summable* (λn. *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 p-nonneg abs-mult intro*!: *exI*[*of - 0*] *mult-right-mono mult-nonneg-nonneg u-le-p*)

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

5.4.2 *U* equals infinite sum of *u*

lemma *u-sums-U*: *u x y sums U x y*

```
proof −
  have AE ω in paths x. (∃ n. ω n = y) ⟶ (∃!n. (∀ i<n. ω i ≠ y) ∧ ω n = y)
  proof (rule AE-I2)
    fix x :: nat ⇒ 'a
    { fix n
      from smallest[of λi. x i = x n, OF refl] guess i .
      then have ∃ i. (∀ ia<i. x ia ≠ x n) ∧ x i = x n
        by auto }
    note smallest-Ex = this

    show ⋀y. (∃ n. x n = y) ⟶ (∃!n. (∀ i<n. x i ≠ y) ∧ x n = y)
      by (auto intro: smallest-Ex) (metis linorder-cases)
  qed
  then show ?thesis
    unfolding u-def[abs-def] U-def by (intro prob-sums) auto
qed

lemma accessibleI-U:
  assumes x ∈ S 0 < U x y shows accessible x y
proof (rule ccontr)
  assume ¬ accessible x y

  have U x y = 0
    unfolding U-def
    apply (rule prob-eq-0-AE)
    using AE-all-enabled[OF ‹x ∈ S›]
  proof eventually-elim
    fix ω assume ω[THEN spec]: ∀ i. ω i ∈ E (nat-case x ω i)
    { fix n have accessible x (ω n)
      proof (induct n)
        case 0 with ω[of 0] ‹x ∈ S› show ?case
          by (simp add: accessible-single)
      next
        case (Suc n) with ω[of Suc n] show ?case
          by (simp add: accessible.step)
      qed }
    with ‹¬ accessible x y› show ¬ (∃ n. ω n = y)
      by auto
  qed
  with ‹0 < U x y› show False by auto
qed

lemma F-nonneg: 0 ≤ F x y by (simp add: F-def measure-nonneg)

lemma F-le-1: F x y ≤ 1 by (simp add: F-def)

lemma f-sums-F: f x y sums F x y
proof −
  have AE ω in paths x. (∃ n. nat-case x ω n = y) ⟶ (∃!n. (∀ i<n. nat-case x
```

$\omega\ i \neq y) \wedge$ *nat-case x* ω *n* $= y)$
proof (*rule AE-I2*)
fix ω **assume** $\omega \in$ *space* (*paths x*)
{ **fix** *n*
from *smallest*[*of* λi. *nat-case x* ω *i* = *nat-case x* ω *n*, *OF refl*] **guess** *i* **.**
then have $\exists i$. ($\forall$ *ia*<*i*. *nat-case x* ω *ia* $\neq$ *nat-case x* ω *n*) $\wedge$ *nat-case x* ω *i* = *nat-case x* ω *n*
by *auto* }
note *smallest-Ex* = *this*

show ($\exists$ *n*. *nat-case x* ω *n* = *y*) $\longrightarrow$ ($\exists!$*n*. ($\forall$ *i*<*n*. *nat-case x* ω *i* $\neq$ *y*) $\wedge$ *nat-case x* ω *n* = *y*)
by (*auto intro*: *smallest-Ex*) (*metis linorder-cases*)
qed
then show *?thesis*
unfolding *f-def*[*abs-def*] *F-def* **by** (*intro prob-sums*) *auto*
qed

lemma *U-le-F*: *U x y* $\leq$ *F x y*
apply (*auto simp*: *U-def F-def intro*!: *finite-measure-mono*)
apply (*rule-tac x*=*Suc n* **in** *exI*)
apply *simp*
done

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

lemma *U-bounded*: *0* $\leq$ *U x y U x y* $\leq$ *1*
unfolding *U-def* **by** (*simp-all add*: *measure-nonneg*)

5.5 Generating functions of G, U, and F

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

definition *gf-G x y z* = ($\sum$ *n*. *p x y n* $*_R$ *z* ^ *n*)

definition *gf-F x y z* = ($\sum$ *n*. *f x y n* $*$ *z* ^ *n*)

lemma *lim-gf-G*: ((λz. *ereal* (*gf-G x y z*)) ---> *G x y*) (*at-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*)

lemma *gf-U-eq-U*: *gf-U x y 1* = *U x y*

using *u-sums-U*[*THEN sums-unique*] **by** (*simp add*: *gf-U-def U-def*)

lemma *gf-F-eq-F*: *gf-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 summable-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 mult-nonneg-nonneg 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 *gf-G-nonneg*:
fixes *z* :: *real*
shows *0* ≤ *z* ⟹ *z* < *1* ⟹ *0* ≤ *gf-G x y z*
unfolding *gf-G-def*
by (*intro suminf-ge-zero convergence-G convergence-G-less-1*) (*simp-all add*: *mult-nonneg-nonneg p-nonneg*)

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-ge-zero*) (*simp-all add*: *mult-nonneg-nonneg 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 *gf-G-eq-gf-F*:
assumes *x*[*simp*]: *x* ∈ *S*
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-setsum-p-f*[*of x y*] *setsum-left-distrib*)
also have ... = (∑ *n*. ∑ *i*=*0* .. *n*. (*f x y i* ∗ *z*^*i*) ∗ (*p y y* (*n* − *i*) ∗ *z*^(*n* − *i*)))

by (*intro arg-cong*[**where** *f*=*suminf*] *setsum-cong ext atLeast0AtMost*[*symmetric*])
(*simp-all add*: *power-add*[*symmetric*])
also have ... = ($\sum$ *n*. *f x y n* * *z*^*n*) * ($\sum$ *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 p-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

5.5.1 Relate *gf-G* and *gf-U*

lemma *gf-G-eq-gf-U*:
fixes *z* :: *'z* :: {*banach*, *real-normed-field*}
assumes *x*[*simp*]: *x* ∈ *S*
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* = ($\sum$ *i*≤*n*. (*p x x* (*n* − *i*) * *u x x i*) *R* *z*^*Suc n*)
unfolding *scaleR-setsum-left*[*symmetric*] **by** (*simp add*: *p-eq-setsum-p-u*)
also have ... = ($\sum$ *i*≤*n*. (*u x x i R* *z*^*Suc i*) * (*p x x* (*n* − *i*) *R* *z*^(*n* − *i*)))
by (*intro setsum-cong refl*) (*simp add*: *field-simps power-diff cong*: *disj-cong*)
finally have *p x x* (*Suc n*) *R* *z*^(*Suc n*) = ($\sum$ *i*=*0* .. *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* + ($\sum$ *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* + ($\sum$ *n*. $\sum$ *i*=*0* .. *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 p-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-G-pos:
  fixes z :: real
  assumes z: 0 < z z < 1 and ∗: accessible x y
  shows 0 < gf-G x y z
  unfolding gf-G-def
proof (subst suminf-gt-zero-iff)
  show summable (λn. p x y n ∗R z ^ n)
    using z by (intro convergence-G convergence-G-less-1) simp
  show pos: ∀ n. 0 ≤ p x y n ∗R z ^ n
    using z by (auto intro!: mult-nonneg-nonneg p-nonneg)
  show ∃ n. 0 < p x y n ∗R z ^ n
  proof (rule ccontr)
    assume ¬ (∃ n. 0 < p x y n ∗R z ^ n)
    with pos have ∀ n. p x y n ∗ z ^ n = 0
      by (intro antisym allI) (simp-all add: not-less)
    with z have ⋀n. p x y n = 0
      by simp
    with ∗[THEN accessibleD-pos] show False
      by simp
  qed
qed

lemma gf-U: (gf-U x y ---> U x y) (at-left 1)
proof −
  have ((λz. ereal (∑ n. u x y n ∗ z ^ n)) ---> (∑ n. ereal (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. ereal (u x y n)) = ereal (suminf (u x y))
   by (intro u-nonneg suminf-ereal suminf-ereal-finite 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. 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-F: (gf-F x y ---> F x y) (at-left 1)
proof −
  have ((λz. ereal (∑ n. f x y n ∗ z ^ n)) ---> (∑ n. ereal (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 $(\sum n.\ ereal\ (f\ x\ y\ n)) = ereal\ (suminf\ (f\ x\ y))$
by (*intro f-nonneg suminf-ereal suminf-ereal-finite 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 show *?thesis*
by (*simp add*: *gf-F-def*[*abs-def*] *F-def*)
qed

5.6 Recurrent states and H

definition *recurrent* :: $'s \Rightarrow bool$ **where**
$recurrent\ s \longleftrightarrow (AE\ \omega\ in\ paths\ s.\ \exists n.\ \omega\ n = s)$

lemma *recurrent-iff-U-eq-1*: $recurrent\ s \longleftrightarrow U\ s\ s = 1$
unfolding *recurrent-def U-def* **by** (*subst prob-Collect-eq-1*) *simp-all*

definition $H\ s\ t = \mathcal{P}(\omega\ in\ paths\ s.\ infinite\ \{n.\ \omega\ n = t\})$

lemma *H-eq*:
assumes $s \in S$
shows $recurrent\ s \longleftrightarrow H\ s\ s = 1$
$\neg\ recurrent\ s \longleftrightarrow H\ s\ s = 0$
$H\ s\ t = U\ s\ t * H\ t\ t$
proof −
def $H' \equiv \lambda s\ t\ n.\ \{\omega \in space\ (paths\ s).\ finite\ \{n.\ \omega\ n = t\} \longrightarrow n \leq card\ \{n.\ \omega\ n = t\}\}$
have [*measurable*]: $\bigwedge x\ y\ n.\ H'\ x\ y\ n \in sets\ S\text{-}seq$
unfolding *H'-def space-paths*
apply (*intro predE pred-intros-logic*)
apply *simp*
apply (*intro pred-const-le*)
apply (*rule measurable-card*)
apply *simp-all*
done
let *?H'* $= \lambda s\ t\ n.\ measure\ (paths\ s)\ (H'\ s\ t\ n)$
{ fix *x y*
have *?H'* $x\ y\ 1 = U\ x\ y$
unfolding *H'-def U-def*
apply (*intro arg-cong2*[**where** *f*=*measure*] *refl arg-cong*[**where** *f*=*Collect*] *ext conj-cong*)
apply (*auto simp add*: *card-le-Suc-iff*)
apply (*rule ccontr*)
apply *simp*
apply (*rule-tac x*=*n* **in** *exI*)
apply (*rule-tac x*=$\{m.\ \omega\ m = \omega\ n\} - \{n\}$ **in** *exI*)
apply *auto*
done **}**
note *H'-1* = *this*

{ fix m **and** $x\ y$ :: $'s$ **assume** [*simp*]: $x \in S$
def $i == \lambda\omega.$ *if* $\exists\, n.\ \omega\ n = y$ *then Suc* (*LEAST* $n.\ \omega\ n = y$) *else 0*
have [*measurable*]: $i \in$ *measurable S-seq* (*count-space UNIV*)
by (*simp add*: *i-def*)
have *emeasure* (*paths x*) (*H' x y* (*Suc m*)) = ($\int^+ \omega.$ *indicator* (*H' x y* (*Suc m*)) ω *∂paths x*)
by *simp*
also have ... =
($\int^+ \omega.$ *indicator* $\{\omega.\ i\ \omega \neq 0\}\ \omega$ *
($\int^+ \omega'.$ *indicator* (*H' x y* (*Suc m*)) (*comb-seq* (*i ω*) *ω ω'*) *∂paths* (*nat-case x ω* (*i ω*))) *∂paths x*)
proof (*rule positive-integral-prefixes*)
fix $\omega\ \omega'$ **assume** *i*: $0 < i\ \omega$ **and** *le*: $\bigwedge j.\ j < i\ \omega \Longrightarrow \omega\ j = \omega'\ j$
def $j \equiv i\ \omega - 1$
from *i* **have** $\exists\, i.\ \omega\ i = y$
unfolding *i-def* **by** (*auto split*: *split-if-asm*)
then have $j < i\ \omega\ \omega\ j = y\ \forall\, k{<}j.\ \omega\ k \neq y$
by (*auto simp*: *i-def j-def intro*: *LeastI-ex dest*: *not-less-Least split*: *split-if-asm*)
with *le* **have** $i\ \omega' =$ *Suc j*
by (*subst i-def*) (*auto simp add*: *not-less*[*symmetric*] *intro*!: *Least-equality*)
then show $i\ \omega = i\ \omega'$
using ‹$0 < i\ \omega$› **by** (*simp add*: *j-def*)
next
fix ω **assume** $i\ \omega = 0$
then have $\bigwedge k.\ \omega\ k \neq y$ **by** (*auto simp*: *i-def split*: *split-if-asm*)
then show *indicator* (*H' x y* (*Suc m*)) $\omega = (0{::}ereal)$
by (*auto simp add*: *H'-def split*: *split-indicator*)
qed *simp-all*
also have ... = ($\int^+ \omega.$ *indicator* $\{\omega{\in}$*space S-seq*. $\exists\, n.\ \omega\ n = y\}\ \omega$ *
($\int^+ \omega'.$ *indicator* (*H' y y m*) *ω' ∂paths y*) *∂paths x*)
proof (*intro positive-integral-cong ereal-right-mult-cong*)
fix ω **assume** $\omega \in$ *space* (*paths x*)
then show *indicator* $\{\omega.\ i\ \omega \neq 0\}\ \omega$ = *indicator* $\{\omega \in$ *space S-seq*. $\exists\, n.\ \omega\ n = y\}\ \omega$
by (*auto simp add*: *i-def split*: *split-indicator*)
next
fix ω **assume** *indicator* $\{\omega \in$ *space S-seq*. $\exists\, n.\ \omega\ n = y\}\ \omega \neq 0$ **and** ω: $\omega \in$ *space* (*paths x*)
then have *ex-y*: $\exists\, n.\ \omega\ n = y\ i\ \omega \neq 0$
by (*auto simp*: *indicator-def i-def split*: *split-if-asm*)
then have *x-i-ω*: *nat-case x ω* (*i ω*) = *y*
by (*auto simp add*: *i-def intro*: *LeastI*)
then have *paths-y*: *paths* (*nat-case x ω* (*i ω*)) = *paths y*
by *simp*
{ fix ω' **assume** ω': $\omega' \in$ *space S-seq*
have $\bigwedge n.$ *Suc n* $< i\ \omega \Longrightarrow \omega\ n \neq y$
using *ex-y* **by** (*auto simp add*: *i-def dest*: *not-less-Least split*: *if-splits*)
with *x-i-ω* **have** $\bigwedge n.$ *comb-seq* (*i ω*) *ω ω' n* = *y* $\longleftrightarrow$
(*i ω* $\leq$ *Suc n* $\wedge$ *nat-case y ω'* (*Suc n* − *i ω*) = *y*)

```
          by (auto simp add: comb-seq-def not-less le-imp-diff-is-add le-less split:
nat.splits)
       then have A: {n. comb-seq (i ω) ω ω' n = y} = (λn. n + (i ω − 1)) ‘ {n.
nat-case y ω' n = y}
         (is ?A = ?s ‘ ?B)
         using ‹i ω ≠ 0›
         apply simp
         apply safe
         apply (rule-tac x=Suc x − i ω in image-eqI)
         apply simp-all
         done
       have s: inj-on ?s ?B by auto
       have B: ?B = {0} ∪ Suc ‘ {n. ω' n = y}
         by (simp add: set-eq-iff image-Collect split: nat.split)
       have finite {n. ω' n = y} ⟹ card ?A = Suc (card {n. ω' n = y})
           unfolding A card-image[OF s] by (simp add: zero-notin-Suc-image
card-image B)
       moreover have finite ?A ⟷ finite {n. ω' n = y}
         unfolding A B by (auto dest!: finite-imageD)
       ultimately have comb-seq (i ω) ω ω' ∈ H' x y (Suc m) ⟷ ω' ∈ H' y y
m
       using ω ω' by (auto simp add: H'-def space-PiM PiE-iff Pi-iff comb-seq-def)
       then have indicator (H' x y (Suc m)) (comb-seq (i ω) ω ω') = (indicator
(H' y y m) ω' :: ereal)
         by (simp split: split-indicator) }
     then show (∫+ ω'. indicator (H' x y (Suc m)) (comb-seq (i ω) ω ω') ∂paths
(nat-case x ω (i ω))) =
         integral^P (paths y) (indicator (H' y y m))
       unfolding paths-y by (intro positive-integral-cong) simp
   qed
   also have ... = U x y * ?H' y y m
     by (simp add: positive-integral-multc measure-nonneg emeasure-eq-measure
U-def)
   finally have ?H' x y (Suc m) = U x y * ?H' y y m
     by (simp add: emeasure-eq-measure) }
 note H'-Suc = this

 { fix m x assume x ∈ S
   then 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 (paths x) (⋂ i. H' x y i)
     apply (rule finite-Lim-measure-decseq)
     apply safe
     apply simp
     apply (auto simp add: decseq-Suc-iff subset-eq H'-def)
     done
```

```
  also have (⋂i. H' x y i) = {ω∈space (paths x). infinite {n. ω n = y}}
   by (auto simp: H'-def not-less[symmetric])
  finally have ?H' x y ----> H x y
   unfolding H-def . }
 note H'-lim = this

 from H'-lim[of s s, THEN LIMSEQ-Suc] ‹s∈S›
 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 measure-nonneg)
 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 add: tendsto-const)
 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] ‹s∈S› H'-Suc[OF ‹s∈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
```

5.6.1 Relating *recurrent* and G

```
lemma recurrent-iff-G-infinite:
 assumes x ∈ S
 shows recurrent x ⟷ G x x = ∞
proof −
 have ((λz. ereal (gf-G x x z)) ---> G x x) (at-left 1)
  by (rule lim-gf-G)
 then have G: ((λz. ereal (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 ‹x ∈ S›)
  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 $gf\text{-}U\ x\ x\ z \neq 1$
using *gf-G-eq-gf-U*[*OF* ‹$x \in S$› *convergence-G-less-1*[*of z*]] z **by** *simp*
moreover
have $gf\text{-}U\ x\ x\ z \leq U\ x\ x$
unfolding *gf-U-def gf-U-eq-U*[*symmetric*]
using z
by (*intro summable-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 $gf\text{-}U\ x\ x\ z < 1$
using *U-bounded*[*of x x*] **by** *simp* **}**
note *strict* = *this*

{ assume $U\ x\ x = 1$
moreover have $((\lambda xa.\ 1 - gf\text{-}U\ x\ x\ xa :: real) \longrightarrow 1 - U\ x\ x)\ (at\text{-}left\ 1)$
by (*intro tendsto-intros gf-U*)
moreover have *eventually* $(\lambda z.\ gf\text{-}U\ x\ x\ z < 1)\ (at\text{-}left\ (1::real))$
by (*auto intro*!: *eventually-at-left-1 strict simp*: ‹$U\ x\ x = 1$› *gf-U-eq-U*)
ultimately have $((\lambda z.\ ereal\ (1\ /\ (1 - gf\text{-}U\ x\ x\ z))) \longrightarrow \infty)\ (at\text{-}left\ 1)$
unfolding *tendsto-PInfty-eq-at-top*
by (*intro LIM-at-top-divide*[**where** *a=1*] *tendsto-const zero-less-one*)
(*auto simp*: *field-simps*)
with G **have** $G\ x\ x = \infty$
by (*rule tendsto-unique*[*rotated*]) *simp* **}**
moreover
{ assume $U\ x\ x < 1$
then have $((\lambda xa.\ ereal\ (1\ /\ (1 - gf\text{-}U\ x\ x\ xa))) \longrightarrow 1\ /\ (1 - U\ x\ x))\ (at\text{-}left\ 1)$
by (*intro tendsto-intros gf-U lim-ereal*[*THEN iffD2*]) *simp*
from *tendsto-unique*[*OF - G this*] **have** $G\ x\ x \neq \infty$
by *simp* **}**
ultimately show *?thesis*
using *U-cases recurrent-iff-U-eq-1* **by** *blast*
qed

5.6.2 *recurrent* is invariant

lemma *recurrent-iffI-communicating*:
assumes $(x, y) \in communicating$
shows *recurrent* $x \longleftrightarrow$ *recurrent* y
proof −
from *assms* **obtain** $n\ m$ **where** $0 < p\ x\ y\ n\ y \in S\ 0 < p\ y\ x\ m\ x \in S$
by (*force simp*: *communicating-def dest*: *accessibleD-pos intro*: *accessible-in-S1*)

moreover
{ fix $x\ y\ n\ m$ **assume** [*simp*]: $x \in S\ y \in S$
assume $0 < p\ x\ y\ n\ 0 < p\ y\ x\ m\ G\ y\ y = \infty$
then have $\infty = ereal\ (p\ x\ y\ n * p\ y\ x\ m) * G\ y\ y$
by (*auto intro*: *mult-pos-pos*)
also have $ereal\ (p\ x\ y\ n * p\ y\ x\ m) * G\ y\ y = (\sum i.\ ereal\ (p\ x\ y\ n * p\ y\ x\ m) * p\ y\ y\ i)$
unfolding *G-eq-suminf*
by (*rule suminf-cmult-ereal*[*symmetric*]) (*auto simp*: *p-nonneg mult-nonneg-nonneg*)
also have $\ldots \leq (\sum i.\ ereal\ (p\ x\ x\ (n + i + m)))$
proof (*intro suminf-le-pos*)
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 add*: *p-nonneg*)
also have $\ldots \leq p\ x\ x\ (n + i + m)$
by (*intro prob-reachable-le*) *simp-all*
finally show $ereal\ (p\ x\ y\ n * p\ y\ x\ m) * p\ y\ y\ i \leq ereal\ (p\ x\ x\ (n + i + m))$
by (*simp add*: *ac-simps*)
qed (*simp add*: *p-nonneg mult-nonneg-nonneg*)
also have $\ldots \leq (\sum i.\ ereal\ (p\ x\ x\ (i + (n + m))))$
by (*simp add*: *ac-simps*)
also have $\ldots \leq (\sum i.\ ereal\ (p\ x\ x\ i))$
by (*rule suminf-ereal-offset-le*) (*simp add*: *p-nonneg*)
also have $\ldots \leq G\ x\ x$
unfolding *G-eq-suminf* **by** (*auto intro*!: *suminf-le-pos*)
finally have $G\ x\ x = \infty$
by *simp* **}**
ultimately show *?thesis*
using *recurrent-iff-G-infinite* **by** *blast*
qed

5.6.3 *recurrent* and H and U

lemma *recurrent-accessible*:
assumes *recurrent x accessible x y*
shows $U\ y\ x = 1\ H\ y\ x = 1$ *recurrent y* $(x, y) \in$ *communicating*
proof −
{ fix $w\ y$ **assume** *step*: *accessible x w y* $\in E\ w\ U\ w\ x = 1\ H\ w\ x = 1$ *recurrent w x* $\neq y$
then have [*simp*]: $y \in S\ w \in S\ x \in S$
by (*auto intro*: *accessible-in-S2 accessible-in-S1*)

have *measure* $(K\ w)\ S = U\ w\ x$
using *step K.prob-space*[*of w*] **by** *simp*
also have $\ldots = (\int v.\ indicator\ \{x\}\ v + U\ v\ x * indicator\ (S - \{x\})\ v\ \partial K\ w)$
unfolding *U-def*
apply (*subst prob-iterate-Collect*)
apply (*auto intro*!: *integral-cong arg-cong2*[**where** f=*measure*] *AE-I2*

