Markov Decision Processes with Rewards

Maximilian Schäffeler and Mohammad Abdulaziz

March 17, 2025

Abstract

We present a formalization of Markov Decision Processes with rewards. In particular we first build on Hölzl's formalization [1] of MDPs and extend them with rewards. We proceed with an analysis of the expected total discounted reward criterion for infinite horizon MDPs. The central result is the construction of the iteration rule for the Bellman operator. We prove the optimality equations for this operator and show the existence of an optimal stationary deterministic solution. The analysis can be used to obtain dynamic programming algorithms such as value iteration and policy iteration to solve Markov Decision Processes with formal guarantees. Our formalization is based upon chapters 5 and 6 in Puterman's book [2].

Contents

1 Bounded Functions

theory *Bounded-Functions*
imports
HOL.Topological-Spaces
HOL−Analysis.Uniform-Limit
HOL−Probability.Probability
begin

1.1 Definition

definition *bfun* = {*f*. *bounded* (*range f*)}

typedef (**overloaded**) ('*a*, '*b*) *bfun* (‹(- ⇒$_b$ -)› [*22*] *21*) =
bfun::('*a* ⇒ '*b* :: *metric-space*) *set*
morphisms *apply-bfun Bfun*
by (*fastforce simp*: *bounded-def bfun-def*)

declare [[*coercion apply-bfun* :: ('*a* ⇒$_b$ ('*b* :: *metric-space*)) ⇒ '*a* ⇒ '*b*]]

setup-lifting *type-definition-bfun*

lemma *bounded-apply-bfun*[*intro*, *simp*]: *bounded* ((*apply-bfun x*) ‘ *X*)
using *apply-bfun* **by** (*fastforce simp*: *bfun-def bounded-def*)

lemma *apply-bfun-bdd-above*[*simp*, *intro*]:
fixes *f* :: '*c* ⇒$_b$ *real*
shows *bdd-above* (*f* ‘ *X*)
by (*auto intro*: *bounded-imp-bdd-above*)

lemma *bfun-eqI*[*intro*]: (⋀*x*. *apply-bfun f x* = *apply-bfun g x*) ⟹ *f* = *g*
by *transfer auto*

lemma *bfun-eqD*[*dest*]: *f* = *g* ⟹ (⋀*x*. *apply-bfun f x* = *apply-bfun g x*)
by *auto*

lemma *bfunE*:

assumes $f \in$ *bfun*
obtains g **where** $f =$ *apply-bfun* g
by (*blast intro*: *apply-bfun-cases assms*)

lemma *const-bfun*: $(\lambda x.\ b) \in$ *bfun*
by (*auto simp*: *bfun-def image-def*)

lift-definition *const-bfun*::$'b \Rightarrow$ ($'a \Rightarrow_b$ ($'b$ :: *metric-space*)) **is** $\lambda(c{::}'b)$ -. c
by (*rule const-bfun*)

lemma *bounded-dist-le-SUP-dist*:
bounded (*range* f) $\Longrightarrow$ *bounded* (*range* g) $\Longrightarrow$ *dist* ($f\ x$) ($g\ x$) $\leq$ (*SUP* x. *dist* ($f\ x$) ($g\ x$))
by (*auto intro*!: *cSUP-upper bounded-imp-bdd-above bounded-dist-comp*)

instantiation *bfun* :: (*type*, *metric-space*) *metric-space*
begin

lift-definition *dist-bfun* :: ($'a \Rightarrow_b$ $'b$) $\Rightarrow$ ($'a \Rightarrow_b$ $'b$) $\Rightarrow$ *real*
is $\lambda f\ g.$ (*SUP* x. *dist* ($f\ x$) ($g\ x$)) .

definition *uniformity-bfun* :: (($'a \Rightarrow_b$ $'b$) $\times$ $'a \Rightarrow_b$ $'b$) *filter*
where *uniformity-bfun* = (*INF* $e{\in}\{0 <..\}$. *principal* $\{(x,\ y).$ *dist* x $y < e\}$)

definition *open-bfun* :: ($'a \Rightarrow_b$ $'b$) *set* $\Rightarrow$ *bool*
where *open-bfun* S = ($\forall\, x{\in}S.\ \forall_F$ (x', y) *in uniformity*. $x' = x \longrightarrow y \in S$)

lemma *dist-bounded*:
fixes $f\ g$ :: $'a \Rightarrow_b$ $'b$
shows *dist* ($f\ x$) ($g\ x$) $\leq$ *dist* $f\ g$
by *transfer* (*auto intro*!: *bounded-dist-le-SUP-dist simp*: *bfun-def*)

lemma *dist-bound*:
fixes $f\ g$ :: $'a \Rightarrow_b$ ($'b$ :: *metric-space*)
assumes $\bigwedge x.$ *dist* ($f\ x$) ($g\ x$) $\leq b$
shows *dist* $f\ g \leq b$
using *assms*
by *transfer* (*auto intro*!: *cSUP-least*)

lemma *dist-fun-lt-imp-dist-val-lt*:
fixes $f\ g$ :: $'a \Rightarrow_b$ $'b$
assumes *dist* $f\ g < e$
shows *dist* ($f\ x$) ($g\ x$) $< e$
using *dist-bounded assms*
by (*rule le-less-trans*)

instance
proof
  fix *f g h* :: *'a* $\Rightarrow_b$ *'b*
  show *dist f g = 0* $\longleftrightarrow$ *f = g*
  proof
    have $\bigwedge x$. *dist* (*f x*) (*g x*) $\leq$ *dist f g*
      by (*rule dist-bounded*)
    also assume *dist f g = 0*
    finally show *f = g*
      by (*auto simp*: *apply-bfun-inject*[*symmetric*])
  qed (*auto simp*: *dist-bfun-def intro*!: *cSup-eq*)
  show *dist f g* $\leq$ *dist f h + dist g h*
  proof (*rule dist-bound*)
    fix *x*
    have *dist* (*f x*) (*g x*) $\leq$ *dist* (*f x*) (*h x*) + *dist* (*g x*) (*h x*)
      by (*rule dist-triangle2*)
    also have *dist* (*f x*) (*h x*) $\leq$ *dist f h*
      by (*rule dist-bounded*)
    also have *dist* (*g x*) (*h x*) $\leq$ *dist g h*
      by (*rule dist-bounded*)
    finally show *dist* (*f x*) (*g x*) $\leq$ *dist f h + dist g h*
      by *simp*
  qed
qed (*rule open-bfun-def uniformity-bfun-def*)+

end

lift-definition *PiC*::*'a set* $\Rightarrow$ (*'a* $\Rightarrow$ (*'b* :: *metric-space*) *set*) $\Rightarrow$ (*'a* $\Rightarrow_b$ *'b*) *set*
  is λ*I X*. *Pi I X* $\cap$ *bfun*
  by *auto*

lemma *mem-PiC-iff*: *x* $\in$ *PiC I X* $\longleftrightarrow$ *apply-bfun x* $\in$ *Pi I X*
  by *transfer simp*

lemmas *mem-PiCD* = *mem-PiC-iff*[*THEN iffD1*]
  and *mem-PiCI* = *mem-PiC-iff*[*THEN iffD2*]

lemma *tendsto-bfun-uniform-limit*:
  fixes *f*::*'i* $\Rightarrow$ *'a* $\Rightarrow_b$ (*'b* :: *metric-space*)
  assumes (*f* $\longrightarrow$ *l*) *F*
  shows *uniform-limit UNIV f l F*
proof (*rule uniform-limitI*)
  fix *e*::*real* **assume** *e > 0*
  from *tendstoD*[*OF assms this*] **have** $\forall_F$ *x in F*. *dist* (*f x*) *l* < *e* .
  then show $\forall_F$ *n in F*. $\forall x \in$ *UNIV*. *dist* ((*f n*) *x*) (*l x*) < *e*
    by *eventually-elim* (*auto simp*: *dist-fun-lt-imp-dist-val-lt*)
qed

lemma *uniform-limit-tendsto-bfun*:
fixes $f::'i \Rightarrow 'a \Rightarrow_b ('b :: metric\text{-}space)$
and $l::'a \Rightarrow_b 'b$
assumes *uniform-limit UNIV f l F*
shows $(f \longrightarrow l)\ F$
proof (*rule tendstoI*)
fix $e::real$ **assume** $e > 0$
then have $e\ /\ 2 > 0$ **by** *simp*
from *uniform-limitD*[*OF assms this*]
have $\forall_F\ i\ in\ F.\ \forall x.\ dist\ (f\ i\ x)\ (l\ x) < e\ /\ 2$ **by** *simp*
then have $\forall_F\ x\ in\ F.\ dist\ (f\ x)\ l \leq e\ /\ 2$
by *eventually-elim* (*blast intro*: *dist-bound less-imp-le*)
then show $\forall_F\ x\ in\ F.\ dist\ (f\ x)\ l < e$
by *eventually-elim* (*use* ‹$0 < e$› **in** *auto*)
qed

1.2 Supremum Norm

instantiation *bfun* :: (*type*, *real-normed-vector*) *real-vector*
begin

lemma *uminus-cont*: $f \in bfun \Longrightarrow (\lambda x.\ -\ f\ x) \in bfun$ **for** $f::'a \Rightarrow 'b$
by (*auto simp*: *bfun-def*)

lemma *plus-cont*: $f \in bfun \Longrightarrow g \in bfun \Longrightarrow (\lambda x.\ f\ x + g\ x) \in bfun$
for $f\ g::'a \Rightarrow 'b$
by (*auto simp*: *bfun-def bounded-plus-comp*)

lemma *minus-cont*: $f \in bfun \Longrightarrow g \in bfun \Longrightarrow (\lambda x.\ f\ x - g\ x) \in bfun$
for $f\ g::'a \Rightarrow 'b$
by (*auto simp*: *bfun-def bounded-minus-comp*)

lemma *scaleR-cont*: $f \in bfun \Longrightarrow (\lambda x.\ a *_R f\ x) \in bfun$ **for** $f :: 'a \Rightarrow 'b$
by (*auto simp*: *bfun-def bounded-scaleR-comp*)

lemma *bfun-normI*[*intro*]: $(\bigwedge x.\ norm\ (f\ x) \leq b) \Longrightarrow f \in bfun$
by (*auto simp*: *bfun-def intro*: *boundedI*)

lift-definition *uminus-bfun*::$('a \Rightarrow_b 'b) \Rightarrow ('a \Rightarrow_b 'b)$ **is** $\lambda f\ x.\ -\ f\ x$
by (*rule uminus-cont*)

lift-definition *plus-bfun*::$('a \Rightarrow_b 'b) \Rightarrow ('a \Rightarrow_b 'b) \Rightarrow 'a \Rightarrow_b 'b$ **is** $\lambda f\ g\ x.\ f\ x + g\ x$
by (*rule plus-cont*)

lift-definition *minus-bfun*::$('a \Rightarrow_b 'b) \Rightarrow ('a \Rightarrow_b 'b) \Rightarrow 'a \Rightarrow_b 'b$ **is** $\lambda f\ g\ x.\ f\ x - g\ x$

by (*rule minus-cont*)

lift-definition *zero-bfun*::$'a \Rightarrow_b 'b$ **is** λ-. *0*
by (*rule const-bfun*)

lemma *const-bfun-0-eq-0*[*simp*]: *const-bfun 0 = 0*
by *transfer simp*

lift-definition *scaleR-bfun*::*real* $\Rightarrow$ $('a \Rightarrow_b 'b) \Rightarrow 'a \Rightarrow_b 'b$ **is** $\lambda r\ g\ x.$ $r *_R g\ x$
by (*rule scaleR-cont*)

lemmas [*simp*] =
const-bfun.rep-eq
uminus-bfun.rep-eq
plus-bfun.rep-eq
minus-bfun.rep-eq
zero-bfun.rep-eq
scaleR-bfun.rep-eq

instance
by *standard* (*auto simp*: *algebra-simps*)
end

lemma *scaleR-cont'*: $f \in$ *bfun* $\Longrightarrow$ $(\lambda x.\ a * f\ x) \in$ *bfun* **for** $f :: 'a \Rightarrow$ *real*
using *scaleR-cont*[*of f a*] **by** *auto*

lemma *bfun-norm-le-SUP-norm*:
$f \in$ *bfun* $\Longrightarrow$ *norm* $(f\ x) \leq$ $(SUP\ x.$ *norm* $(f\ x))$
by (*auto intro*!: *cSUP-upper bounded-imp-bdd-above simp*: *bfun-def bounded-norm-comp*)

instantiation *bfun* :: (*type*, *real-normed-vector*) *real-normed-vector*
begin

definition *norm-bfun* :: $('a, 'b)$ *bfun* $\Rightarrow$ *real*
where *norm-bfun f = dist f 0*

definition *sgn* $(f::('a,'b)$ *bfun*$) =$ $f /_R$ *norm f*

instance
proof
fix *a* :: *real*
fix *f g* :: $('a, 'b)$ *bfun*
show *dist f g = norm* $(f - g)$
unfolding *norm-bfun-def*
by *transfer* (*simp add*: *dist-norm*)

show *norm* $(f + g) \leq$ *norm* $f +$ *norm* g
unfolding *norm-bfun-def*
by *transfer*
(*auto intro*!: *cSUP-least norm-triangle-le add-mono bfun-norm-le-SUP-norm simp*: *dist-norm*)
show *norm* $(a *_R f) = |a| *$ *norm* f
unfolding *norm-bfun-def dist-bfun.rep-eq*
by (*subst continuous-at-Sup-mono*[*of* $\lambda x.\ |a| * x$])
(*fastforce intro*!: *monoI mult-left-mono continuous-intros bounded-imp-bdd-above*

simp: *bounded-norm-comp image-comp*)+
qed (*auto simp*: *norm-bfun-def sgn-bfun-def*)
end

lemma *norm-bfun-def'*: *norm* $f = (\bigsqcup x.$ *norm* $((f :: 'a \Rightarrow_b 'b ::$ *real-normed-vector*$)\ x))$
by(*subst norm-conv-dist*, *simp add*: *dist-bfun.rep-eq*)

lemma *norm-le-norm-bfun*: *norm* (*apply-bfun f x*) $\leq$ *norm f*
by (*simp add*: *apply-bfun bfun-norm-le-SUP-norm norm-bfun-def dist-bfun-def*)

lemma *abs-le-norm-bfun*: *abs* (*apply-bfun f x*) $\leq$ *norm f*
by (*subst real-norm-def*[*symmetric*]) (*rule norm-le-norm-bfun*)

lemma *le-norm-bfun*: *apply-bfun f x* $\leq$ *norm f*
using *abs-ge-self abs-le-norm-bfun*
by (*rule order.trans*)

1.3 Complete Space

lemma *tendsto-add*: $P \longrightarrow (L :: 'a ::$ *real-normed-vector*$) \Longrightarrow (\lambda n.$ $P\ n + c) \longrightarrow L + c$
by (*intro tendsto-intros*)

lemma *lim-add*: *convergent* $P \Longrightarrow$ *lim* $(\lambda n.\ P\ n + (c :: 'a ::$*real-normed-vector*$))$ $=$ *lim* $P + c$
by (*auto intro*: *limI dest*: *Bounded-Functions.tendsto-add simp add*: *convergent-LIMSEQ-iff*)

lemma *complete-bfun*:
assumes *cauchy-f*: *Cauchy* $(f ::$ *nat* $\Rightarrow ('a,\ 'b ::$ {*complete-space*, *real-normed-vector*}$)$ *bfun*$)$
shows *convergent f*
proof −
let *?f* $= \lambda x.$ *lim* $(\lambda n.\ f\ n\ x)$

from *cauchy-f* **have** *cauchy-fx*: *Cauchy* $(\lambda n.\ f\ n\ x)$ **for** x
by(*fastforce intro*: *dist-fun-lt-imp-dist-val-lt CauchyI' dest*: *met-*

ric-CauchyD)+

hence *conv-fx*: *convergent* (λ*n*. *f n x*) **for** *x*
by(*auto intro*: *Cauchy-convergent*)

have *lim-f-bfun*: *?f* ∈ *bfun*
proof −
have ∃ *b*. ∀ *x*. *norm* (*lim* (λ*n*. *f n x*)) ≤ *b*
proof −
obtain *N b* **where** *dist-N*: *dist* (*f n x*) (*f m x*) < *b* **if** *n* ≥ *N m* ≥ *N* **for** *x m n*
using *metric-CauchyD*[*OF cauchy-f zero-less-numeral*] *dist-fun-lt-imp-dist-val-lt*
by *metis*
have *aux*: *norm* (*lim* (λ*n*. *f n x*)) ≤ *b* + *norm* (*f N x*) **for** *x*
proof−
from *conv-fx*[*unfolded convergent-LIMSEQ-iff*]
have *tendsto-f-N*: (λ*n*. *f* (*n* + *N*) *x*) ⟶ *?f x*
by (*auto dest*: *LIMSEQ-ignore-initial-segment*)
hence *tendsto-f-dist*: (λ*n*. *dist* (*f* (*n* + *N*) *x*) (*f N x*)) ⟶ *dist* (*?f x*) (*f N x*)
by (*auto intro*: *tendsto-intros*)
have *dist* (*f* (*n* + *N*) *x*) (*f N x*) ≤ *b* **for** *n*
by (*auto intro*!: *less-imp-le simp*: *dist-N*)
hence *dist* (*?f x*) (*f N x*) ≤ *b*
using *lim-le*[*OF convergentI*[*OF tendsto-f-dist*]]
by (*auto simp*: *limI*[*OF tendsto-f-dist*, *symmetric*])
thus *norm* (*?f x*) ≤ *b* + *norm* (*f N x*)
using *norm-triangle-ineq2 order-trans*
by (*fastforce simp*: *dist-norm*)
qed
show *?thesis*
by (*auto intro*!: *exI*[*of* - *b* + *norm* (*f N*)] *order.trans*[*OF aux*] *norm-le-norm-bfun*)
qed
thus *?thesis*
by (*auto intro*: *boundedI simp*: *bfun-def*)
qed

hence *bfun-lim-f-inv*: *apply-bfun* (*Bfun ?f*) = *?f*
using *bfun.Bfun-inverse* **by** *blast*

have *f* ⟶ *Bfun ?f*
proof −
have ⋀*e*. *e* > *0* ⟹ ∃ *N*. ∀ *n* ≥ *N*. *dist* (*Bfun ?f*) (*f n*) < *e*
proof −
fix *e* :: *real*
assume *e* > *0*
hence ∃ *N*. ∀ *n* ≥ *N*. ∀ *m* ≥ *N*. *dist* (*f n*) (*f m*) < *0.5* ∗ *e* (**is** ∃ *N*. ∀ *n* ≥ *N*. ∀ *m* ≥ *N*. *?P n m N e*)

```
      by(force intro!: metric-CauchyD[OF cauchy-f])
    then obtain N where dist-N: ?P n m N e if n ≥ N m ≥ N for
n m
      by auto
    have ∀ n x. dist (?f x) (f (n + N) x) ≤ 0.5 * e
    proof safe
      fix n x
      have (λm. f m x) ⟶ ?f x
        using conv-fx convergent-LIMSEQ-iff
        by blast
      hence tendsto-f-N: (λm. f (m + N) x) ⟶ ?f x
        using LIMSEQ-ignore-initial-segment
        by auto
      hence tendsto-f-dist:
        (λm. dist (f (m + N) x) (f (n + N) x)) ⟶ dist (?f x) (f
(n + N) x)
        by (simp add: tendsto-dist)
      have dist (f (m + N) x) (f (n + N) x) < 0.5 * e for m
        by (fastforce intro!: dist-fun-lt-imp-dist-val-lt[OF dist-N])
      thus dist (?f x) (f (n + N) x) ≤ 0.5 * e
        by (fastforce intro: less-imp-le convergentI[OF tendsto-f-dist]
intro!: lim-le
            simp only: limI[OF tendsto-f-dist, symmetric])
    qed
    hence ∀ n. (SUP x. dist (?f x) (f (n + N) x)) ≤ 0.5 * e
      by (fastforce intro!: cSUP-least)
    hence aux: ∀ n. dist (Bfun ?f) (f (n + N)) ≤ 0.5 * e
      unfolding dist-bfun-def
      by (simp add: bfun-lim-f-inv)
    have 0.5 * e < e  by (simp add: ‹0 < e›)
    hence ∀ n. dist (Bfun ?f) (f (n + N)) < e
      using aux le-less-trans by blast
    thus ∃ N. ∀ n≥N. dist (Bfun ?f) (f n) < e
      by (metis add.commute less-eqE)
  qed
  thus ?thesis
    by (simp add: dist-commute metric-LIMSEQ-I)
qed
thus convergent f
  unfolding convergent-def
  by blast
qed

lemma norm-bound:
  fixes f :: ('a, 'b::real-normed-vector) bfun
  assumes ⋀x. norm (apply-bfun f x) ≤ b
  shows norm f ≤ b
  using dist-bound[of f 0 b] assms
  by (simp add: dist-norm)
```

lemma *bfun-bounded-norm-range*: *bounded* (*range* (λ*s*. *norm* (*apply-bfun v s*)))
proof −
  obtain *b* **where** ∀ *s*. *norm* (*v s*) ≤ *b*
    using *norm-le-norm-bfun*
    by *fast*
  thus *?thesis*
    by (*simp add*: *bounded-norm-comp*)
qed

instance *bfun* :: (*type*, *banach*) *banach*
  by *standard* (*auto simp*: *complete-bfun*)

lemma *bfun-prob-space-integrable*:
  assumes *prob-space S v* ∈ *borel-measurable S*
  assumes (*v* :: *'a* ⇒ *'b* :: {*second-countable-topology*, *banach*}) ∈ *bfun*

  shows *integrable S v*
  using *prob-space.finite-measure norm-le-norm-bfun*[*of Bfun v*] *Bfun-inverse*[*OF assms*(*3*)] *assms*
  by (*auto intro*: *finite-measure.integrable-const-bound*)

lemma *bfun-integral-bound*:
  assumes (*v* :: *'a* ⇒ *'c* :: {*euclidean-space*}) ∈ *bfun*
  shows (λ*S*. ∫ *x*. *v x* ∂(*S* :: *'a pmf*)) ∈ *bfun*
proof −
  obtain *b* **where** *bH*: ∀ *x*. *norm* (*v x*) ≤ *b*
    using *bfun-norm-le-SUP-norm assms* **by** *fast*
  have (∫ *x*. *norm* (*v x*) ∂*S*) ≤ *b* **for** *S* :: *'a pmf*
   using ‹*v* ∈ *bfun*› *bfun-def bounded-norm-comp bH bfun-prob-space-integrable*
   by (*fastforce intro*!: *prob-space.integral-le-const prob-space-measure-pmf simp*: *bfun-def*)
  hence ∀ *S* :: *'a pmf*. *norm* (∫ *x*. (*v x*) ∂*S*) ≤ *b*
    using *integral-norm-bound order-trans* **by** *blast*
  thus *?thesis*
    unfolding *bfun-def*
    by (*auto intro*: *boundedI*)
qed

lemma *scale-bfun*[*intro*!]: *f* ∈ *bfun* ⟹ (λ*x*. (*k*::*real*) ∗ *f x*) ∈ *bfun*
  using *scaleR-cont*[*of f k*] **by** *auto*

lemma *bfun-spec*[*intro*]: *f* ∈ *bfun* ⟹ (λ*x*. *f* (*g x*)) ∈ *bfun*
  unfolding *bfun-def bounded-def* **by** *auto*

lemma *apply-bfun-bfun*[*simp*]: *apply-bfun f* ∈ *bfun*
  using *apply-bfun* **.**

lemma *bfun-integral-bound'*[*intro*]: ($v :: 'a \Rightarrow 'c :: \{euclidean\text{-}space\}$) $\in$ *bfun* $\Longrightarrow$
  ($\lambda S.\ \int x.\ v\ x\ \partial((F\ S) :: 'a\ pmf)) \in$ *bfun*
  using *bfun-integral-bound*
  by (*subst bfun-spec*[*of - F*]) *auto*

lift-definition *bfun-comp* :: ($'a \Rightarrow 'b$) $\Rightarrow$ ($'b \Rightarrow_b 'c$::*metric-space*) $\Rightarrow$ ($'a \Rightarrow_b 'c$) **is**
  $\lambda g\ bf\ x.\ bf\ (g\ x)$
  by *auto*