```
        simp: exI[of - 0] prob-eq-1 split: split-indicator)
    apply (rule-tac x=n − 1 in exI)
    apply (simp split: nat.splits)
    apply (rule-tac x=Suc n in exI)
    apply (simp split: nat.splits)
    done
  also have ... = measure (K w) {x} + (∫ v. U v x * indicator (S − {x}) v
∂K w)
    by (subst integral-add)
      (auto intro!: K.integrable-const-bound[where B=1]
          simp: abs-mult mult-le-one U-bounded K.emeasure-eq-measure)
  finally have measure (K w) S − measure (K w) {x} = (∫ v. U v x * indicator
(S − {x}) v ∂K w)
    by simp
  also have measure (K w) S − measure (K w) {x} = measure (K w) (S −
{x})
    by (subst K.finite-measure-Diff) auto
  finally have 0 = (∫ v. indicator (S − {x}) v ∂K w) − (∫ v. U v x * indicator
(S − {x}) v ∂K w)
    by (simp add: Diff-subset K.emeasure-eq-measure)
  also have ... = (∫ v. (1 − U v x) * indicator (S − {x}) v ∂K w)
    by (subst integral-diff(2)[symmetric])
      (auto intro!: K.integrable-const-bound[where B=1] integral-cong
          simp: abs-mult mult-le-one U-bounded split: split-indicator)
  also have ... ≥ (∫ v. (1 − U y x) * indicator {y} v ∂K w) (is - ≥ ?rhs)
    using ‹recurrent x›
    by (auto intro!: integral-mono K.integrable-const-bound[where B=1]
         simp: abs-mult mult-le-one U-bounded recurrent-iff-U-eq-1 field-simps
simp del: ‹y∈S›
          split: split-indicator)
  also (xtrans) have ?rhs = (1 − U y x) * measure (K w) {y}
    by (simp add: integral-cmul-indicator K.emeasure-eq-measure)
  finally have (1 − U y x) * measure (K w) {y} = 0
    by (auto intro!: antisym simp: U-bounded measure-nonneg mult-le-0-iff)
  with ‹y ∈ E w› have U y x = 1
    by (simp add: E-def field-simps K.emeasure-eq-measure)
  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 accessible y x
    by (intro accessibleI-U) auto
  with step have (x, y) ∈ communicating
    by (auto simp add: communicating-def intro: accessible.intros)
  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 ‹accessible x y›
 show U y x = 1 H y x = 1 recurrent y (x, y) ∈ communicating
```

proof *induction*
case *start* **then show** *U x x = 1 H x x = 1 recurrent x* $(x, x) \in$ *communicating*
using ‹*recurrent x*› *H-eq*(*1*)[*of x*] **by** (*auto simp*: *recurrent-iff-U-eq-1 communicating-def accessible.start*)
next
case (*step w y*)
with *enabled*[*of w y*] ‹*recurrent x*› *H-eq*(*1*)[*of x*]
have *U y x = 1* $\wedge$ *H y x = 1* $\wedge$ *recurrent y* $\wedge$ $(x, y) \in$ *communicating*
by (*cases x = y*) (*auto simp*: *recurrent-iff-U-eq-1 communicating-def accessible.start*)
then show *U y x = 1 H y x = 1 recurrent y* $(x, y) \in$ *communicating*
by *auto*
qed
qed

lemma *recurrent-class*:
assumes *recurrent x*
shows $\{y.$ *accessible x y*$\}$ = *communicating* `` $\{x\}$
using *recurrent-accessible*(*4*)[*OF* ‹*recurrent x*›] **by** (*auto simp*: *communicating-def*)

lemma *irreduccible-recurrent-class*:
assumes $x \in S$ *recurrent x* **shows** $\{y.$ *accessible x y*$\} \in S$ // *communicating*
unfolding *recurrent-class*[*OF* ‹*recurrent x*›] **by** (*rule quotientI*[*OF* ‹$x \in S$›])

5.7 Essential classes

definition *essential-class* :: *'s set* $\Rightarrow$ *bool* **where**
essential-class S' $\longleftrightarrow$ *S'* $\in$ *S* // *communicating* $\wedge$ $\neg$ $(\exists\, s{\in}S'.\ \exists\, t{\in}S - S'.$ *accessible s t*)

lemma *essential-classI*:
assumes *C*: *C* $\in$ *S* // *communicating*
assumes *eq*: $\bigwedge x\ y.\ x \in C \Longrightarrow$ *accessible x y* $\Longrightarrow y \in C$
shows *essential-class C*
by (*auto simp*: *essential-class-def intro*: *C*) (*metis eq*)

lemma *essential-classI2*:
assumes $X \neq \{\}$ $X \subseteq S$
assumes *accI*: $\bigwedge x\ y.\ x \in X \Longrightarrow y \in X \Longrightarrow$ *accessible x y*
assumes *ED*: $\bigwedge x\ y.\ x \in X \Longrightarrow y \in E\ x \Longrightarrow y \in X$
shows *essential-class X*
proof (*rule essential-classI*)
{ fix *x y* **assume** *accessible x y x* $\in X$
then show *y* $\in X$
by *induct* (*auto dest*: *ED*)**}**
note *accD* = *this*

from ‹$X \neq \{\}$› **obtain** *x* **where** $x \in X$ **by** *auto*
from ‹$x \in X$› ‹$X \subseteq S$› **show** *X* $\in$ *S* // *communicating*

by (*auto simp add*: *quotient-def Image-def communicating-def accI dest*: *accD intro*!: *bexI*[*of* - *x*])
qed

lemma *essential-recurrent-class*:
assumes $x \in S$ *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 intro*: *accessible-trans*)
done

lemma *essential-classD1*:
essential-class $C \Longrightarrow x \in C \Longrightarrow x \in S$
unfolding *essential-class-def*
by (*auto elim*!: *quotientE simp*: *communicating-in-S*)

lemma *essential-classD2*:
essential-class $C \Longrightarrow x \in C \Longrightarrow$ *accessible* $x\ y \Longrightarrow y \in C$
unfolding *essential-class-def* **by** (*auto intro*: *accessible-in-S2*)

lemma *essential-classD3*:
essential-class $C \Longrightarrow x \in C \Longrightarrow y \in C \Longrightarrow (x, y) \in$ *communicating*
unfolding *essential-class-def*
by (*auto elim*!: *quotientE simp*: *communicating-def intro*: *accessible-trans*)

lemma *finite-essential-class-imp-recurrent*:
assumes C: *essential-class* C *finite* C **and** x: $x \in C$
shows *recurrent* x
proof −
note *essential-classD1*[*OF* $C(1)$ x]
have *AE* ω *in paths* x. $\exists\, y{\in}C$. *infinite* $\{n.\ \omega\ n = y\}$
using *AE-accessible*[*OF* ‹$x \in S$›]
proof *eventually-elim*
fix ω **assume** $*$: $\forall\, m$. *accessible* x (*case* m *of* 0 ⇒ x | *Suc* x ⇒ $\omega\ x$)
have $\forall\, m{\in}UNIV.\ \exists\, x{\in}C.\ \omega\ m = x$
proof
fix m
from $*$[*THEN spec*, *of Suc* m] *essential-classD2*[*OF* $C(1)$ x]
show $\exists\, x{\in}C.\ \omega\ m = x$
by *auto*
qed
from *pigeonhole-infinite-rel*[*OF nat-infinite* $C(2)$ *this*]
show $\exists\, y{\in}C$. *infinite* $\{n.\ \omega\ n = y\}$
by *simp*
qed
then have $1 = \mathcal{P}(\omega$ *in paths* x. $\exists\, y{\in}C$. *infinite* $\{n.\ \omega\ n = y\})$
by (*subst* (*asm*) *prob-Collect-eq-1*[*symmetric*]) (*auto simp*: ‹*finite C*›)
also have ... = *measure* (*paths* x) ($\bigcup y{\in}C$. $\{\omega{\in}$*space* (*paths* x). *infinite* $\{n.\ \omega$

$n = y\}\}$)
  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 finite-measure-subadditive-finite*) *auto*
 also have ... $= (\sum y{\in}C.\ U\ x\ y * H\ y\ y)$
  by (*auto intro*!: *setsum-cong H-eq* ‹$x \in S$›)
 finally have $\exists\, y{\in}C.$ *recurrent y*
  by (*rule-tac ccontr*) (*simp add*: *H-eq*(*2*) *essential-classD1*[*OF C*(*1*)])
 then guess *y* **..**
 from *essential-classD3*[*OF C*(*1*) *x this*(*1*)] *recurrent-accessible*(*3*)[*OF this*(*2*)]
 show *recurrent x*
  by (*simp add*: *communicating-def*)
qed

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

5.8 Hitting times

definition *t x* ω = (*if* $\exists\, i.\ \omega\ i = x$ *then ereal* (*Suc* (*LEAST i*. $\omega\ i = x$)) *else* ∞)

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

lemma *summable-gf-U'*:
 assumes *z*: *norm z* < *1*
 shows *summable* ($\lambda n.\ u\ x\ y\ n * Suc\ n * z\ \hat{}\ n$)
proof −
 have *summable* ($\lambda n.\ n * |z|\ \hat{}\ n$)
 proof (*rule root-test-convergence*)
  have ($\lambda n.$ *root n n* $* |z|$) −−−−> *1* $* |z|$
   by (*intro tendsto-intros LIMSEQ-root*)
  then show ($\lambda n.$ *root n* (*norm* $(n * |z|\ \hat{}\ n)$)) −−−−> $|z|$
   by (*rule filterlim-cong*[*THEN iffD1*, *rotated 3*])
    (*auto intro*!: *exI*[*of - 1*]
     simp add: *abs-mult u-nonneg real-root-mult power-abs eventually-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* ($\lambda n.\ u\ x\ y\ n * Suc\ n * z\ \hat{}\ n$)
  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*)

done
qed

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 *integral-t-eq*:
recurrent x $\Longrightarrow$ $(\int^+\omega.\ t\ x\ \omega \partial paths\ x) = (\int^+\omega.\ ereal\ (Suc\ (LEAST\ n.\ \omega\ n = x))\ \partial paths\ x)$
by (*auto simp*: *recurrent-def t-def intro*!: *positive-integral-cong-AE elim*!: *eventually-elim1*)

lemma *one-le-integral-t*:
assumes x: *recurrent x* **shows** $1 \leq (\int^+\omega.\ t\ x\ \omega\ \partial paths\ x)$
unfolding *integral-t-eq*[*OF x*]
by (*auto intro*!: *positive-integral-ge-const simp*: *one-ereal-def*)

5.8.1 Integral equals infinite sum of u

lemma *integral-t-eq-suminf*:
assumes x: *recurrent x* **and** [*simp*]: $x \in S$
shows $(\int^+\omega.\ t\ x\ \omega \partial paths\ x) = (\sum i.\ ereal\ (u\ x\ x\ i * Suc\ i))$
proof −
let *?L* = $\lambda\omega.\ LEAST\ n.\ \omega\ n = x$
{ fix t
have $(\lambda n.\ \mathcal{P}(\omega\ in\ paths\ x.\ n \geq t \wedge (\forall i{<}n.\ \omega\ i \neq x) \wedge \omega\ n = x))$ *sums* $\mathcal{P}(\omega\ in\ paths\ x.\ t \leq ?L\ \omega)$
apply (*rule prob-sums*[*rotated 2*])
using x **unfolding** *recurrent-def*
apply *eventually-elim*
proof (*intro iffI conjI allI impI*)
fix ω **and** n **assume** *ex*: $\exists n.\ \omega\ n = x$ **and** *L*: $t \leq ?L\ \omega$
then show $\exists!n.\ t \leq n \wedge (\forall i{<}n.\ \omega\ i \neq x) \wedge \omega\ n = x$
apply (*intro ex1I*[*of - ?L ω*])
apply (*auto dest*: *not-less-Least intro*: *LeastI*)
apply (*rule LeastI2-wellorder*)
apply *auto*
apply (*metis linorder-cases not-le*)
done
next
fix ω n **assume** *ex*: $\exists n.\ \omega\ n = x$ **and** t: $t \leq n \wedge (\forall i{<}n.\ \omega\ i \neq x) \wedge \omega\ n = x$
show $t \leq ?L\ \omega$
by (*rule LeastI2-ex*[*OF ex*]) (*metis t not-less le-less-trans*)
qed *simp-all*
also have $(\lambda n.\ \mathcal{P}(\omega\ in\ paths\ x.\ n \geq t \wedge (\forall i{<}n.\ \omega\ i \neq x) \wedge \omega\ n = x)) = (\lambda n.\ u\ x\ x\ n * indicator\ \{t\ ..\}\ n)$

by (*auto simp*: *u-def split*: *split-indicator*)
finally have $\mathcal{P}(\omega$ *in paths* $x.\ t \leq$ *?L* $\omega) = (\int^+ n.\ u\ x\ x\ n * indicator\ \{t\ ..\}\ n\partial count\text{-}space\ UNIV)$
apply (*subst positive-intergal-count-space-nat*)
apply (*auto dest*!: *sums-unique intro*!: *mult-nonneg-nonneg u-nonneg*)
apply (*intro suminf-ereal*[*symmetric*] *mult-nonneg-nonneg u-nonneg indicator-pos-le suminf-ereal-finite*)
apply (*rule summable-comparison-test*[*OF - u-sums-U*[*THEN sums-summable*, *of x x*]])
apply (*auto intro*!: *exI*[*of - 0*] *simp*: *u-nonneg split*: *split-indicator*)
done **}**
note *eq* = *this*
have $(\int^+ \omega.\ t\ x\ \omega\partial paths\ x) = (\int^+ \omega.\ ereal\ (Suc\ (?L\ \omega))\ \partial paths\ x)$
using *x* **by** (*rule integral-t-eq*)
also have ... $= (\sum t.\ ereal\ (\mathcal{P}(\omega$ *in paths* $x.\ t \leq (?L\ \omega))))$
unfolding *real-eq-of-nat* **by** (*subst positive-integral-nat-function*) (*simp-all add*: *emeasure-eq-measure less-Suc-eq-le*)
also have ... $= (\sum t.\ \int^+ n.\ u\ x\ x\ n * indicator\ \{t\ ..\}\ n\partial count\text{-}space\ UNIV)$
unfolding *eq* **..**
also have ... $= (\int^+ n.\ (\sum t.\ ereal\ (u\ x\ x\ n * indicator\ \{t\ ..\}\ n))\partial count\text{-}space\ UNIV)$
by (*subst positive-integral-suminf*) (*simp-all add*: *mult-nonneg-nonneg u-nonneg*)
also have $(\lambda n.\ (\sum t.\ ereal\ (u\ x\ x\ n * indicator\ \{t\ ..\}\ n))) = (\lambda n.\ ereal\ (u\ x\ x\ n * Suc\ n))$
proof
fix *n*
have $(\sum t.\ ereal\ (u\ x\ x\ n * indicator\ \{t\ ..\}\ n)) = (\sum t{\leq}n.\ ereal\ (u\ x\ x\ n * indicator\ \{t\ ..\}\ n))$
by (*intro suminf-finite*) (*auto split*: *split-indicator*)
also have ... $= (\sum t{\leq}n.\ ereal\ (u\ x\ x\ n))$
by (*intro setsum-cong*) (*auto split*: *split-indicator*)
also have ... $= ereal\ (u\ x\ x\ n * Suc\ n)$
by (*simp add*: *real-eq-of-nat ac-simps*)
finally show $(\sum t.\ ereal\ (u\ x\ x\ n * indicator\ \{t\ ..\}\ n)) = ereal\ (u\ x\ x\ n * Suc\ n)$ **.**
qed
also have $(\int^+ n.\ ereal\ (u\ x\ x\ n * Suc\ n)\ \partial count\text{-}space\ UNIV) = (\sum n.\ ereal\ (u\ x\ x\ n * Suc\ n))$
by (*simp add*: *u-nonneg mult-nonneg-nonneg positive-intergal-count-space-nat*)
finally show *?thesis* **.**
qed

5.8.2 *gf-U′* approaches the integral of *t*

lemma *gf-U′-tendsto-integral-t*:
assumes *x*: *recurrent x x* $\in S$
shows $((\lambda z.\ ereal\ (gf\text{-}U'\ x\ x\ z)) \longrightarrow (\int^+ \omega.\ t\ x\ \omega\partial paths\ x))\ (at\text{-}left\ 1)$
unfolding *integral-t-eq-suminf*[*OF x*] *gf-U′-def*
by (*auto intro*!: *power-series-tendsto-at-left summable-gf-U′ mult-nonneg-nonneg*

u-nonneg)

lemma *gf-U'-pos*:
fixes z :: *real*
assumes z: $0 < z$ $z < 1$ **and** $U\ x\ y \neq 0$
shows $0 <$ *gf-U'* $x\ y\ z$
unfolding *gf-U'-def*
proof (*subst suminf-gt-zero-iff*)
show *summable* $(\lambda n.\ u\ x\ y\ n * real\ (Suc\ n) * z\ \hat{}\ n)$
using z **by** (*intro summable-gf-U'*) *simp*
show *pos*: $\forall n.\ 0 \leq u\ x\ y\ n * real\ (Suc\ n) * z\ \hat{}\ n$
using z **by** (*auto intro*!: *mult-nonneg-nonneg u-nonneg*)
show $\exists n.\ 0 < u\ x\ y\ n * real\ (Suc\ n) * z\ \hat{}\ n$
proof (*rule ccontr*)
assume $\neg\ (\exists n.\ 0 < u\ x\ y\ n * real\ (Suc\ n) * z\ \hat{}\ n)$
with *pos* **have** $\forall n.\ u\ x\ y\ n * real\ (Suc\ n) * z\ \hat{}\ n = 0$
by (*intro antisym allI*) (*simp-all add*: *not-less*)
with z **have** $u\ x\ y = (\lambda n.\ 0)$
by (*intro ext*) *simp*
with *u-sums-U*[*of x y*, *THEN sums-unique*] ‹$U\ x\ y \neq 0$› **show** *False*
by *simp*
qed
qed

lemma *inverse-gf-U'-tendsto*:
assumes *recurrent* y $y \in S$
shows $((\lambda x.\ -\ 1\ /\ -$ *gf-U'* $y\ y\ x)$ ---> *real* $(1\ /$ *integral*^{P} $(paths\ y)\ (t\ y)))$
(*at-left* (*1*::*real*))
proof *cases*
assume *inf*: *integral*^{P} $(paths\ y)\ (t\ y) = \infty$
with *gf-U'-tendsto-integral-t*[*of y*] ‹*recurrent y*› ‹$y \in S$›
have *LIM* z (*at-left 1*). *gf-U'* $y\ y\ z$:> *at-top*
by (*auto simp*: *tendsto-PInfty-eq-at-top*)
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*)
then show *?thesis*
by (*auto intro*!: *tendsto-divide-0 simp add*: *inf*)
next
assume *fin*: *integral*^{P} $(paths\ y)\ (t\ y) \neq \infty$
then obtain r **where** r: *integral*^{P} $(paths\ y)\ (t\ y) =$ *ereal* r
by (*cases integral*^{P} $(paths\ y)\ (t\ y)$) *auto*
then have *eq*: *real* $(1\ /$ *integral*^{P} $(paths\ y)\ (t\ y)) = -\ 1\ /\ -\ r$ **and** $1 \leq r$
using *one-le-integral-t*[*OF* ‹*recurrent y*›] **by** (*auto simp add*: *one-ereal-def*)
have *gf-U'*: (*gf-U'* $y\ y$ ---> r) (*at-left* (*1*::*real*))
using *gf-U'-tendsto-integral-t*[*OF* ‹*recurrent y*› ‹$y \in S$›] r
by *simp*
show *?thesis*
using ‹$1 \leq r$› **unfolding** *eq* **by** (*intro tendsto-intros gf-U'*) *simp*
qed

5.9 Positive recurrent

definition *pos-recurrent x* $\longleftrightarrow$ *recurrent x* $\wedge$ $(\int^+\omega.\ t\ x\ \omega\partial paths\ x) \neq \infty$

lemma *pos-recurrentI-communicating*:
assumes *y*: *pos-recurrent y* **and** *x*: $(y, x) \in$ *communicating*
shows *pos-recurrent x*
proof −
let *?E* = $\lambda x.$ *integral*^{P} (*paths x*) (*t x*)
from *y x* **have** *recurrent*: *recurrent y recurrent x* **and** *fin*: *?E y* $\neq \infty$
by (*auto simp*: *pos-recurrent-def recurrent-iffI-communicating*)
have *in-S*[*simp*]: $x \in S\ y \in S$
using *x* **by** (*auto dest*: *communicating-in-S*)
have *pos*: *0* < *real* (*1* / *?E y*)
using *one-le-integral-t*[*OF* ‹*recurrent y*›]
by (*subst zero-less-real-of-ereal*)
(*auto simp add*: *divide-ereal-def ereal-0-gt-inverse positive-integral-positive fin*)

from *fin* **obtain** *r* **where** *r*: *?E y* = *ereal r*
by (*cases ?E y*) *auto*

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

let *?L* = *at-left* (*1*::*real*)
have *le*: *eventually* ($\lambda z.$ *p x y n* $*$ *p y x m* $*$ *z*^(*n* + *m*) $\leq$ (*1* − *gf-U y y z*) / (*1* − *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*^(*n* + *m*)) $*$ (*p y y i* $*$ *z*^*i*)) *sums* ((*p x y n* $*$ *p y x m* $*$ *z*^(*n* + *m*)) $*$ *gf-G y y z*)
unfolding *gf-G-def*
by (*intro sums-mult summable-sums*) (*auto intro*!: *conv summable-sums convergence-G*[**where** *'a*=*real*, *simplified*])
have ($\sum i.$ (*p x y n* $*$ *p y x m* $*$ *z*^(*n* + *m*)) $*$ (*p y y i* $*$ *z*^*i*)) $\leq$ ($\sum i.$ *p x x* (*i* + (*n* + *m*)) $*$ *z*^(*i* + (*n* + *m*)))
proof (*intro allI summable-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 add*: *p-nonneg*)
also have ... $\leq$ *p x x* (*n* + *i* + *m*)
by (*intro prob-reachable-le*) *simp-all*
finally show *p x y n* $*$ *p y x m* $*$ *z* ^ (*n* + *m*) $*$ (*p y y i* $*$ *z* ^ *i*) $\leq$ *p x x* (*i* + (*n* + *m*)) $*$ *z* ^ (*i* + (*n* + *m*))

```
      using z by (auto simp add: ac-simps power-add intro!: mult-left-mono)
    qed
    also have ... ≤ 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: setsum-nonneg mult-nonneg-nonneg p-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 accessible.start[OF ‹y∈S›] 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(1)[OF ‹x∈S› conv] gf-G-eq-gf-U(1)[OF ‹y∈S› conv]
      using gf-G-eq-gf-U(2)[OF ‹x∈S› conv] gf-G-eq-gf-U(2)[OF ‹y∈S› 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 ?E x ≠ ∞
  proof
    assume ?E 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 DERIV-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 DERIV-intros DERIV-gf-U)

      have (gf-U' y y ---> ?E y) ?L
        using ‹recurrent y› ‹y ∈ S› by (rule gf-U'-tendsto-integral-t)
      then have *: (gf-U' y y ---> r) ?L
        by (simp add: r)
      moreover
      have (gf-U' x x ---> ?E x) ?L
```

```
      using ‹recurrent x› ‹x ∈ S› by (rule gf-U'-tendsto-integral-t)
    then have LIM z ?L. − gf-U' x x z :> at-bot
      by (simp add: tendsto-PInfty-eq-at-top ‹?E x = ∞› filterlim-uminus-at-top)
    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)
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)
```

5.10 Stationary distribution

```
definition stationary-distribution :: 's measure ⇒ bool where
  stationary-distribution N ⟷ N = point-measure S (λx. ∫⁺s. emeasure (K s) {x} ∂N)

definition stat C = point-measure S (λx. inverse (∫⁺ω. t x ω ∂paths x) * indicator C x)

lemma sets-stat[simp]: sets (stat C) = Pow S
 by (simp add: stat-def point-measure-def)

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

lemma stationary-distributionD-eq: stationary-distribution N ⟹
   N = point-measure S (λx. ∫⁺s. emeasure (K s) {x} ∂N)
   unfolding stationary-distribution-def .

lemma stationary-distributionD-emeasure:
 assumes N: stationary-distribution N prob-space N and A: A ⊆ S
 shows emeasure N A = (∫⁺s. emeasure (K s) A ∂N)
proof −
 interpret N: prob-space N by fact
```

have *sets-N*: *sets N = Pow S*
by (*subst stationary-distributionD-eq*[*OF N*(*1*)]) (*simp add*: *sets-point-measure*)

have *space-N*: *space N = S*
using *sets-eq-imp-space-eq*[*of N count-space S*] **by** (*simp-all add*: *sets-N*)

then have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable N*
unfolding *measurable-def* **by** (*simp add*: *sets-N space-N*)

have [*simp*]: *countable A* ⋀*x*. *x* ∈ *A* ⟹ *x* ∈ *S*
using *A* **by** (*auto simp add*: *countable-subset*)
then have [*simp*]: *sets* (*count-space A* ⨂$_M$ *N*) = *Pow* (*A* × *S*)
by (*simp add*: *sets-N space-N sets-pair-countable*)

then have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable* (*count-space A* ⨂$_M$ *N*)
using *sets-eq-imp-space-eq*[*of count-space A* ⨂$_M$ *N count-space* (*A* × *S*)]
by (*simp add*: *measurable-def*)

interpret *A*: *sigma-finite-measure count-space A*
by (*rule sigma-finite-measure-count-space*) *simp*

have (∫$^+$ *s*. *emeasure* (*K s*) *A* ∂*N*) = (∫$^+$ *s*. ∫$^+$*a*. *emeasure* (*K s*) {*a*} ∂*count-space A* ∂*N*)
by (*intro positive-integral-cong emeasure-countable-singleton*) (*simp-all add*: *space-N*)
also have ... = (∫$^+$*a*. ∫$^+$*s*. *emeasure* (*K s*) {*a*} ∂*N* ∂*count-space A*)
apply (*rule pair-sigma-finite.Fubini'*)
apply *unfold-locales*
apply *simp*
done
also have ... = (∫$^+$*a*. *emeasure N* {*a*} ∂*count-space A*)
apply (*intro positive-integral-cong*)
apply (*subst* (*2*) *stationary-distributionD-eq*[*OF N*(*1*)])
apply (*simp add*: *point-measure-def emeasure-density*)
proof −
fix *a* **assume** *a* ∈ *A*
have *eq*: ⋀*xa*. (∫$^+$ *s*. *emeasure* (*K s*) {*xa*} ∂*N*) ∗ *indicator* {*a*} *xa* = (∫$^+$ *s*. *emeasure* (*K s*) {*a*} ∂*N*) ∗ *indicator* {*a*} *xa*
by (*simp split*: *split-indicator*)
show (∫$^+$ *s*. *emeasure* (*K s*) {*a*} ∂*N*) = (∫$^+$ *xa*. (∫$^+$ *s*. *emeasure* (*K s*) {*xa*} ∂*N*) ∗ *indicator* {*a*} *xa* ∂*count-space S*)
unfolding *eq* **using** ‹*a* ∈ *A*›
by (*subst positive-integral-cmult*) (*simp-all add*: *positive-integral-positive one-ereal-def*[*symmetric*])
qed
also have ... = *emeasure N A*
by (*intro emeasure-countable-singleton*[*symmetric*]) (*simp-all add*: *sets-N*)
finally show *?thesis* **..**

qed

lemma *stationary-distributionI*:
assumes *finite-measure N*
assumes *N*[*simp*]: *sets N* = *Pow S*
assumes *le*: $\bigwedge y.\ y \in S \Longrightarrow (\int x.\ measure\ (K\ x)\ \{y\}\ \partial N) \leq measure\ N\ \{y\}$
shows *stationary-distribution N*
unfolding *stationary-distribution-def*
proof (*rule measure-eqI-countable*[*OF - - countable-space*])
fix *a* **assume** [*simp*]: $a \in S$
interpret *N*: *finite-measure N* **by** *fact*
interpret *S*: *sigma-finite-measure count-space S*
using *countable-space* **by** (*rule sigma-finite-measure-count-space*)
interpret *NS*: *pair-sigma-finite N count-space S* **..**

have *space-N*[*simp*]: *space N* = *S*
using *sets-eq-imp-space-eq*[*of N count-space S*] **by** *simp*

have [*measurable*]: $\bigwedge f.\ f \in borel\text{-}measurable\ N$
unfolding *measurable-def* **using** *sets-eq-imp-space-eq*[*of N count-space S*]
by (*auto simp add*: *N*)

have *sets* ($N \bigotimes_M$ *count-space S*) = *Pow* (*S* × *S*)
by (*intro sets-pair-countable countable-space N*) *simp*
then have [*measurable*]: $\bigwedge f.\ f \in borel\text{-}measurable\ (N \bigotimes_M count\text{-}space\ S)$
by (*auto simp add*: *measurable-def*)

have ($\int^+ t.\ ereal\ (\int x.\ measure\ (K\ x)\ \{t\}\ \partial N)\ \partial count\text{-}space\ S$)
= ($\int^+ t.\ (\int^+ x.\ emeasure\ (K\ x)\ \{t\}\ \partial N)\ \partial count\text{-}space\ S$)
apply (*simp add*: *K.emeasure-eq-measure*)
apply (*subst* (*2*) *positive-integral-eq-integral*)
apply (*rule N.integrable-const-bound*[**where** *B=1*])
apply (*simp-all add*: *measure-nonneg*)
done
also have ... = ($\int^+ x.\ (\int^+ t.\ emeasure\ (K\ x)\ \{t\}\ \partial count\text{-}space\ S)\ \partial N$)
by (*rule NS.Fubini'*) *simp*
also have ... = ($\int^+ x.\ emeasure\ (K\ x)\ (space\ (K\ x))\ \partial N$)
apply (*intro positive-integral-cong*)
apply *simp*
apply (*subst emeasure-countable-singleton*[*symmetric*])
apply *simp-all*
done
also have ... = *emeasure N S*
unfolding *K.emeasure-space-1* **by** *simp*
finally have *comm*: ($\int^+ t.\ ereal\ (\int x.\ measure\ (K\ x)\ \{t\}\ \partial N)\ \partial count\text{-}space\ S$)
= *ereal* (*measure N S*)
by (*simp add*: *N.emeasure-eq-measure*)

{ assume ($\int^+ x.\ emeasure\ (K\ x)\ \{a\}\ \partial N$) < *emeasure N* {*a*}

```
    then have 0 < ereal (measure N {a} − (∫ x. measure (K x) {a} ∂N))
      apply (simp add: K.emeasure-eq-measure N.emeasure-eq-measure)
      apply (subst (asm) positive-integral-eq-integral)
      apply (rule N.integrable-const-bound[where B=1])
      apply (simp-all add: measure-nonneg)
      done
    also have ... = (∫⁺t. ereal (measure N {a} − (∫ x. measure (K x) {a} ∂N))
∗ indicator {a} t ∂count-space S)
      by (subst positive-integral-cmult-indicator) (simp-all add: le-diff-eq le)
    also have ... ≤ (∫⁺t. ereal (measure N {t}) − ereal (∫ x. measure (K x) {t}
∂N) ∂count-space S)
      by (intro positive-integral-mono) (auto split: split-indicator simp: le-diff-eq le)
    also have ... = (∫⁺t. emeasure N {t} ∂count-space S) − emeasure N S
      by (subst positive-integral-diff)
        (auto intro!: AE-I2 lebesgue-integral-nonneg measure-nonneg le simp: comm
N.emeasure-eq-measure)
    also have ... = 0
    by (subst emeasure-countable-singleton[symmetric]) (simp-all add: N.emeasure-eq-measure)
    finally have False by simp }
  note is-eq = this

  show emeasure N {a} = emeasure (point-measure S (λx. ∫⁺ s. emeasure (K s)
{x} ∂N)) {a}
    apply (simp add: emeasure-point-measure-finite2 positive-integral-positive)
    apply (intro antisym le ‹a ∈ S›)
    apply (rule ccontr)
    apply (simp add: not-le)
    apply (auto dest: is-eq)
    using le[OF ‹a ∈ S›]
    apply (simp add: K.emeasure-eq-measure N.emeasure-eq-measure)
    apply (subst positive-integral-eq-integral)
    apply (rule N.integrable-const-bound[where B=1])
    apply (simp-all add: measure-nonneg)
    done
qed (simp-all add: sets-point-measure)

lemma stationary-distribution-iterate:
  assumes N: stationary-distribution N and finite-measure N and [simp]: y ∈ S
  shows measure N {y} = (∫ x. p x y n ∂N)
proof −
  interpret N: finite-measure N by fact
  interpret S: sigma-finite-measure count-space S
    using countable-space by (rule sigma-finite-measure-count-space)
  interpret NS: pair-sigma-finite N count-space S ..

  from N have N-eq: N = point-measure S (λx. ∫⁺ s. emeasure (K s) {x} ∂N)
    by (simp add: stationary-distribution-def)
  have sets-N[simp]: sets N = Pow S
    by (subst N-eq) (simp add: sets-point-measure)
```

have *space-N*[*simp*]: *space N = S*
using *sets-eq-imp-space-eq*[*of N count-space S*] **by** *simp*
have [*simp*]: $\bigwedge f.\ f \in$ *borel-measurable N*
by (*auto simp*: *measurable-def*)

have *sets* (*N* $\bigotimes_M$ *count-space S*) = *Pow* (*S* × *S*)
by (*intro sets-pair-countable countable-space sets-N*) *simp*
then have [*measurable*]: $\bigwedge f.\ f \in$ *borel-measurable* (*N* $\bigotimes_M$ *count-space S*)
by (*auto simp add*: *measurable-def*)

have *integrable-K*: $\bigwedge x.\ x \in S \Longrightarrow$ *integrable N* (λ*s*. *Sigma-Algebra.measure* (*K s*) {*x*})
by (*rule N.integrable-const-bound*[**where** *B=1*]) (*simp-all add*: *measure-nonneg*)

have *integrable-p*: $\bigwedge n\ y.\ y \in S \Longrightarrow$ *integrable N* (λ*x*. *p x y n*)
by (*rule N.integrable-const-bound*[**where** *B=1*]) (*simp-all add*: *p-nonneg p-le-1*)

have *integrable-p-K*: $\bigwedge n\ x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow$ *integrable* (*K x*) (λ*x*. *p x y n*)
by (*rule K.integrable-const-bound*[**where** *B=1*]) (*simp-all add*: *p-nonneg p-le-1*)

note *N-eq* **also have** *point-measure S* (λ*x*. $\int^+ s.$ *emeasure* (*K s*) {*x*} ∂*N*) =
point-measure S (λ*x*. *ereal* ($\int s.$ *measure* (*K s*) {*x*} ∂*N*))
unfolding *point-measure-def*
by (*intro density-cong*)
(*auto simp*: *AE-count-space K.emeasure-eq-measure integrable-K positive-integral-eq-integral measure-nonneg*)
finally have *N-eq'*: *N = point-measure S* (λ*x*. *ereal* ($\int s.$ *measure* (*K s*) {*x*} ∂*N*)) **.**

show *?thesis*
proof (*induct n*)
case *0*
have *measure N* {*y*} = ($\int x.$ *indicator* {*y*} *x* ∂*N*)
by (*simp add*: *N.emeasure-eq-measure*)
also have ... = ($\int x.$ *p x y 0* ∂*N*)
by (*intro integral-cong*) (*simp add*: *p-0*)
finally show *?case* **.**
next
case (*Suc n*)
then have *Sigma-Algebra.measure N* {*y*} = ($\int^+ x.$ *p x y n* ∂*N*)
by (*simp add*: *positive-integral-eq-integral integrable-p p-nonneg*)
also have ... = ($\int^+ s.$ ($\int^+ x.$ *p x y n* ∗ *emeasure* (*K s*) {*x*} ∂*count-space S*) ∂*N*)
by (*subst N-eq*)
(*simp add*: *point-measure-def positive-integral-density positive-integral-positive positive-integral-cmult*[*symmetric*] *p-nonneg NS.Fubini' ac-simps*)
also have ... = ($\int^+ s.$ ($\int^+ x.$ *p x y n* ∂*K s*) ∂*N*)
apply (*intro positive-integral-cong*)

```
    apply (subst (2) K-eq-density)
    apply (simp-all add: emeasure-nonneg positive-integral-density ac-simps)
    done
  also have ... = (∫⁺s. p s y (Suc n) ∂N)
    apply (intro positive-integral-cong)
    apply (subst p-Suc)
    apply (simp-all add: p-nonneg integrable-p-K positive-integral-eq-integral)
    done
  also have ... = (∫ s. p s y (Suc n) ∂N)
    by (simp add: p-nonneg integrable-p positive-integral-eq-integral)
  finally show ?case by simp
 qed
qed

lemma stat-subprob:
 assumes C: essential-class C and pos: ∀ c∈C. pos-recurrent c
 shows emeasure (stat C) C ≤ 1
proof −
 let ?E = λx. integral^P (paths x) (t x)
 let ?L = at-left (1::real)
 have C ⊆ S
  using C by (auto simp add: essential-class-def quotient-def communicating-in-S)
 from this countable-space have countable C
  by (rule countable-subset)
 from finite-sequence[OF this] guess A . note A = this
 from A ‹C ⊆ S› have (λn. emeasure (stat C) (A n)) −−−−> emeasure (stat
C) (⋃ i. A i)
  by (intro Lim-emeasure-incseq) auto
 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) ‹C ⊆ S› have A-n: finite (A n) A n ⊆ S
   by auto

  from C have C ≠ {}
   by (simp add: essential-class-def not-empty-irreducible)
  then obtain x where x ∈ C by auto

   have ((λz. (∑ y∈A n. gf-F x y z ∗ ((1 − z) / (1 − gf-U y y z)))) −−−>
(∑ y∈A n. F x y ∗ real (1 / ?E y))) ?L
  proof (intro tendsto-intros gf-F, rule lhopital-left)
   fix y assume y ∈ A n
   with ‹A n ⊆ C› ‹C ⊆ S› have y ∈ S y ∈ C
    by auto
   show (op − 1 −−−> 0) ?L
    by (intro tendsto-eq-intros) simp-all
   have recurrent y
    using pos[THEN bspec, OF ‹y∈C›] by (simp add: pos-recurrent-def)
   then have U y y = 1
```