1.4 Order Instance

class *ordered-real-normed-vector* = *real-normed-vector* + *ordered-real-vector*

instance *real* :: *ordered-real-normed-vector*
  by *standard*

instantiation *bfun* :: (-, *ordered-real-normed-vector*) *ordered-real-normed-vector*
begin

definition *less-eq-bfun f g* $\equiv \forall x.$ *apply-bfun f x* $\leq$ *apply-bfun g x*
definition *less-bfun f g* $\equiv \forall x.$ *apply-bfun f x* $\leq$ *apply-bfun g x* $\wedge$ ($\exists y.$ *f y* $<$ *g y*)

instance
proof (*standard*, *goal-cases*)
  case (*1 x y*)
  then show *?case*
    by (*auto dest*: *leD simp add*: *less-bfun-def less-eq-bfun-def*)
      (*metis order.not-eq-order-implies-strict*)
qed (*auto intro*: *order-trans antisym dest*: *leD not-le-imp-less*
   simp: *less-eq-bfun-def less-bfun-def eq-iff scaleR-left-mono scaleR-right-mono*)
end

lemma *less-eq-bfunI*[*intro*]: ($\bigwedge x.$ *apply-bfun f x* $\leq$ *apply-bfun g x*) $\Longrightarrow$ *f* $\leq$ *g*
  unfolding *less-eq-bfun-def*
  by *auto*

lemma *less-eq-bfunD*[*dest*]: *f* $\leq$ *g* $\Longrightarrow$ ($\bigwedge x.$ *apply-bfun f x* $\leq$ *apply-bfun g x*)
  unfolding *less-eq-bfun-def*
  by *auto*

1.5 Miscellaneous

instantiation *bfun* :: (*type*, *one*) *one* **begin**

lift-definition *one-bfun* :: $'s \Rightarrow_b 'd$::{*metric-space*, *one*} **is** $\lambda x.\ 1$

using *const-bfun* **.**

instance
by *standard*
end

declare *one-bfun.rep-eq* [*simp*]

lemma *apply-bfun-one* [*simp*]: *apply-bfun* (*1* :: - $\Rightarrow_b$ *real*) *x* = *1*
using *one-bfun.rep-eq*
by *auto*

lemma *norm-bfun-one*[*simp*]: *norm* (*1* :: *'a* $\Rightarrow_b$ *real*) = *1*
unfolding *norm-bfun-def'* **by** *auto*

lemma *range-bfunI*[*intro*]: *bounded* (*range f*) $\Longrightarrow$ *f* $\in$ *bfun*
by (*simp add*: *bfun-def*)

lemma *finite-bfun*[*simp*]: (λ(*i*::-::*finite*). *f i*) $\in$ *bfun*
by (*meson finite finite-imageI finite-imp-bounded range-bfunI*)

lemma *bounded-apply-bfun'*:
assumes *bounded* ((*F* :: *'c* $\Rightarrow$ *'d* $\Rightarrow_b$ *'b*::*real-normed-vector*) ' *S*)
shows *bounded* ((λ*b*. (*F b*) *x*) ' *S*)
proof −
obtain *b* **where** $\forall$ *x*$\in$*S*. *norm* (*F x*) $\leq$ *b*
by (*meson assms bounded-pos image-eqI*)
thus *bounded* ((λ*b*. (*F b*) *x*) ' *S*)
by (*fastforce intro*: *norm-le-norm-bfun dual-order.trans boundedI*[*of* - *b*])
qed

lemma *bfun-tendsto-apply-bfun*:
assumes *h*: (*F* :: (*nat* $\Rightarrow$ *'a* $\Rightarrow_b$ *real*)) $\longrightarrow$ (*y* :: *'a* $\Rightarrow_b$ *real*)
shows (λ*n*. *F n x*) $\longrightarrow$ *y x*
proof −
have *aux*: (λ*n*. *dist* (*F n*) *y*) $\longrightarrow$ *0*
using *h*
using *tendsto-dist-iff* **by** *blast*
have $\bigwedge$*n*. *dist* (*F n x*) (*y x*) $\leq$ *dist* (*F n*) *y*
unfolding *dist-bfun-def*
using *Bounded-Continuous-Function.bounded-dist-le-SUP-dist* **by** *fastforce*
hence $\bigwedge$*n*. *norm* (*dist* (*F n x*) (*y x*)) $\leq$ *norm*(*dist* (*F n*) *y*)
by *auto*
hence (λ*n*. *dist* (*F n x*) (*y x*)) $\longrightarrow$ *0*
by (*subst Lim-transform-bound*[*OF* - *aux*]) *auto*

thus *?thesis*
using *tendsto-dist-iff* **by** *blast*
qed

1.6 Bounded Functions and Vectors

lemma *vec-bfun*[*simp*, *intro*]: (\$) $x \in$ *bfun*
using *finite-bfun*.

lemma *norm-bfun-le-norm-vec*: *norm* (*bfun.Bfun* ((\$) (*x* :: *real*^'*c* :: *finite*))) $\leq$ *norm x*
proof −
have *norm* (*bfun.Bfun* ((\$) (*x* :: *real*^'*c* :: *finite*))) $\leq$ ($\bigsqcup$ *xa*. |*x* \$ *xa*|)
unfolding *norm-bfun-def dist-bfun-def*
by (*auto simp*: *Bfun-inverse*)
also have ... $\leq$ *norm x*
using *component-le-norm-cart*
by (*auto intro*: *cSUP-least*)
finally show *?thesis*
by *auto*
qed

lemma *bounded-linear-bfun-nth*: *bounded-linear f* $\Longrightarrow$ *bounded-linear* (λ*v*. *bfun.Bfun* ((\$) (*f v*)))
using *order-trans*[*OF Finite-Cartesian-Product.norm-nth-le onorm*, *of f*]
by (*auto simp*: *Bfun-inverse mult.commute linear-simps dist-bfun-def norm-bfun-def*
intro!: *bounded-linear-intro cSup-least*)

lemma *norm-vec-le-norm-bfun*:
norm (*vec-lambda* (*apply-bfun* (*x* :: 'd::*finite* $\Rightarrow_b$ *real*))) $\leq$ *norm x* $*$ *card* (*UNIV* :: 'd *set*)
proof −
have *norm* (*vec-lambda* (*apply-bfun x*)) $\leq$ ($\sum$ *i* $\in$ *UNIV* . |(*apply-bfun x i*)|)
using *L2-set-le-sum-abs*
unfolding *norm-vec-def L2-set-def*
by *auto*
also have ... $\leq$ (*card* (*UNIV* :: 'd *set*) $*$ ($\bigsqcup$ *xa*. |*apply-bfun x xa*|))
by (*auto intro*!: *sum-bounded-above cSup-upper*)
finally show *?thesis*
by (*simp add*: *norm-bfun-def dist-bfun-def mult.commute*)
qed

end

2 Bounded Linear Functions

theory *Blinfun-Util*
imports
HOL−Analysis.Bounded-Linear-Function
Bounded-Functions
begin

2.1 Composition

lemma *blinfun-compose-id*[*simp*]:
id-blinfun o_L *f* = *f*
f o_L *id-blinfun* = *f*
by (*auto intro*!: *blinfun-eqI*)

lemma *blinfun-compose-assoc*: *F* o_L *G* o_L *H* = *F* o_L (*G* o_L *H*)
using *blinfun-apply-inject* **by** *fastforce*

lemma *blinfun-compose-diff-right*: *f* o_L (*g* − *h*) = (*f* o_L *g*) − (*f* o_L *h*)
by (*auto intro*!: *blinfun-eqI simp*: *blinfun.bilinear-simps*)

2.2 Power

overloading
blinfunpow ≡ *compow* :: *nat* ⇒ ('*a*::*real-normed-vector* ⇒$_L$ '*a*) ⇒ ('*a* ⇒$_L$ '*a*)
begin

primrec *blinfunpow* :: *nat* ⇒ ('*a*::*real-normed-vector* ⇒$_L$ '*a*) ⇒ ('*a* ⇒$_L$ '*a*)
where
blinfunpow 0 f = *id-blinfun*
| *blinfunpow* (*Suc n*) *f* = *f* o_L *blinfunpow n f*

end

lemma *bounded-pow-blinfun*[*intro*]:
assumes *bounded* (*range* (*F*::*nat* ⇒ '*a*::*real-normed-vector* ⇒$_L$ '*a*))
shows *bounded* (*range* (λ*t*. (*F t*)^^(*Suc n*)))
using *assms* **proof** −
assume *bounded* (*range F*)
then obtain *b* **where** *bh*: ⋀*x*. *norm* (*F x*) ≤ *b*
by (*auto simp*: *bounded-iff*)
hence *norm* ((*F x*)^^(*Suc n*)) ≤ *b*^(*Suc n*) **for** *x*
using *bh*
by (*induction n*) (*auto intro*!: *order.trans*[*OF norm-blinfun-compose*] *simp*: *mult-mono*′)
thus *?thesis*
by (*auto intro*!: *boundedI*)
qed

lemma *blincomp-scaleR-right*: $(a *_R (F :: 'a :: real\text{-}normed\text{-}vector \Rightarrow_L 'a)) \widehat{\ } t = a \widehat{\ } t *_R F \widehat{\ } t$
by (*induction t*) (*auto intro*: *blinfun-eqI simp*: *blinfun.scaleR-left blinfun.scaleR-right*)
lemma *summable-inv-Q*:
fixes $Q :: 'a :: banach \Rightarrow_L 'a$
assumes *onorm-le*: *norm* (*id-blinfun* − *Q*) < *1*
shows *summable* $(\lambda n.\ (id\text{-}blinfun - Q) \widehat{\ } n)$
using *onorm-le norm-blinfun-compose*
by (*force intro*!: *summable-ratio-test*)

lemma *blinfunpow-assoc*: $(F::'a::real\text{-}normed\text{-}vector \Rightarrow_L 'a) \widehat{\ } (Suc\ n) = (F \widehat{\ } n) \circ_L F$
by (*induction n*) (*auto simp*: *blinfun-compose-assoc*[*symmetric*])

lemma *norm-blinfunpow-le*: $norm\ ((f :: 'b :: real\text{-}normed\text{-}vector \Rightarrow_L 'b) \widehat{\ } n) \leq norm\ f \widehat{\ } n$
by (*induction n*) (*auto simp*: *norm-blinfun-id-le intro*!: *order.trans*[*OF norm-blinfun-compose*] *mult-left-mono*)

lemma *blinfunpow-nonneg*:
assumes $\bigwedge v.\ 0 \leq v \Longrightarrow 0 \leq blinfun\text{-}apply\ (f :: ('b :: \{ord, real\text{-}normed\text{-}vector\} \Rightarrow_L 'b))\ v$
shows $0 \leq v \Longrightarrow 0 \leq (f \widehat{\ } n)\ v$
by(*induction n*) (*auto simp*: *assms*)

lemma *blinfunpow-mono*:
assumes $\bigwedge u\ v.\ u \leq v \Longrightarrow (f :: 'b :: \{ord, real\text{-}normed\text{-}vector\} \Rightarrow_L 'b)\ u \leq f\ v$
shows $u \leq v \Longrightarrow (f \widehat{\ } n)\ u \leq (f \widehat{\ } n)\ v$
by (*induction n*) (*auto simp*: *assms*)

lemma *banach-blinfun*:
fixes $C :: 'b :: \{real\text{-}normed\text{-}vector, complete\text{-}space\} \Rightarrow_L 'b$
assumes *norm C* < *1*
shows $\exists!v.\ C\ v = v \bigwedge v.\ (\lambda n.\ (C \widehat{\ } n)\ v) \longrightarrow (THE\ v.\ C\ v = v)$
using *assms*
proof −
obtain *v* **where** $C\ v = v\ \forall v'.\ C\ v' = v' \longrightarrow v' = v$
using *assms banach-fix-type*[*of norm C blinfun-apply C*]
by (*metis blinfun.zero-right less-irrefl mult.left-neutral mult-less-le-imp-less*

norm-blinfun norm-conv-dist norm-ge-zero zero-less-dist-iff)
obtain *l* **where** $(\forall v\ u.\ norm\ (C\ (v - u)) \leq l * dist\ v\ u)\ 0 \leq l\ l < 1$
by (*metis assms dist-norm norm-blinfun norm-imp-pos-and-ge*)
hence *1*: $dist\ (C\ v)\ (C\ u) \leq l * dist\ v\ u$ **for** *v u*
by (*simp add*: *blinfun.diff-right dist-norm*)

```
  have 2: dist ((C ^^ n) v0) v ≤ l ^ n * dist v0 v for n v0
    using ‹0 ≤ l›
    by (induction n) (auto simp: mult.assoc
        intro!: mult-mono' order.trans[OF 1[of - v , unfolded ‹C v =
v›]])
  have (λn. l ^ n) ⟶ 0
    by (simp add: LIMSEQ-realpow-zero ‹0 ≤ l› ‹l < 1›)
  hence k: ⋀v0. (λn. l ^ n * dist v0 v) ⟶ 0
    by (auto simp add: tendsto-mult-left-zero)
  have (λn. dist ((C ^^ n) v0) v) ⟶ 0 for v0
    using k 2 order-trans abs-ge-self
    by (subst Limits.tendsto-0-le[where ?K = 1, where ?f = (λn. l
^ n * dist v0 v)])
       (fastforce intro: eventuallyI)+
  hence ⋀v0. (λn. (C ^^ n) v0) ⟶ v
    using tendsto-dist-iff
    by blast
  thus (λn. (C ^^ n) v0) ⟶ (THE v. C v = v) for v0
   using theI'[of λx. C x = x] ‹C v = v› ‹∀v'. C v' = v' ⟶ v' = v›
    by blast
next
  show norm C < 1 ⟹ ∃!v. blinfun-apply C v = v
    by (auto intro!: banach-fix-type[OF - assms]
        simp: dist-norm norm-blinfun blinfun.diff-right[symmetric])
qed
```

2.3 Geometric Sum

```
lemma inv-one-sub-Q:
  fixes Q :: 'a :: banach ⇒_L 'a
  assumes onorm-le: norm (id-blinfun − Q) < 1
  shows (Q o_L (∑ i. (id-blinfun − Q)^^i)) = id-blinfun
    and (∑ i. (id-blinfun − Q)^^i) o_L Q = id-blinfun
proof −
  obtain b where bh: b < 1  norm (id-blinfun − Q) < b
    using onorm-le dense
    by blast
  have 0 < b
    using le-less-trans[OF norm-ge-zero bh(2)] .
  have norm-le-aux: norm ((id-blinfun − Q)^^Suc n) ≤ b^(Suc n)
for n
  proof (induction n)
    case 0
    thus ?case
      using bh
      by simp
  next
    case (Suc n)
    thus ?case
```

```
    proof −
     have norm ((id-blinfun − Q) ^^ Suc (Suc n)) ≤ norm (id-blinfun
− Q) ∗ norm((id-blinfun − Q) ^^ Suc n)
        using norm-blinfun-compose
        by auto
      thus ?thesis
        using Suc.IH ‹0 < b› bh order.trans
        by (fastforce simp: mult-mono′)
    qed
  qed
  have (Q o_L (∑ i≤n. (id-blinfun − Q)^^i)) = (id-blinfun − (id-blinfun
− Q)^^(Suc n)) for n
   by (induction n) (auto simp: bounded-bilinear.diff-left bounded-bilinear.add-right

        bounded-bilinear-blinfun-compose)
  hence ⋀n. norm (id-blinfun − (Q o_L (∑ i≤n. (id-blinfun − Q)^^i)))
≤ b^Suc n
    using norm-le-aux
    by auto
  hence l2: (λn. (id-blinfun − (Q o_L (∑ i≤n. (id-blinfun − Q)^^i))))
⟶ 0
    using ‹0 < b› bh
     by (subst Lim-transform-bound[where g=λn. b^Suc n]) (auto
intro!: tendsto-eq-intros)
  have summable (λn. (id-blinfun − Q)^^n)
    using onorm-le norm-blinfun-compose
    by (force intro!: summable-ratio-test)
  hence (λn. ∑ i≤n. (id-blinfun − Q)^^i) ⟶ (∑ i. (id-blinfun
− Q)^^i)
    using summable-LIMSEQ′
    by blast
  hence (λn. Q o_L (∑ i≤n. (id-blinfun − Q)^^i)) ⟶ (Q o_L (∑ i.
(id-blinfun − Q)^^i))
    using bounded-bilinear-blinfun-compose
   by (subst Limits.bounded-bilinear.tendsto[where prod = (o_L)]) auto

   hence (λn. id-blinfun − (Q o_L (∑ i≤n. (id-blinfun − Q)^^i)))
⟶
    id-blinfun − (Q o_L (∑ i. (id-blinfun − Q)^^i))
    by (subst Limits.tendsto-diff) auto
  thus (Q o_L (∑ i. (id-blinfun − Q)^^i)) = id-blinfun
    using LIMSEQ-unique l2 by fastforce

  have ((∑ i≤n. (id-blinfun − Q)^^i) o_L Q) = (id-blinfun − (id-blinfun
− Q)^^(Suc n)) for n
  proof (induction n)
    case (Suc n)
    have sum ((^^) (id-blinfun − Q)) {..Suc n} o_L Q =
        (sum ((^^) (id-blinfun − Q)) {..n} o_L Q) + ((id-blinfun − Q)
```

$\widehat{\ }\widehat{\ }Suc\ n\ o_L\ Q$)
by (*simp add*: *bounded-bilinear.add-left bounded-bilinear-blinfun-compose*)
also have ... = *id-blinfun* − (((*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$(*Suc n*) o_L *id-blinfun*) −
((*id-blinfun* − *Q*) $\widehat{\ }\widehat{\ }$*Suc n* o_L *Q*))
using *Suc.IH*
by *auto*
also have ... = *id-blinfun* − (((*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$(*Suc n*) o_L (*id-blinfun* − *Q*)))
by (*auto intro*!: *blinfun-eqI simp*: *blinfun.diff-right blinfun.diff-left blinfun.minus-left*)
also have ... = *id-blinfun* − (((*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$(*Suc* (*Suc n*))))
using *blinfunpow-assoc*
by *metis*
finally show *?case*
by *auto*
qed *simp*
hence $\bigwedge n$. *norm* (*id-blinfun* − ((∑ *i*≤*n*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*)) ≤ *b*^*Suc n*
using *norm-le-aux* **by** *auto*
hence *l2*: (λ*n*. *id-blinfun* − ((∑ *i*≤*n*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*)) ⟶ *0*
using ‹*0* < *b*› *bh*
by (*subst Lim-transform-bound*[**where** *g*=λ*n*. *b*^*Suc n*]) (*auto intro*!: *tendsto-eq-intros*)
have *summable* (λ*n*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*n*)
using *local.onorm-le norm-blinfun-compose*
by (*force intro*!: *summable-ratio-test*)
hence (λ*n*. ∑ *i*≤*n*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) ⟶ (∑ *i*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*)
using *summable-LIMSEQ'* **by** *blast*
hence (λ*n*. (∑ *i*≤*n*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*) ⟶ ((∑ *i*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*)
using *bounded-bilinear-blinfun-compose*
by (*subst Limits.bounded-bilinear.tendsto*[**where** *prod* = (o_L)]) *auto*

hence (λ*n*. *id-blinfun* − ((∑ *i*≤*n*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*)) ⟶
id-blinfun − ((∑ *i*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*)
by (*subst Limits.tendsto-diff*) *auto*
thus ((∑ *i*. (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*) o_L *Q*) = *id-blinfun*
using *LIMSEQ-unique l2* **by** *fastforce*
qed

lemma *inv-norm-le*:
fixes *Q* :: *'a* :: *banach* ⇒$_L$ *'a*
assumes *norm Q* < *1*
shows (*id-blinfun*−*Q*) o_L (∑ *i*. *Q*$\widehat{\ }\widehat{\ }$*i*) = *id-blinfun*
(∑ *i*. *Q*$\widehat{\ }\widehat{\ }$*i*) o_L (*id-blinfun*−*Q*) = *id-blinfun*

using *inv-one-sub-Q*[*of id-blinfun* − *Q*] *assms*
by *auto*

lemma *inv-norm-le'*:
fixes Q :: *'a* :: *banach* $\Rightarrow_L$ *'a*
assumes *norm* Q < *1*
shows (*id-blinfun*−*Q*) ((∑ *i*. $Q^{\frown\frown}i$) x) = x
(∑ *i*. $Q^{\frown\frown}i$) ((*id-blinfun*−*Q*) x) = x
using *inv-norm-le assms*
by (*auto simp del*: *blinfun-apply-blinfun-compose*
simp: *inv-norm-le blinfun-apply-blinfun-compose*[*symmetric*])