```
      by (simp add: recurrent-iff-U-eq-1)

    show ((λx. 1 − gf-U y y x) −−−> 0) ?L
      using gf-U[of y y] ‹U y y = 1› by (intro tendsto-eq-intros) auto
    show eventually (λx. 1 − gf-U y y x ≠ 0) ?L
      using gf-G-eq-gf-U(2)[OF ‹y ∈ S› convergence-G-less-1, where 'z=real]
by (auto intro!: eventually-at-left-1)
    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 DERIV-intros DERIV-gf-U)
    show eventually (λx. DERIV (op − 1) x :> − 1) ?L
      by (auto intro!: eventually-at-left-1 DERIV-intros)
   show ((λx. − 1 / − gf-U' y y x) −−−> real (1 / integral^P (paths y) (t y)))
?L
      using ‹recurrent y› ‹y ∈ S› by (rule inverse-gf-U'-tendsto)
  qed
  also have (∑ y∈A n. F x y * real (1 / integral^P (paths y) (t y))) = (∑ y∈A
n. real (1 / integral^P (paths y) (t y)))
  proof (intro setsum-cong refl)
    fix y assume y ∈ A n
    with ‹A n ⊆ C› have y ∈ C by auto
    with ‹x ∈ C› have (x, y) ∈ communicating
      by (rule essential-classD3[OF C])
    with ‹y∈C› have recurrent y accessible y x
    using pos[THEN bspec, of y] by (auto simp add: pos-recurrent-def communicating-def)
    then have U x y = 1
      by (rule recurrent-accessible)
    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 * real (1 / integral^P (paths y) (t y)) = real (1 / integral^P
(paths y) (t y))
      by simp
  qed
  finally have le: (∑ y∈A n. real (1 / ?E y)) ≤ 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 ‹C ⊆ S› ‹x ∈ C› have norm z < 1 x ∈ S
      by auto
    then have conv: ⋀x y. convergence-G x y z
      by (simp add: convergence-G-less-1)
    have (∑ y∈A n. gf-F x y z / (1 − gf-U y y z)) = (∑ y∈A n. gf-G x y z)
      using ‹norm z < 1› ‹x ∈ S›
      apply (auto intro!: setsum-cong simp: gf-G-eq-gf-F)
      using ‹A n ⊆ S›
      apply (subst gf-G-eq-gf-U(1)[OF - conv])
      apply auto
      done
```

```
    also have ... = (∑ y∈A n. ∑ n. p x y n * z^n)
      by (simp add: gf-G-def)
    also have ...  = (∑ i. ∑ y∈A n. p x y i *R z^i)
      by (subst suminf-setsum[OF convergence-G[OF conv]]) simp
    also have ...  ≤ (∑ i. z^i)
   proof (intro summable-le summable-setsum convergence-G conv summable-geometric
allI)
      fix l
      have (∑ y∈A n. p x y l *R z ^ l) = (∑ y∈A n. p x y l) * z ^ l
        by (simp add: setsum-left-distrib)
      also have ... ≤ z ^ l
      proof (intro mult-left-le-one-le)
        have (∑ y∈A n. p x y l) = P(ω in paths x. nat-case x ω l ∈ A n)
          unfolding p-def using ‹finite (A n)› ‹A n ⊆ S›
          by (subst finite-measure-finite-Union[symmetric])
           (auto simp: disjoint-family-on-def intro!: arg-cong2[where f=measure])
        then show (∑ y∈A n. p x y l) ≤ 1
          by simp
      qed (insert z, auto simp: setsum-nonneg p-nonneg)
      finally show (∑ y∈A n. p x y l *R z ^ l) ≤ z ^ l .
    qed fact
    also have ... = 1 / (1 - z)
      using sums-unique[OF geometric-sums, OF ‹norm z < 1›] ..
    finally have (∑ y∈A n. gf-F x y z / (1 - gf-U y y z)) ≤ 1 / (1 - z) .
    then have (∑ y∈A n. gf-F x y z / (1 - gf-U y y z)) * (1 - z) ≤ 1
      using z by (simp add: field-simps)
    then have (∑ y∈A n. gf-F x y z / (1 - gf-U y y z) * (1 - z)) ≤ 1
      by (simp add: setsum-left-distrib)
    then show (∑ y∈A n. gf-F x y z * ((1 - z) / (1 - gf-U y y z))) ≤ 1
      by simp
   qed

   from A-n have emeasure (stat C) (A n) = (∑ y∈A n. emeasure (stat C) {y})
     by (intro emeasure-eq-setsum-singleton) (simp-all, blast)
   also have ... = (∑ y∈A n. inverse (?E y))
     unfolding stat-def using ‹A n ⊆ S› A(1)[of n]
     apply (intro setsum-cong refl)
     apply (subst emeasure-point-measure-finite2)
     apply (auto simp: inverse-nonneg-iff positive-integral-positive)
     done
   also have ... = ereal (∑ y∈A n. real (1 / ?E y))
     apply (subst setsum-ereal[symmetric])
   proof (intro setsum-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 ?E y = ereal r 1 ≤ ?E y
       by (cases ?E y) (auto simp: pos-recurrent-def)
     then show inverse (?E y) = ereal (real (1 / ?E y))
```

```
      by (simp add: one-ereal-def inverse-eq-divide)
    qed
    also have ... ≤ 1
      using le by (simp add: one-ereal-def)
    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
proof cases
  assume y ∈ S
  show ?thesis
    unfolding stat-def using ‹y ∈ S› ‹y ∉ C›
    by (subst emeasure-point-measure-finite2) auto
next
  assume y ∉ S
  then have {y} ∉ sets (stat C) by simp
  then show ?thesis
    by (rule emeasure-notin-sets)
qed
```

5.10.1 *stationary-distribution* **implies** *pos-recurrent*

```
lemma stationary-distributionD:
  assumes C: essential-class C
  assumes N: stationary-distribution N and prob-space N and null: measure N
(S − C) = 0
  shows ∀ x∈C. pos-recurrent x N = stat C
proof −
  interpret N: prob-space N by fact
  have N-finite: finite-measure N by unfold-locales
  from N have N-eq: N = point-measure S (λx. ∫⁺ s. emeasure (K s) {x} ∂N)
    by (simp add: stationary-distribution-def)
  have sets-N[simp]: sets N = Pow S
    by (subst N-eq) (simp add: sets-point-measure)
  have space-N[simp]: space N = S
    using sets-eq-imp-space-eq[of N count-space S] by simp
  have [simp]: ⋀f. f ∈ borel-measurable N
    by (auto simp: measurable-def)
  have C ⊆ S
   using C by (auto simp add: essential-class-def quotient-def communicating-in-S)
  then have measure-C: measure N C = 1
    using N.finite-measure-Diff[of S C] N.prob-space null by simp
  from null ‹C ⊆ S› have ae-C: AE x in N. x ∈ C
    using N.prob-eq-0[of S − C] by (auto simp: Diff-subset)
```

have *integrable-K*: $\bigwedge x.\ x \in S \Longrightarrow$ *integrable* N ($\lambda s.$ *Sigma-Algebra.measure* (K s) $\{x\}$)
by (*rule N.integrable-const-bound*[**where** *B=1*]) (*simp-all add*: *measure-nonneg*)

note *N-eq* **also have** *point-measure* S ($\lambda x.$ $\int^+ s.$ *emeasure* (K s) $\{x\}$ ∂N) =
point-measure S ($\lambda x.$ *ereal* ($\int s.$ *measure* (K s) $\{x\}$ ∂N))
unfolding *point-measure-def*
by (*intro density-cong*)
(*auto simp*: *AE-count-space K.emeasure-eq-measure integrable-K positive-integral-eq-integral measure-nonneg*)
finally have *N-eq'*: N = *point-measure* S ($\lambda x.$ *ereal* ($\int s.$ *measure* (K s) $\{x\}$ ∂N)) **.**

have *integral-N-finite*: $\bigwedge A\ f.$ *finite* $A \Longrightarrow A \subseteq S \Longrightarrow$ ($\int x.$ f x $*$ *indicator* A x ∂N) = ($\sum a{\in}A.$ ($\int s.$ *measure* (K s) $\{a\}$ ∂N) $*$ f a)
apply (*subst N-eq'*)
apply (*subst lebesgue-integral-point-measure-finite-indicator*)
apply (*simp-all add*: *measure-nonneg lebesgue-integral-nonneg*)
done

have *integrable-p*: $\bigwedge n\ y.\ y \in S \Longrightarrow$ *integrable* N ($\lambda x.$ p x y n)
by (*rule N.integrable-const-bound*[**where** *B=1*]) (*simp-all add*: *p-nonneg p-le-1*)

{ fix e :: *real* **assume** $0 < e$
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)
have *countable* C
using ‹$C \subseteq S$› *countable-space* **by** (*rule countable-subset*)
from *finite-sequence*[*OF this*] **guess** F **. note** F = *this*
with ‹$C \subseteq S$› **have** $*$: ($\lambda n.$ *measure* N (F n)) ----> *measure* N ($\bigcup i.$ F i)
by (*intro N.finite-Lim-measure-incseq*) *auto*
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 ‹*measure* N C = *1*› ‹*0* < e› **show** *False*
by *simp*
qed
then obtain A **where** $A \subseteq C$ *finite* A **and** e: $1 - e <$ *measure* N A **by** *auto*

{ fix y n **assume** $y \in C$
with ‹$C \subseteq S$› **have** $y \in S$ **by** *auto*
with N *N-finite* **have** *measure* N $\{y\}$ = ($\int x.$ p x y n ∂N)
by (*rule stationary-distribution-iterate*)
also have ... $\leq$ ($\int x.$ p x y n $*$ *indicator* A x + *indicator* ($C - A$) x ∂N)
using *ae-C* ‹$C \subseteq S$› ‹$A \subseteq C$›
by (*intro integral-mono-AE*)
(*auto elim*!: *eventually-elim1*
intro!: *integral-add integral-indicator p-le-1 integrable-mult-indicator*

```
            split: split-indicator simp: integrable-p ‹y∈S›)
    also have ... = (∫ x. p x y n * indicator A x ∂N) + measure N (C - A)
      using ae-C ‹C ⊆ S› ‹A ⊆ C›
      apply (subst integral-add)
      apply (auto elim!: eventually-elim1
              intro!: integral-add integral-indicator p-le-1 integrable-mult-indicator
                split: split-indicator simp: integrable-p ‹y∈S›)
      apply (subst integral-indicator)
      apply (auto simp add: N.emeasure-eq-measure)
      done
    also have ... ≤ (∫ x. p x y n * indicator A x ∂N) + e
      using e ‹C ⊆ S› ‹A ⊆ C› measure-C by (simp add: N.finite-measure-Diff)
    finally have measure N {y} ≤ (∫ x. p x y n * indicator A x ∂N) + e .
    then have emeasure N {y} ≤ ereal (∫ x. p x y n * indicator A x ∂N) + e
      by (simp add: N.emeasure-eq-measure)
    also have ... = (∫⁺x. ereal (p x y n) * indicator A x ∂N) + e
      by (subst positive-integral-eq-integral[symmetric])
        (auto intro!: N.integrable-const-bound[where B=1]
              simp: abs-mult mult-nonneg-nonneg p-nonneg p-le-1 mult-le-one
times-ereal.simps[symmetric] ereal-indicator
            simp del: times-ereal.simps)
    finally have emeasure N {y} ≤ (∫⁺x. ereal (p x y n) * indicator A x ∂N)
+ e . }
  note v-le = this

  { fix y and z :: real assume y: y ∈ C and z: 0 < z z < 1
    have summable-int-p: summable (λn. (∫ x. p x y n * indicator A x ∂N) *
(1 - z) * z ^ n)
      using ‹y∈C› z ‹C ⊆ S› ‹A ⊆ C›
    by (auto intro!: summable-comparison-test[OF - summable-mult[OF summable-geometric[of
z], of 1]] exI[of - 0] mult-le-one
                     N.integral-le-const integrable-mult-indicator integrable-p
AE-I2 p-le-1
          simp: abs-mult lebesgue-integral-nonneg p-nonneg mult-nonneg-nonneg)

    from y z have sums-y: (λn. measure N {y} * (1 - z) * z ^ 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} = ereal (∑ n. (measure N {y} * (1 - z)) * z ^
n)
      by (simp add: sums-unique N.emeasure-eq-measure)
    also have ... = (∑ n. emeasure N {y} * (1 - z) * z ^ n)
      using z
      by (subst suminf-ereal[symmetric])
        (auto intro!: mult-nonneg-nonneg measure-nonneg suminf-ereal-finite
         summable-mult[OF summable-geometric[of z]] simp: N.emeasure-eq-measure)
    also have ... ≤ (∑ n. ((∫⁺x. ereal (p x y n) * indicator A x ∂N) + e) * (1
- z) * z ^ n)
```

using ‹$y \in C$› z ‹$C \subseteq S$› ‹$A \subseteq C$›
by (*intro suminf-le-pos ereal-mult-right-mono v-le*)
(*auto simp*: *N.emeasure-eq-measure mult-nonneg-nonneg measure-nonneg*)
also have ... = $(\sum n.\ (\int^+ x.\ ereal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * (1 - z) * z\ \hat{}\ n) + e$
using ‹$0 < e$› z *sums-mult*[*OF geometric-sums*[*of z*], *of* $e * (1 - z)$]
by (*simp add*: *ereal-left-distrib positive-integral-positive mult-nonneg-nonneg suminf-add-ereal sums-suminf-ereal*)
also have ... = $(\sum n.\ (1 - z) * ((\int^+ x.\ ereal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * z\ \hat{}\ n)) + e$
by (*simp add*: *ac-simps*)
also have ... = $(1 - z) * (\sum n.\ ((\int^+ x.\ ereal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * z\ \hat{}\ n)) + e$
using z **by** (*subst suminf-cmult-ereal*) (*simp-all add*: *ereal-0-le-mult positive-integral-positive*)
also have $(\sum n.\ ((\int^+ x.\ ereal\ (p\ x\ y\ n) * indicator\ A\ x\ \partial N) * z\ \hat{}\ n)) =$
$(\sum n.\ (\int^+ x.\ ereal\ (p\ x\ y\ n * z\ \hat{}\ n) * indicator\ A\ x\ \partial N))$
using z **by** (*simp add*: *ac-simps positive-integral-cmult*[*symmetric*])
also have ... = $(\int^+ x.\ ereal\ (gf\text{-}G\ x\ y\ z) * indicator\ A\ x\ \partial N)$
using z
apply (*subst positive-integral-suminf*[*symmetric*])
apply (*auto simp add*: *ereal-0-le-mult mult-nonneg-nonneg p-nonneg gf-G-def*
intro!: *positive-integral-cong suminf-ereal' split*: *split-indicator*)
apply (*intro summable-comparison-test*[*OF - summable-mult*[*OF summable-geometric*[*of z*], *of 1*]] *impI*)
apply (*simp-all add*: *abs-mult p-nonneg p-le-1 mult-le-one power-le-one split*: *split-indicator*)
done
also have ... = $(\int^+ x.\ ereal\ (gf\text{-}F\ x\ y\ z * gf\text{-}G\ y\ y\ z) * indicator\ A\ x\ \partial N)$
using z **by** (*intro positive-integral-cong*) (*simp add*: *gf-G-eq-gf-F*[*symmetric*])
also have ... = $gf\text{-}G\ y\ y\ z * (\int^+ x.\ ereal\ (gf\text{-}F\ x\ y\ z) * indicator\ A\ x\ \partial N)$
using z **by** (*subst positive-integral-cmult*[*symmetric*]) (*simp-all add*: *gf-G-nonneg ac-simps*)
also have ... = $(1 / (1 - gf\text{-}U\ y\ y\ z)) * (\int^+ x.\ ereal\ (gf\text{-}F\ x\ y\ z) * indicator\ A\ x\ \partial N)$
using z ‹$y \in C$› ‹$C \subseteq S$› **by** (*subst gf-G-eq-gf-U*) (*auto intro*!: *convergence-G-less-1*)
finally have *emeasure N* $\{y\} \leq ereal\ ((1 - z) / (1 - gf\text{-}U\ y\ y\ z)) * (\int^+ x.\ gf\text{-}F\ x\ y\ z * indicator\ A\ x\ \partial N) + e$
by (*subst* (*asm*) *mult-assoc*[*symmetric*]) (*simp add*: *ereal-indicator*[*symmetric*])
then have *measure N* $\{y\} \leq (1 - z) / (1 - gf\text{-}U\ y\ y\ z) * (\int x.\ gf\text{-}F\ x\ y\ z * indicator\ A\ x\ \partial N) + e$
using z
by (*subst* (*asm*) *positive-integral-eq-integral*[*OF N.integrable-const-bound*[**where** *B=1*]])
(*auto simp*: *gf-F-nonneg mult-nonneg-nonneg N.emeasure-eq-measure gf-F-le-1 mult-le-one*) **}**
then have $\exists A \subseteq C.\ finite\ A \wedge (\forall y \in C.\ \forall z.\ 0 < z \longrightarrow z < 1 \longrightarrow measure\ N\ \{y\} \leq (1 - z) / (1 - gf\text{-}U\ y\ y\ z) * (\int x.\ gf\text{-}F\ x\ y\ z * indicator\ A\ x\ \partial N) + e)$
using ‹$A \subseteq C$› ‹*finite A*› **by** *auto* **}**

note *eps* = *this*

{ fix *y A* **assume** *y* ∈ *C finite A A* ⊆ *C*
with ‹*C* ⊆ *S*› **have** ((λ*z*. ∫ *x*. *gf-F x y z* ∗ *indicator A x* ∂*N*) −−−> ∫ *x*. *F x y* ∗ *indicator A x* ∂*N*) (*at-left 1*)
by (*subst* (*1 2*) *integral-N-finite*) (*auto intro*!: *tendsto-intros gf-F*) **}**
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 ((λ*z*. (*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*: ((λ*z*. (*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*
using ‹*C* ⊆ *S*› *countable-subset*[*OF* ‹*C* ⊆ *S*› *countable-space*]
by (*subst emeasure-countable-singleton*) (*auto simp*: *N.emeasure-eq-measure positive-integral-0-iff-AE*)
then show *False*
using ‹*measure N C* = *1*› **by** (*simp add*: *N.emeasure-eq-measure*)
qed
then have ⋀*x*. *x* ∈ *C* ⟹ *U x x* = *1*

by (*metis recurrent-iff-U-eq-1*)

{ fix y **assume** $y \in C$
then have $y \in S$ $U\ y\ y = 1$ *recurrent* y
using ‹$y \in C \Longrightarrow U\ y\ y = 1$› ‹$C \subseteq S$› *all-recurrent* **by** *auto*
have *measure* N $\{y\} \leq$ *real* $(1\ /\ integral^P\ (paths\ y)\ (t\ y))$
proof (*rule field-le-epsilon*)
fix e :: *real* **assume** $0 < e$
from *eps*[*OF* ‹$0 < e$›] ‹$y \in C$› **obtain** A **where**
A: *finite* A $A \subseteq C$ **and**
le: $\bigwedge z.\ 0 < z \Longrightarrow z < 1 \Longrightarrow$ *measure* N $\{y\} \leq (1 - z)\ /\ (1 - gf\text{-}U\ y\ y\ z) * (\int x.\ gf\text{-}F\ x\ y\ z * indicator\ A\ x\ \partial N) + e$
by *auto*
let *?L* = *at-left* (*1*::*real*)
have $((\lambda z.\ (1 - z)\ /\ (1 - gf\text{-}U\ y\ y\ z) * (\int x.\ gf\text{-}F\ x\ y\ z * indicator\ A\ x\ \partial N) + e) \longrightarrow$
real $(1\ /\ integral^P\ (paths\ y)\ (t\ y)) * (\int x.\ F\ x\ y * indicator\ A\ x\ \partial N) + e)$ *?L*
proof (*intro tendsto-add tendsto-const tendsto-mult int-gf-F*,
rule lhopital-left[**where** f'=$\lambda x.\ -1$ **and** g'=$\lambda z.\ -\ gf\text{-}U'\ y\ y\ z$])
show $(op - 1 \longrightarrow 0)$ *?L* $((\lambda x.\ 1 - gf\text{-}U\ y\ y\ x) \longrightarrow 0)$ *?L*
using *gf-U*[*of y y*] **by** (*auto intro*!: *tendsto-eq-intros simp*: ‹$U\ y\ y = 1$›)
show $y \in C$ *finite* A $A \subseteq C$ **by** *fact*+
show *eventually* $(\lambda x.\ 1 - gf\text{-}U\ y\ y\ x \neq 0)$ *?L*
using *gf-G-eq-gf-U*(*2*)[*OF* ‹$y \in S$› *convergence-G-less-1*, **where** *'z*=*real*] **by** (*auto intro*!: *eventually-at-left-1*)
show $((\lambda x.\ -1\ /\ -\ gf\text{-}U'\ y\ y\ x) \longrightarrow$ *real* $(1\ /\ integral^P\ (paths\ y)\ (t\ y)))$ *?L*
using ‹*recurrent y*› ‹$y \in S$› **by** (*rule inverse-gf-U'-tendsto*)
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 DERIV-intros DERIV-gf-U*)
show *eventually* $(\lambda x.\ DERIV\ (op - 1)\ x :> -1)$ *?L*
by (*auto intro*!: *eventually-at-left-1 DERIV-intros*)
qed
then have *measure* N $\{y\} \leq$ *real* $(1\ /\ integral^P\ (paths\ y)\ (t\ y)) * (\int x.\ F\ x\ y * indicator\ A\ x\ \partial N) + e$
by (*rule tendsto-le*[*OF trivial-limit-at-left-real - tendsto-const*]) (*intro eventually-at-left-1 le*)
then have *measure* N $\{y\} - e \leq$ *real* $(1\ /\ integral^P\ (paths\ y)\ (t\ y)) * (\int x.\ F\ x\ y * indicator\ A\ x\ \partial N)$
by *simp*
also have $\ldots \leq$ *real* $(1\ /\ integral^P\ (paths\ y)\ (t\ y))$
using A ‹$C \subseteq S$›
by (*intro mult-left-le real-of-ereal-pos zero-le-divide-ereal positive-integral-positive N.integral-le-const N.integrable-const-bound*[**where** B=*1*])