2.4 Inverses

definition *is-inverse*$_L$ X Y ⟷ X o_L Y = *id-blinfun* ∧ Y o_L X = *id-blinfun*

abbreviation *invertible*$_L$ X ≡ ∃ X'. *is-inverse*$_L$ X X'

lemma *is-inverse*$_L$*-I*[*intro*]:
assumes X o_L Y = *id-blinfun* Y o_L X = *id-blinfun*
shows *is-inverse*$_L$ X Y
using *assms*
unfolding *is-inverse*$_L$*-def*
by *auto*

lemma *is-inverse*$_L$*-D*[*dest*]:
assumes *is-inverse*$_L$ X Y
shows X o_L Y = *id-blinfun* Y o_L X = *id-blinfun*
using *assms*
unfolding *is-inverse*$_L$*-def*
by *auto*

lemma *invertible*$_L$*-D*[*dest*]:
assumes *invertible*$_L$ f
obtains g **where** f o_L g = *id-blinfun* g o_L f = *id-blinfun*
using *assms*
by *auto*

lemma *invertible*$_L$*-I*[*intro*]:
assumes f o_L g = *id-blinfun* g o_L f = *id-blinfun*
shows *invertible*$_L$ f
using *assms*
by *auto*

lemma *is-inverse*$_L$*-comm*: *is-inverse*$_L$ X Y ⟷ *is-inverse*$_L$ Y X
by *auto*

lemma *is-inverse*$_L$*-unique*: *is-inverse*$_L$ f g ⟹ *is-inverse*$_L$ f h ⟹ g

$= h$
unfolding *is-inverse$_L$-def*
using *blinfun-compose-assoc blinfun-compose-id(1)*
by *metis*

lemma *is-inverse$_L$-ex1*: *is-inverse$_L$ f g* $\Longrightarrow \exists!h.$ *is-inverse$_L$ f h*
using *is-inverse$_L$-unique*
by *auto*

lemma *is-inverse$_L$-ex1'*: $\exists\, x.$ *is-inverse$_L$ f x* $\Longrightarrow \exists!x.$ *is-inverse$_L$ f x*
using *is-inverse$_L$-ex1*
by *auto*

definition *inv$_L$ f* = (*THE g. is-inverse$_L$ f g*)

lemma *inv$_L$-eq*:
assumes *is-inverse$_L$ f g*
shows *inv$_L$ f* = *g*
unfolding *inv$_L$-def*
using *assms is-inverse$_L$-ex1*
by (*auto intro*!: *the-equality*)

lemma *inv$_L$-I*:
assumes *f o$_L$ g* = *id-blinfun g o$_L$ f* = *id-blinfun*
shows *g* = *inv$_L$ f*
using *assms inv$_L$-eq*
unfolding *is-inverse$_L$-def*
by *auto*

lemma *inv-app1* [*simp*]: *invertible$_L$ X* $\Longrightarrow$ (*inv$_L$ X*) *o$_L$ X* = *id-blinfun*
using *is-inverse$_L$-ex1' inv$_L$-eq*
by *blast*

lemma *inv-app2*[*simp*]: *invertible$_L$ X* $\Longrightarrow$ *X o$_L$* (*inv$_L$ X*) = *id-blinfun*
using *is-inverse$_L$-ex1' inv$_L$-eq*
by *blast*

lemma *inv-app1'*[*simp*]: *invertible$_L$ X* $\Longrightarrow$ *inv$_L$ X* (*X v*) = *v*
using *inv-app1 blinfun-apply-blinfun-compose id-blinfun.rep-eq*
by *metis*

lemma *inv-app2'*[*simp*]: *invertible$_L$ X* $\Longrightarrow$ *X* (*inv$_L$ X v*) = *v*
using *inv-app2 blinfun-apply-blinfun-compose id-blinfun.rep-eq*
by *metis*

lemma *inv$_L$-inv$_L$*[*simp*]: *invertible$_L$ X* $\Longrightarrow$ *inv$_L$* (*inv$_L$ X*) = *X*
by (*metis inv$_L$-eq is-inverse$_L$-comm*)

lemma *inv$_L$-cancel-iff*:

assumes $invertible_L\ f$
shows $f\ x = y \longleftrightarrow x = inv_L\ f\ y$
by (*auto simp add*: *assms*)

lemma $invertible_L$*-inf-sum*:
assumes $norm\ (X :: 'b :: banach \Rightarrow_L 'b) < 1$
shows $invertible_L\ (id\text{-}blinfun - X)$
using *Blinfun-Util.inv-norm-le*[*OF assms*] *assms*
by *blast*

lemma inv_L*-inf-sum*:
fixes $X :: 'b :: banach \Rightarrow_L -$
assumes $norm\ X < 1$
shows $inv_L\ (id\text{-}blinfun - X) = (\sum i.\ X\ \widehat{\ }\ i)$
using *Blinfun-Util.inv-norm-le*[*OF assms*] *assms*
by (*auto simp*: inv_L*-I*[*symmetric*])

lemma $is\text{-}inverse_L$*-compose*:
assumes $invertible_L\ f\ invertible_L\ g$
shows $is\text{-}inverse_L\ (f\ o_L\ g)\ (inv_L\ g\ o_L\ inv_L\ f)$
by (*auto intro*!: *blinfun-eqI* $is\text{-}inverse_L$*-I*[*of - *$inv_L\ g\ o_L\ inv_L\ f$]
simp: *inv-app2'*[*OF assms*(*1*)] *inv-app2'*[*OF assms*(*2*)] *inv-app1'*[*OF assms*(*1*)] *inv-app1'*[*OF assms*(*2*)])

lemma $invertible_L$*-compose*: $invertible_L\ f \Longrightarrow invertible_L\ g \Longrightarrow invertible_L\ (f\ o_L\ g)$
using $is\text{-}inverse_L$*-compose*
by *blast*

lemma inv_L*-compose*:
assumes $invertible_L\ f\ invertible_L\ g$
shows$inv_L\ (f\ o_L\ g) = (inv_L\ g)\ o_L\ (inv_L\ f)$
using *assms* inv_L*-eq* $is\text{-}inverse_L$*-compose*
by *blast*

lemma inv_L*-id-blinfun*[*simp*]: $inv_L\ id\text{-}blinfun = id\text{-}blinfun$
by (*metis blinfun-compose-id*(*2*) inv_L*-I*)

2.5 Norm

lemma *bounded-range-subset*:
$bounded\ (range\ f :: real\ set) \Longrightarrow bounded\ (f\ `\ X')$
by (*auto simp*: *bounded-iff*)

lemma *bounded-const*: $bounded\ ((\lambda\text{-}.\ x)\ `\ X)$
by (*meson finite-imp-bounded finite.emptyI finite-insert finite-subset image-subset-iff insert-iff*)

lift-definition $bfun\text{-}pos :: ('d \Rightarrow_b real) \Rightarrow ('d \Rightarrow_b real)$ **is** $\lambda f\ i.\ if\ f\ i$

< 0 *then* $-f\ i$ *else* $f\ i$
using *bounded-const bounded-range-subset* **by** (*auto simp*: *bfun-def*)

lemma *bfun-pos-zero*[*simp*]: *bfun-pos* $f = 0 \longleftrightarrow f = 0$
by (*auto intro*!: *bfun-eqI simp*: *bfun-pos.rep-eq split*: *if-splits*)

lift-definition *bfun-nonneg* :: $('d \Rightarrow_b real) \Rightarrow ('d \Rightarrow_b real)$ **is** $\lambda f\ i.$ *if* $f\ i \leq 0$ *then* 0 *else* $f\ i$
using *bounded-const bounded-range-subset* **by** (*auto simp*: *bfun-def*)

lemma *bfun-nonneg-split*: *bfun-nonneg* $x -$ *bfun-nonneg* $(-x) = x$
by (*auto simp*: *bfun-nonneg.rep-eq*)

lemma *blinfun-split*: *blinfun-apply* $f\ x = f$ (*bfun-nonneg* x) $-$ f (*bfun-nonneg* $(-x)$)
using *bfun-nonneg-split*
by (*metis blinfun.diff-right*)

lemma *bfun-nonneg-pos*: *bfun-nonneg* $x +$ *bfun-nonneg* $(-x) =$ *bfun-pos* x
by (*auto simp*: *bfun-nonneg.rep-eq bfun-pos.rep-eq*)

lemma *bfun-nonneg*: $0 \leq$ *bfun-nonneg* f
by (*auto simp*: *bfun-nonneg.rep-eq*)

lemma *bfun-pos-eq-nonneg*: *bfun-pos* $n =$ *bfun-nonneg* $n +$ *bfun-nonneg* $(-n)$
by (*auto simp*: *bfun-pos.rep-eq bfun-nonneg.rep-eq*)

lemma *blinfun-mono-norm-pos*:
fixes $f :: ('c \Rightarrow_b real) \Rightarrow_L ('d \Rightarrow_b real)$
assumes $\bigwedge v :: 'c \Rightarrow_b real.\ v \geq 0 \Longrightarrow f\ v \geq 0$
shows *norm* $(f\ n) \leq$ *norm* $(f$ (*bfun-pos* $n))$
proof $-$
have $*$: $|f\ n\ i| \leq |f$ (*bfun-pos* $n)\ i|$ **for** i
by (*auto simp*: *blinfun-split*[*of f n*] *bfun-nonneg-pos*[*symmetric*] *blinfun.add-right abs-real-def*)
(*metis add-nonneg-nonneg assms bfun-nonneg leD less-eq-bfun-def zero-bfun.rep-eq*)+
thus *norm* $(f\ n) \leq$ *norm* $((f$ (*bfun-pos* $n)))$
unfolding *norm-bfun-def*′ **using** $*$
by (*auto intro*!: *cSUP-mono bounded-imp-bdd-above abs-le-norm-bfun boundedI*[*of - norm* $((f$ (*bfun-pos* $n)))$])
qed

lemma *norm-bfun-pos*[*simp*]: *norm* (*bfun-pos* f) = *norm* f
proof $-$
have *norm* (*bfun-pos* f) = $(\bigsqcup i.\ |$*bfun-pos* $f\ i|)$
by (*auto simp add*: *norm-bfun-def*′)

```
  also have ... = (⨆ i. |f i|)
    by (rule SUP-cong[OF refl]) (auto simp: bfun-pos.rep-eq)
  finally show ?thesis by (auto simp add: norm-bfun-def')
qed

lemma norm-blinfun-mono-eq-nonneg:
  fixes f :: ('c ⇒b real) ⇒L ('d ⇒b real)
  assumes ⋀v :: 'c ⇒b real. v ≥ 0 ⟹ f v ≥ 0
  shows norm f = (⨆ v ∈ {v. v ≥ 0}. norm (f v) / norm v)
  unfolding norm-blinfun.rep-eq onorm-def
proof (rule antisym, rule cSUP-mono)
  have *: norm (blinfun-apply f v) / norm v ≤ norm f for v
    using norm-blinfun[of f]
    by (cases v = 0) (auto simp: pos-divide-le-eq)
  thus bdd-above ((λv. norm (f v) / norm v) ' {v. 0 ≤ v})
    by (auto intro!: bounded-imp-bdd-above boundedI)
  show ∃ m∈{v. 0 ≤ v}. norm (f n) / norm n ≤ norm (f m) / norm
m for n
    using blinfun-mono-norm-pos[OF assms]
    by (cases norm (bfun-pos n) = 0)
       (auto intro!: frac-le exI[of - bfun-pos n] simp: less-eq-bfun-def
bfun-pos.rep-eq)
  show (⨆ v∈{v. 0 ≤ v}. norm (f v) / norm v) ≤ (⨆ x. norm (f x) /
norm x)
    using *
    by (auto intro!: cSUP-mono bounded-imp-bdd-above boundedI)
qed auto

lemma norm-blinfun-normalized-le: norm (blinfun-apply f v) / norm
v ≤ norm f
 by (simp add: blinfun.bounded-linear-right le-onorm norm-blinfun.rep-eq)

lemma norm-blinfun-mono-eq-nonneg':
  fixes f :: ('c ⇒b real) ⇒L ('d ⇒b real)
  assumes ⋀v :: 'c ⇒b real. 0 ≤ v ⟹ 0 ≤ f v
  shows norm f = (⨆ x ∈ {x. norm x = 1 ∧ x ≥ 0}. norm (f x))
proof (subst norm-blinfun-mono-eq-nonneg[OF assms])
  show (⨆ v∈{v. 0 ≤ v}. norm (f v) / norm v) =
    (⨆ x∈{x. norm x = 1 ∧ 0 ≤ x}. norm (f x))
  proof (rule antisym, rule cSUP-mono)
    show {v::'c ⇒b real. 0 ≤ v} ≠ {} by auto
    show bdd-above ((λx. norm (f x)) ' {x. norm x = 1 ∧ 0 ≤ x})
    by (fastforce intro: order.trans[OF norm-blinfun[of f]] bounded-imp-bdd-above
boundedI)
   show ∃ m∈{x. norm x = 1 ∧ 0 ≤ x}. norm (f n) / norm n ≤ norm
(f m) if n ∈ {v. 0 ≤ v} for n
    proof (cases norm (bfun-pos n) = 0)
      case True
      then show ?thesis by (auto intro!: exI[of - 1])
```

```
    next
      case False
      then show ?thesis
        using that
       by (auto simp: scaleR-nonneg-nonneg blinfun.scaleR-right intro!:
exI[of - (1/norm n) *R n])
    qed
    show (⨆x∈{x. norm x = 1 ∧ 0 ≤ x}. norm (f x)) ≤ (⨆v∈{v. 0
≤ v}. norm (f v) / norm v)
    proof (rule cSUP-mono)
      show {x::'c ⇒b real. norm x = 1 ∧ 0 ≤ x} ≠ {}
        by (auto intro!: exI[of - 1])
   qed (fastforce intro!: norm-blinfun-normalized-le bounded-imp-bdd-above
boundedI)+
  qed
qed auto

lemma norm-blinfun-mono-le-norm-one:
  fixes f :: ('c ⇒b real) ⇒L ('d ⇒b real)
  assumes ⋀v :: 'c ⇒b real. v ≥ 0 ⟹ f v ≥ 0
  assumes norm x = 1 0 ≤ x
  shows norm (f x) ≤ norm (f 1)
proof −
  have **: 0 ≤ 1 − x
    using assms
   by (auto simp: less-eq-bfun-def intro: order.trans[OF le-norm-bfun])
  show ?thesis
    unfolding norm-bfun-def'
  proof (intro cSUP-mono)
     showbdd-above (range (λx. norm (apply-bfun (blinfun-apply f 1)
x)))
      using order.trans abs-le-norm-bfun norm-blinfun
      by (fastforce intro!: bounded-imp-bdd-above boundedI)
    show ∃ m∈UNIV. norm (f x n) ≤ norm (f 1 m) for n
      using assms(1) assms(3) assms(1)[of 1 − x] **
      unfolding less-eq-bfun-def zero-bfun.rep-eq abs-real-def
      by (auto simp: blinfun.diff-right linorder-class.not-le[symmetric])
  qed auto
qed

lemma norm-blinfun-mono-eq-one:
  fixes f :: ('c ⇒b real) ⇒L ('d ⇒b real)
  assumes ⋀v :: 'c ⇒b real. v ≥ 0 ⟹ f v ≥ 0
  shows norm f = norm (f 1)
proof −
  have (⨆x∈{x. norm x = 1 ∧ 0 ≤ x}. norm (f x)) = norm (f 1)
  proof (rule antisym, rule cSUP-least)
    show {x::'c ⇒b real. norm x = 1 ∧ 0 ≤ x} ≠ {}
      by (auto intro!: exI[of - 1])
```

next
show $\bigwedge x.\ x \in \{x.\ norm\ x = 1 \wedge 0 \leq x\} \Longrightarrow norm\ (f\ x) \leq norm\ (f\ 1)$
by (*simp add*: *assms norm-blinfun-mono-le-norm-one*)
next
show $norm\ (f\ 1) \leq (\bigsqcup x{\in}\{x.\ norm\ x = 1 \wedge 0 \leq x\}.\ norm\ (f\ x))$
by (*rule cSUP-upper*) (*fastforce intro*!: *bdd-aboveI2 order.trans*[*OF norm-blinfun*])+
qed
thus *?thesis*
using *norm-blinfun-mono-eq-nonneg'*[*OF assms*]
by *auto*
qed

2.6 Miscellaneous

lemma *bounded-linear-apply-bfun*: *bounded-linear* $(\lambda x.\ apply\text{-}bfun\ x\ i)$
using *norm-le-norm-bfun*
by (*fastforce intro*: *bounded-linear-intro*[*of - 1*])

lemma *lim-blinfun-apply*: *convergent* $X \Longrightarrow (\lambda n.\ blinfun\text{-}apply\ (X\ n)\ u) \longrightarrow lim\ X\ u$
using *blinfun.bounded-bilinear-axioms*
by (*auto simp*: *convergent-LIMSEQ-iff intro*: *Limits.bounded-bilinear.tendsto*)

lemma *bounded-apply-blinfun*:
assumes *bounded* $((F :: 'c \Rightarrow 'd{::}real\text{-}normed\text{-}vector \Rightarrow_L 'b{::}real\text{-}normed\text{-}vector)\ `\ S)$
shows *bounded* $((\lambda b.\ blinfun\text{-}apply\ (F\ b)\ x)\ `\ S)$
proof −
obtain b **where** $\forall\, x{\in}S.\ norm\ (F\ x) \leq b$
by (*meson* ‹*bounded* $(F\ `\ S)$› *bounded-pos image-eqI*)
thus *bounded* $((\lambda b.\ (F\ b)\ x)\ `\ S)$
by (*auto simp*: *mult-right-mono mult.commute*[*of - b*]
intro!: *boundedI*[*of - norm x * b*] *dual-order.trans*[*OF - norm-blinfun*])
qed

lemma *tendsto-blinfun-apply*: $(\lambda n.\ X\ n) \longrightarrow L \Longrightarrow (\lambda n.\ blinfun\text{-}apply\ (X\ n)\ u) \longrightarrow L\ u$
using *blinfun.bounded-bilinear-axioms*
by (*auto simp*: *convergent-LIMSEQ-iff intro*: *Limits.bounded-bilinear.tendsto*)

definition *nonneg-blinfun* $(Q :: -{::}\{ordered\text{-}real\text{-}normed\text{-}vector\} \Rightarrow_L -{::}\{ordered\text{-}ab\text{-}group\text{-}add, ordered\text{-}real\text{-}normed\text{-}vector\}) \equiv (\forall\, v{\geq}0.\ blinfun\text{-}apply\ Q\ v \geq 0)$

definition *blinfun-le* $Q\ R = nonneg\text{-}blinfun\ (R - Q)$

lemma *nonneg-blinfun-nonneg*[*dest*]: *nonneg-blinfun Q* $\Longrightarrow$ *0* $\leq$ *v* $\Longrightarrow$ *0* $\leq$ *Q v*
unfolding *nonneg-blinfun-def*
by *auto*

lemma *nonneg-blinfun-mono*[*dest*]: *nonneg-blinfun Q* $\Longrightarrow$ *u* $\leq$ *v* $\Longrightarrow$ *Q u* $\leq$ *Q v*
using *nonneg-blinfun-nonneg*[*of Q v − u*, *unfolded blinfun.diff-right*]
by *auto*

lemma *nonneg-id-blinfun*: *nonneg-blinfun id-blinfun*
by (*auto simp*: *nonneg-blinfun-def*)

lemma *blinfun-nonneg-eq*:
assumes $\forall$ *v* $\geq$ *0*. *blinfun-apply* (*f*::(*'c* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*'c* $\Rightarrow_b$ *real*)) *v* = *blinfun-apply g v*
shows *f* = *g*
proof (*rule blinfun-eqI*)
fix *v* :: *'c* $\Rightarrow_b$ *real*
define *v1* **where** *v1* = *Bfun* (λ*x*. *max* (*v x*) *0*)
define *v2* **where** *v2* = *Bfun* (λ*x*. − *min* (*v x*) *0*)
have *in-bfun*[*simp*]: (λ*x*. *max* (*v x*) *0*) $\in$ *bfun* (λ*x*. − *min* (*v x*) *0*) $\in$ *bfun*
by (*auto simp*: *le-norm-bfun minus-min-eq-max abs-le-norm-bfun abs-le-D2 intro*!: *boundedI*[*of - norm v*])
have *eq-v*: *v* = *v1* − *v2*
unfolding *v1-def v2-def*
by (*auto simp*: *Bfun-inverse*)
have *nonneg*: *0* $\leq$ *v1 0* $\leq$ *v2*
unfolding *less-eq-bfun-def*
by (*auto simp*: *v1-def v2-def Bfun-inverse*)
show *blinfun-apply f v* = *blinfun-apply g v*
unfolding *eq-v*
using *nonneg assms*
by (*auto simp*: *blinfun.diff-right*)
qed

lemma *bfun-zero-le-one*: *0* $\leq$ (*1* :: *'c* $\Rightarrow_b$ *real*)
by (*simp add*: *less-eq-bfunI*)

lemma *norm-nonneg-blinfun-one*:
assumes *nonneg-blinfun* (*X* :: (*'c* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*'c* $\Rightarrow_b$ *real*))
shows *norm X* = *norm* (*blinfun-apply X 1*)
using *assms* **unfolding** *nonneg-blinfun-def*

by (*auto simp*: *norm-blinfun-mono-eq-one*)

lemma *blinfun-apply-mono*: *nonneg-blinfun X* $\Longrightarrow$ *0* $\leq$ *v* $\Longrightarrow$ *blinfun-le X Y* $\Longrightarrow$ *X v* $\leq$ *Y v*
by (*simp add*: *blinfun.diff-left blinfun-le-def nonneg-blinfun-def*)

lemma *nonneg-blinfun-scaleR*[*intro*]: *nonneg-blinfun B* $\Longrightarrow$ *0* $\leq$ *c* $\Longrightarrow$ *nonneg-blinfun* (*c* $*_R$ *B*)
by (*simp add*: *nonneg-blinfun-def scaleR-blinfun.rep-eq scaleR-nonneg-nonneg*)

lemma *nonneg-blinfun-compose*[*intro*]: *nonneg-blinfun B* $\Longrightarrow$ *nonneg-blinfun C* $\Longrightarrow$ *nonneg-blinfun* (*C* o_L *B*)
by (*simp add*: *nonneg-blinfun-def*)

lemma *matrix-le-norm-mono*:
assumes *nonneg-blinfun* (*C* :: (*'c* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*'c* $\Rightarrow_b$ *real*))
and *nonneg-blinfun* (*D* $-$ *C*)
shows *norm C* $\leq$ *norm D*
proof $-$
have *nonneg-blinfun D*
using *assms*
by (*metis add-nonneg-nonneg diff-add-cancel nonneg-blinfun-def plus-blinfun.rep-eq*)
have *zero-le*: *0* $\leq$ *C 1 0* $\leq$ *D 1*
using *assms zero-le-one* ‹*nonneg-blinfun D*›
by (*auto simp add*: *less-eq-bfunI nonneg-blinfun-nonneg*)
hence *C 1* $\leq$ *D 1*
using *assms*(*2*) **unfolding** *nonneg-blinfun-def blinfun.diff-left*
by (*simp add*: *less-eq-bfun-def*)
thus *?thesis*
using *assms* ‹*nonneg-blinfun D*› *zero-le le-norm-bfun*
by (*fastforce simp*: *norm-nonneg-blinfun-one norm-bfun-def' less-eq-bfun-def intro*!: *bdd-above.I2 cSUP-mono*)
qed

lemma *bounded-subset*: *Y* $\subseteq$ *X* $\Longrightarrow$ *bounded* (*f* ‘ *X*) $\Longrightarrow$ *bounded* (*f* ‘ *Y*)
by (*auto simp*: *bounded-def*)

lemma *bounded-subset-range*: *bounded* (*range f*) $\Longrightarrow$ *bounded* (*f* ‘ *Y*)
using *bounded-subset subset-UNIV* **by** *metis*

lift-definition *bfun-if* :: (*'b* $\Rightarrow$ *bool*) $\Rightarrow$ (*'b* $\Rightarrow_b$ *'c*::*metric-space*) $\Rightarrow$ (*'b* $\Rightarrow_b$ *'c*) $\Rightarrow$ (*'b* $\Rightarrow_b$ *'c*) **is** λ*b u v s*. *if b s then u s else v s*
using *bounded-subset-range*

by (*auto simp*: *bfun-def*)

lemma *bfun-if-add*: *bfun-if b* (*w* + *z*) (*u* + *v*) = *bfun-if b w u* + *bfun-if b z v*
by (*auto simp*: *bfun-if.rep-eq*)

lemma *bfun-if-zero-add*: *bfun-if b 0* (*u* + *v*) = *bfun-if b 0 u* + *bfun-if b 0 v*
by (*auto simp*: *bfun-if.rep-eq*)

lemma *bfun-if-zero-le*: *0* ≤ *v* ⟹ *bfun-if b 0 v* ≤ *v*
by (*metis* (*no-types*, *lifting*) *bfun-if.rep-eq le-less less-eq-bfun-def*)

lemma *bfun-if-eq*: (⋀*i*. *P i* ⟹ *apply-bfun v i* = *apply-bfun u i*) ⟹ (⋀*i*. ¬*P i* ⟹ *v i* = *apply-bfun w i*) ⟹ *bfun-if P u w* = *v*
by (*auto simp*: *bfun-if.rep-eq*)

lemma *bfun-if-scaleR*: *c* *$_R$ *bfun-if b v1 v2* = *bfun-if b* (*c* *$_R$ *v1*) (*c* *$_R$ *v2*)
by (*auto simp*: *bfun-if.rep-eq*)

lemma *summable-blinfun-apply*:
assumes *summable* (*f* :: *nat* ⇒ *'a*::*real-normed-vector* ⇒$_L$ *'a*)
shows *summable* (λ*n*. *f n v*)
using *assms tendsto-blinfun-apply*
unfolding *summable-def sums-def blinfun.sum-left*[*symmetric*]
by *auto*

lemma *blinfun-apply-suminf*:
assumes *summable* (*f* :: *nat* ⇒ *'a*::*real-normed-vector* ⇒$_L$ *'a*)
shows (∑ *k*. *blinfun-apply* (*f k*) *v*) = (∑ *k*. *f k*) *v*
using *bounded-linear.suminf*[*OF blinfun.bounded-linear-left assms*]
by *auto*
end
theory *MDP-reward-Util*
imports *Blinfun-Util*
begin

3 Auxiliary Lemmas

3.1 Summability

lemma *summable-powser-const*:
fixes *c* :: *real*
assumes |*c*| < *1*

shows *summable* ($\lambda n.\ c\hat{}n * x$)
using *assms*
by (*auto simp*: *mult.commute*)

3.2 Infinite sums

lemma *suminf-split-head'*:
summable ($f :: nat \Rightarrow 'x :: real\text{-}normed\text{-}vector$) $\Longrightarrow$ *suminf f* = *f 0* + ($\sum n.\ f\ (Suc\ n)$)
by (*auto simp*: *suminf-split-head*)

lemma *sum-disc-lim*:
assumes $|c :: real| < 1$
shows ($\sum x.\ c\hat{}x * B$) = $B\ /(1-c)$
by (*simp add*: *assms suminf-geometric summable-geometric suminf-mult2*[*symmetric*])

3.3 Bounded Functions

lemma *suminf-apply-bfun*:
fixes $f :: nat \Rightarrow 'c \Rightarrow_b real$
assumes *summable f*
shows ($\sum i.\ f\ i$) x = ($\sum i.\ f\ i\ x$)
by (*auto intro*!: *bounded-linear.suminf assms bounded-linear-intro*[**where** *K* = *1*] *abs-le-norm-bfun*)

lemma *sum-apply-bfun*:
fixes $f :: nat \Rightarrow 'c \Rightarrow_b real$
shows ($\sum i{<}n.\ f\ i$) x = ($\sum i{<}n.\ apply\text{-}bfun\ (f\ i)\ x$)
by (*induction n*) *auto*

3.4 Push-Forward of a Bounded Function

lemma *integrable-bfun-prob-space* [*simp*]:
integrable (*measure-pmf P*) ($\lambda t.\ apply\text{-}bfun\ f\ (F\ t) :: real$)
proof −
obtain *b* **where** $\forall t.\ |f\ (F\ t)| \leq b$
by (*metis norm-le-norm-bfun real-norm-def*)
hence ($\int^+ x.\ ennreal\ |f\ (F\ x)|\ \partial P$) $\leq b$
using *nn-integral-mono ennreal-leI*
by (*auto intro*: *measure-pmf.nn-integral-le-const*)
then show *?thesis*
using *ennreal-less-top le-less-trans*
by (*fastforce simp*: *integrable-iff-bounded*)
qed

lift-definition *push-exp* :: ($'b \Rightarrow 'c\ pmf$) $\Rightarrow$ ($'c \Rightarrow_b real$) $\Rightarrow$ ($'b \Rightarrow_b real$) **is**
$\lambda c\ f\ s.\ measure\text{-}pmf.expectation\ (c\ s)\ f$
using *bfun-integral-bound'* **.**

declare *push-exp.rep-eq*[*simp*]

lemma *norm-push-exp-le-norm*: *norm* (*push-exp d x*) ≤ *norm x*
proof −
  have ⋀*s*. (∫ *s'*. *norm* (*x s'*) ∂*d s*) ≤ *norm x*
    using *measure-pmf.prob-space-axioms norm-le-norm-bfun*[*of x*]
    by (*auto intro*!: *prob-space.integral-le-const*)
  hence *aux*: ⋀*s*. *norm* (∫ *s'*. *x s'* ∂*d s*) ≤ *norm x*
    using *integral-norm-bound order-trans* **by** *blast*
  have *norm* (*push-exp d x*) = (⨆ *s*. *norm* (∫ *s'*. *x s'* ∂*d s*))
    unfolding *norm-bfun-def'*
    by *auto*
  also have ... ≤ *norm x*
    using *aux* **by** (*fastforce intro*!: *cSUP-least*)
  finally show *?thesis* **.**
qed

lemma *push-exp-bounded-linear* [*simp*]: *bounded-linear* (*push-exp d*)
  using *norm-push-exp-le-norm*
  by (*auto intro*!: *bounded-linear-intro*[**where** *K* = *1*])

lemma *onorm-push-exp* [*simp*]: *onorm* (*push-exp d*) = *1*
proof (*intro antisym*)
  show *onorm* (*push-exp d*) ≤ *1*
    using *norm-push-exp-le-norm*
    by (*auto intro*!: *onorm-bound*)
next
  show *1* ≤ *onorm* (*push-exp d*)
    using *onorm*[*of - 1*, *OF push-exp-bounded-linear*]
    by (*auto simp*: *norm-bfun-def'*)
qed

lemma *push-exp-return*[*simp*]: *push-exp return-pmf* = *id*
  by (*auto simp*: *eq-id-iff*[*symmetric*])

3.5 Boundedness

lemma *bounded-abs*[*intro*]:
  bounded (*X'* :: *real set*) ⟹ *bounded* (*abs* ‘ *X'*)
  by (*auto simp*: *bounded-iff*)

lemma *bounded-abs-range*[*intro*]:
  bounded (*range f* :: *real set*) ⟹ *bounded* (*range* (λ*x*. *abs* (*f x*)))
  by (*auto simp*: *bounded-iff*)

3.6 Probability Theory

lemma *integral-measure-pmf-bind*:
  assumes (⋀*x*. |(*f* :: *'b* ⇒ *real*) *x*| ≤ *B*)

shows $(\int x.\ f\ x\ \partial((\textit{measure-pmf}\ M) \gg\!= (\lambda x.\ \textit{measure-pmf}\ (N\ x)))) = (\int x.\ \int y.\ f\ y\ \partial N\ x\ \partial M)$
using *assms*
by (*subst integral-bind*[*of - count-space UNIV B*]) (*auto simp*: *measure-pmf-in-subprob-space*)

lemma *lemma-4-3-1'*:
assumes *set-pmf p* $\subseteq$ *W*
and *bounded* ((*w* :: *'c* $\Rightarrow$ *real*) ‘ *W*)
and *W* $\neq$ {}
and *measure-pmf.expectation p w* = ($\bigsqcup p \in \{p.\ \textit{set-pmf}\ p \subseteq W\}$. *measure-pmf.expectation p w*)
shows $\exists\, x \in W.$ *measure-pmf.expectation p w* = *w x*
proof −
have *abs-w-le-sup*: *y* $\in$ *W* $\Longrightarrow$ |*w y*| $\leq$ ($\bigsqcup x \in W.$ |*w x*|) **for** *y*
using *assms bounded-abs*[*OF assms*(*2*)]
by (*auto intro*!: *cSUP-upper bounded-imp-bdd-above simp*: *image-image*)
have *False* **if** *x* $\in$ *set-pmf p w x* < $\bigsqcup$(*w* ‘ *W*) **for** *x*
proof −
have *ex-gr*: $\exists\, x'.\ x' \in W \wedge w\ x < w\ x'$
using *cSUP-least*[*of W w w x*] *that assms*
by *fastforce*
let *?s* = $\lambda s.$ (*if x* = *s then SOME x'*. *x'* $\in$ *W* $\wedge$ *w x* < *w x' else s*)
have *measure-pmf.expectation p w* < *measure-pmf.expectation p* ($\lambda xa.$ *w* (*?s xa*))
proof (*intro measure-pmf.integral-less-AE*[**where** *A* = {*x*}])
show *integrable* (*measure-pmf p*) *w*
using *assms abs-w-le-sup*
by (*fastforce simp*: *AE-measure-pmf-iff*
intro!: *measure-pmf.integrable-const-bound*)
show *integrable* (*measure-pmf p*) ($\lambda xa.$ *w* (*?s xa*))
using *assms*(*1*) *ex-gr someI*[**where** *P* = $\lambda x'.$ (*x'* $\in$ *W*) $\wedge$ (*w x* < *w x'*)]
by (*fastforce simp*: *AE-measure-pmf-iff*
intro!: *abs-w-le-sup measure-pmf.integrable-const-bound*)
show *emeasure* (*measure-pmf p*) {*x*} $\neq$ *0*
by (*simp add*: *emeasure-pmf-single-eq-zero-iff* ‹*x* $\in$ *p*›)
show {*x*} $\in$ *measure-pmf.events p*
by *auto*
show *AE xa*∈{*x*} *in p*. *w xa* $\neq$ *w* (*?s xa*) *AE xa in p*. *w xa* $\leq$ *w* (*?s xa*)
using *someI*[**where** *P* = $\lambda x'.$ (*x'* $\in$ *W*) $\wedge$ (*w x* < *w x'*)] *ex-gr*
by (*fastforce intro*!: *AE-pmfI*)+
qed
hence *measure-pmf.expectation p w* < $\bigsqcup$((*λp. measure-pmf.expectation p w*) ‘ {*p. set-pmf p* $\subseteq$ *W*})
proof (*subst less-cSUP-iff*, *goal-cases*)
case *1*

```
    then show ?case
      using assms(1)
      by blast
  next
    case 2
    then show ?case
      using abs-w-le-sup
      by (fastforce
          simp: AE-measure-pmf-iff
         intro: cSUP-upper2 bdd-aboveI[where M = (⨆x∈W. |w x|)]
         intro!: measure-pmf.integral-le-const measure-pmf.integrable-const-bound)
  next
    case 3
    then show ?case
      using ex-gr someI[where P = λx'. (x' ∈ W) ∧ (w x < w x')]
assms(1)
      by (auto intro!: exI[of - map-pmf ?s p])
  qed
  thus False
    using assms by auto
qed
hence 1: x ∈ set-pmf p ⟹ w x = ⨆ (w ‘ W) for x
  using assms
 by (fastforce intro: antisym simp: bounded-imp-bdd-above cSUP-upper)
hence w (SOME x. x ∈ set-pmf p) = ⨆ (w ‘ W)
  by (simp add: set-pmf-not-empty some-in-eq)
thus ?thesis
  using 1 assms(1) set-pmf-not-empty some-in-eq
  by (fastforce intro!: bexI[of - SOME x. x ∈ set-pmf p]
   simp: AE-measure-pmf-iff Bochner-Integration.integral-cong-AE[where
?g = λ-. ⨆ (w ‘ W)])
qed

lemma lemma-4-3-1:
  assumes set-pmf p ⊆ W integrable (measure-pmf p) w bounded ((w
:: 'c ⇒ real) ‘ W)
  shows measure-pmf.expectation p w ≤ ⨆(w ‘ W)
  using assms bounded-has-Sup(1) prob-space-measure-pmf
 by (fastforce simp: AE-measure-pmf-iff intro!: prob-space.integral-le-const)

lemma bounded-integrable:
  assumes bounded (range v) v ∈ borel-measurable (measure-pmf p)
  shows integrable (measure-pmf p) (v :: 'c ⇒ real)
  using assms
 by (auto simp: bounded-iff AE-measure-pmf-iff intro!: measure-pmf.integrable-const-bound)
```

3.7 Argmax

lemma *finite-is-arg-max*: *finite* $X \Longrightarrow X \neq \{\} \Longrightarrow \exists\, x.$ *is-arg-max* (f :: $'c \Rightarrow$ *real*) ($\lambda x.\ x \in X$) x
unfolding *is-arg-max-def*
proof (*induction rule*: *finite-induct*)
case (*insert* x F)
then show *?case*
proof (*cases* $\forall\, y \in F.\ f\ y \leq f\ x$)
case *True*
then show *?thesis*
by (*auto intro*!: *exI*[*of* - x])
next
case *False*
then show *?thesis*
using *insert* **by** *force*
qed
qed *simp*

lemma *finite-arg-max-le*:
assumes *finite* (X :: $'c$ *set*) $X \neq \{\}$
shows $s \in X \Longrightarrow$ (f :: $'c \Rightarrow$ *real*) $s \leq f$ (*arg-max-on* (f :: $'c \Rightarrow$ *real*) X)
unfolding *arg-max-def arg-max-on-def*
by (*metis assms*(*1*) *assms*(*2*) *finite-is-arg-max is-arg-max-linorder someI-ex*)

lemma *arg-max-on-in*:
assumes *finite* (X :: $'c$ *set*) $X \neq \{\}$
shows (*arg-max-on* (f :: $'c \Rightarrow$ *real*) X) $\in X$
unfolding *arg-max-on-def arg-max-def*
by (*metis assms*(*1*) *assms*(*2*) *finite-is-arg-max is-arg-max-def someI*)

lemma *finite-arg-max-eq-Max*:
assumes *finite* (X :: $'c$ *set*) $X \neq \{\}$
shows (f :: $'c \Rightarrow$ *real*) (*arg-max-on* f X) = *Max* (f ‘ X)
using *assms*
by (*auto intro*!: *Max-eqI*[*symmetric*] *finite-arg-max-le arg-max-on-in*)

lemma *arg-max-SUP*: *is-arg-max* (f :: $'b \Rightarrow$ *real*) ($\lambda x.\ x \in X$) $m \Longrightarrow$ $f\ m = (\bigsqcup (f\ `\ X))$
unfolding *is-arg-max-def*
by (*auto intro*!: *antisym cSUP-upper bdd-aboveI*[*of* - $f\ m$] *cSUP-least*)

definition *has-max* $X \equiv \exists\, x \in X.\ \forall\, x' \in X.\ x' \leq x$
definition *has-arg-max* $f\ X \equiv \exists\, x.$ *is-arg-max* f ($\lambda x.\ x \in X$) x

lemma *has-max* ((f :: $'b \Rightarrow$ *real*) ‘ X) $\longleftrightarrow$ *has-arg-max* $f\ X$
unfolding *has-max-def has-arg-max-def is-arg-max-def*

using *not-less* **by** (*auto dest*!: *leD simp*: *not-less*)

lemma *has-arg-max-is-arg-max*: *has-arg-max f X* $\Longrightarrow$ *is-arg-max f* ($\lambda x.\ x \in X$) (*arg-max f* ($\lambda x.\ x \in X$))
unfolding *has-arg-max-def arg-max-def*
by (*auto intro*: *someI*)

lemma *has-arg-max-arg-max*: *has-arg-max f X* $\Longrightarrow$ (*arg-max f* ($\lambda x.\ x \in X$)) $\in X$
unfolding *has-arg-max-def arg-max-def is-arg-max-def* **by** (*auto intro*: *someI2-ex*)

lemma *app-arg-max-ge*: *has-arg-max* (*f* :: *'b* $\Rightarrow$ *real*) *X* $\Longrightarrow$ *x* $\in$ *X* $\Longrightarrow$ *f x* $\leq$ *f* (*arg-max-on f X*)
unfolding *has-arg-max-def arg-max-on-def arg-max-def is-arg-max-def*
using *someI*[**where** *?P* = $\lambda x.\ x \in X \wedge (\nexists y.\ y \in X \wedge f\ x < f\ y)$] *le-less-linear*
by *auto*

lemma *app-arg-max-eq-SUP*: *has-arg-max* (*f* :: *'b* $\Rightarrow$ *real*) *X* $\Longrightarrow$ *f* (*arg-max-on f X*) = $\bigsqcup$(*f* ‘ *X*)
by (*simp add*: *arg-max-SUP arg-max-on-def has-arg-max-is-arg-max*)

lemma *SUP-is-arg-max*:
assumes *x* $\in$ *X bdd-above* (*f* ‘ *X*) (*f* :: *'c* $\Rightarrow$ *real*) *x* = $\bigsqcup$(*f* ‘ *X*)
shows *is-arg-max f* ($\lambda x.\ x \in X$) *x*
unfolding *is-arg-max-def*
using *not-less assms cSUP-upper*[*of - X f*]
by *auto*

lemma *is-arg-max-linorderI*[*intro*]: **fixes** *f* :: *'c* $\Rightarrow$ *'b* :: *linorder*
assumes *P x* $\bigwedge y.$ (*P y* $\Longrightarrow$ *f x* $\geq$ *f y*)
shows *is-arg-max f P x*
using *assms* **by** (*auto simp*: *is-arg-max-linorder*)

lemma *is-arg-max-linorderD*[*dest*]: **fixes** *f* :: *'c* $\Rightarrow$ *'b* :: *linorder*
assumes *is-arg-max f P x*
shows *P x* (*P y* $\Longrightarrow$ *f x* $\geq$ *f y*)
using *assms* **by** (*auto simp*: *is-arg-max-linorder*)

lemma *is-arg-max-cong*:
assumes $\bigwedge x.$ *P x* $\Longrightarrow$ *f x* = *g x*
shows *is-arg-max f P x* $\longleftrightarrow$ *is-arg-max g P x*
unfolding *is-arg-max-def* **using** *assms* **by** *auto*

lemma *is-arg-max-cong'*:
assumes $\bigwedge x.$ *P x* $\Longrightarrow$ *f x* = *g x*
shows *is-arg-max f P* = *is-arg-max g P*

using *assms* **by** (*auto cong*: *is-arg-max-cong*)

lemma *is-arg-max-congI*:
assumes *is-arg-max f P x* $\bigwedge x.\ P\ x \Longrightarrow f\ x = g\ x$
shows *is-arg-max g P x*
using *is-arg-max-cong assms* **by** *force*