```
        (auto simp: F-nonneg mult-nonneg-nonneg mult-le-one F-le-1)
    finally show measure N {y} ≤ real (1 / integral^P (paths y) (t 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 real (1 / (∫⁺ω. t y ω∂paths y)) = 0
    by (simp add: pos-recurrent-def)
   with measure-y-le[OF ‹y ∈ C›] have measure N {y} = 0
    by (auto intro!: antisym simp: measure-nonneg pos-recurrent-def) }
  then have emeasure N C = 0
   using ‹C ⊆ S› countable-subset[OF ‹C ⊆ S› countable-space]
   by (subst emeasure-countable-singleton) (auto simp: N.emeasure-eq-measure
positive-integral-0-iff-AE)
  then show False
   using ‹measure N C = 1› by (simp add: N.emeasure-eq-measure)
 qed

 let ?E = λx. integral^P (paths x) (t x)
 { fix A assume A ⊆ S
  from this countable-space have [simp]: countable A by (rule countable-subset)
  have emeasure N A = (∫⁺x. emeasure N {x} ∂count-space A)
   using ‹A ⊆ S› by (intro emeasure-countable-singleton) auto
  also have ... ≤ (∫⁺x. emeasure (stat C) {x} ∂count-space A)
  proof (intro positive-integral-mono)
   fix y assume y ∈ space (count-space A)
   with ‹A ⊆ S› have y ∈ S by auto
   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: ?E y = ereal r 1 ≤ ?E y
     by (cases ?E y) (auto simp: pos-recurrent-def)

    from measure-y-le[OF ‹y ∈ C›]
    have emeasure N {y} ≤ ereal (real (1 / integral^P (paths y) (t y)))
     by (simp add: N.emeasure-eq-measure)
    also have ... = emeasure (stat C) {y}
     unfolding stat-def using ‹C ⊆ S› ‹y ∈ C› r
     by (subst emeasure-point-measure-finite2)
```

```
        (auto simp: inverse-nonneg-iff positive-integral-positive divide-ereal-def
one-ereal-def)
      finally show emeasure N {y} ≤ emeasure (stat C) {y}
        by simp
    next
      assume y ∉ C
      with ‹y ∈ S› ae-C have emeasure N {y} = 0
       by (subst AE-iff-measurable[symmetric, where P=λx. x ≠ y]) (auto elim!:
eventually-elim1)
      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
    using ‹A ⊆ S› 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 pos] N-le-C[OF ‹C ⊆ S›] ‹measure N C = 1›
 have emeasure (stat C) C = 1
   by (auto simp add: N.emeasure-eq-measure one-ereal-def)
 moreover have emeasure (stat C) (S − C) = 0
   using ‹C ⊆ S›
   by (subst emeasure-countable-singleton)
    (auto simp: sets-point-measure emeasure-stat-not-C intro!: positive-integral-0-iff-AE[THEN
iffD2])
 ultimately have emeasure (stat C) (space (stat C)) = 1
    using plus-emeasure[of C stat C S − C] ‹C ⊆ S› by (simp add: Diff-subset
Un-absorb1)
 interpret stat: prob-space stat C
   by default fact

 show N = stat C
 proof (rule measure-eqI-countable)
   show sets N = Pow S sets (stat C) = Pow S countable S
     by auto

   { assume ∃x∈S. emeasure N {x} ≠ emeasure (stat C) {x}
     then obtain x where [simp]: x ∈ S and 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
     have 1 = emeasure N {x} + emeasure N (S − {x})
     using N.emeasure-space-1 by (subst plus-emeasure) (auto simp: insert-absorb)
     also have ... < emeasure (stat C) {x} + emeasure (stat C) (S − {x})
         using x N-le-C[of S − {x}] by (simp add: stat.emeasure-eq-measure
N.emeasure-eq-measure Diff-subset)
     also have ... = 1
```

using *stat.emeasure-space-1* **by** (*subst plus-emeasure*) (*auto simp*: *insert-absorb*)
finally have *False* **by** *simp* **}**
then show !!x. $x \in S \Longrightarrow$ *emeasure* N $\{x\}$ = *emeasure* (*stat* C) $\{x\}$ **by** *metis*
qed
qed

lemma *stationary-distribution-imp-int-t*:
assumes C: *essential-class* C *stationary-distribution* N *prob-space* N *measure* N $(S - C) = 0$
assumes x: $x \in C$ **shows** $(\int^+\omega.\ t\ x\ \omega\ \partial paths\ x) = 1\ /$ *measure* N $\{x\}$
proof −
have $C \subseteq S$
using C **by** (*auto simp*: *essential-class-def C-subset-S*)
with x **have** [*simp*]: $x \in S$ **by** *auto*
from *stationary-distributionD*[*OF C*] **have** N = *stat* C **and** $*$: $\forall x{\in}C.$ *pos-recurrent* x **by** *auto*
show *?thesis*
unfolding ‹N = *stat* C› *stat-def*
using $*$[*THEN bspec*, *OF x*] x
apply (*cases* $(\int^+\omega.\ t\ x\ \omega\ \partial paths\ x)$)
apply (*auto simp add*: *measure-def emeasure-point-measure-finite2 inverse-nonneg-iff*
mult-nonneg-nonneg positive-integral-positive pos-recurrent-def
divide-inverse)
done
qed

5.11 Aperiodic classes

definition *period-set* $x = \{i.\ 0 < i \wedge 0 < p\ x\ x\ i\ \}$

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

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

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

end

inductive-set *monoid-closure* :: *nat set* ⇒ *int set* **for** S **where**
base: $s \in S \Longrightarrow$ *int* $s \in$ *monoid-closure* S
| *plus*: $s \in$ *monoid-closure* $S \Longrightarrow t \in$ *monoid-closure* $S \Longrightarrow s + t \in$ *monoid-closure* S
| *diff*: $s \in$ *monoid-closure* $S \Longrightarrow t \in$ *monoid-closure* $S \Longrightarrow s - t \in$ *monoid-closure* S

lemma *Gcd-nonneg*: $0 \leq$ *Gcd* (S :: *int set*)
by (*simp add*: *Gcd-int-def*)

lemma *Gcd-eq-Gcd-monoid-closure*:
fixes S :: *nat set*
shows *Gcd* S = *Gcd* (*monoid-closure* S)
apply (*simp add*: *Gcd-int-def*)
proof (*safe intro*!: *dvd.antisym gcd-lcm-complete-lattice-nat.Inf-greatest*)
fix s **assume** $s \in$ *monoid-closure* S
then show *Gcd* S *dvd nat* $|s|$
proof *induct*
case (*plus* s t)
then show *?case*
by (*cases* s t *rule*: *int-cases'*[*case-product int-cases'*])
(*simp-all add*: *uminus-add-conv-diff nat-abs-int-diff nat-int-add*)
next
case (*diff* s t)
then show *?case*
by (*cases* s t *rule*: *int-cases'*[*case-product int-cases'*])
(*simp-all add*: *uminus-add-conv-diff nat-abs-int-diff nat-int-add*)
qed *simp*
next
fix s **assume** $s \in S$
then show *Gcd* ((λx. *nat* $|x|$) ' *monoid-closure* S) *dvd* s
by (*intro Gcd-dvd-nat image-eqI*[**where** x=*int* s] *monoid-closure.base*) *simp*
qed

lemma *monoid-closure-nat-mult*:
$s \in$ *monoid-closure* $S \Longrightarrow$ *int* $n * s \in$ *monoid-closure* S
proof (*induct* n)
case *0* **with** *monoid-closure.diff*[*of s S s*] **show** *?case* **by** *simp*
next
case (*Suc* n) **then show** *?case*
by (*simp add*: *field-simps monoid-closure.plus*)
qed

lemma *monoid-closure-uminus*:
assumes s: $s \in$ *monoid-closure* S **shows** $-s \in$ *monoid-closure* S
using *monoid-closure.diff*[*OF monoid-closure.diff*[*OF s s*] *s*] **by** *simp*

lemma *monoid-closure-int-mult*:
$s \in$ *monoid-closure* $S \Longrightarrow i * s \in$ *monoid-closure* S
by (*cases i rule*: *int-cases'*) (*auto intro*: *monoid-closure-uminus monoid-closure-nat-mult*)

lemma *monoid-closure-setsum*:
assumes X: *finite* X $X \neq \{\}$ $X \subseteq S$
shows $(\sum x{\in}X.\ a\ x * int\ x) \in$ *monoid-closure* S
using X
apply (*induct* X *rule*: *finite-ne-induct*)
apply (*auto intro*!: *monoid-closure-int-mult monoid-closure.base monoid-closure.plus*)
done

```
lemma Gcd-monoid-closure-in-monoid-closure:
  fixes S :: nat set
  assumes s ∈ S s ≠ 0
  shows Gcd (monoid-closure S) ∈ monoid-closure S
proof −
  def m ≡ LEAST n. n ≠ 0 ∧ int n ∈ monoid-closure S

  from ‹s ∈ S› have s: int s ∈ monoid-closure S
    by (rule monoid-closure.base)
  moreover from ‹s ≠ 0› have 0 < s
    by (metis neq0-conv)
  ultimately have s': s ≠ 0 ∧ int s ∈ monoid-closure S
    by auto

  have m: m ≠ 0 ∧ m ∈ monoid-closure S
    unfolding m-def
    apply (rule LeastI[of - s])
    using s'
    by simp
  then have m': m ∈ monoid-closure S 0 < int m
    by auto

  have Gcd (monoid-closure S) = int m
    apply (simp add: Gcd-int-def)
    apply (safe intro!: gcd-lcm-complete-lattice-nat.Inf-eqI)
  proof −
    fix y assume ∀ i. i ∈ (λx. nat |x|) ‘ monoid-closure S ⟶ y dvd i
    moreover have m ∈ (λx. nat |x|) ‘ monoid-closure S
      using m apply (intro image-eqI[where x=int m]) by auto
    ultimately show y dvd m by auto
  next
    fix i s assume s: s ∈ monoid-closure S
    from zmod-zdiv-equality'[of s m]
    have s mod int m = s − s div int m ∗ int m
      by auto
    also have s − s div int m ∗ int m ∈ monoid-closure S
      by (intro monoid-closure.diff s monoid-closure-int-mult m')
    finally have mod-in: s mod int m ∈monoid-closure S by auto

    have int m dvd s
    proof cases
      assume s mod int m = 0
      with dvd-eq-mod-eq-0[of int m s] show ?thesis by simp
    next
      assume s mod int m ≠ 0
      moreover have 0 ≤ s mod int m
        using m'(2) by (rule pos-mod-sign[of int m s])
      ultimately have pos: 0 < s mod int m by auto
      moreover have s mod int m < int m
```

```
      using m'(2) by (rule pos-mod-bound)
    moreover have m ≤ nat (s mod int m)
      using pos
      apply (subst m-def)
      apply (intro Least-le)
      apply (auto intro: mod-in)
      done
   ultimately show ?thesis
     by auto
  qed
  then show m dvd nat |s|
    unfolding int-dvd-iff [symmetric] .
 qed
 with m' show ?thesis by simp
qed

lemma Gcd-in-monoid-closure:
 fixes S :: nat set
 shows s ∈ S ⟹ s ≠ 0 ⟹ Gcd S ∈ monoid-closure S
 unfolding Gcd-eq-Gcd-monoid-closure by (rule Gcd-monoid-closure-in-monoid-closure)

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 −
 have Gcd S ∈ monoid-closure S
   using s by (rule Gcd-in-monoid-closure)
 also have monoid-closure S = {s − t | s t. (s = 0 ∨ s ∈ S) ∧ (t = 0 ∨ t ∈ S)
}
  proof auto
   fix x assume x ∈ monoid-closure S
   then show ∃ s t. x = int s − int t ∧ (s = 0 ∨ s ∈ S) ∧ (t = 0 ∨ t ∈ S)
   proof induct
     case (base x) then show ?case
       apply (rule-tac x=x in exI)
       apply (rule-tac x=0 in exI)
       apply auto
       done
   next
     case (plus x y)
     then obtain a b c d where
       a = 0 ∨ a ∈ S b = 0 ∨ b ∈ S x = int a − b
       c = 0 ∨ c ∈ S d = 0 ∨ d ∈ S y = int c − d
       by auto
     then show ?case
       apply (rule-tac x=a + c in exI)
       apply (rule-tac x=b + d in exI)
```

```
      apply simp
      apply (metis S add-0-right add-0-left)
      done
    next
      case (diff x y)
      then obtain a b c d where
        a = 0 ∨ a ∈ S b = 0 ∨ b ∈ S x = int a − b
        c = 0 ∨ c ∈ S d = 0 ∨ d ∈ S y = int c − d
        by auto
      then show ?case
        apply (rule-tac x=a + d in exI)
        apply (rule-tac x=b + c in exI)
        apply simp
        apply (metis S add-0-right add-0-left)
        done
    qed
  next
    show 0 ∈ monoid-closure S
      using monoid-closure.diff[of int s S int s] monoid-closure.base[OF ‹s ∈ S›]
by simp
  qed (auto intro: monoid-closure.base monoid-closure-uminus monoid-closure.diff)
  finally obtain s t :: nat where st: s = 0 ∨ s ∈ S t = 0 ∨ t ∈ S Gcd S = int
s − int t by auto
  then have Gcd-S: Gcd S = s − t
    by auto
  with gcd-lcm-complete-lattice-nat.Inf-top-conv[of S] 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 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 ∈ S t ∈ S t ≠ 0 by auto
    then have Gcd S dvd t by auto
    then obtain a where a: t = Gcd S ∗ a ..
    with ‹t ≠ 0› have 0 < a by auto
```

```
    show ∃ N. ∀ n≥N. n * Gcd S ∈ S
    proof (safe intro!: exI[of - a * a])
      fix n
      def m ≡ (n − a * a) div a
      def r ≡ (n − a * a) mod a
      with ‹0 < a› have r < a by simp
      moreover def am ≡ a + m
      ultimately have r < am by simp
      assume a * a ≤ 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 ... = 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 ... ∈ S
        using ‹s ∈ S› ‹t ∈ S› ‹r < am›
        by (cases r = 0) (auto intro!: cmult-S S)
      finally show n * Gcd S ∈ S .
    qed
  qed
qed

context Discrete-Markov-Kernel
begin

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!: accessibleD-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 y x y k k + l] c
    have k-l: 0 < p y y (k + l)
      by (simp add: communicating-in-S)
    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 y x x k k + n] c
    have 0 < p y x (k + n)
      by (simp add: communicating-in-S)
```

```
    from mult-pos-pos[OF this(1) l] prob-reachable-le[of y x y k + n (k + n) + l]
c
    have 0 < p y y (k + n + l)
      by (simp add: communicating-in-S)
    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 (blast elim: dvd-plusE) }
  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 (metis dvd.antisym gcd-lcm-complete-lattice-nat.Inf-greatest)
qed

lemma period-eq:
  assumes C ∈ S // 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)

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)
  with ‹C = {x}› show ?thesis by auto
next
  from C have irr: C ∈ S // 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 accessibleD-pos[of x y] accessibleD-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 simp: mult-pos-pos)
  also have p x y k * p y x (k + l − k) ≤ p x x (k + l)
```

```
    using prob-reachable-le[of x y x k k + l] c by (auto simp: communicating-in-S)
  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 ∈ S // communicating
    using C by (auto simp: not-ephemeral-def)

  have period C ≠ 0
    unfolding period-eq[OF C' ‹x ∈ C›] gcd-lcm-complete-lattice-nat.Inf-top-conv
    using n by (auto simp: period-set-def)
  then show ?thesis by auto
qed

lemma period-posD:
  assumes C: C ∈ S // 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›] gcd-lcm-complete-lattice-nat.Inf-top-conv
    unfolding period-set-def by simp
qed

lemma eventually-periodic:
  assumes C: C ∈ S // 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 ∈ S // communicating
    using C by auto

  have period C ≠ 0
    unfolding period-eq[OF C' ‹x ∈ C›] gcd-lcm-complete-lattice-nat.Inf-top-conv
    using n by (auto simp: period-set-def)
   moreover 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 mult-pos-pos*)
moreover have $p\ x\ x\ k * p\ x\ x\ l \leq p\ x\ x\ (k + l)$
using *prob-reachable-le*[*of* $x\ x\ k\ k + l$] *C-subset-S*[*OF* C'] ‹$x \in C$›
by *auto*
ultimately show $k + l \in$ *period-set* x
using ‹$0 < l$› **by** (*auto simp*: *period-set-def*)
qed
ultimately show *eventually* $(\lambda m.\ 0 < p\ x\ x\ (m * period\ C))$ *sequentially*
unfolding *period-eq*[*OF* C' ‹$x \in C$›, *symmetric*]
using *eventually-ge-at-top*[*of 1*]
by (*auto simp add*: *period-set-def elim*!: *eventually-elim2*)
qed

5.11.1 *aperiodic* is equal to non-zero p

lemma *aperiodic-eventually-recurrent*:
aperiodic $C \longleftrightarrow C \in S$ // *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: $C \in S$ // *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*)
show *aperiodic* C
unfolding *period-eq*[*OF* C ‹$x \in C$›] *aperiodic-def*
proof (*intro conjI gcd-lcm-complete-lattice-nat.Inf-eqI*)
fix y **assume** $\bigwedge i.\ i \in$ *period-set* $x \Longrightarrow y$ *dvd* i
from *this*[*of Suc N*] *this*[*of Suc* (*Suc N*)] ‹$\{N <..\} \subseteq$ *period-set* x›
have y *dvd* (*Suc* (*Suc N*) − *Suc N*)
by (*intro dvd-diff-nat*) (*auto simp*: *subset-eq*)
then show y *dvd 1* **by** *simp*
qed (*simp-all add*: C)
qed (*simp add*: *aperiodic-def*)

end

5.12 Product Construction for Markov Chains

locale *Pair-Discrete-Markov-Kernel* =

K1!: *Discrete-Markov-Kernel S1 K1* + *K2*!: *Discrete-Markov-Kernel S2 K2* **for** *S1 S2 K1 K2*
begin

definition *Sp* = *S1* × *S2*
definition *Kp* = (λ(*s1*, *s2*). *K1 s1* ⨂$_M$ *K2 s2*)

lemma *SpI*[*simp*]: *x1* ∈ *S1* ⟹ *x2* ∈ *S2* ⟹ (*x1*, *x2*) ∈ *Sp*
by (*simp add*: *Sp-def*)

lemma *measurable-K1K2*[*measurable*]: *x1* ∈ *S1* ⟹ *x2* ∈ *S2* ⟹ *f* ∈ *borel-measurable* (*K1 x1* ⨂$_M$ *K2 x2*)
unfolding *measurable-def*
by (*simp add*: *space-pair-measure sets-pair-countable*[*OF K1.countable-space K2.countable-space*])

sublocale *K1K2*!: *pair-prob-space K1 s1 K2 s2* **for** *s1 s2* **by** *default*
sublocale *Kp*!: *prob-space Kp s* **for** *s*
apply (*cases s*)
apply (*simp add*: *Kp-def*)
apply *unfold-locales*
done

sublocale *P*!: *Discrete-Markov-Kernel Sp Kp*
proof
show *countable Sp*
by (*simp add*: *Sp-def*)
show *Sp* ≠ {}
using *K1.non-empty-space K2.non-empty-space* **by** (*simp add*: *Sp-def*)
{ fix *s* **show** *sets* (*Kp s*) = *Pow Sp*
by (*cases s*) (*simp add*: *Sp-def Kp-def sets-pair-countable*) **}**
qed

lemma *sets-Sp*: *sets* (*count-space Sp*) = *sets* (*count-space S1* ⨂$_M$ *count-space S2*)
by (*subst sets-pair-countable*[*of S1 S2*]) (*simp-all add*: *Sp-def*)

lemma *sets-S-seq-Sp*: *sets P.S-seq* = *sets* (Π$_M$ *i*∈*UNIV*. *count-space S1* ⨂$_M$ *count-space S2*)
by (*intro sets-PIM-cong refl sets-Sp*)

lemma *space-S-seq-Sp*: *space P.S-seq* = *space* (Π$_M$ *i*∈*UNIV*. *count-space S1* ⨂$_M$ *count-space S2*)
using *sets-eq-imp-space-eq*[*OF sets-S-seq-Sp*] **.**

lemma *prod-measurable*: (λ(*ω1*, *ω2*) *n*. (*ω1 n*, *ω2 n*)) ∈ *measurable* (*K1.paths x1* ⨂$_M$ *K2.paths x2*) *P.S-seq*
proof −
have *space-Sp*: *space* (*count-space Sp*) = *space* (*count-space S1* ⨂$_M$ *count-space S2*)