3.8 Contraction Mappings

definition *is-contraction C* $\equiv \exists l.\ 0 \leq l \wedge l < 1 \wedge (\forall v\ u.\ dist\ (C\ v)\ (C\ u) \leq l * dist\ v\ u)$

lemma *banach'*:
fixes $C :: 'b :: complete\text{-}space \Rightarrow 'b$
assumes *is-contraction C*
shows $\exists!v.\ C\ v = v\ \bigwedge v.\ (\lambda n.\ (C\ \widehat{\ }\widehat{\ }\ n)\ v) \longrightarrow (THE\ v.\ C\ v = v)$
proof −
obtain *v* **where** *C*: $C\ v = v\ \forall v'.\ C\ v' = v' \longrightarrow v' = v$
by (*metis assms is-contraction-def banach-fix-type*)
obtain *l* **where** *cont*: $dist\ (C\ v)\ (C\ u) \leq l * dist\ v\ u\ 0 \leq l\ l < 1$
for *v u*
using *assms is-contraction-def* **by** *blast*
have ∗: $\bigwedge n\ v0.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ n)\ v0)\ v \leq l\ \widehat{\ }\ n * dist\ v0\ v$
proof −
fix *n v0*
show $dist\ ((C\ \widehat{\ }\widehat{\ }\ n)\ v0)\ v \leq l\ \widehat{\ }\ n * dist\ v0\ v$
proof (*induction n*)
case (*Suc n*)
thus $dist\ ((C\ \widehat{\ }\widehat{\ }\ Suc\ n)\ v0)\ v \leq l\ \widehat{\ }\ Suc\ n * dist\ v0\ v$
using ‹$0 \leq l$›
by (*subst C*(*1*)[*symmetric*]) (*auto simp*: *algebra-simps intro*!: *order-trans*[*OF cont*(*1*)] *mult-left-mono*)
qed *simp*
qed
have $(\lambda n.\ l\ \widehat{\ }\ n) \longrightarrow 0$
by (*simp add*: *LIMSEQ-realpow-zero* ‹$0 \leq l$› ‹$l < 1$›)
hence $\bigwedge v0.\ (\lambda n.\ l\ \widehat{\ }\ n * dist\ v0\ v) \longrightarrow 0$
by (*simp add*: *tendsto-mult-left-zero*)
hence $(\lambda n.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ n)\ v0)\ v) \longrightarrow 0$ **for** *v0*
using ∗ *order-trans abs-ge-self*
by (*subst Limits.tendsto-0-le*[*of* $(\lambda n.\ l\ \widehat{\ }\ n * dist\ v0\ v)$ - - *1*])
(*fastforce intro*!: *eventuallyI*)+
hence $\bigwedge v0.\ (\lambda n.\ (C\ \widehat{\ }\widehat{\ }\ n)\ v0) \longrightarrow v$
using *tendsto-dist-iff* **by** *blast*
thus $\bigwedge v0.\ (\lambda n.\ (C\ \widehat{\ }\widehat{\ }\ n)\ v0) \longrightarrow (THE\ v.\ C\ v = v)$
by (*metis* (*mono-tags*, *lifting*) *C theI'*)
next
show $\exists!v.\ C\ v = v$
using *assms banach-fix-type* **unfolding** *is-contraction-def* **by** *blast*

qed

lemma *contraction-dist*:
fixes $C :: 'b :: complete\text{-}space \Rightarrow 'b$
assumes $\bigwedge v\ u.\ dist\ (C\ v)\ (C\ u) \leq c\ *\ dist\ v\ u$
assumes $0 \leq c\ c < 1$
shows $(1 - c) * dist\ v\ (THE\ v.\ C\ v = v) \leq dist\ v\ (C\ v)$
proof −
have *is-contraction C*
unfolding *is-contraction-def* **using** *assms* **by** *auto*
then obtain *v-fix* **where** *v-fix*: $v\text{-}fix = (THE\ v.\ C\ v = v)$
using *the1-equality* **by** *blast*
hence $(\lambda n.\ (C\ \widehat{\ }\widehat{\ }\ n)\ v) \longrightarrow v\text{-}fix$
using *banach'*[*OF* ‹*is-contraction C*›] **by** *simp*
have *dist-contr-le-pow*: $\bigwedge n.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ n)\ v)\ ((C\ \widehat{\ }\widehat{\ }\ Suc\ n)\ v) \leq c\ \widehat{\ }\ n * dist\ v\ (C\ v)$
proof −
fix *n*
show $dist\ ((C\ \widehat{\ }\widehat{\ }\ n)\ v)\ ((C\ \widehat{\ }\widehat{\ }\ Suc\ n)\ v) \leq c\ \widehat{\ }\ n * dist\ v\ (C\ v)$
using *assms*
by (*induction n*) (*auto simp*: *algebra-simps intro*!: *order.trans*[*OF assms*(*1*)] *mult-left-mono*)
qed
have *summable-C*: $summable\ (\lambda i.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ i)\ v)\ ((C\ \widehat{\ }\widehat{\ }\ Suc\ i)\ v))$
using *dist-contr-le-pow assms summable-powser-const*
by (*intro summable-comparison-test*[*of* $(\lambda i.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ i)\ v)\ ((C\ \widehat{\ }\widehat{\ }\ Suc\ i)\ v))\ \lambda i.\ c\widehat{\ }i * dist\ v\ (C\ v)$]) *auto*
have $\forall e > 0.\ dist\ v\ v\text{-}fix \leq (\sum i.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ i)\ v)\ ((C\ \widehat{\ }\widehat{\ }\ (Suc\ i))\ v)) + e$
proof *safe*
fix *e* ::*real* **assume** $0 < e$
have $\forall_F\ n\ in\ sequentially.\ dist\ ((C\widehat{\ }\widehat{\ }n)\ v)\ v\text{-}fix < e$
using ‹$(\lambda n.\ (C\ \widehat{\ }\widehat{\ }\ n)\ v) \longrightarrow v\text{-}fix$› ‹$0 < e$› *tendsto-iff* **by** *force*
then obtain *N* **where** $dist\ ((C\widehat{\ }\widehat{\ }N)\ v)\ v\text{-}fix < e$
by *fastforce*
hence ∗: $dist\ v\ v\text{-}fix \leq dist\ v\ ((C\widehat{\ }\widehat{\ }N)\ v) + e$
by (*metis add-le-cancel-left dist-commute dist-triangle-le less-eq-real-def*)
have $dist\ v\ ((C\widehat{\ }\widehat{\ }N)\ v) \leq (\sum i{\leq}N.\ dist\ ((C\ \widehat{\ }\widehat{\ }\ i)\ v)\ ((C\ \widehat{\ }\widehat{\ }\ (Suc\ i))\ v))$
proof (*induction N arbitrary*: *v*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc N*)
have $dist\ v\ ((C\ \widehat{\ }\widehat{\ }\ Suc\ N)\ v) \leq dist\ v\ (C\ v) + dist\ (C\ v)\ ((C\widehat{\ }\widehat{\ }(Suc\ N))\ v)$

by *metric*
also have ... = *dist v* (*C v*) + *dist* (*C v*) ((*C*^^*N*) (*C v*))
by (*metis funpow-simps-right*(*2*) *o-def*)
also have ... ≤ *dist v* (*C v*) + (∑ *i*≤*N*. *dist* ((*C* ^^ *i*) (*C v*)) ((*C* ^^ *Suc i*) (*C v*)))
using *Suc.IH add-le-cancel-left* **by** *blast*
also have ... ≤ *dist v* (*C v*) + (∑ *i*≤*N*. *dist* ((*C* ^^*Suc i*) *v*) ((*C* ^^ (*Suc* (*Suc i*))) *v*))
by (*simp only*: *funpow-simps-right*(*2*) *o-def*)
also have ... ≤ (∑ *i*≤*Suc N*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ (*Suc i*)) *v*))
by (*subst sum.atMost-Suc-shift*) *simp*
finally show *dist v* ((*C* ^^ *Suc N*) *v*) ≤ (∑ *i*≤*Suc N*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ *Suc i*) *v*)) **.**
qed
moreover have
(∑ *i*≤*N*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ *Suc i*) *v*)) ≤ (∑ *i*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ (*Suc i*)) *v*))
using *summable-C*
by (*auto intro*: *sum-le-suminf*)
ultimately have *dist v* ((*C*^^*N*) *v*) ≤ (∑ *i*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ (*Suc i*)) *v*))
by *linarith*
thus *dist v v-fix* ≤ (∑ *i*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ *Suc i*) *v*)) + *e*
using ∗ **by** *fastforce*
qed
hence *le-suminf*: *dist v v-fix* ≤ (∑ *i*. *dist* ((*C* ^^ *i*) *v*) ((*C* ^^ *Suc i*) *v*))
using *field-le-epsilon* **by** *blast*
have *dist v v-fix* ≤ (∑ *i*. *c* ^ *i* ∗ *dist v* (*C v*))
using *dist-contr-le-pow summable-C assms summable-powser-const*
by (*auto intro*!: *order-trans*[*OF le-suminf*] *suminf-le*)
hence *dist v v-fix* ≤ *dist v* (*C v*) / (*1* − *c*)
using *sum-disc-lim*
by (*metis sum-disc-lim abs-of-nonneg assms*(*2*) *assms*(*3*))
hence (*1* − *c*) ∗ *dist v v-fix* ≤ *dist v* (*C v*)
using *assms*(*3*) *mult.commute pos-le-divide-eq*
by (*metis diff-gt-0-iff-gt*)
thus *?thesis*
using *v-fix* **by** *blast*
qed

3.9 Limits

lemma *tendsto-bfun-sandwich*:
assumes
(*f* :: *nat* ⇒ *'b* ⇒$_b$ *real*) ⟶ *x* (*g* :: *nat* ⇒ *'b* ⇒$_b$ *real*) ⟶ *x*
eventually (λ*n*. *f n* ≤ *h n*) *sequentially eventually* (λ*n*. *h n* ≤ *g n*) *sequentially*

shows $(h :: nat \Rightarrow 'b \Rightarrow_b real) \longrightarrow x$
proof −
have *1*: $(\lambda n.\ dist\ (f\ n)\ (g\ n) + dist\ (g\ n)\ x) \longrightarrow 0$
using *tendsto-dist*[*OF assms*(*1*) *assms*(*2*)] *tendsto-dist-iff assms*
by (*auto intro*!: *tendsto-add-zero*)
have *eventually* $(\lambda n.\ dist\ (h\ n)\ (g\ n) \leq dist\ (f\ n)\ (g\ n))$ *sequentially*

using *assms*(*3*) *assms*(*4*)
proof *eventually-elim*
case (*elim n*)
hence $dist\ (h\ n\ a)\ (g\ n\ a) \leq dist\ (f\ n\ a)\ (g\ n\ a)$ **for** *a*
proof −
have $f\ n\ a \leq h\ n\ a\ h\ n\ a \leq g\ n\ a$
using *elim* **unfolding** *less-eq-bfun-def* **by** *auto*
thus *?thesis*
using *dist-real-def* **by** *fastforce*
qed
thus *?case*
unfolding *dist-bfun.rep-eq*
by (*auto intro*!: *cSUP-mono bounded-imp-bdd-above simp*: *dist-real-def bounded-minus-comp bounded-abs-range*)
qed
moreover have *eventually* $(\lambda n.\ dist\ (h\ n)\ x \leq dist\ (h\ n)\ (g\ n) + dist\ (g\ n)\ x)$ *sequentially*
by (*simp add*: *dist-triangle*)
ultimately have *2*: *eventually* $(\lambda n.\ dist\ (h\ n)\ x \leq dist\ (f\ n)\ (g\ n) + dist\ (g\ n)\ x)$ *sequentially*
using *eventually-elim2* **by** *fastforce*
have $(\lambda n.\ dist\ (h\ n)\ x) \longrightarrow 0$
proof (*subst tendsto-iff*, *safe*)
fix *e* ::*real*
assume $e > 0$
hence *3*: $\forall_F\ xa\ in\ sequentially.\ dist\ (f\ xa)\ (g\ xa) + dist\ (g\ xa)\ x < e$
using *1*
by (*auto simp*: *tendsto-iff*)
show $\forall_F\ xa\ in\ sequentially.\ dist\ (dist\ (h\ xa)\ x)\ 0 < e$
by (*rule eventually-mp*[*OF - 3*]) (*fastforce intro*: *2 eventually-mono*)
qed
thus *?thesis*
using *tendsto-dist-iff*
by *auto*
qed

3.10 Supremum

lemma *SUP-add-le*:
assumes $X \neq \{\}$ *bounded* $(B\ `\ X)$ *bounded* $(A'\ `\ X)$
shows $(\bigsqcup c \in X.\ (B :: 'a \Rightarrow real)\ c + A'\ c) \leq (\bigsqcup b \in X.\ B\ b) +$

$(\bigsqcup a \in X.\ A'\ a)$
using *assms*
by (*auto simp*: *add-mono bounded-has-Sup(1) intro*!: *cSUP-least*)+

lemma *le-SUP-diff'*:
assumes *ne*: $X \neq \{\}$
and *bdd*: *bounded* $(B\ `\ X)$ *bounded* $(A'\ `\ X)$
and *sup-le*: $(\bigsqcup a \in X.\ (A' :: 'a \Rightarrow real)\ a) \leq (\bigsqcup b \in X.\ B\ b)$
shows $(\bigsqcup b \in X.\ B\ b) - (\bigsqcup a \in X.\ (A' :: 'a \Rightarrow real)\ a) \leq (\bigsqcup c \in X.\ B\ c - A'\ c)$
proof −
have *bounded* $((\lambda x.\ (B\ x - A'\ x))\ `\ X)$
using *bdd bounded-minus-comp* **by** *blast*
have $(\bigsqcup b \in X.\ B\ b) - (\bigsqcup a \in X.\ A'\ a) - e \leq (\bigsqcup c \in X.\ B\ c - A'\ c)$ **if** *e*: $e > 0$ **for** *e*
proof −
obtain *z* **where** *z*: $(\bigsqcup b \in X.\ B\ b) - e \leq B\ z\ z \in X$
using *e ne*
by (*subst less-cSupE*[**where** *?y* = $\bigsqcup\ (B\ `\ X) - e$, **where** *?X* = *B`X*]) *fastforce*+
hence $((\bigsqcup a \in X.\ A'\ a) \leq B\ z + e)$
using *sup-le*
by *force*
hence $A'\ z \leq B\ z + e$
using ‹$z \in X$› *bdd bounded-has-Sup(1)* **by** *fastforce*
thus $(\bigsqcup b \in X.\ B\ b) - (\bigsqcup a \in X.\ A'\ a) - e \leq (\bigsqcup c \in X.\ B\ c - A'\ c)$
using ‹*bounded* $((\lambda x.\ B\ x - A'\ x)\ `\ X)$› *z bounded-has-Sup(1)*[*OF bdd(2)*]
by (*subst cSUP-upper2*[**where** $x = z$]) (*fastforce intro*!: *bounded-imp-bdd-above*)+
qed
thus *?thesis*
by (*subst field-le-epsilon*) *fastforce*+
qed

lemma *le-SUP-diff*:
fixes $A' :: 'a \Rightarrow real$
assumes $X \neq \{\}$ *bounded* $(B\ `\ X)$ *bounded* $(A'\ `\ X)$ $(\bigsqcup a \in X.\ A'\ a) \leq (\bigsqcup b \in X.\ B\ b)$
shows $0 \leq (\bigsqcup c \in X.\ B\ c - A'\ c)$
using *assms*
by (*auto intro*!: *order.trans*[*OF - le-SUP-diff'*])

lemma *bounded-SUP-mul*[*simp*]:
$X \neq \{\} \Longrightarrow 0 \leq l \Longrightarrow$ *bounded* $(f\ `\ X) \Longrightarrow (\bigsqcup x \in X.\ (l :: real) * f\ x) = (l * (\bigsqcup x \in X.\ f\ x))$
proof −
assume $X \neq \{\}$ *bounded* $(f\ `\ X)$ $0 \leq l$
have $(\bigsqcup x \in X.\ ereal\ l * ereal\ (f\ x)) = (l * (\bigsqcup x \in X.\ ereal\ (f\ x)))$

by (*simp add*: *Sup-ereal-mult-left'* ‹*0* ≤ *l*› ‹*X* ≠ {}›)
obtain *b* **where** ∀ *a* ∈*X*. |*f a*| ≤ *b*
using ‹*bounded* (*f* ‘ *X*)› *bounded-real* **by** *auto*
have ∀ *a* ∈*X*. |*ereal* (*f a*)| ≤ *b*
by (*simp add*: ‹∀ *a*∈*X*. |*f a*| ≤ *b*›)
hence *sup-leb*: (⨆ *a*∈*X*. |*ereal* (*f a*)|)≤ *b*
by (*simp add*: *SUP-least*)
have (⨆ *a*∈*X*. *ereal* (*f a*)) ≤ (⨆ *a*∈*X*. |*ereal* (*f a*)|)
by (*auto intro*: *Complete-Lattices.SUP-mono'*)
moreover have −(⨆ *a*∈*X*. *ereal* (*f a*)) ≤ (⨆ *a*∈*X*. |*ereal* (*f a*)|)
using ‹*X* ≠ {}›
by (*auto intro*!: *Inf-less-eq cSUP-upper2 simp add*: *ereal-INF-uminus-eq*[*symmetric*])
ultimately have |(⨆ *a*∈*X*. *ereal* (*f a*))| ≤ (⨆ *a*∈*X*. |*ereal* (*f a*)|)
by (*auto intro*: *ereal-abs-leI*)
hence |⨆ *a*∈*X*. *ereal* (*f a*)| ≤ *b*
using *sup-leb* **by** *auto*
hence |⨆ *a*∈*X*. *ereal* (*f a*)| ≠ ∞
by *auto*
hence (⨆ *x* ∈ *X*. *ereal* (*f x*)) = *ereal* (⨆ *x* ∈ *X*. (*f x*))
using *ereal-SUP* **by** *metis*
hence (⨆ *x* ∈ *X*. *ereal* (*l* ∗ *f x*)) = *ereal* (*l* ∗ (⨆ *x* ∈ *X*. *f x*))
using ‹(⨆ *x*∈*X*. *ereal l* ∗ *ereal* (*f x*)) = *ereal l* ∗ (⨆ *x*∈*X*. *ereal* (*f*
x))› **by** *auto*
hence *ereal* (⨆ *x* ∈ *X*. *l* ∗ *f x*) = *ereal* (*l* ∗ (⨆ *x* ∈ *X*. *f x*))
by (*simp add*: *ereal-SUP*)
thus *?thesis*
by *auto*
qed

lemma *abs-cSUP-le*[*intro*]:
X ≠ {} ⟹ *bounded* (*F* ‘ *X*) ⟹ |⨆ *x* ∈ *X*. (*F x*) :: *real*| ≤ (⨆ *x* ∈
X. |*F x*|)
by (*auto intro*!: *cSup-abs-le cSUP-upper2 bounded-imp-bdd-above*
simp: *image-image*[*symmetric*])

4 Least argmax

definition *least-arg-max f P* = (*LEAST x*. *is-arg-max f P x*)

lemma *least-arg-max-prop*: ∃ *x*::′*a*::*wellorder*. *P x* ⟹ *finite* {*x*. *P x*}
⟹ *P* (*least-arg-max* (*f* :: - ⇒ *real*) *P*)
unfolding *least-arg-max-def*
apply (*rule LeastI2-ex*)
using *finite-is-arg-max*[*of* {*x*. *P x*}, **where** *f* = *f*]
by *auto*

lemma *is-arg-max-apply-eq*: *is-arg-max* (*f* :: - ⇒ - :: *linorder*) *P x*
⟹ *is-arg-max f P y* ⟹ *f x* = *f y*
by (*auto simp*: *is-arg-max-def not-less dual-order.eq-iff*)

lemma *least-arg-max-apply*:
assumes *is-arg-max* (*f* :: - ⇒ - :: *linorder*) *P* (*x*::-::*wellorder*)
shows *f* (*least-arg-max f P*) = *f x*
proof −
have *is-arg-max f P* (*least-arg-max f P*)
by (*metis LeastI-ex assms least-arg-max-def*)
thus *?thesis*
using *is-arg-max-apply-eq assms* **by** *metis*
qed

lemma *apply-arg-max-eq-max*: *finite* {*x* . *P x*} ⟹ *is-arg-max* (*f* :: - ⇒ - :: *linorder*) *P x* ⟹ *f x* = *Max* (*f* ' {*x*. *P x*})
by (*auto simp*: *is-arg-max-def intro*!: *Max-eqI*[*symmetric*])

lemma *apply-arg-max-eq-max'*: *finite X*⟹ *is-arg-max* (*f* :: - ⇒ - :: *linorder*) (λ*x*. *x* ∈ *X*) *x* ⟹ (*MAX x* ∈ *X*. *f x*) = *f x*
by (*auto simp*: *is-arg-max-linorder intro*!: *Max-eqI*)

lemma *least-arg-max-is-arg-max*: *P* ≠ {} ⟹ *finite P* ⟹ *is-arg-max f* (λ*x*::-::*wellorder*. *x* ∈ *P*) (*least-arg-max* (*f* :: - ⇒ *real*) (λ*x*. *x* ∈ *P*))
unfolding *least-arg-max-def*
apply (*rule LeastI-ex*)
using *finite-is-arg-max*
by *auto*

lemma *is-arg-max-const*: *is-arg-max* (*f* :: - ⇒ - :: *linorder*) (λ*y*. *y* = *c*) *x* ⟷ *x* = *c*
unfolding *is-arg-max-def*
by *auto*

lemma *least-arg-max-cong'*:
assumes ⋀*x*. *is-arg-max f P x* = *is-arg-max g P x*
shows *least-arg-max f P* = *least-arg-max g P*
unfolding *least-arg-max-def* **using** *assms* **by** *metis*

end

5 Discrete-Time Markov Decision Processes with Arbitrary State Spaces

In this file we construct discrete-time Markov decision processes, e.g. with arbitrary state spaces. Proofs and definitions are adapted from Markov_Models.Discrete_Time_Markov_Process.

theory *MDP-cont*
imports *HOL−Probability.Probability*
begin

lemma *Ionescu-Tulcea-C-eq*:
assumes $\bigwedge i\ h.\ h \in space\ (PiM\ \{0..<i\}\ N) \Longrightarrow P\ i\ h = P'\ i\ h$
assumes *h*: *Ionescu-Tulcea P N Ionescu-Tulcea P' N*
shows *Ionescu-Tulcea.C P N 0 n* ($\lambda x.$ *undefined*) = *Ionescu-Tulcea.C P' N 0 n* ($\lambda x.$ *undefined*)
proof (*induction n*)
case *0*
then show *?case* **using** *h* **by** (*auto simp*: *Ionescu-Tulcea.C-def*)
next
case (*Suc n*)
have *aux*: *space* (*PiM* $\{0..<0 + n\}$ *N*) = *space* (*rec-nat* ($\lambda n.$ *return* (*Pi*$_M$ $\{0..<n\}$ *N*))
($\lambda m\ ma\ n\ \omega.$ *ma n* $\omega \gg=$ *Ionescu-Tulcea.eP P' N* (*n* + *m*)) *n 0* ($\lambda x.$ *undefined*))
using *h*
by (*subst Ionescu-Tulcea.space-C*[*symmetric*, **where** $P = P'$, **where** $x = (\lambda x.$ *undefined*)])
(*auto simp add*: *Ionescu-Tulcea.C-def*)
have $\bigwedge i\ h.\ h \in space\ (PiM\ \{0..<i\}\ N) \Longrightarrow$ *Ionescu-Tulcea.eP P N i h* = *Ionescu-Tulcea.eP P' N i h*
by (*auto simp add*: *Ionescu-Tulcea.eP-def assms*)
then show *?case*
using *Suc.IH h*
using *aux*[*symmetric*]
by (*auto intro*!: *bind-cong simp*: *Ionescu-Tulcea.C-def*)
qed

lemma *Ionescu-Tulcea-CI-eq*:
assumes $\bigwedge i\ h.\ h \in space\ (PiM\ \{0..<i\}\ N) \Longrightarrow P\ i\ h = P'\ i\ h$
assumes *h*: *Ionescu-Tulcea P N Ionescu-Tulcea P' N*
shows *Ionescu-Tulcea.CI P N* = *Ionescu-Tulcea.CI P' N*
proof −
have $\bigwedge J.$ *Ionescu-Tulcea.CI P N J* = *Ionescu-Tulcea.CI P' N J*
unfolding *Ionescu-Tulcea.CI-def*[*OF h*(*1*)] *Ionescu-Tulcea.CI-def*[*OF h*(*2*)]
using *assms*
by (*auto intro*!: *distr-cong Ionescu-Tulcea-C-eq*)
thus *?thesis* **by** *auto*
qed

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

proof (*rule measure-eqI-PiM-infinite*[*OF* ∗ - *A*])
fix *J* :: *nat set* **and** *F′*
assume *J*: *finite J* ⋀*i*. *i* ∈ *J* ⟹ *F′ i* ∈ *sets* (*M i*)

define *n* **where** *n* = (*if J* = {} *then 0 else Max J*)
define *F* **where** *F i* = (*if i* ∈ *J then F′ i else space* (*M i*)) **for** *i*
then have *F*[*simp*, *measurable*]: *F i* ∈ *sets* (*M i*) **for** *i*
using *J* **by** *auto*
have *emb-eq*: *prod-emb UNIV M J* (*Pi*$_E$ *J F′*) = *prod-emb UNIV M* {..*n*} (*Pi*$_E$ {..*n*} *F*)
proof *cases*
assume *J* = {} **then show** *?thesis*
by (*auto simp add*: *n-def F-def*[*abs-def*] *prod-emb-def PiE-def*)
next
assume *J* ≠ {} **then show** *?thesis*
by (*auto simp*: *prod-emb-def PiE-iff F-def n-def less-Suc-eq-le* ‹*finite J*› *split*: *if-split-asm*)
qed

show *emeasure P* (*prod-emb UNIV M J* (*Pi*$_E$ *J F′*)) = *emeasure Q* (*prod-emb UNIV M J* (*Pi*$_E$ *J F′*))
unfolding *emb-eq* **by** (*rule eq*) *fact*
qed

lemma *distr-cong-simp*:
M = *K* ⟹ *sets N* = *sets L* ⟹ (⋀*x*. *x* ∈ *space M* =*simp*=> *f x* = *g x*) ⟹ *distr M N f* = *distr K L g*
unfolding *simp-implies-def* **by** (*rule distr-cong*)

5.1 Definition and Basic Properties

locale *discrete-MDP* =
fixes *Ms* :: *′s measure*
and *Ma* :: *′a measure*
and *A* :: *′s* ⇒ *′a set*
and *K* :: *′s* × *′a* ⇒ *′s measure*

assumes *A-s*: ⋀*s*. *A s* ∈ *sets Ma*

assumes *A-ne*: ⋀*s*. *A s* ≠ {}

assumes *ex-pol*: ∃ *δ* ∈ *Ms* →$_M$ *Ma*. ∀ *s*. *δ s* ∈ *A s*

assumes *K*[*measurable*]: *K* ∈ *Ms* ⨂$_M$ *Ma* →$_M$ *prob-algebra Ms*
begin

lemma *space-prodI*[*intro*]: *x* ∈ *space A′* ⟹ *y* ∈ *space B* ⟹ (*x*,*y*) ∈ *space* (*A′* ⨂$_M$ *B*)
by (*auto simp add*: *space-pair-measure*)

abbreviation $M \equiv Ms \bigotimes_M Ma$
abbreviation $Ma\text{-}A\ s \equiv restrict\text{-}space\ Ma\ (A\ s)$

lemma *space-ma*[*intro*]: $s \in space\ Ms \Longrightarrow a \in space\ Ma \Longrightarrow (s,a) \in space\ M$
by (*simp add*: *space-pair-measure*)

lemma *space-x0*[*simp*]: $x0 \in space\ (prob\text{-}algebra\ Ms) \Longrightarrow space\ x0 = space\ Ms$
by (*metis* (*mono-tags*, *lifting*) *space-prob-algebra mem-Collect-eq sets-eq-imp-space-eq*)

lemma *A-subs-Ma*: $A\ s \subseteq space\ Ma$
by (*simp add*: *A-s sets.sets-into-space*)

lemma *space-Ma-A-subset*: $s \in space\ Ms \Longrightarrow space\ (Ma\text{-}A\ s) \subseteq A\ s$
by (*simp add*: *space-restrict-space*)

lemma *K-restrict* [*measurable*]: $K \in (Ms \bigotimes_M Ma\text{-}A\ s) \rightarrow_M prob\text{-}algebra\ Ms$
by *measurable* (*metis measurable-id measurable-pair-iff measurable-restrict-space2-iff*)

lemma *measurable-K-act*[*measurable*, *intro*]: $s \in space\ Ms \Longrightarrow (\lambda a.\ K\ (s,\ a)) \in Ma \rightarrow_M prob\text{-}algebra\ Ms$
by *measurable*

lemma *measurable-K-st*[*measurable*, *intro*]: $a \in space\ Ma \Longrightarrow (\lambda s.\ K\ (s,\ a)) \in Ms \rightarrow_M prob\text{-}algebra\ Ms$
by *measurable*

lemma *space-K*[*simp*]: $sa \in space\ M \Longrightarrow space\ (K\ sa) = space\ Ms$
using *K* **unfolding** *prob-algebra-def measurable-restrict-space2-iff*
by (*auto dest*: *subprob-measurableD*)

lemma *space-K2*[*simp*]: $s \in space\ Ms \Longrightarrow a \in space\ Ma \Longrightarrow space\ (K\ (s,\ a)) = space\ Ms$
by (*simp add*: *space-pair-measure*)

lemma *space-K-E*: $s' \in space\ (K\ (s,a)) \Longrightarrow s \in space\ Ms \Longrightarrow a \in space\ Ma \Longrightarrow s' \in space\ Ms$
by *auto*

lemma *sets-K*: $sa \in space\ M \Longrightarrow sets\ (K\ sa) = sets\ Ms$
using *K* **unfolding** *prob-algebra-def* **unfolding** *measurable-restrict-space2-iff*
by (*auto dest*: *subprob-measurableD*)

lemma *sets-K'*[*measurable-cong*]: $s \in space\ Ms \Longrightarrow a \in space\ Ma \Longrightarrow sets\ (K\ (s,a)) = sets\ Ms$

by (*simp add*: *sets-K space-pair-measure*)

lemma *prob-space-K*[*intro*]: *sa* ∈ *space M* ⟹ *prob-space* (*K sa*)
using *measurable-space*[*OF K*] **by** (*simp add*: *space-prob-algebra*)

lemma *prob-space-K2*[*intro*]: *s* ∈ *space Ms* ⟹ *a* ∈ *space Ma* ⟹ *prob-space* (*K* (*s*,*a*))
using *prob-space-K* **by** (*simp add*: *space-pair-measure*)

lemma *K-in-space* [*intro*]: *m* ∈ *space M* ⟹ *K m* ∈ *space* (*prob-algebra Ms*)
by (*meson K measurable-space*)

5.2 Policies

type-synonym ('*c*, '*d*) *pol* = *nat* ⇒ ((*nat* ⇒ '*c* × '*d*) × '*c*) ⇒ '*d measure*

abbreviation *H i* ≡ Pi_M {*0*..<*i*} (λ-. *M*)

abbreviation *Hs i* ≡ *H i* $\bigotimes_M$ *Ms*

lemma *space-H1*: *j* < (*i* :: *nat*) ⟹ ω ∈ *space* (*H i*) ⟹ ω *j* ∈ *space M*
using *PiE-def*
by (*auto simp*: *space-PiM*)

lemma *space-case-nat*[*intro*]:
assumes ω ∈ *space* (*H i*) *s* ∈ *space Ms*
shows *case-nat s* (*fst* ∘ ω) *i* ∈ *space Ms*
using *assms*
by (*cases i*) (*auto intro*!: *space-H1 measurable-space*[*OF measurable-fst*])

lemma *undefined-in-H0*: (λ-. *undefined*) ∈ *space* (*H* (*0* :: *nat*))
by *auto*

lemma *sets-K-Suc*[*measurable-cong*]: *h* ∈ *space* (*H* (*Suc n*)) ⟹ *sets* (*K* (*h n*)) = *sets Ms*
using *sets-K space-H1* **by** *blast*

A decision rule is a function from states to distributions over enabled actions.

definition *is-dec0 d* ≡ *d* ∈ *Ms* →$_M$ *prob-algebra Ma*

definition *is-dec* (*t* :: *nat*) *d* ≡ (*d* ∈ *Hs t* →$_M$ *prob-algebra Ma*)

lemma *is-dec0 d* ⟹ *is-dec t* (λ(-,*s*). *d s*)

unfolding *is-dec0-def is-dec-def* **by** *auto*

A policy is a function from histories to valid decision rules.

definition *is-policy* :: ('*s*, '*a*) *pol* ⇒ *bool* **where**
is-policy p ≡ ∀ *i*. *is-dec i* (*p i*)

abbreviation *p0* :: ('*s*, '*a*) *pol* ⇒ '*s* ⇒ '*a measure* **where**
p0 p s ≡ *p* (*0* ::*nat*) (λ-. *undefined*, *s*)

context
fixes *p* **assumes** *p*[*simp*]: *is-policy p*
begin

lemma *is-policyD*[*measurable*]: *p i* ∈ *Hs i* →$_M$ *prob-algebra Ma*
using *p* **unfolding** *is-policy-def is-dec-def* **by** *auto*

lemma *space-policy*[*simp*]: *hs* ∈ *space* (*Hs i*) ⟹ *space* (*p i hs*) = *space Ma*
using *K is-policyD* **unfolding** *prob-algebra-def measurable-restrict-space2-iff*
by (*auto dest*: *subprob-measurableD*)

lemma *space-policy'*[*simp*]: *h* ∈ *space* (*H i*) ⟹ *s* ∈ *space Ms* ⟹ *space* (*p i* (*h*,*s*)) = *space Ma*
using *space-policy*
by (*simp add*: *space-pair-measure*)

lemma *space-policyI*[*intro*]:
assumes *s* ∈ *space Ms h* ∈ *space* (*H i*) *a* ∈ *space Ma*
shows *a* ∈ *space* (*p i* (*h*,*s*))
using *space-policy assms*
by (*auto simp*: *space-pair-measure*)

lemma *sets-policy*[*simp*]: *hs* ∈ *space* (*Hs i*) ⟹ *sets* (*p i hs*) = *sets Ma*
using *p K is-policyD*
unfolding *prob-algebra-def measurable-restrict-space2-iff*
by (*auto dest*: *subprob-measurableD*)

lemma *sets-policy'*[*measurable-cong*, *simp*]:
h ∈ *space* (*H i*) ⟹ *s* ∈ *space Ms* ⟹ *sets* (*p i* (*h*,*s*)) = *sets Ma*
using *sets-policy*
by (*auto simp*: *space-pair-measure*)

lemma *sets-policy''*[*measurable-cong*, *simp*]:
h ∈ *space* ((*Pi*$_M$ {} (λ-. *M*))) ⟹ *s* ∈ *space Ms* ⟹ *sets* (*p 0* (*h*,*s*)) = *sets Ma*
using *sets-policy*
by (*auto simp*: *space-pair-measure*)

lemma *policy-prob-space*: *hs* ∈ *space* (*Hs i*) ⟹ *prob-space* (*p i hs*)
proof −
assume *h*: *hs* ∈ *space* (*Hs i*)
hence *p i hs* ∈ *space* (*prob-algebra Ma*)
using *p* **by** (*auto intro*: *measurable-space*)
thus *prob-space* (*p i hs*)
unfolding *prob-algebra-def* **by** (*simp add*: *space-restrict-space*)
qed

lemma *policy-prob-space′*: *h* ∈ *space* (*H i*) ⟹ *s* ∈ *space Ms* ⟹ *prob-space* (*p i* (*h*,*s*))
using *policy-prob-space* **by** (*simp add*: *space-pair-measure*)

lemma *prob-space-p0*: *x* ∈ *space Ms* ⟹ *prob-space* (*p0 p x*)
by (*simp add*: *policy-prob-space′*)

lemma *p0-sets*[*measurable-cong*]: *x* ∈ *space Ms* ⟹ *sets* (*p 0* (λ-. *undefined*,*x*)) = *sets Ma*
using *subprob-measurableD*(*2*) *measurable-prob-algebraD* **by** *simp*

lemma *space-p0*[*simp*]: *s* ∈ *space Ms* ⟹ *space* (*p0 p s*) = *space Ma*
by *auto*

lemma *return-policy-prob-algebra* [*measurable*]:
h ∈ *space* (*H n*) ⟹ *x* ∈ *space Ms* ⟹ (λ*a*. *return M* (*x*, *a*)) ∈ *p n* (*h*, *x*) →$_M$ *prob-algebra M*
by *measurable*
end

5.3 Successor Policy

To shift the policy by one step, we provide a single state-action pair as history

definition *Suc-policy p sa* = (λ*i* (*h*, *s*). *p* (*Suc i*) (λ*i′*. *case-nat sa h i′*, *s*))

lemma *p-as-Suc-policy*: *p* (*Suc i*) (*h*, *s*) = *Suc-policy p* ((*h 0*)) *i* (λ*i*. *h* (*Suc i*), *s*)
proof −
have ∗: *case-nat* (*f 0*) (λ*i*. *f* (*Suc i*)) = *f* **for** *f*
by (*auto split*: *nat.splits*)
show *?thesis*
unfolding *Suc-policy-def*
by (*auto simp*: ∗)
qed

lemma *is-policy-Suc-policy*[*intro*]:
assumes *s*: *sa* ∈ *space M* **and** *p*: *is-policy p*

shows *is-policy* (*Suc-policy p sa*)
proof −
have ∗: (λ*x*. *case-nat sa* (*fst x*)) ∈ *Pi*$_M$ {*0*..<*i*} (λ-. *M*) ⨂$_M$ *Ms* →$_M$ *Pi*$_M$ {*0*..<*Suc i*} (λ-. *M*) **for** *i*
using *s space-H1*
by (*intro measurable-PiM-single*) (*fastforce simp*: *space-PiM space-pair-measure split*: *nat.splits*)+
have ⋀*i*. *p i* ∈ *Pi*$_M$ {*0*..<*i*} (λ-. *M*) ⨂$_M$ *Ms* →$_M$ *prob-algebra Ma*
using *is-policyD p* **by** *blast*
hence ⋀*i*. *Suc-policy p sa i* ∈ *Pi*$_M$ {*0*..<*i*} (λ-. *M*) ⨂$_M$ *Ms* →$_M$ *prob-algebra Ma*
unfolding *Suc-policy-def*
using ∗
by *measurable*
thus *?thesis* **unfolding** *is-policy-def is-dec-def*
by *blast*
qed

lemma *Suc-policy-measurable-step*[*measurable*]:
assumes *is-policy p*
shows (λ*x*. *Suc-policy p* (*fst* (*fst x*)) *n* (*snd* (*fst x*), *snd x*)) ∈
(*M* ⨂$_M$ *Pi*$_M$ {*0*..<*n*} (λ-. *M*)) ⨂$_M$ *Ms* →$_M$ *prob-algebra Ma*
unfolding *Suc-policy-def*
using *assms*
by *measurable* (*auto*
intro: *measurable-PiM-single′*
simp: *space-pair-measure PiE-iff space-PiM extensional-def*
split: *nat.split*)

5.4 Single-Step Distribution

K′ takes a policy, a distribution over 's, the epoch, and a history, produces a distribution over the next state-action pair.

definition *K′* :: ('*s*, '*a*) *pol* ⇒ '*s measure* ⇒ *nat* ⇒ (*nat* ⇒ ('*s* × '*a*)) ⇒ ('*s* × '*a*) *measure*
where
K′ p s0 n ω = *do* {
s ← *case-nat s0* (*K* ∘ *ω*) *n*;
a ← *p n* (*ω*, *s*);
return M (*s*, *a*)
}

lemma *prob-space-K′*:
assumes *p*: *is-policy p* **and** *x*: *x0* ∈ *space* (*prob-algebra Ms*) **and** *h*: *h* ∈ *space* (*H n*)
shows *prob-space* (*K′ p x0 n h*)
unfolding *K′-def*
proof (*intro prob-space.prob-space-bind*[**where** *S* = *M*])
show *prob-space* (*case n of 0* ⇒ *x0* | *Suc x* ⇒ (*K* ∘ *h*) *x*)

```
  using x h space-H1 by (auto split: nat.splits simp: space-prob-algebra)
next
  show AE x in case n of 0 ⇒ x0 | Suc x ⇒ (K ∘ h) x.
    prob-space (p n (h, x) ⤜ (λa. return M (x, a)))
  proof (cases n)
    case 0
    then have h': h ∈ space (Pi_M {0..<0} (λ-. M))
      using h by auto
    show ?thesis
      using 0 p h x sets-policy'
      by (fastforce intro: prob-space.prob-space-bind[where S=M]
          policy-prob-space prob-space-return
          cong: measurable-cong-sets)
  next
    case (Suc nat)
    then show ?thesis
   proof (intro AE-I2 prob-space.prob-space-bind[of - - M], goal-cases)
      case (1 x)
      then show ?case
        using p space-H1 h x
        by (fastforce intro!: policy-prob-space)
    next
      case (2 x a)
      then show ?case
        using h p space-H1
        by (fastforce intro!: prob-space-return)
    next
      case (3 x)
      then show ?case
        using p h x space-K space-H1
     by (fastforce intro!: measurable-prob-algebraD return-policy-prob-algebra)
    qed
  qed
next
  show (λs. p n (h, s) ⤜ (λa. return M (s, a))) ∈
    (case n of 0 ⇒ x0 | Suc x ⇒ (K ∘ h) x) →_M subprob-algebra M
  proof (intro measurable-bind[where N = Ma])
    show  (λx. p n (h, x)) ∈ (case n of 0 ⇒ x0 | Suc x ⇒ (K ∘ h) x)
→_M subprob-algebra Ma
      using p h x
      by (auto split: nat.splits intro!: measurable-prob-algebraD simp:
space-prob-algebra)
  next
    show (λs. return M (fst s, snd s)) ∈
      (case n of 0 ⇒ x0 | Suc x ⇒ (K ∘ h) x) ⨂_M Ma →_M sub-
prob-algebra M
      using h x sets-K-Suc
      by (auto split: nat.splits simp: sets-K space-prob-algebra cong:
measurable-cong-sets)
```

qed
qed

lemma *measurable-K'*[*measurable*]:
assumes *p*: *is-policy p* **and** *x*: $x \in$ *space* (*prob-algebra Ms*)
shows $K'\ p\ x\ i \in H\ i \to_M$ *prob-algebra M*
proof −
fix *i*
show $K'\ p\ x\ i \in Pi_M\ \{0..<i\}\ (\lambda\text{-}.\ M) \to_M$ *prob-algebra M*
unfolding *K'-def*
proof (*intro measurable-bind-prob-space2*[**where** $N = Ms$])
show ($\lambda a.$ *case i of 0* $\Rightarrow x \mid$ *Suc x* $\Rightarrow (K \circ a)\ x) \in Pi_M\ \{0..<i\}$ $(\lambda\text{-}.\ M) \to_M$ *prob-algebra Ms*
using *x* **by** *measurable auto*
next
show ($\lambda(\omega,\ y).\ p\ i\ (\omega,\ y) \gg\!= (\lambda a.$ *return M* $(y,\ a))) \in$
$Pi_M\ \{0..<i\}\ (\lambda\text{-}.\ M) \bigotimes_M Ms \to_M$ *prob-algebra M*
using *x p* **by** *auto*
qed
qed