by (*simp add*: *space-pair-measure Sp-def*)
have *eq*: $\bigwedge i.\ (\lambda\omega.\ (case\ \omega\ of\ (\omega 1, \omega 2) \Rightarrow \lambda n.\ (\omega 1\ n, \omega 2\ n))\ i) = (\lambda\omega.\ (fst\ \omega\ i, snd\ \omega\ i))$
by *auto*
have *measurable-Sp*: $\bigwedge f\ M.\ f \in measurable\ M\ (count\text{-}space\ Sp) \longleftrightarrow f \in measurable\ M\ (count\text{-}space\ S1 \bigotimes_M count\text{-}space\ S2)$
unfolding *measurable-def space-Sp sets-Sp* **..**

have *PiE*: $(\lambda\omega.\ case\ \omega\ of\ (\omega 1, \omega 2) \Rightarrow \lambda n.\ (\omega 1\ n, \omega 2\ n)) \in space\ (K1.paths\ x1) \times space\ (K2.paths\ x2) \to UNIV \to_E Sp$
by (*auto simp*: *space-PiM PiE-iff*)

show *?thesis*
apply (*intro measurable-PiM-single'*)
unfolding *eq measurable-Sp*
apply (*intro measurable-Pair*)
apply (*simp-all add*: *space-pair-measure PiE del*: *K1.space-paths K2.space-paths*)
done
qed

lemma *P-paths-eq-prod*:
assumes [*simp*]: *x1* ∈ *S1 x2* ∈ *S2*
shows $P.paths\ (x1, x2) = distr\ (K1.paths\ x1 \bigotimes_M K2.paths\ x2)\ P.S\text{-}seq\ (\lambda(\omega 1, \omega 2)\ n.\ (\omega 1\ n, \omega 2\ n))$
(**is** *?L* = *?R*)
proof (*rule nat-eq-cyl*)
show *sets ?L* = *sets P.S-seq sets ?R* = *sets P.S-seq* **by** *simp-all*

have *sp-eq*: $(\lambda(\omega 1, \omega 2)\ n.\ (\omega 1\ n, \omega 2\ n)) -` space\ (P.S\text{-}seq) \cap space\ (K1.paths\ x1) \times space\ (K2.paths\ x2) =$
$space\ (K1.paths\ x1) \times space\ (K2.paths\ x2)$
by (*auto simp*: *space-PiM PiE-iff*)

show *emeasure ?R* (*space P.S-seq*) ≠ ∞
apply (*subst emeasure-distr*[*OF prod-measurable*])
apply (*simp-all add*: *space-pair-measure sp-eq del*: *K1.space-paths K2.space-paths*)
apply (*subst K2.path.emeasure-pair-measure-Times*)
apply *simp-all*
done
show *countable Sp* **by** *simp*

fix *n* :: *nat* **and** ω' :: *nat* ⇒ - **assume** ω': $\bigwedge i.\ \omega'\ i \in Sp$
{ fix *i* **from** ω'[*of i*] **have** *fst* (ω' *i*) ∈ *S1 snd* (ω' *i*) ∈ *S2*
by (*auto simp*: *Sp-def*) **}**
note *this*[*simp*] ω'[*simp*]
{ fix *n*
have *eq*: $((\lambda(\omega 1, \omega 2)\ n.\ (\omega 1\ n, \omega 2\ n)) -` \{\omega \in space\ ?R.\ \forall i{<}n.\ \omega\ i = \omega'\ i\} \cap space\ (K1.paths\ x1 \bigotimes_M K2.paths\ x2)) =$
$\{\omega \in space\ (K1.paths\ x1).\ \forall i{<}n.\ \omega\ i = fst\ (\omega'\ i)\} \times \{\omega \in space\ (K2.paths$

x2). $\forall i<n.\ \omega\ i = snd\ (\omega'\ i)\}$
by (*auto simp*: *space-PiM space-pair-measure PiE-iff eq-commute*)

{ fix *i*
let *?p* = *nat-case* (*x1*, *x2*) ω' *i*
let *?x1* = *case i of 0* ⇒ *x1* | *Suc x* ⇒ *fst* (ω' *x*) **and** *?x2* = *case i of 0* ⇒ *x2* | *Suc x* ⇒ *snd* (ω' *x*)
have *emeasure* (*Kp ?p*) {ω' *i*} = *emeasure* (*K1 ?x1*) {*fst* (ω' *i*)} ∗ *emeasure* (*K2 ?x2*) {*snd* (ω' *i*)}
unfolding *Kp-def* **using** *K2.K.emeasure-pair-measure-Times*[*of* {*fst* (ω' *i*)} *K1 ?x1* {*snd* (ω' *i*)} *?x2*]
by (*auto split*: *prod.split nat.splits*) **}**
note *prod* = *this*

have *emeasure ?L* {$\omega \in$ *space ?L*. $\forall i<n.\ \omega\ i = \omega'\ i\}$ = *emeasure ?R* {$\omega \in$ *space ?R*. $\forall i<n.\ \omega\ i = \omega'\ i\}$
apply (*subst emeasure-distr*[*OF prod-measurable*])
apply *simp*
unfolding *eq*
apply (*subst K2.path.emeasure-pair-measure-Times*)
apply *simp*
apply *simp*
apply (*subst P.emeasure-path*)
apply *simp*
apply *simp*
apply (*subst K1.emeasure-path*)
apply *simp*
apply *simp*
apply (*subst K2.emeasure-path*)
apply *simp*
apply *simp*
apply (*simp add*: *prod setprod.distrib*)
done }
from *this*[*of Suc n*] **show** *emeasure ?L* {$\omega \in$ *space ?L*. $\forall i\leq n.\ \omega\ i = \omega'\ i\}$ = *emeasure ?R* {$\omega \in$ *space ?R*. $\forall i\leq n.\ \omega\ i = \omega'\ i\}$
unfolding *less-Suc-eq-le* **.**
qed

lemma *prod-eq-prob-paths*:
assumes *x*[*simp*]: *x1* ∈ *S1 x2* ∈ *S2* **and** [*measurable*]: *Measurable.pred K1.S-seq P1 Measurable.pred K2.S-seq P2*
shows $\mathcal{P}(\omega$ *in K1.paths x1*. *P1* $\omega)$ ∗ $\mathcal{P}(\omega$ *in K2.paths x2*. *P2* $\omega)$ = $\mathcal{P}(\omega$ *in P.paths* (*x1*, *x2*). *P1* (*fst* ∘ ω) ∧ *P2* (*snd* ∘ ω))
proof −
interpret *PROD*: *pair-prob-space K1.paths x1 K2.paths x2*
by *default*
let *?P1* = {ω∈*space K1.S-seq*. *P1* ω} **and** *?P2* = {ω∈*space K2.S-seq*. *P2* ω}
let *?P* = {$\omega \in$ *space* (*P.paths* (*x1*, *x2*)). *P1* (*fst* ∘ ω) ∧ *P2* (*snd* ∘ ω)}
have *eq*: ((λ(ω*1*, ω*2*) *n*. (ω*1 n*, ω*2 n*)) −‘ *?P* ∩ *space* (*K1.paths x1* $\bigotimes_M$

```
K2.paths x2)) = ?P1 × ?P2
    by (auto simp: space-PiM space-pair-measure PiE-iff o-def)
  show ?thesis
    apply (subst P-paths-eq-prod[OF x])
    apply (subst measure-distr[OF prod-measurable])
   apply (simp add: comp-def sets-S-seq-Sp sets-eq-imp-space-eq[OF sets-S-seq-Sp])
    unfolding eq
    using K2.path.emeasure-pair-measure-Times[of ?P1 K1.paths x1 ?P2 x2]
    apply (simp add: K2.path.emeasure-eq-measure K1.path.emeasure-eq-measure
PROD.emeasure-eq-measure)
    done
qed

lemma p-eq-p1-p2:
  x1 ∈ S1 ⟹ x2 ∈ S2 ⟹ y1 ∈ S1 ⟹ y2 ∈ S2 ⟹ P.p (x1, x2) (y1, y2) n
= K1.p x1 y1 n * K2.p x2 y2 n
  unfolding P.p-def K1.p-def K2.p-def
  apply (subst prod-eq-prob-paths)
  apply (auto intro!: arg-cong2[where f=measure] split: nat.splits)
  done

lemma Sp-iff: (a, b) ∈ Sp ⟷ a ∈ S1 ∧ b ∈ S2
  by (simp add: Sp-def)

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

lemma aperiodicI-pair:
  assumes C1: K1.aperiodic C1 and C2: K2.aperiodic C2
  shows P.aperiodic (C1 × C2)
  unfolding P.aperiodic-eventually-recurrent
proof safe
 from C1[unfolded K1.aperiodic-eventually-recurrent] C2[unfolded K2.aperiodic-eventually-recurrent]
 have C1: C1 ∈ S1 // K1.communicating and C2: C2 ∈ S2 // K2.communicating
and
    ev: ⋀x. x ∈ C1 ⟹ eventually (λm. 0 < K1.p x x m) sequentially ⋀x. x ∈
C2 ⟹ eventually (λm. 0 < K2.p x x m) sequentially
    by auto
  then have subset: C1 ⊆ S1 C2 ⊆ S2
    using K1.C-subset-S[OF C1] K2.C-subset-S[OF C2] by auto
  { fix x1 x2 assume x: x1 ∈ C1 x2 ∈ C2
    with subset have xS: x1 ∈ S1 x2 ∈ S2 by auto
    from ev(1)[OF x(1)] ev(2)[OF x(2)]
```

```
  show eventually (λm. 0 < P.p (x1, x2) (x1, x2) m) sequentially
    by eventually-elim (simp add: p-eq-p1-p2 x xS mult-pos-pos) }

 { fix x1 x2 y1 y2
   assume acc: K1.accessible x1 y1 K2.accessible x2 y2 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.accessibleD-pos K2.accessibleD-pos)
   with acc ev(1)[of y1] ev(2)[of y2]
   have eventually (λm. 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 simp: K1.accessible-in-S2 K2.accessible-in-S2 elim: eventually-elim2
intro: mult-pos-pos)
   then have eventually (λm. 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 x1 y1 y1 l l + m]
      K2.prob-reachable-le[of x2 y2 y2 k k + m]
     show 0 < K1.p x1 y1 (m + l) ∧ 0 < K2.p x2 y2 (m + k)
     by (auto simp add: K1.accessible-in-S1 K2.accessible-in-S1 K1.accessible-in-S2
K2.accessible-in-S2 ac-simps)
   qed
  then have eventually (λm. 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.p (x1, x2) (y1, y2) N
   by (auto simp add: K1.accessible-in-S1 K2.accessible-in-S1 K1.accessible-in-S2
K2.accessible-in-S2 p-eq-p1-p2 mult-pos-pos)
   with acc have P.accessible (x1, x2) (y1, y2)
   by (auto simp add: K1.accessible-in-S1 K2.accessible-in-S1 K1.accessible-in-S2
K2.accessible-in-S2
            intro!: P.accessibleI-pos) }
 note 1 = this

 { fix x1 x2 y1 y2
   assume acc:P.accessible (x1, x2) (y1, y2)
   then have (x1, x2) ∈ Sp (y1, y2) ∈ Sp
    by (auto simp: P.accessible-in-S1 P.accessible-in-S2)
   moreover from acc obtain k where 0 < P.p (x1, x2) (y1, y2) k by (auto
dest!: P.accessibleD-pos)
   ultimately have K1.accessible x1 y1 ∧ K2.accessible x2 y2 using subset
    by (subst (asm) p-eq-p1-p2)
      (auto intro!: K1.accessibleI-pos K2.accessibleI-pos
           simp: Sp-def zero-less-mult-iff K1.p-nonneg K2.p-nonneg 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* ∈ *Sp* // *P.communicating*
apply (*simp add*: *quotient-def Image-def*)
apply (*safe intro*!: *bexI*[*of* - (*x1*, *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*)) ∈ *P.communicating*
using *1*[*of x1 y1 x2 y2*] *1*[*of y1 x1 y2 x2*] *xC yC*
by (*auto simp*: *P.communicating-def K1.communicating-def K2.communicating-def*)
next
fix *y1 y2* **assume** ((*x1*, *x2*), (*y1*, *y2*)) ∈ *P.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*: *P.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*
next
from *subset xC* **show** (*x1*, *x2*) ∈ *Sp* **by** (*auto simp*: *Sp-def*)
qed
qed

lemma *stationary-distributionI-pair*:
assumes *N1*: *K1.stationary-distribution N1 prob-space N1*
assumes *N2*: *K2.stationary-distribution N2 prob-space N2*
shows *P.stationary-distribution* (*N1* ⨂$_M$ *N2*)
unfolding *P.stationary-distribution-def*
proof (*rule pair-measure-eqI*)
interpret *N1*: *prob-space N1* **by** *fact*
show *sigma-finite-measure N1* **by** *default*

interpret *N2*: *prob-space N2* **by** *fact*
show *sigma-finite-measure N2* **by** *default*

interpret *N1N2*: *pair-prob-space N1 N2* **by** *default*

interpret *SpC*: *sigma-finite-measure count-space Sp*
using *P.countable-space* **by** (*rule sigma-finite-measure-count-space*)

have *sets-N1*: *sets N1* = *Pow S1*
by (*subst K1.stationary-distributionD-eq*[*OF N1*(*1*)]) (*simp add*: *sets-point-measure*)
moreover have *sets-N2*: *sets N2* = *Pow S2*
by (*subst K2.stationary-distributionD-eq*[*OF N2*(*1*)]) (*simp add*: *sets-point-measure*)
ultimately have *sets-N1N2*: *sets* (*N1* ⨂$_M$ *N2*) = *Pow Sp*

by (*simp add*: *Sp-def sets-pair-countable*)
then show *sets* (*N1* $\bigotimes_M$ *N2*) = *sets* (*point-measure Sp* (λ*x*. $\int^+$ *s*. *emeasure* (*Kp s*) {*x*} ∂(*N1* $\bigotimes_M$ *N2*)))
by (*simp add*: *sets-point-measure*)

have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable* (*N1* $\bigotimes_M$ *N2*)
unfolding *measurable-def* **using** *sets-eq-imp-space-eq*[*of N1* $\bigotimes_M$ *N2 count-space Sp*]
by (*simp add*: *sets-N1N2*)

have *space-N12*: *space N1* = *S1 space N2* = *S2*
using *sets-eq-imp-space-eq*[*of N1 count-space S1*] *sets-eq-imp-space-eq*[*of N2 count-space S2*]
by (*simp-all add*: *sets-N1 sets-N2*)

have *sets-N1N2C*: *sets* ((*N1* $\bigotimes_M$ *N2*) $\bigotimes_M$ *count-space Sp*) = *Pow* (*Sp* × *Sp*)
by (*simp add*: *sets-pair-countable sets-N1N2*)
then have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable* ((*N1* $\bigotimes_M$ *N2*) $\bigotimes_M$ *count-space Sp*)
unfolding *measurable-def* **using** *sets-eq-imp-space-eq*[*of* (*N1* $\bigotimes_M$ *N2*) $\bigotimes_M$ *count-space Sp count-space* (*Sp* × *Sp*)]
by *simp*

have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable N1*
unfolding *measurable-def* **using** *sets-eq-imp-space-eq*[*of N1 count-space S1*]
by (*simp add*: *sets-N1*)

have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable N2*
unfolding *measurable-def* **using** *sets-eq-imp-space-eq*[*of N2 count-space S2*]
by (*simp add*: *sets-N2*)

fix *A B* **assume** *A-and-B*: *A* ∈ *sets N1 B* ∈ *sets N2*
then have *AB*: *A* × *B* ⊆ *Sp* **and** *A* ⊆ *S1 B* ⊆ *S2*
by (*auto simp add*: *sets-N1 sets-N2 Sp-def*)
have *1*: ⋀*x s*. *indicator* (*A* × *B*) *x* ∗ *indicator* {*s*} *x* = (*indicator* (*if s* ∈ *A* × *B then* {*s*} *else* {}) *x* :: *ereal*)
by (*auto simp*: *indicator-def*)
have ($\int^+$ *x*. ($\int^+$ *y*. ($\int^+$ *ya*. *emeasure* (*Kp* (*x*, *y*)) {*ya*} ∗ *indicator* (*A* × *B*) *ya* ∂*count-space Sp*) ∂*N2*) ∂*N1*) =
($\int^+$ *x*. ($\int^+$ *y*. ($\int^+$ *ya*. *emeasure* (*density* (*Kp* (*x*, *y*)) (*indicator* (*A* × *B*))) {*ya*} ∂*count-space Sp*) ∂*N2*) ∂*N1*)
by (*intro positive-integral-cong*) (*auto simp*: *emeasure-density 1 space-N12*)
also have ... = ($\int^+$ *x*. ($\int^+$ *y*. *emeasure* (*density* (*Kp* (*x*, *y*)) (*indicator* (*A* × *B*))) *Sp* ∂*N2*) ∂*N1*)
apply (*intro positive-integral-cong*)
apply (*subst emeasure-countable-singleton*[*symmetric*])
apply (*simp-all add*: *space-N12*)
done
also have ... = ($\int^+$ *x*. ($\int^+$ *y*. *emeasure* (*Kp* (*x*, *y*)) (*A* × *B*) ∂*N2*) ∂*N1*)

apply (*intro positive-integral-cong*)
apply (*subst emeasure-density*)
apply (*simp-all add*: *space-N12 AB*)
apply (*subst positive-integral-cong*)
apply *simp*
apply (*simp add*: *AB*)
done
also have ... = ($\int^+ x.$ ($\int^+ y.$ *emeasure* (*K1 x*) *A* $*$ *emeasure* (*K2 y*) *B* ∂*N2*) ∂*N1*)
apply (*intro positive-integral-cong*)
apply (*simp-all add*: *space-N12 Kp-def*)
apply (*subst K2.K.emeasure-pair-measure-Times*)
using *A-and-B sets-N1 sets-N2*
apply *simp-all*
done
also have ... = ($\int^+ x.$ *emeasure* (*K1 x*) *A* $*$ ($\int^+ y.$ *emeasure* (*K2 y*) *B* ∂*N2*) ∂*N1*)
apply (*intro positive-integral-cong*)
apply (*subst positive-integral-cmult*)
apply (*simp-all add*: *emeasure-nonneg*)
done
also have ... = ($\int^+ x.$ *emeasure* (*K1 x*) *A* ∂*N1*) $*$ ($\int^+ y.$ *emeasure* (*K2 y*) *B* ∂*N2*)
apply (*subst positive-integral-multc*)
apply (*simp-all add*: *positive-integral-positive*)
done
also have ... = *emeasure N1 A* $*$ *emeasure N2 B*
apply (*subst K1.stationary-distributionD-emeasure*[*OF N1* ‹*A* $\subseteq$ *S1*›])
apply (*subst K2.stationary-distributionD-emeasure*[*OF N2* ‹*B* $\subseteq$ *S2*›]) **..**
finally have *2*: ($\int^+ x.$ ($\int^+ y.$ ($\int^+ ya.$ *emeasure* (*Kp* (*x*, *y*)) {*ya*} $*$ *indicator* (*A* $\times$ *B*) *ya* ∂*count-space Sp*) ∂*N2*) ∂*N1*) =
emeasure N1 A $*$ *emeasure N2 B* **.**

show *emeasure N1 A* $*$ *emeasure N2 B* =
emeasure (*point-measure Sp* ($\lambda x.$ $\int^+ s.$ *emeasure* (*Kp s*) {*x*} ∂(*N1* $\bigotimes_M$ *N2*))) (*A* $\times$ *B*)
apply (*simp add*: *point-measure-def emeasure-density AB*)
apply (*subst positive-integral-multc*[*symmetric*])
apply *simp-all*
apply (*subst pair-sigma-finite.Fubini'*)
apply *unfold-locales*
apply *simp*
apply (*subst N2.positive-integral-fst*[*symmetric*])
apply (*simp-all add*: *positive-integral-positive 2*)
done
qed

end

5.13 Stationary Distribution is the Limit of the Marginal Distributions

context *Discrete-Markov-Kernel*
begin

lemma *stationary-distribution-imp-limit*:
assumes *aperiodic C essential-class C* **and** *N*: *stationary-distribution N prob-space N measure N* (*S* − *C*) = *0*
assumes [*simp*]: *y* ∈ *C*
shows (λ*n*. ∫ *x*. |*p y x n* − *measure N* {*x*}| ∂*count-space C*) −−−−> *0*
(**is** *?L* −−−−> *0*)
proof −
interpret *N*: *prob-space N* **by** *fact*
have *sets-N*[*simp*]: *sets N* = *Pow S*
by (*subst stationary-distributionD-eq*[*OF N*(*1*)]) (*simp add*: *sets-point-measure*)

have *space-N*[*simp*]: *space N* = *S*
using *sets-eq-imp-space-eq*[*of N count-space S*] **by** (*simp-all add*: *sets-N*)

then have [*simp*]: ⋀*f*. *f* ∈ *borel-measurable N*
unfolding *measurable-def* **by** (*simp add*: *sets-N space-N*)

have [*simp*]: ⋀*f M*. *f* ∈ *measurable N M* ⟷ *f* ∈ *measurable* (*count-space S*) *M*
unfolding *measurable-def* **by** (*simp add*: *sets-N space-N*)

def *S'* ≡ *insert None* (*Some* ‘ *S*)
def *K'* ≡ λ*x*. *distr* (*case x of Some x* ⇒ *K x* | *None* ⇒ *N*) (*count-space S'*) *Some*
have [*simp*]: *Some* ∈ *S* → *S'*
by (*simp add*: *S'-def*)
have [*simp*]: (*Some* −‘ *S'* ∩ *S*) = *S*
by (*auto simp*: *S'-def*)

from ‹*essential-class C*› **have** *C*: *C* ∈ *S* // *communicating*
by (*simp add*: *essential-class-def*)
from *C-subset-S*[*OF this*] **have** [*simp*]: ⋀*x*. *x* ∈ *C* ⟹ *x* ∈ *S* **by** *auto*

have [*simp*]: ⋀*x*. *Some x* ∈ *S'* ⟷ *x* ∈ *S* **by** (*auto simp*: *S'-def*)

interpret *K'*: *prob-space K' s* **for** *s*
apply *rule*
apply (*cases s*)
using *N.emeasure-space-1 K.emeasure-space-1*
apply (*simp-all add*: *K'-def emeasure-distr K-measurable1*)
done
interpret *K2*: *Discrete-Markov-Kernel S' K'*
proof qed (*simp-all add*: *S'-def K'-def*)
interpret *KN*: *Pair-Discrete-Markov-Kernel S S' K K'*

by *default*

from *stationary-distributionD*[*OF* ‹*essential-class C*› *N*]
have *pos*: $\bigwedge x.\ x \in C \Longrightarrow$ *pos-recurrent x* **and** N = *stat C* **by** *auto*

have *pos*: $\bigwedge x.\ x \in C \Longrightarrow 0 <$ *emeasure N* $\{x\}$
using *pos C-subset-S*[*OF C*] **unfolding** *stat-def* ‹N = *stat C*›
by (*subst emeasure-point-measure-finite2*)
(*auto simp*: *ereal-0-gt-inverse positive-integral-positive inverse-nonneg-iff pos-recurrent-def*)
then have *rpos*: $\bigwedge x.\ x \in C \Longrightarrow 0 <$ *measure N* $\{x\}$
by (*simp add*: *N.emeasure-eq-measure*)

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

have *intK*: $\bigwedge f\ x.$ $(\int x.\ f\ x\ \partial K'\ (Some\ x)) = (\int x.\ f\ (Some\ x)\ \partial K\ x)$
by (*simp add*: *K′-def integral-distr K-measurable1*)

{ fix *m x y* **assume** $x \in S\ y \in S$
then have *K2.p* (*Some x*) (*Some y*) $m = p\ x\ y\ m$
proof (*induct m arbitrary*: *x*)
case *0* **then show** *?case* **by** (*simp add*: *K2.p-0 p-0*)
next
case (*Suc m*)
then have [*simp*]: $x \in S\ y \in S$ *Some* $x \in S'$ *Some* $y \in S'$ **by** (*auto simp*: *S′-def*)
show *?case*
by (*auto intro*!: *integral-cong simp add*: *K2.p-Suc p-Suc intK Suc.hyps*)
qed }
note *K-p-eq* = *this*

have *in-S′*: $\bigwedge x.\ x \in S' \longleftrightarrow x =$ *None* $\vee$ $(\exists\, y{\in}S.\ x =$ *Some* $y)$
by (*auto simp*: *S′-def*)

have *None-S′*[*simp*]: *None* $\in S'$
by (*auto simp*: *S′-def*)

{ fix *n x* **assume** *x*: $x \in S$
then have *K2.p* (*Some x*) *None* $n = 0$
proof (*induct n arbitrary*: *x*)
case (*Suc n x*) **then show** *?case*
apply (*simp add*: *K2.p-Suc intK cong*: *integral-cong*)
apply (*subst integral-cong*)
apply (*rule Suc*)
apply *simp-all*
done
qed (*simp add*: *K2.p-0*) **}**
note *K-S-None* = *this*

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

have *K2-accessible*: $\bigwedge x\ y.$ *K2.accessible* (*Some x*) $y \longleftrightarrow (\exists z{\in}S.\ y = Some\ z \wedge$ *accessible x z*)
apply (*auto simp*: *K2.accessible-iff accessible-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* ‹*c*∈*C*›])
next
fix *c c' x* **assume** $c \in C$ $c' \in C$
with *irreducibleD*[*OF C this*] **show** (*Some c*, *Some c'*) ∈ *K2.communicating*
by (*auto simp*: *K2.communicating-iff communicating-iff K-p-eq*)
qed

have *Some* ‘ $C \in S'$ // *K2.communicating*
by (*auto simp add*: *quotient-def Image-def c0 K2-communicating*
intro!: *bexI*[*of* - *Some c0*])
then have *K2.essential-class* (*Some* ‘ *C*)
by (*rule K2.essential-classI*)
(*auto simp*: *K2-accessible 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-elim1 simp*: *K-p-eq*)
qed *fact*
then have *aperiodic*: *KN.P.aperiodic* ($C \times$ *Some* ‘ *C*)
by (*rule KN.aperiodicI-pair*[*OF* ‹*aperiodic C*›])

have *KN-essential*: *KN.P.essential-class* ($C \times$ *Some* ‘ *C*)
proof (*rule KN.P.essential-classI*)
show $C \times$ *Some* ‘ $C \in$ *KN.Sp* // *KN.P.communicating*
using *aperiodic* **by** (*simp add*: *KN.P.aperiodic-def*)
next
fix *x y* **assume** $x \in C \times$ *Some* ‘ *C KN.P.accessible x y*
with *KN.P-accessibleD*[*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-accessible essential-classD2[OF ‹essential-class C›])
  qed

  have fm-N: finite-measure N
    by unfold-locales

  { fix n x y assume [simp]: y ∈ S
    have measure N {y} = P(ω in K2.paths None. nat-case None ω (Suc n) =
Some y)
      unfolding stationary-distribution-iterate[OF N(1) fm-N ‹y ∈ S›, of n]
      apply (subst K2.p-def[symmetric])
      apply (subst K2.p-Suc)
      apply simp
      apply (subst K'-def)
      apply simp
      apply (subst integral-distr)
      apply simp
      apply simp
      apply (auto intro!: integral-cong simp: K-p-eq)
      done
    then have measure N {y} = P(ω in K2.paths None. ω n = Some y)
      by simp }
  note measure-y-eq = this

  def D ≡ {x∈S × S'. Some (fst x) = snd x}

  { fix n x assume [simp]: x ∈ S
    have P(ω in KN.P.paths (y, None). (∃ i<n. snd (ω n) = Some x ∧ smallest
ω D i)) =
      (∑ i<n. P(ω in KN.P.paths (y, None). snd (ω n) = Some x ∧ smallest ω
D i))
      apply (subst KN.P.finite-measure-finite-Union[symmetric])
      apply (auto simp add: KN.space-S-seq-Sp KN.sets-S-seq-Sp
                            disjoint-family-on-def
                  intro: smallest-unique
                  intro!: arg-cong2[where f=measure])
      done
    also have ... = (∑ i<n. P(ω in KN.P.paths (y, None). fst (ω n) = x ∧
smallest ω D i))
    proof (intro setsum-cong refl)
      fix i assume i: i ∈ {..< n}
      from i show P(ω in KN.P.paths (y, None). snd (ω n) = Some x ∧ smallest
ω D i) =
        P(ω in KN.P.paths (y, None). fst (ω n) = x ∧ smallest ω D i)
        apply (subst (1 2) KN.P.prob-split-Collect[where i=Suc i])
        apply (simp add: KN.Sp-def)
      apply (simp-all add: KN.space-S-seq-Sp KN.sets-S-seq-Sp smallest-comb-seq
comb-seq-Suc-less)
```

proof (*intro integral-cong*)
fix ω **assume** ω: $\omega \in$ *space* (*KN.P.paths* (y, *None*))
then have $\omega\ i \in S \times S'$
by (*auto simp*: *space-PiM PiE-iff*) (*simp add*: *KN.Sp-def*)
then have [*simp*]: *fst* ($\omega\ i$) $\in S$ *snd* ($\omega\ i$) $\in S'$
by *auto*
show *measure* (*KN.P.paths* ($\omega\ i$)) {$\omega' \in$ *space* (Pi_M *UNIV* (λi. *count-space* $S \bigotimes_M$ *count-space* S')).
snd (ω' (n − *Suc* i)) = *Some* $x \wedge$ *smallest* ω *D* i} =
measure (*KN.P.paths* ($\omega\ i$)) {$\omega' \in$ *space* (Pi_M *UNIV* (λi. *count-space* $S \bigotimes_M$ *count-space* S')).
fst (ω' (n − *Suc* i)) = $x \wedge$ *smallest* ω *D* i}
proof *cases*
assume *smallest* ω *D* i
from *smallestD1*[*OF this*]
have *eq*: *snd* ($\omega\ i$) = *Some* (*fst* ($\omega\ i$)) **by** (*simp add*: *D-def*)

have $\mathcal{P}(\omega'$ *in KN.P.paths* ($\omega\ i$). *fst* (ω' (n − *Suc* i)) = x) =
$\mathcal{P}(\omega'$ *in paths* (*fst* ($\omega\ i$)). ω' (n − *Suc* i) = x) ∗ $\mathcal{P}(\omega'$ *in K2.paths* (*snd* ($\omega\ i$)). *True*)
using ω **by** (*subst KN.prod-eq-prob-paths*) *simp-all*
also have ... = *p* (*fst* ($\omega\ i$)) x (*Suc* (n − *Suc* i))
using *K2.prob-space* **by** (*simp add*: *p-def*)
also have ... = *K2.p* (*snd* ($\omega\ i$)) (*Some* x) (*Suc* (n − *Suc* i))
by (*simp add*: *K-p-eq eq*)
also have ... = $\mathcal{P}(\omega'$ *in paths* (*fst* ($\omega\ i$)). *True*) ∗ $\mathcal{P}(\omega'$ *in K2.paths* (*snd* ($\omega\ i$)). ω' (n − *Suc* i) = *Some* x)
using *prob-space* **by** (*simp add*: *K2.p-def*)
also have ... = $\mathcal{P}(\omega'$ *in KN.P.paths* ($\omega\ i$). *snd* (ω' (n − *Suc* i)) = *Some* x)
using ω **by** (*subst KN.prod-eq-prob-paths*) *simp-all*
finally show *?thesis* **using** ‹*smallest* ω *D* i› **by** (*simp add*: *KN.space-S-seq-Sp KN.sets-S-seq-Sp*)
qed *simp*
qed
qed
also have ... = $\mathcal{P}(\omega$ *in KN.P.paths* (y, *None*). (∃ i<n. *fst* ($\omega\ n$) = $x \wedge$ *smallest* ω *D* i))
apply (*subst KN.P.finite-measure-finite-Union*[*symmetric*])
apply (*auto simp add*: *KN.space-S-seq-Sp KN.sets-S-seq-Sp*
disjoint-family-on-def
intro: *smallest-unique*
intro!: *arg-cong2*[**where** f=*measure*])
done
finally have *eq*: $\mathcal{P}(\omega$ *in KN.P.paths* (y, *None*). (∃ i<n. *snd* ($\omega\ n$) = *Some* $x \wedge$ *smallest* ω *D* i)) =
$\mathcal{P}(\omega$ *in KN.P.paths* (y, *None*). (∃ i<n. *fst* ($\omega\ n$) = $x \wedge$ *smallest* ω *D* i)) .

have *p y x* (*Suc n*) − *measure N* {x} = $\mathcal{P}(\omega$ *in paths y*. $\omega\ n = x$) − $\mathcal{P}(\omega$ *in*

K2.paths None. ω n = Some x)
unfolding *p-def* **by** (*subst measure-y-eq*) *simp-all*
also have $\mathcal{P}(\omega$ *in paths y. ω n = x*$) = \mathcal{P}(\omega$ *in paths y. ω n = x*$) * \mathcal{P}(\omega$ *in K2.paths None. True*)
using *K2.prob-space* **by** *simp*
also have ... = $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *fst* (*ω n*) = *x*)
by (*subst KN.prod-eq-prob-paths*) *auto*
also have ... = $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). ($\exists$ *i*<*n. fst* (*ω n*) = *x* ∧ *smallest ω D i*)) +
$\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *fst* (*ω n*) = *x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*))
apply (*subst KN.P.finite-measure-Union*[*symmetric*])
unfolding *KN.P.space-paths KN.P.sets-paths*
apply (*simp-all add*: *KN.space-S-seq-Sp KN.sets-S-seq-Sp*)
apply (*auto intro*!: *arg-cong2*[**where** *f*=*measure*])
done
also have $\mathcal{P}(\omega$ *in K2.paths None. ω n = Some x*$) = \mathcal{P}(\omega$ *in paths y. True*$) * \mathcal{P}(\omega$ *in K2.paths None. ω n = Some x*)
using *prob-space* **by** *simp*
also have ... = $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *snd* (*ω n*) = *Some x*)
by (*subst KN.prod-eq-prob-paths*) *auto*
also have ... = $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). ($\exists$ *i*<*n. snd* (*ω n*) = *Some x* ∧ *smallest ω D i*)) +
$\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *snd* (*ω n*) = *Some x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*))
apply (*subst KN.P.finite-measure-Union*[*symmetric*])
unfolding *KN.P.space-paths KN.P.sets-paths*
apply (*simp-all add*: *KN.space-S-seq-Sp KN.sets-S-seq-Sp*)
apply (*auto intro*!: *arg-cong2*[**where** *f*=*measure*])
done
finally have | *p y x* (*Suc n*) − *measure N* {*x*} | =
| $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *fst* (*ω n*) = *x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*)) −
$\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *snd* (*ω n*) = *Some x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*)) |
unfolding *eq* **by** (*simp add*: *field-simps*)
also have ... ≤ | $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *fst* (*ω n*) = *x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*)) | +
| $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *snd* (*ω n*) = *Some x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*)) |
by (*rule abs-triangle-ineq4*)
also have ... ≤ $\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *fst* (*ω n*) = *x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*)) +
$\mathcal{P}(\omega$ *in KN.P.paths* (*y*, *None*). *snd* (*ω n*) = *Some x* ∧ ¬ ($\exists$ *i*<*n. smallest ω D i*))
by (*simp add*: *measure-nonneg*)
finally have | *p y x* (*Suc n*) − *measure N* {*x*} | ≤ ... **. }**
note *mono* = *this*

have *Sp-iff*: $\bigwedge x.$ *x* ∈ *KN.Sp* ⟷ *fst x* ∈ *S* ∧ *snd x* ∈ *S'*

by (*auto simp add*: *KN.Sp-def*)

from ‹$C \subseteq S$› *countable-space* **have** [*simp*]: *countable C* **by** (*rule countable-subset*)

{ fix *n* :: *nat*
have ($\int^+x$. | *p y x* (*Suc n*) − *measure N* {*x*} | ∂*count-space C*) ≤
($\int^+x$. *ereal* (*P*(ω *in KN.P.paths* (*y*, *None*). *fst* (ω *n*) = *x* ∧ ¬ (∃ *i*<*n*. *smallest* ω *D i*))) +
ereal (*P*(ω *in KN.P.paths* (*y*, *None*). *snd* (ω *n*) = *Some x* ∧ ¬ (∃ *i*<*n*. *smallest* ω *D i*))) ∂*count-space C*)
using *mono* **by** (*intro positive-integral-mono*) *simp*
also have ... = ($\int^+x$. *P*(ω *in KN.P.paths* (*y*, *None*). *fst* (ω *n*) = *x* ∧ ¬ (∃ *i*<*n*. *smallest* ω *D i*)) ∂*count-space C*) +
($\int^+x$. *P*(ω *in KN.P.paths* (*y*, *None*). *snd* (ω *n*) = *Some x* ∧ ¬ (∃ *i*<*n*. *smallest* ω *D i*)) ∂*count-space C*)
by (*subst positive-integral-add*) (*simp-all add*: *measure-nonneg*)
also have ... = *emeasure* (*KN.P.paths* (*y*, *None*)) (⋃ *x*∈*C*. {ω∈*space* (*KN.P.paths* (*y*, *None*)). *fst* (ω *n*) = *x* ∧ ¬ (∃ *i*<*n*. *smallest* ω *D i*)}) +
emeasure (*KN.P.paths* (*y*, *None*)) (⋃ *x*∈*C*. {ω∈*space* (*KN.P.paths* (*y*, *None*)). *snd* (ω *n*) = *Some x* ∧ ¬ (∃ *i*<*n*. *smallest* ω *D i*)})
by (*subst* (*1 2*) *emeasure-countable-UNION*)
(*auto simp add*: *KN.space-S-seq-Sp KN.sets-S-seq-Sp disjoint-family-on-def KN.P.emeasure-eq-measure*)
also have ... ≤ *ereal* (*P*(ω *in KN.P.paths* (*y*, *None*). ¬ (∃ *i*<*n*. *smallest* ω *D i*))) + *ereal* (*P*(ω *in KN.P.paths* (*y*, *None*). ¬ (∃ *i*<*n*. *smallest* ω *D i*)))
unfolding *KN.P.emeasure-eq-measure*
by (*intro add-mono ereal-less-eq*(*3*)[*THEN iffD2*] *KN.P.finite-measure-mono*)
(*auto simp*: *Sp-iff KN.space-S-seq-Sp KN.sets-S-seq-Sp*)
also have ... ≤ *2* ∗ *P*(ω *in KN.P.paths* (*y*, *None*). ¬ (∃ *i*<*n*. *smallest* ω *D i*))
by *simp*
finally have *?L* (*Suc n*) ≤ *2* ∗ *P*(ω *in KN.P.paths* (*y*, *None*). ¬ (∃ *i*<*n*. *smallest* ω *D i*))
by (*intro integral-bound*) *simp-all* **}**
note *le-2* = *this*