5.5 Initial State-Action Distribution

K0 produces the initial state-action distribution from a state distribution and a policy.

definition $K0\ p\ s0 = K'\ p\ s0\ 0\ (\lambda\text{-}.$ *undefined*)

lemma *K0-def'*:
$K0\ p\ s0 =$ *do* {
$s \leftarrow s0$;
$a \leftarrow p0\ p\ s$;
return M $(s,\ a)$}
unfolding *K0-def K'-def* **by** *auto*

lemma *K0-prob*[*measurable*]: *is-policy p* $\Longrightarrow$ $K0\ p \in$ *prob-algebra Ms* $\to_M$ *prob-algebra M*
unfolding *K0-def'*
by *measurable*

lemma *prob-space-K0*: *is-policy p* $\Longrightarrow$ *x0* $\in$ *space* (*prob-algebra Ms*) $\Longrightarrow$ *prob-space* (*K0 p x0*)
by (*simp add*: *K0-def prob-space-K'*)

lemma *space-K0*[*simp*]: *is-policy p* $\Longrightarrow$ $s \in$ *space* (*prob-algebra Ms*) $\Longrightarrow$ *space* $(K0\ p\ s) =$ *space M*
by (*meson K0-prob measurable-prob-algebraD sets-eq-imp-space-eq sets-kernel*)

lemma *sets-K0*[*measurable-cong*]:

assumes *is-policy p s* ∈ *space* (*prob-algebra Ms*)
shows *sets* (*K0 p s*) = *sets M*
using *assms* **by** (*meson K0-prob measurable-prob-algebraD sub-prob-measurableD*(*2*))

lemma *K0-return-eq-p0*:
assumes *is-policy p s* ∈ *space Ms*
shows *K0 p* (*return Ms s*) = *p0 p s* ⤜ (λ*a*. *return M* (*s*,*a*))
unfolding *K0-def′*
using *assms*
by (*subst bind-return*[**where** *N* = *M*]) (*auto intro*!: *measurable-prob-algebraD*)

lemma *M-ne-policy*[*intro*]: *is-policy p* ⟹ *s* ∈ *space* (*prob-algebra Ms*) ⟹ *space M* ≠ {}
using *space-K0 prob-space.not-empty prob-space-K0*
by *force*

5.6 Sequence Space of the MDP

We can instantiate *Ionescu-Tulcea* with *K′*.

lemma *IT-K′*: *is-policy p* ⟹ *x* ∈ *space* (*prob-algebra Ms*) ⟹ *Ionescu-Tulcea* (*K′ p x*) (λ-. *M*)
unfolding *Ionescu-Tulcea-def* **using** *measurable-K′ prob-space-K′*
by (*fast dest*: *measurable-prob-algebraD*)

definition *lim-sequence* :: (′*s*, ′*a*) *pol* ⇒ ′*s measure* ⇒ (*nat* ⇒ (′*s* × ′*a*)) *measure*
where
lim-sequence p x = *projective-family.lim UNIV* (*Ionescu-Tulcea.CI* (*K′ p x*) (λ-. *M*)) (λ-. *M*)

lemma
assumes *x*: *x* ∈ *space* (*prob-algebra Ms*) **and** *p*: *is-policy p*
shows *space-lim-sequence*: *space* (*lim-sequence p x*) = *space* (Π$_M$ *i*∈*UNIV*. *M*)
and *sets-lim-sequence*[*measurable-cong*]: *sets* (*lim-sequence p x*) = *sets* (Π$_M$ *i*∈*UNIV*. *M*)
and *emeasure-lim-sequence-emb*: ⋀*J X*. *finite J* ⟹ *X* ∈ *sets* (Π$_M$ *j*∈*J*. *M*) ⟹
emeasure (*lim-sequence p x*) (*prod-emb UNIV* (λ-. *M*) *J X*) =
emeasure (*Ionescu-Tulcea.CI* (*K′ p x*) (λ-. *M*) *J*) *X*
and *emeasure-lim-sequence-emb-I0o*: ⋀*n X*. *X* ∈ *sets* (Π$_M$ *i* ∈ {*0*..<*n*}. *M*) ⟹
emeasure (*lim-sequence p x*) (*prod-emb UNIV* (λ-. *M*) {*0*..<*n*} *X*) =
emeasure (*Ionescu-Tulcea.C* (*K′ p x*) (λ-. *M*) *0 n* (λ*x*. *undefined*)) *X*
proof −
interpret *Ionescu-Tulcea K′ p x* λ-. *M*

```
    using IT-K'[OF p x] .
  show space (lim-sequence p x) = space (Π_M i∈UNIV. M)
    unfolding lim-sequence-def by simp
  show sets (lim-sequence p x) = sets (Π_M i∈UNIV. M)
    unfolding lim-sequence-def by simp

  { fix J :: nat set and X assume finite J X ∈ sets (Π_M j∈J. M)
    then show emeasure (lim-sequence p x) (PF.emb UNIV J X) =
emeasure (CI J) X
      unfolding lim-sequence-def by (rule lim) }
  note emb = this

  have up-to-I0o[simp]: up-to {0..<n} = n for n
    unfolding up-to-def by (rule Least-equality) auto

  { fix n :: nat and X assume X ∈ sets (Π_M j∈{0..<n}. M)
    thus emeasure (lim-sequence p x) (PF.emb UNIV {0..<n} X) =
emeasure (C 0 n (λx. undefined)) X
      by (simp add: space-C emb CI-def space-PiM distr-id2 sets-C
cong: distr-cong-simp) }
qed

lemma lim-sequence-prob-space:
  assumes is-policy p s ∈ space (prob-algebra Ms)
  shows prob-space (lim-sequence p s)
  using assms proof −
  assume p: is-policy p
  fix s assume [simp]: s ∈ space (prob-algebra Ms)
  interpret Ionescu-Tulcea K' p s λ-. M
    using IT-K' p by simp
  have sp:
    space (lim-sequence p s) = prod-emb UNIV (λ-. M) {} (Π_E j∈{}.
space M)
    space (CI {}) = {} →_E space M
   by (auto simp: p space-lim-sequence space-PiM prod-emb-def PF.space-P)
  show prob-space (lim-sequence p s)
    using PF.prob-space-P[THEN prob-space.emeasure-space-1, of {}]
   by (auto intro!: prob-spaceI simp add: p sp emeasure-lim-sequence-emb
simp del: PiE-empty-domain)
qed
```

5.7 Measurablility of the Sequence Space

```
lemma lim-sequence[measurable]:
  assumes p: is-policy p
   shows lim-sequence p ∈ prob-algebra Ms →_M prob-algebra (Π_M
i∈UNIV. M)
proof (intro measurable-prob-algebra-generated[OF sets-PiM Int-stable-prod-algebra
```

prod-algebra-sets-into-space])
show $\bigwedge a.\ a \in space\ (prob\text{-}algebra\ Ms) \Longrightarrow prob\text{-}space\ (lim\text{-}sequence\ p\ a)$
using *lim-sequence-prob-space p* **by** *blast*
next
fix *a* **assume** [*simp*]: $a \in space\ (prob\text{-}algebra\ Ms)$
show $sets\ (lim\text{-}sequence\ p\ a) = sets\ (Pi_M\ UNIV\ (\lambda i.\ M))$
by (*simp add*: *p sets-lim-sequence*)
next
fix $X :: (nat \Rightarrow 's \times 'a)\ set$ **assume** $X \in prod\text{-}algebra\ UNIV\ (\lambda i.\ M)$
then obtain $J :: nat\ set$ **and** F **where** J: $J \neq \{\}$ *finite J* $F \in J \to sets\ M$
and X: $X = prod\text{-}emb\ UNIV\ (\lambda\text{-}.\ M)\ J\ (Pi_E\ J\ F)$
unfolding *prod-algebra-def* **by** *auto*
then have *Pi-F*: *finite J* $Pi_E\ J\ F \in sets\ (Pi_M\ J\ (\lambda\text{-}.\ M))$
by (*auto intro*: *sets-PiM-I-finite*)

define n **where** $n = (LEAST\ n.\ \forall i{\geq}n.\ i \notin J)$
have *J-le-n*: $J \subseteq \{0..{<}n\}$
proof −
have $\bigwedge x.\ x \in J \Longrightarrow \forall i{\geq}Suc\ (Max\ J).\ i \notin J$
using *not-le Max-less-iff*[*OF* ‹*finite J*›]
by (*auto simp*: *Suc-le-eq*)
moreover have $x \in J \Longrightarrow \forall i{\geq}a.\ i \notin J \Longrightarrow x < a$ **for** *x a*
using *not-le* **by** *auto*
ultimately show *?thesis*
unfolding *n-def*
by (*fastforce intro*!: *LeastI2*[*of* $\lambda n.\ \forall i{\geq}n.\ i \notin J\ Suc\ (Max\ J)\ \lambda x.\ \text{-} < x$])
qed

have *C*: $(\lambda x.\ Ionescu\text{-}Tulcea.C\ (K'\ p\ x)\ (\lambda\text{-}.\ M)\ 0\ n\ (\lambda x.\ undefined)) \in prob\text{-}algebra\ Ms \to_M subprob\text{-}algebra\ (Pi_M\ \{0..{<}n\}\ (\lambda\text{-}.\ M))$
proof (*induction n*)
case *0*
thus *?case*
by (*auto simp*: *measurable-cong*[*OF Ionescu-Tulcea.C.simps*(*1*)[*OF IT-K'*, *OF p*]])
next
case (*Suc n*)
have $(\lambda w.\ Ionescu\text{-}Tulcea.eP\ (K'\ p\ (fst\ w))\ (\lambda\text{-}.\ M)\ n\ (snd\ w)) \in prob\text{-}algebra\ Ms \bigotimes_M Pi_M\ \{0..{<}n\}\ (\lambda\text{-}.\ M) \to_M subprob\text{-}algebra\ (Pi_M\ \{0..{<}Suc\ n\}\ (\lambda\text{-}.\ M))$
proof (*subst measurable-cong*)
fix *w* **assume** $w \in space\ (prob\text{-}algebra\ Ms \bigotimes_M Pi_M\ \{0..{<}n\}\ (\lambda\text{-}.\ M))$
then show $Ionescu\text{-}Tulcea.eP\ (K'\ p\ (fst\ w))\ (\lambda\text{-}.\ M)\ n\ (snd\ w) =$

distr (*K′ p* (*fst w*) *n* (*snd w*)) (Π$_M$ *i*∈{*0..<Suc n*}. *M*) (*fun-upd* (*snd w*) *n*)
by (*auto simp*: *p space-pair-measure Ionescu-Tulcea.eP-def*[*OF IT-K′*] *split*: *prod.split*)
next
show (λ*w*. *distr* (*K′ p* (*fst w*) *n* (*snd w*)) (Π$_M$ *i*∈{*0..<Suc n*}. *M*) (*fun-upd* (*snd w*) *n*))
∈ *prob-algebra Ms* ⨂$_M$ *Pi$_M$* {*0..<n*} (λ-. *M*) →$_M$ *subprob-algebra* (*Pi$_M$* {*0..<Suc n*} (λ-. *M*))
proof (*rule measurable-distr2*[**where** *M* = *M*])
show (λ(*x*, *y*). (*snd x*)(*n* := *y*)) ∈ (*prob-algebra Ms* ⨂$_M$ *Pi$_M$* {*0..<n*} (λ-. *M*)) ⨂$_M$ *M* →$_M$ *Pi$_M$* {*0..<Suc n*} (λ*i*. *M*)
proof (*rule measurable-PiM-single′*)
fix *i* **assume** *i* ∈ {*0..<Suc n*}
then show (λω. (*case* ω *of* (*x*, *y*) ⇒ (*snd x*)(*n* := *y*)) *i*) ∈ (*prob-algebra Ms* ⨂$_M$ *Pi$_M$* {*0..<n*} (λ-. *M*)) ⨂$_M$ *M* →$_M$ *M*
unfolding *split-beta′* **by** (*cases i* = *n*) *auto*
next
show (λω. *case* ω *of* (*x*, *y*) ⇒ (*snd x*)(*n* := *y*)) ∈ *space* ((*prob-algebra Ms* ⨂$_M$ *Pi$_M$* {*0..<n*} (λ-. *M*)) ⨂$_M$ *M*) → {*0..<Suc n*} →$_E$ *space M*
by (*auto simp*: *space-pair-measure space-PiM less-Suc-eq PiE-iff*)
qed
next
show (λ*x*. *K′ p* (*fst x*) *n* (*snd x*)) ∈ *prob-algebra Ms* ⨂$_M$ *Pi$_M$* {*0..<n*} (λ-. *M*) →$_M$ *subprob-algebra M*
unfolding *K′-def comp-def*
using *p*
by (*auto intro*!: *measurable-prob-algebraD*)
qed
qed
then show *?case*
using *Suc.IH*
by (*subst measurable-cong*[*OF Ionescu-Tulcea.C.simps*(*2*)[*OF IT-K′*, **where** *p1* = *p*, *OF p*]])
(*auto intro*!: *measurable-bind*)
qed
have ∗: (λ*x*. *Ionescu-Tulcea.CI* (*K′ p x*) (λ-. *M*) *J*) ∈ *prob-algebra Ms* →$_M$ *subprob-algebra* (*Pi$_M$ J* (λ-. *M*))
using *measurable-distr*[*OF measurable-restrict-subset*[*OF J-le-n*], *of* (λ-. *M*)] *C p*
by (*subst measurable-cong*)
(*auto simp*: *Ionescu-Tulcea.up-to-def*[*OF IT-K′*] *n-def Ionescu-Tulcea.CI-def*[*OF IT-K′*])
have (λ*a*. *emeasure* (*lim-sequence p a*) *X*) ∈ *borel-measurable* (*prob-algebra Ms*) ⟷
(λ*a*. *emeasure* (*Ionescu-Tulcea.CI* (*K′ p a*) (λ-. *M*) *J*) (*Pi$_E$ J F*)) ∈

borel-measurable (*prob-algebra Ms*)
unfolding *X* **using** *J Pi-F* **by** (*intro p measurable-cong emeasure-lim-sequence-emb*) *auto*
also have ...
using ∗ *measurable-emeasure-subprob-algebra Pi-F*(*2*) **by** *auto*
finally show (λ*a*. *emeasure* (*lim-sequence p a*) *X*) ∈ *borel-measurable* (*prob-algebra Ms*) **.**
qed

lemma *lim-sequence-aux*[*measurable*]:
assumes *p*: *is-policy p*
assumes *f* : ⋀*x*. *x* ∈ *space M* ⟹ *is-policy* (*f x*)
assumes *f'*: ⋀*n*. (λ*x*. *f* (*fst* (*fst x*)) *n* (*snd* (*fst x*), *snd x*)) ∈ (*M* ⨂*M* *Pi*$_M$ {*0*..<*n*} (λ-. *M*)) ⨂*M* *Ms* →*M* *prob-algebra Ma*
assumes *gm*: *g* ∈ *M* →*M* *prob-algebra Ms*
shows (λ*x*. *lim-sequence* (*f x*) (*g x*)) ∈ *M* →*M* *prob-algebra* (*Pi*$_M$ *UNIV* (λ-. *M*))
proof (*intro measurable-prob-algebra-generated*[*OF sets-PiM Int-stable-prod-algebra prod-algebra-sets-into-space*])
fix *a* **assume** *a*[*simp*]: *a* ∈ *space M*
have *g*: ⋀*x*. *x* ∈ *space M* ⟹ *g x* ∈ *space* (*prob-algebra Ms*)
by (*meson gm measurable-space*)
interpret *Ionescu-Tulcea K'* (*f a*) (*g a*) λ-. *M*
using *IT-K' p*
using *f*[*OF* ‹*a* ∈ *space M*›] *g* **by** *measurable*
have *p'*: *is-policy* (*f a*)
using ‹*a* ∈ *space M*› *p f* **by** *auto*
have *sp*:
space (*lim-sequence* (*f a*) (*g a*)) = *prod-emb UNIV* (λ-. *M*) {} (Π$_E$ *j*∈{}. *space M*)
space (*CI* {}) = {} →$_E$ *space M*
using *g a p'* **by** (*auto simp*: *space-lim-sequence p' space-PiM prod-emb-def PF.space-P*)
have *emeasure* (*lim-sequence* (*f a*) (*g a*)) (*space* (*lim-sequence* (*f a*) (*g a*))) = *1*
unfolding *sp*
using *g a p' sets.top*[*of* (*Pi*$_M$ {} (λ-. *M*)), *unfolded space-PiM-empty*]

PF.prob-space-P[*THEN prob-space.emeasure-space-1*, *of* {}]
by (*subst emeasure-lim-sequence-emb*) (*auto simp*: *emeasure-lim-sequence-emb sp*)
thus *prob-space* (*lim-sequence* (*f a*) (*g a*))
by (*auto intro*: *prob-spaceI*)
show *sets* (*lim-sequence* (*f a*) (*g a*)) = *sets* (*Pi*$_M$ *UNIV* (λ*i*. *M*))
by (*simp add*: *lim-sequence-def*)
next
fix *X* :: (*nat* ⇒ *'s* × *'a*) *set* **assume** *X* ∈ *prod-algebra UNIV* (λ*i*. *M*)
then obtain *J* :: *nat set* **and** *F* **where** *J*: *J* ≠ {} *finite J F* ∈ *J*

$\rightarrow$ *sets M*
and *X*: *X* = *prod-emb UNIV* (λ-. *M*) *J* (*Pi*$_E$ *J F*)
unfolding *prod-algebra-def* **by** *auto*
then have *Pi-F*: *finite J Pi*$_E$ *J F* ∈ *sets* (*Pi*$_M$ *J* (λ-. *M*))
by (*auto intro*: *sets-PiM-I-finite*)

define *n* **where** *n* = (*LEAST n*. ∀ *i*≥*n*. *i* ∉ *J*)
have *J-le-n*: *J* ⊆ {*0*..<*n*}
unfolding *n-def*
by (*rule LeastI2*[*of* - *Suc* (*Max J*)]) (*auto simp*: ‹*finite J*› *Suc-le-eq not-le*[*symmetric*])

have *g*: ⋀*x*. *x* ∈ *space M* ⟹ *g x* ∈ *space* (*prob-algebra Ms*)
by (*meson gm measurable-space*)

have *C*: (λ*x*. *Ionescu-Tulcea.C* (*K'* (*f x*) (*g x*)) (λ-. *M*) *0 n* (λ*x*. *undefined*)) ∈
M →$_M$ *subprob-algebra* (*Pi*$_M$ {*0*..<*n*} (λ-. *M*))
proof (*induction n*)
case *0*
then show *?case*
using *g f*
by (*auto simp*: *measurable-cong*[*OF Ionescu-Tulcea.C.simps*(*1*)[*OF IT-K'*]])
next
case (*Suc n*)
then show *?case*
proof (*subst measurable-cong*[*OF Ionescu-Tulcea.C.simps*(*2*)[*OF IT-K'*]])
show (λ*w*. *Ionescu-Tulcea.C* (*K'* (*f w*) (*g w*)) (λ-. *M*) *0 n* (λ*x*. *undefined*) ⤜ *Ionescu-Tulcea.eP* (*K'* (*f w*) (*g w*)) (λ-. *M*) (*0* + *n*))
∈ *M* →$_M$ *subprob-algebra* (*Pi*$_M$ {*0*..<*Suc n*} (λ-. *M*))
if *h*: (λ*x*. *Ionescu-Tulcea.C* (*K'* (*f x*) (*g x*)) (λ-. *M*) *0 n* (λ*x*. *undefined*)) ∈ *M* →$_M$ *subprob-algebra* (*Pi*$_M$ {*0*..<*n*} (λ-. *M*))
proof (*rule measurable-bind'*[*OF h*])
show (λ(*x*, *y*). *Ionescu-Tulcea.eP* (*K'* (*f x*) (*g x*)) (λ-. *M*) (*0* + *n*) *y*) ∈ *M* ⨂$_M$ *Pi*$_M$ {*0*..<*n*} (λ-. *M*) →$_M$ *subprob-algebra* (*Pi*$_M$ {*0*..<*Suc n*} (λ-. *M*))
proof (*subst measurable-cong*)
fix *n* :: *nat* **and** *w* **assume** *w* ∈ *space* (*M* ⨂$_M$ *Pi*$_M$ {*0*..<*n*} (λ-. *M*))
then show (*case w of* (*x*, *a*) ⇒ *Ionescu-Tulcea.eP* (*K'* (*f x*) (*g x*)) (λ-. *M*) (*0* + *n*) *a*) =
(*case w of* (*x*, *a*) ⇒ *distr* (*K'* (*f x*) (*g x*) *n a*) (Π$_M$ *i*∈{*0*..<*Suc n*}. *M*) (*fun-upd a n*))
by (*auto simp*: *IT-K' Ionescu-Tulcea.eP-def f g space-ma p space-pair-measure*
Ionescu-Tulcea.eP-def[*OF IT-K'*])
next

```
        fix n
        have (λw. distr (K' (f (fst w)) (g (fst w)) n (snd w)) (Pi_M
{0..<Suc n} (λi. M)) (fun-upd (snd w) n))
          ∈ M ⨂_M Pi_M {0..<n} (λ-. M) →_M subprob-algebra (Pi_M
{0..<Suc n} (λ-. M))
            proof (intro measurable-distr2[where M=M] measur-
able-PiM-single', goal-cases)
          case (1 i)
          then show ?case
            by (cases i = n) (auto simp: split-beta')
        next
          case 2
          then show ?case
              by (auto simp: split-beta' PiE-iff extensional-def Pi-iff
space-pair-measure space-PiM)
        next
          case 3
          then show ?case
            unfolding K'-def
          proof (intro measurable-bind[where N = Ms], goal-cases)
            case 1
            then show ?case
              unfolding measurable-pair-swap-iff[of - M]
               by measurable (auto simp: gm  measurable-snd'' intro:
measurable-prob-algebraD)
          next
            case 2
            then show ?case
              unfolding Suc-policy-def
              using f'
                  by (auto intro!: measurable-bind[where N = Ma]
measurable-prob-algebraD)
          qed
        qed
          thus (λw. case w of (x, a) ⇒ distr ((K' (f x) (g x)) n a)
(Pi_M {0..<Suc n} (λi. M)) (fun-upd a n)) ∈ M ⨂_M Pi_M {0..<n}
(λ-. M) →_M subprob-algebra (Pi_M {0..<Suc n} (λ-. M))
          by measurable
      qed
    qed
  qed (auto simp: f g)
 qed

 have p': ⋀a. a ∈ space M ⟹ is-policy (f a)
   using f by auto
 have (λa. emeasure (lim-sequence (f a) (g a)) X) ∈ borel-measurable
M ⟷
   (λa. emeasure (Ionescu-Tulcea.CI (K' (f a) (g a)) (λ-. M) J) (Pi_E
J F)) ∈ borel-measurable M
```