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

let *?N′* = *distr N* (*count-space S′*) *Some*
interpret *N′*: *prob-space ?N′*
by (*rule N.prob-space-distr*) *simp*
interpret *NP*: *pair-prob-space N ?N′*
by *default*

have ∗: *Some* −‘ {*None*} ∩ *S* = {} **by** *auto*
have *pos-recurrent*: ∀ *x*∈*C* × *Some* ‘ *C*. *KN.P.pos-recurrent x*
proof (*rule KN.P.stationary-distributionD*(*1*)[*OF KN-essential KN.stationary-distributionI-pair*[*OF N*(*1*,*2*)]])

show *prob-space ?N'* **by** *unfold-locales*
show *K2.stationary-distribution ?N'*
proof (*rule K2.stationary-distributionI*)
show *finite-measure ?N'* **by** *unfold-locales*
show *sets ?N' = Pow S'* **by** *simp*
fix *y* **assume** *y*: *y ∈ S'*
show (*∫ x. measure* (*K' x*) {*y*} *∂?N'*) *≤ measure ?N'* {*y*}
proof (*cases y*)
case *None* **then show** *?thesis*
apply *simp*
apply (*subst measure-distr*)
apply *simp-all*
apply (*subst integral-distr*)
apply (*simp-all add*: *K'-def*)
apply (*subst measure-distr*)
apply (*simp-all add*: *K-measurable1 ∗*)
done
next
case (*Some x*)
with *y* **have** *∗*: (*Some −‘* {*Some x*} *∩ S*) = {*x*} **by** *auto*
from *Some y* **show** *?thesis*
apply *simp*
apply (*subst measure-distr*)
apply *simp-all*
apply (*subst integral-distr*)
apply (*simp-all add*: *K'-def*)
apply (*subst measure-distr*)
apply (*simp-all add*: *K-measurable1 ∗*)
using *stationary-distributionD-emeasure*[*OF N*(*1*,*2*), *of* {*x*}]
apply (*simp add*: *N.emeasure-eq-measure K.emeasure-eq-measure*)
apply (*subst* (*asm*) *positive-integral-eq-integral*)
apply (*rule N.integrable-const-bound*[**where** *B=1*])
apply (*auto simp*: *measure-nonneg*)
done
qed
qed
show *prob-space* (*N* $\bigotimes_M$ *distr N* (*count-space S'*) *Some*) **by** *unfold-locales*
have *eq*: *KN.Sp − C × Some ‘ C = S ×* (*Some ‘* (*S − C*) *∪* {*None*}) *∪* (*S − C*) *× S'*
unfolding *KN.Sp-def* **by** (*auto simp add*: *S'-def inj-image-mem-iff*)
have *eq2*: *Some −‘ insert None* (*Some ‘* (*S − C*)) *∩ S = S − C*
by *auto*
have *measure* (*N* $\bigotimes_M$ *?N'*) (*KN.Sp − C × Some ‘ C*) *≤*
measure (*N* $\bigotimes_M$ *?N'*) (*S ×* (*Some ‘* (*S − C*) *∪* {*None*})) + *measure* (*N* $\bigotimes_M$ *?N'*) ((*S − C*) *× S'*)
unfolding *eq*
by (*intro NP.finite-measure-subadditive pair-measureI*) (*auto simp add*: *S'-def*)
also have *measure* (*N* $\bigotimes_M$ *?N'*) (*S ×* (*Some ‘* (*S − C*) *∪* {*None*})) = *0*
using *N*(*3*) **unfolding** *measure-def*

```
    apply (subst N'.emeasure-pair-measure-Times)
    apply (auto simp add: emeasure-distr * S'-def)
    apply (subst (asm) emeasure-distr)
    apply (auto simp: eq2 N)
    done
  also have measure (N ⨂M ?N') ((S - C) × S') = 0
    using N(3) unfolding measure-def
    apply (subst N'.emeasure-pair-measure-Times)
    apply (auto simp add: * S'-def)
    done
  finally show measure (N ⨂M ?N') (KN.Sp - C × Some ` C) = 0
    by (simp add: antisym measure-nonneg)
 qed

 from c0-D have measure (KN.P.paths (y, None)) {ω∈space (KN.P.paths (y,
None)). ∀i. ω i ∉ D} ≤
   measure (KN.P.paths (y, None)) {ω∈space (KN.P.paths (y, None)). ∀i. ω i
≠ (c0, Some c0)}
   apply (auto intro!: KN.P.finite-measure-mono)
   apply (erule-tac x=i in allE)
   apply simp
   done
 also have ... = 0
   apply (rule KN.P.prob-eq-0-AE)
   apply (subst KN.P.AE-iterate)
   apply simp
   apply simp
   apply (subst KN.P.AE-K-iff)
   apply simp
 proof
   fix t assume t: t ∈ KN.P.E (y, None)
   obtain a b where t-eq: t = (a, b) by (cases t) auto
    with t have ab: a ∈ S b ∈ S' and eq-0: emeasure (KN.Kp (y, None)) {(a,
b)} ≠ 0
     unfolding KN.P.E-def by (auto simp add: KN.Sp-def)
   have b ≠ None
   proof
     assume b = None
     from ab have emeasure (KN.Kp (y, None)) ({a} × {b}) = emeasure (K y)
{a} * emeasure (K' None) {b}
     unfolding KN.Kp-def by (simp del: insert-times-insert add: K'.emeasure-pair-measure-Times)
     also have emeasure (K' None) {b} = 0
       by (simp add: K'-def emeasure-distr ‹b = None› *)
     finally show False
       using t by (simp add: KN.P.E-def t-eq)
   qed
   then obtain b' where b': b = Some b' b' ∈ S
     using ‹b ∈ S'› by auto
   then have eq2: (Some -` {Some b'}) ∩ S = {b'} by auto
```

from b' *eq-0* **have** *emeasure* $(K\ y \bigotimes_M ?N')\ (\{a\} \times \{Some\ b'\}) \neq 0$
unfolding *KN.Kp-def* **by** (*simp add*: *K'-def*)
then have *neq-0*: *emeasure* $(K\ y)\ \{a\} \neq 0 \wedge$ *emeasure* $N\ \{b'\} \neq 0$
by (*subst* (*asm*) N'*.emeasure-pair-measure-Times*)
(*simp-all add*: *emeasure-distr ab* b' *eq2*)
then have $a \in E\ y$
by (*simp add*: *E-def* ‹$a \in S$›)
with ‹$y \in C$› **have** $a \in C$
using *essential-classD2*[*OF* ‹*essential-class* C› ‹$y \in C$›] **by** (*auto simp*: *accessible-single*)
have $b' \in C$
proof (*rule ccontr*)
assume $b' \notin C$
with ‹$b' \in S$› **have** *measure* $N\ \{b'\} \leq$ *measure* $N\ (S - C)$
by (*intro N.finite-measure-mono*) *auto*
then have *measure* $N\ \{b'\} = 0$
by (*simp add*: *measure-nonneg antisym N*)
with *neq-0* **show** *False*
by (*simp add*: *N.emeasure-eq-measure*)
qed

from *pos-recurrent*[*THEN bspec*, *of* $(c0, Some\ c0)$]
have *recurrent-c0*: *KN.P.recurrent* $(c0, Some\ c0)$
by (*simp add*: *KN.P.pos-recurrent-def c0*)
have $C \times Some\ `\ C \in KN.Sp\ //\ KN.P.communicating$
using *aperiodic* **by** (*simp add*: *KN.P.aperiodic-def*)
then have $((c0, Some\ c0), t) \in KN.P.communicating$
by (*rule KN.P.irreducibleD*) (*simp-all add*: *t-eq ab c0* b' ‹$b' \in C$› ‹$a \in C$›)
then have *KN.P.accessible* $(c0, Some\ c0)\ t$
by (*simp add*: *KN.P.communicating-def*)
then have *KN.P.U* $t\ (c0, Some\ c0) = 1$
by (*rule KN.P.recurrent-accessible*(*1*)[*OF recurrent-c0*])
then have *AE* ω *in KN.P.paths* t. $\exists i.\ \omega\ i = (c0, Some\ c0)$
unfolding *KN.P.U-def* **by** (*subst* (*asm*) *KN.P.prob-Collect-eq-1*) *simp-all*
then show *AE* ω *in KN.P.paths* t. $\neg\ (\forall i.$ *nat-case* $t\ \omega\ i \neq (c0, Some\ c0))$
apply *eventually-elim*
apply *auto*
apply (*rule-tac x=Suc i* **in** *exI*)
apply *simp*
done
qed
finally have *measure* $(KN.P.paths\ (y, None))\ \{\omega \in space\ (KN.P.paths\ (y, None)).\ \forall i.\ \omega\ i \notin D\} = 0$
by (*intro antisym measure-nonneg*)

have $(\lambda n.\ \mathcal{P}(\omega$ *in KN.P.paths* $(y,$ *None*$).\ \neg\ (\exists i<n.$ *smallest* $\omega\ D\ i))) \longrightarrow$
measure $(KN.P.paths\ (y, None))\ (\bigcap n.\ \{\omega \in space\ (KN.P.paths\ (y, None)).\ \neg\ (\exists i<n.\ smallest\ \omega\ D\ i)\})$
by (*rule KN.P.finite-Lim-measure-decseq*) (*auto simp*: *decseq-def*)

```
  also have (⋂ n. {ω∈space (KN.P.paths (y, None)). ¬ (∃ i<n. smallest ω D i)})
=
    {ω∈space (KN.P.paths (y, None)). ∀ i. ω i ∉ D}
    by (auto intro: smallestI dest: smallestD1)
  also have measure (KN.P.paths (y, None)) {ω∈space (KN.P.paths (y, None)).
∀ i. ω i ∉ D} = 0 by fact
  finally have ∗: (λn. 2 ∗ 𝒫(ω in KN.P.paths (y, None). ¬ (∃ i<n. smallest ω D
i))) ----> 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: lebesgue-integral-nonneg)
    done
qed

lemma stationary-distribution-imp-p-limit:
  assumes aperiodic C essential-class C and N: stationary-distribution N prob-space
N measure N (S - C) = 0
  assumes [simp]: x ∈ C y ∈ C
  shows p x y ----> measure N {y}
proof -
  interpret N: prob-space N by fact
  def D ≡ λy n. |p x y n - Sigma-Algebra.measure N {y}|

  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 .

  have C ⊆ S
    using ‹essential-class C›
    by (simp add: C-subset-S essential-class-def)
  then have [simp]: countable C
    using countable-space by (rule countable-subset)

  { fix n
    have D y n ≤ (∫ z. D y n ∗ indicator {y} z ∂count-space C)
      by (subst integral-cmul-indicator) auto
    also have ... ≤ (∫ y. D y n ∂count-space C)
      by (intro integral-mono integral-cmul-indicator)
         (auto split: split-indicator simp: D-def p-def disjoint-family-on-def
           intro!: integrable-abs integral-diff N.integrable-measure 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*=*measure N* {*y*} **and** *net*=*sequentially* **and** *f*=*p x y*]
by (*simp add*: *D-def tendsto-rabs-zero-iff*)
qed

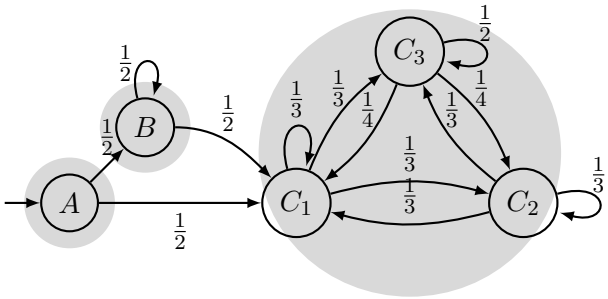
end

end

theory *Example-A*
imports *Classifying-Markov-Chains*
begin

6 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 *default* (*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)
```

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

```
interpretation tau: finite-pmf UNIV tau s for s
  by unfold-locales (simp-all add: tau-def UNIV-state split: state.split)

interpretation A: Discrete-Markov-Kernel UNIV tau.M
  by default simp-all
```

6.1 The essential classs $\{C1, C2, C3\}$

```
lemma A-E-eq:
  A.E x = (case x of A ⇒ {B, C1} | B ⇒ {B, C1} | - ⇒ {C1, C2, C3})
 unfolding A.E-def using state.nchotomy by (auto simp: tau-def split: state.split)

lemma A-essential: A.essential-class {C1, C2, C3}
  by (rule A.essential-classI2) (auto intro!: A.accessible-single 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-eq-one) (simp add: A.period-set-def A.p-Suc A.p-0 eq tau-def)
  finally show A.period {C1, C2, C3} = 1 .
qed
```

6.2 The stationary distribution n

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

```
definition n :: state ⇒ real where
  n = (λC1 ⇒ 0.3 | C2 ⇒ 0.3 | C3 ⇒ 0.4 | - ⇒ 0)

interpretation n: finite-pmf UNIV n
  by unfold-locales (simp-all add: n-def UNIV-state split: state.split)

lemma stationary-distribution-N: A.stationary-distribution n.M
  unfolding A.stationary-distribution-def
proof (rule measure-eqI-finite[where A=UNIV])
```

fix *a*
have *sum-eq*: $\bigwedge x.\ (\sum y{\in}UNIV.\ tau\ y\ x * n\ y) = n\ x$
by (*simp add*: *UNIV-state n-def tau-def split*: *state.split*)
show *emeasure n.M* $\{a\}$ = *emeasure* (*point-measure UNIV* ($\lambda x.\ \int^+ s.$ *emeasure* (*tau.M s*) $\{x\}$ ∂*n.M*)) $\{a\}$
by (*subst emeasure-point-measure-finite*) (*simp-all add*: *positive-integral-positive sum-eq*)
qed (*simp-all add*: *sets-point-measure*)

lemma *exclusive-N*: *measure n.M* (*UNIV* − $\{C1, C2, C3\}$) = *0*
by (*subst n.measure-M*) (*auto simp add*: *n-def UNIV-state setsum-diff*)

lemma *n-is-limit*:
assumes *x*: $x \in \{C1, C2, C3\}$ **and** *y*: $y \in \{C1, C2, C3\}$
shows (*A.p x y*) −−−−> *n y*
proof −
have (*A.p x y*) −−−−> *measure n.M* $\{y\}$
using *A-aperiodic A-essential stationary-distribution-N n.prob-space-M exclusive-N x y*
by (*rule A.stationary-distribution-imp-p-limit*)
then show *?thesis*
by *simp*
qed

lemma *C-is-pos-recurrent*: $x \in \{C1, C2, C3\} \Longrightarrow$ *A.pos-recurrent x*
using *A.stationary-distributionD*(*1*)[*OF A-essential stationary-distribution-N n.prob-space-M exclusive-N*]
by *auto*

lemma *C-recurrence-time*:
assumes *x*: $x \in \{C1, C2, C3\}$
shows ($\int^+\omega.$ *A.t x* ω ∂*A.paths x*) = *1 / n x*
proof −
from *A.stationary-distributionD*(*2*)[*OF A-essential stationary-distribution-N n.prob-space-M exclusive-N*]
have *A.stat* $\{C1, C2, C3\}$ = *n.M* **..**
with *x* **have** *1 / n x* = *1 / emeasure* (*A.stat* $\{C1, C2, C3\}$) $\{x\}$
by (*simp add*: *divide-inverse one-ereal-def*) (*auto simp add*: *n-def*)
also have ... = ($\int^+\omega.$ *A.t x* ω ∂*A.paths x*)
unfolding *A.stat-def* **using** *x*
by (*subst emeasure-point-measure-finite*)
(*simp-all add*: *inverse-nonneg-iff positive-integral-positive ereal-one-divide-invese*)
finally show *?thesis* **..**
qed

end

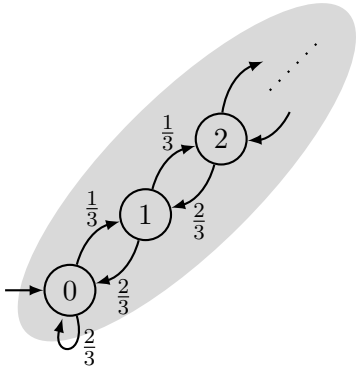
theory *Example-B*

imports *Classifying-Markov-Chains*
begin

lemma *one-divide-ereal*: $x \neq 0 \Longrightarrow 1 \,/\, \mathit{ereal}\ x = \mathit{ereal}\ (1 \,/\, x)$
by (*simp add*: *divide-ereal-def divide-inverse*)

7 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 *tau* :: *nat* ⇒ *nat* ⇒ *real* **where**
tau x y = (*if x* + *1* = *y then 1* / *3*
else if x − *1* = *y then 2* / *3*
else 0)

For the special case when $x = 0$ we have $x - 1 = 0$ and hence *tau 0 0* = *2* / *3*.

We pack this transition function into a discrete Markov kernel, by showing that it forms a probability space.

7.1 Discrete Markov Kernel

definition *K x* = *point-measure UNIV* (*tau x*)

lemma *sets-K*[*simp*]: *sets* (*K x*) = *UNIV* **and** *space-K*[*simp*]: *space* (*K x*) = *UNIV*
by (*simp-all add*: *K-def sets-point-measure space-point-measure*)

lemma *AE-K*[*simp*]: (*AE x in K y*. *P x*) ⟷ *P* (*y* + *1*) ∧ *P* (*y* − *1*)
by (*subst K-def*) (*auto simp add*: *tau-def AE-point-measure*)

lemma *emeasure-K*[*simp*]:
  emeasure (*K x*) *X* = *1*/*3* ∗ *indicator X* (*x* + *1*) + *2*/*3* ∗ *indicator X* (*x* − *1*)
proof −
  have *eq*: {*a* ∈ *X*. *0* < *tau x a*} = (*if x* − *1* ∈ *X* ∧ *x* + *1* ∈ *X then* {*x* − *1*, *x* + *1*}
                                else if x − *1* ∈ *X then* {*x* − *1*}
                                else if x + *1* ∈ *X then* {*x* + *1*}
                                else {})
    by (*auto simp*: *tau-def*)
  show *?thesis*
    unfolding *K-def*
    apply (*subst emeasure-point-measure*)
    apply (*auto simp add*: *eq*)
    apply (*simp-all add*: *tau-def one-divide-ereal*)
    done
qed

interpretation *K*: *prob-space K x* **for** *x*
  by *rule* (*simp add*: *one-divide-ereal one-ereal-def*[*symmetric*])

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

interpretation *B*: *Discrete-Markov-Kernel UNIV K*
  by *default simp-all*

7.2 Enabled, accessible and communicating states

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

lemma *B-E-eq*: *B.E x* = {*x* − *1*, *x* + *1*}
  unfolding *B.E-def* **by** (*auto simp*: *one-divide-ereal*)

lemma *B-E-Suc*: *Suc x* ∈ *B.E x x* ∈ *B.E* (*Suc x*)
  unfolding *B-E-eq* **by** *auto*

lemma *B-accessible*[*intro*]: *B.accessible i j*
proof (*cases i j rule*: *linorder-le-cases*)
  assume *i* ≤ *j* **then show** *?thesis*
    by (*induct rule*: *inc-induct*) (*auto intro*: *B.accessible.intros B.accessible-step-rev B-E-Suc*)
next
  assume *j* ≤ *i* **then show** *?thesis*
    by (*induct rule*: *dec-induct*) (*auto intro*: *B.accessible.intros B.accessible-step-rev B-E-Suc*)
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*

7.3 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-eq-one*) (*simp add*: *B.period-set-def B.p-Suc B.p-0 eq*)
finally show *B.period UNIV* = *1* **.**
qed

7.4 The stationary distribution N

definition N = *point-measure UNIV* (λi. *ereal* (*1 / 2*) ^ *Suc i*)

lemma *borel-measurable-N*[*simp*]: $f \in$ *borel-measurable N*
by (*simp-all add*: *N-def sets-point-measure space-point-measure*)

lemma *sets-N*[*simp*]: *sets N* = *UNIV* **and** *space-N*[*simp*]: *space N* = *UNIV*
by (*simp-all add*: *N-def sets-point-measure space-point-measure*)

lemma *emeasure-N-finite*[*simp*]:
finite A $\Longrightarrow$ *emeasure N A* = $(\sum i \in A.$ *ereal* (*1 / 2*)^*Suc i*)
unfolding *N-def* **by** (*subst emeasure-point-measure-finite2*) *simp-all*

lemma *prob-space-N*: *prob-space N*
proof
have (λi. (*1 / 2*) ^ *i* :: *real*) *sums 2*
using *geometric-sums*[*of 1 / 2* :: *real*] **by** *simp*
from *sums-divide*[*OF this*, *of 2*]
have $*$: (λi. (*1 / 2*) ^ *i / 2* :: *real*) *sums 1*
by *simp*

show *emeasure N* (*space N*) = *1*
by (*subst emeasure-countable-singleton*)
(*auto simp add*: *positive-intergal-count-space-nat one-divide-ereal one-ereal-def*
*
intro!: *sums-suminf-ereal*)
qed

lemma *stationary-distribution-N*: *B.stationary-distribution N*
unfolding *B.stationary-distribution-def*
proof (*rule measure-eqI-countable*[**where** *A=UNIV*])
fix *a*
have *eq*: ⋀*s*. *1 / ereal 3 * indicator {a} (Suc s) + ereal (2 / 3) * indicator {a} (s − Suc 0) =*
*(if a = 0 then ereal (2 / 3) * indicator {0, 1} s else 1 / ereal 3 * indicator {a − 1} s + ereal (2 / 3) * indicator {a + 1} s)*
by (*auto split*: *split-indicator*)
show *emeasure N {a} = emeasure (point-measure UNIV (λx. ∫⁺ s. emeasure (K s) {x} ∂N)) {a}*
apply (*subst emeasure-point-measure-finite2*)
apply (*simp-all add*: *positive-integral-positive eq del*: *ereal-times-divide-eq-left*)
apply (*cases a = 0*)
apply (*auto simp*: *positive-integral-cmult-indicator emeasure-N-finite one-divide-ereal*)
apply (*subst positive-integral-add*)
apply (*simp-all del*: *ereal-times-divide-eq-left*)
apply (*subst (1 2) positive-integral-cmult-indicator*)
apply (*simp-all add*: *one-divide-ereal*)
done
qed (*simp-all add*: *sets-point-measure*)

lemma *exclusive-N*: *measure N (UNIV − UNIV) = 0*
by *simp*

7.5 Limit behavior and recurrence times

lemma *limit*: *(B.p i j) −−−−> (1/2)^Suc j*
proof −
have *(B.p i j) −−−−> measure N {j}*
using *B-aperiodic B-essential stationary-distribution-N prob-space-N exclusive-N UNIV-I UNIV-I*
by (*rule B.stationary-distribution-imp-p-limit*)
then show *?thesis*
by (*simp add*: *measure-def del*: *power-Suc*)
qed

lemma *pos-recurrent*: *B.pos-recurrent i*
using *B.stationary-distributionD(1)*[*OF B-essential stationary-distribution-N prob-space-N exclusive-N*]
by *auto*

lemma *recurrence-time*: *(∫⁺ω. B.t i ω ∂B.paths i) = 2^Suc i*
proof −
from *B.stationary-distributionD(2)*[*OF B-essential stationary-distribution-N prob-space-N exclusive-N*]
have *B.stat UNIV = N* **..**
then have *ereal 2^Suc i = 1 / emeasure (B.stat UNIV) {i}*

by (*simp add*: *one-divide-ereal field-simps power-mult-distrib*[*symmetric*])
also have ... = ($\int^+\omega$. *B.t i* ω *∂B.paths i*)
unfolding *B.stat-def*
by (*subst emeasure-point-measure-finite2*)
(*simp-all add*: *inverse-nonneg-iff positive-integral-positive ereal-one-divide-invese*)
finally show *?thesis*
by (*simp add*: *ereal-power*)
qed

end

References

[1] J. Hölzl. Analyzing discrete-time markov chains with countable state space in Isabelle/HOL. Draft (http://home.in.tum.de/hoelzl/classifying/hoelzl2013markov.pdf).