unfolding *X* **using** *J Pi-F*
by (*fastforce simp add*: *g f K space-pair-measure intro*!: *p p′ measurable-cong emeasure-lim-sequence-emb*)
also have ...
proof (*intro measurable-compose*[*OF - measurable-emeasure-subprob-algebra*[*OF Pi-F*(*2*)]],
subst measurable-cong[**where** *g* = (λ*w*. *distr* (*Ionescu-Tulcea.C* (*K′* (*f w*) (*g w*))
(λ-. *M*) *0 n* (λ*x*. *undefined*)) (*Pi*$_M$ *J* (λ-. *M*)) (λ*f*. *restrict f J*))], *goal-cases*)
case (*1 w*)
then show *?case*
unfolding *Ionescu-Tulcea.CI-def*[*OF IT-K′*[*OF f*[*OF 1*] *g*[*OF 1*]]]
using *p*
by (*subst Ionescu-Tulcea.up-to-def*[*OF IT-K′*[*of Suc-policy p w K w*]])
(*auto simp add*: *n-def* ‹*w* ∈ *space M*› *prob-space-K sets-K space-prob-algebra*)
next
case *2*
then show *?case*
using *measurable-compose measurable-distr*[*OF measurable-restrict-subset*[*OF J-le-n*]] *C*
by *blast*
qed
thus (λ*a*. *emeasure* (*lim-sequence* (*f a*) (*g a*)) *X*) ∈ *borel-measurable M*
using *calculation* **by** *blast*
qed

lemma *lim-sequence-Suc-return*[*measurable*]:
assumes *p*: *is-policy p*
assumes *s*: *s* ∈ *space Ms*
shows (λ*x*. *lim-sequence* (*Suc-policy p* (*s*, *snd x*)) (*return Ms* (*fst x*))) ∈
M →$_M$ *prob-algebra* (*Pi*$_M$ *UNIV* (λ-. *M*))
proof (*intro lim-sequence-aux*[*OF p*], *goal-cases*)
case (*1 x*)
then show *?case*
by (*meson is-policy-Suc-policy measurable-snd measurable-space p s space-ma*)
next
case (*2 n*)
then show *?case*
using *p*
unfolding *Suc-policy-def*
by *measurable* (*auto intro*: *measurable-PiM-single′*
simp: *s space-pair-measure space-PiM PiE-iff extensional-def split*: *nat.split*)

qed *measurable*

lemma *lim-sequence-Suc-K*[*measurable*]:
assumes *is-policy p*
shows (λ*x*. *lim-sequence* (*Suc-policy p x*) (*K x*)) ∈ *M* →$_M$ *prob-algebra* (*Pi*$_M$ *UNIV* (λ-. *M*))
using *assms*
by (*fastforce intro*!: *lim-sequence-aux*)

5.8 Iteration Rule

lemma *step-C*:
assumes *x*: *x* ∈ *space* (*prob-algebra Ms*) **and** *p*: *is-policy p*
shows *Ionescu-Tulcea.C* (*K*′ *p x*) (λ-. *M*) *0 1* (λ-. *undefined*) ⤜
Ionescu-Tulcea.C (*K*′ *p x*) (λ-. *M*) *1 n* =
K0 p x ⤜ (λ*a*. *Ionescu-Tulcea.C* (*K*′ *p x*) (λ-. *M*) *1 n* (*case-nat a* (λ-. *undefined*)))
proof −
interpret *Ionescu-Tulcea K*′ *p x* λ-. *M*
using *IT-K*′[*OF p x*] **.**

have [*simp*]: *space* (*K0 p x*) ≠ {}
using *space-K0*[*OF p x*] *x* **by** *auto*

have [*simp*]: ((λ-. *undefined*)(*0* := *x*::(′*s* × ′*a*))) = *case-nat x* (λ-. *undefined*) **for** *x*
by (*auto simp*: *fun-eq-iff split*: *nat.split*)

have *C 0 1* (λ-. *undefined*) ⤜ *C 1 n* = *eP 0* (λ-. *undefined*) ⤜ *C 1 n*
using *measurable-eP*[*of 0*] *measurable-C*[*of 1 n*, *measurable del*]
by (*simp add*: *bind-return*[**where** *N*=*Pi*$_M$ {*0*} (λ-. *M*)])
also have ... = *K0 p x* ⤜ (λ*a*. *C 1 n* (*case-nat a* (λ-. *undefined*)))
unfolding *eP-def*
proof (*subst bind-distr*[**where** *K* = *Pi*$_M$ {*0*..<*Suc n*} (λ-. *M*)], *goal-cases*)
case *1*
then show *?case*
using *measurable-C*[*of 1 n*, *measurable del*] *x*[*THEN sets-K0*[*OF p*]] *measurable-ident-sets*[*OF sets-P*]
unfolding *K0-def*
by *auto*
next
case *2*
then show *?case*
using *measurable-C*[*of 1 n*]
by *auto*
next
case *3*

then show *?case*
by (*simp add*: *space-P*)
next
case *4*
then show *?case*
unfolding *K0-def*
by (*auto intro*!: *bind-cong*)
qed
finally show
C 0 1 (λ-. *undefined*) ⤜ *C 1 n* = *K0 p x* ⤜ (λ*a*. *C 1 n* (*case-nat a* (λ-. *undefined*))) **.**
qed

lemma *lim-sequence-eq*:
assumes *x*: *x* ∈ *space* (*prob-algebra Ms*) **assumes** *p*: *is-policy p*
shows *lim-sequence p x* =
K0 p x ⤜ (λ*y*. *distr* (*lim-sequence* (*Suc-policy p y*) (*K y*)) (Π$_M$ -∈*UNIV*. *M*) (*case-nat y*))
(**is** - = *?B p x*)
proof (*rule measure-eqI-PiM-infinite*)
show *sets* (*lim-sequence p x*) = *sets* (Π$_M$ *j*∈*UNIV*. *M*)
using *x p* **by** (*rule sets-lim-sequence*)
have [*simp*]: *space* (*K' p x 0* (λ*n*. *undefined*)) ≠ {}
using *p*
using *IT-K' Ionescu-Tulcea.non-empty Ionescu-Tulcea.space-P x*
by *fastforce*
show *sets* (*?B p x*) = *sets* (*Pi$_M$ UNIV* (λ*j*. *M*))
using *p x M-ne-policy space-K0* **by** *auto*

interpret *lim-sequence*: *prob-space lim-sequence p x*
using *lim-sequence x p* **by** (*auto simp*: *measurable-restrict-space2-iff prob-algebra-def*)
show *finite-measure* (*lim-sequence p x*)
by (*rule lim-sequence.finite-measure*)

interpret *Ionescu-Tulcea K' p x* λ-. *M*
using *IT-K'*[*OF p x*] **.**

let *?U* = λ-::*nat*. *undefined* :: ('*s* × '*a*)

fix *J* :: *nat set* **and** *F'*
assume *J*: *finite J* ⋀*i*. *i* ∈ *J* ⟹ *F' i* ∈ *sets M*

define *n* **where** *n* = (*if J* = {} *then 0 else Max J*)
define *F* **where** *F i* = (*if i* ∈ *J then F' i else space M*) **for** *i*
then have *F*[*simp*, *measurable*]: *F i* ∈ *sets M* **for** *i*
using *J* **by** *auto*
have *emb-eq*: *PF.emb UNIV J* (*Pi$_E$ J F'*) = *PF.emb UNIV* {*0..<Suc n*} (*Pi$_E$* {*0..<Suc n*} *F*)

proof *cases*
assume $J = \{\}$ **then show** *?thesis*
by (*auto simp add*: *n-def F-def*[*abs-def*] *prod-emb-def PiE-def*)
next
assume $J \neq \{\}$ **then show** *?thesis*
by (*auto simp*: *prod-emb-def PiE-iff F-def n-def less-Suc-eq-le* ‹*finite J*› *split*: *if-split-asm*)
qed

have *emeasure* (*lim-sequence p x*) (*PF.emb UNIV J* (Pi_E *J F'*)) =
emeasure (*C 0* (*Suc n*) *?U*) (Pi_E {*0..<Suc n*} *F*)
using *x p* **unfolding** *emb-eq*
by (*rule emeasure-lim-sequence-emb-I0o*) (*auto intro*!: *sets-PiM-I-finite*)
also have *C 0* (*Suc n*) *?U* = *K0 p x* $\gg\!=$ (λ*y*. *C 1 n* (*case-nat y ?U*))
using *split-C*[*of ?U 0 Suc 0 n*] *step-C*[*OF x p*] **by** *simp*
also have *emeasure* (*K0 p x* $\gg\!=$ (λ*y*. *C 1 n* (*case-nat y ?U*))) (Pi_E {*0..<Suc n*} *F*) =
($\int^+ y$. *C 1 n* (*case-nat y ?U*) (Pi_E {*0..<Suc n*} *F*) *∂K0 p x*)
using *measurable-C*[*of 1 n*, *measurable del*] *sets-K0*[*OF p x*] *F x p non-empty space-K0*
by (*intro emeasure-bind*[*OF - measurable-compose*[*OF - measurable-C*]])
(*auto cong*: *measurable-cong-sets intro*!: *measurable-PiM-single′ split*: *nat.split-asm*)
also have ... = ($\int^+ y$. *distr* (*lim-sequence* (*Suc-policy p y*) (*K y*))
(Pi_M *UNIV* (λ*j*. *M*)) (*case-nat y*) (*PF.emb UNIV J* (Pi_E *J F'*)) *∂K0 p x*)
proof (*intro nn-integral-cong*)
fix *y* **assume** *y* ∈ *space* (*K0 p x*)
then have *y*: *y* ∈ *space M*
using *x p space-K0* **by** *blast*
then interpret *y*: *Ionescu-Tulcea K′* (*Suc-policy p y*) (*K y*) λ-. *M*
using *p* **by** (*auto intro*!: *IT-K′*)
have *fst y* ∈ *space Ms*
by (*meson measurable-fst measurable-space y*)
let *?y* = *case-nat y*
have [*simp*]: *?y ?U* ∈ *space* (Pi_M {*0*} (λ*i*. *M*))
using *y* **by** (*auto simp*: *space-PiM PiE-iff extensional-def split*: *nat.split*)
have *yM*[*measurable*]: *?y* ∈ Pi_M {*0..<m*} (λ-. *M*) →$_M$ Pi_M {*0..<Suc m*} (λ*i*. *M*) **for** *m*
using *y*
by (*auto intro*: *measurable-PiM-single′ simp*: *space-PiM PiE-iff extensional-def split*: *nat.split*)
have *y′*: *?y ?U* ∈ *space* (Pi_M {*0..<1*} (λ*i*. *M*))
by (*simp add*: *space-PiM PiE-def y extensional-def split*: *nat.split*)

have *eq1*: *?y* −‘ Pi_E {*0..<Suc n*} *F* ∩ *space* (Pi_M {*0..<n*} (λ-.

```
M)) =
        (if y ∈ F 0 then Pi_E {0..<n} (F∘Suc) else {})
      unfolding set-eq-iff using y sets.sets-into-space[OF F]
       by (auto simp: space-PiM PiE-iff extensional-def Ball-def split:
nat.split nat.split-asm)

    have eq2: ?y −‘ PF.emb UNIV {0..<Suc n} (Pi_E {0..<Suc n} F)
∩ space (Pi_M UNIV (λ-. M)) =
       (if y ∈ F 0 then PF.emb UNIV {0..<n} (Pi_E {0..<n} (F∘Suc))
else {})
      unfolding set-eq-iff using y sets.sets-into-space[OF F]
       by (auto simp: space-PiM PiE-iff prod-emb-def extensional-def
Ball-def split: nat.splits)

    let ?I = indicator (F 0) y
    have fst y ∈ space Ms
      using y by (meson measurable-fst measurable-space)
     have C 1 n (?y ?U) = distr (y.C 0 n ?U) (Π_M i∈{0..<Suc n}.
M) ?y
    proof (induction n)
      case (Suc m)

      have C 1 (Suc m) (?y ?U) = distr (y.C 0 m ?U) (Pi_M {0..<Suc
m} (λi. M)) ?y ⤜ eP (Suc m)
        using Suc by simp
      also have ... = y.C 0 m ?U ⤜ (λx. eP (Suc m) (?y x))
        by (auto intro!: bind-distr[where K=Pi_M {0..<Suc (Suc m)}
(λ-. M)] simp: y y.space-C y.sets-C cong: measurable-cong-sets)
       also have ... = y.C 0 m ?U ⤜ (λx. distr (y.eP m x) (Pi_M
{0..<Suc (Suc m)} (λi. M)) ?y)
      proof (intro bind-cong refl)
        fix ω' assume ω': ω' ∈ space (y.C 0 m ?U)
        moreover have K' p x (Suc m) (?y ω') = K' (Suc-policy p y)
(K y) m ω'
          unfolding K'-def Suc-policy-def
          by (auto split: nat.splits)
       ultimately show eP (Suc m) (?y ω') = distr (y.eP m ω') (Pi_M
{0..<Suc (Suc m)} (λi. M)) ?y
          unfolding eP-def y.eP-def
            by (subst distr-distr) (auto simp: y.space-C y.sets-P split:
nat.split cong: measurable-cong-sets
              intro!: distr-cong measurable-fun-upd[where J={0..<m}])
      qed
      also have ... = distr (y.C 0 m ?U ⤜ y.eP m) (Pi_M {0..<Suc
(Suc m)} (λi. M)) ?y
      by (auto intro!: distr-bind[symmetric, OF - - yM] simp: y.space-C
y.sets-C cong: measurable-cong-sets)
      finally show ?case
        by simp
```

qed (*use y* **in** ‹*simp add*: *PiM-empty distr-return*›)
then have *C 1 n* (*case-nat y ?U*) (*Pi*$_E$ {*0..<Suc n*} *F*) =
(*distr* (*y.C 0 n ?U*) (Π$_M$ *i*∈{*0..<Suc n*}. *M*) *?y*) (*Pi*$_E$ {*0..<Suc n*} *F*) **by** *simp*
also have ... = *?I* ∗ *y.C 0 n ?U* (*Pi*$_E$ {*0..<n*} (*F* ∘ *Suc*))
by (*subst emeasure-distr*) (*auto simp*: *y.sets-C y.space-C eq1 cong*: *measurable-cong-sets*)
also have
... = *?I* ∗ *lim-sequence* (*Suc-policy p y*) (*K y*) (*PF.emb UNIV* {*0..<n*} (*Pi*$_E$ {*0..<n*} (*F* ∘ *Suc*)))
using *y sets-PiM-I-finite*
by (*subst emeasure-lim-sequence-emb-I0o*) (*auto simp add*: *p sets-PiM-I-finite*)
also have ... = *distr* (*lim-sequence* (*Suc-policy p y*) (*K y*)) (*Pi*$_M$ *UNIV* (λ*j*. *M*)) *?y*
(*PF.emb UNIV* {*0..<Suc n*} (*Pi*$_E$ {*0..<Suc n*} *F*))
proof (*subst emeasure-distr*, *goal-cases*)
case *1*
thus *?case*
using *y*
by *measurable* (*simp add*: *lim-sequence-def measurable-ident-sets*)
case *2*
thus *?case*
by *auto*
case *3*
thus *?case*
using *y*
by (*subst space-lim-sequence*[*OF - is-policy-Suc-policy*[*OF - p*]]) (*auto simp*: *eq2*)
qed
finally show *emeasure* (*C 1 n* (*case-nat y* (λ-. *undefined*))) (*Pi*$_E$ {*0..<Suc n*} *F*) =
emeasure (*distr* (*lim-sequence* (*Suc-policy p y*) (*K y*)) (*Pi*$_M$ *UNIV* (λ*j*. *M*)) (*case-nat y*))
(*y.PF.emb UNIV J* (*Pi*$_E$ *J F*′))
unfolding *emb-eq* **.**
qed
also have ... = *emeasure* (*K0 p x* ⤜ (λ*y*. *distr* (*lim-sequence* (*Suc-policy p y*) (*K y*))
(*Pi*$_M$ *UNIV* (λ*j*. *M*)) (*case-nat y*))) (*PF.emb UNIV J* (*Pi*$_E$ *J F*′))
using *J sets-K0*[*OF* ‹*is-policy p*› ‹*x* ∈ *space* (*prob-algebra Ms*)›] *p*
by (*subst emeasure-bind*[**where** *N*=*PiM UNIV* (λ-. *M*)]) (*auto simp*: *sets-K x cong*: *measurable-cong-sets*
intro!: *measurable-distr2*[*OF - measurable-prob-algebraD*[*OF lim-sequence*]]
measurable-prob-algebraD
measurable-distr2[**where** *M* = *PiM UNIV* (λ-. *M*)])
finally show *emeasure* (*lim-sequence p x*) (*PF.emb UNIV J* (*Pi*$_E$ *J F*′)) =

emeasure (*K0 p x* $\gg\!=$ (λ*y. distr* (*lim-sequence* (*Suc-policy p y*) (*K y*)) (*Pi*$_M$ *UNIV* (λ*j. M*))
(*case-nat y*))) (*PF.emb UNIV J* (*Pi*$_E$ *J F'*)).
qed

5.9 Stream Space of the MDP

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

lemma *space-lim-stream*: *space* (*lim-stream p x*) = *streams* (*space M*)
unfolding *lim-stream-def* **by** (*simp add*: *space-stream-space*)

lemma *sets-lim-stream*[*measurable-cong*]: *sets* (*lim-stream p x*) = *sets* (*stream-space M*)
unfolding *lim-stream-def* **by** *simp*

lemma *lim-stream*[*measurable*]:
assumes *is-policy p*
shows *lim-stream p* ∈ *prob-algebra Ms* →$_M$ *prob-algebra* (*stream-space M*)
unfolding *lim-stream-def*[*abs-def*]
using *assms*
by (*auto intro*: *measurable-distr-prob-space2*[*OF lim-sequence*])

lemma *lim-stream-Suc*[*measurable*]:
assumes *p*: *is-policy p*
shows (λ*a. lim-stream* (*Suc-policy p a*) (*K a*)) ∈ *M* →$_M$ *prob-algebra* (*stream-space M*)
unfolding *lim-stream-def*[*abs-def*]
using *p*
by (*auto intro*: *measurable-distr-prob-space2*[*OF lim-sequence-Suc-K*])

lemma *space-stream-space-M-ne*: *x* ∈ *space M* ⟹ *space* (*stream-space M*) ≠ {}
using *sconst-streams*[*of x space M*] **by** (*auto simp*: *space-stream-space*)

lemma *prob-space-lim-stream*[*intro*]:
assumes *is-policy p x* ∈ *space* (*prob-algebra Ms*)
shows *prob-space* (*lim-stream p x*)
by (*metis* (*no-types*, *lifting*) *space-prob-algebra measurable-space assms lim-stream mem-Collect-eq*)

lemma *prob-space-step*:
assumes *is-policy p x* ∈ *space M*
shows *prob-space* (*lim-stream* (*Suc-policy p x*) (*K x*))
by (*auto simp*: *assms K-in-space is-policy-Suc-policy*)

lemma *lim-stream-eq*:
assumes *p*: *is-policy p*
assumes *x*: $x \in$ *space* (*prob-algebra Ms*)
shows *lim-stream p x* = *do* {
$y \leftarrow$ *K0 p x*;
$\omega \leftarrow$ *lim-stream* (*Suc-policy p y*) (*K y*);
return (*stream-space M*) (*y* ## ω)
}
unfolding *lim-stream-def lim-sequence-eq*[*OF x p*]
proof (*subst distr-bind*[*OF - - measurable-to-stream*])
show (λ*y*. *distr* (*lim-sequence* (*Suc-policy p y*) (*K y*)) (Pi_M *UNIV* (λ*j*. *M*)) (*case-nat y*)) $\in$
K0 p x $\rightarrow_M$ *subprob-algebra* (Pi_M *UNIV* (λ*i*. *M*))
proof (*intro measurable-prob-algebraD measurable-distr-prob-space2*[**where** *M*=Pi_M *UNIV* (λ*j*. *M*)])
show (λ*x*. *lim-sequence* (*Suc-policy p x*) (*K x*)) $\in$ *K0 p x* $\rightarrow_M$ *prob-algebra* (Pi_M *UNIV* (λ*j*. *M*))
using *lim-sequence-Suc-K*[*OF p*] *sets-K0*[*OF p x*] *measurable-cong-sets*
by *blast*
next show (λ(*ya*, *y*). *case-nat ya y*) $\in$ *K0 p x* $\bigotimes_M$ Pi_M *UNIV* (λ*j*. *M*) $\rightarrow_M$ Pi_M *UNIV* (λ*j*. *M*)
using *sets-K0*[*OF p x*]
by (*subst measurable-cong-sets*[*of - M* $\bigotimes_M$ Pi_M *UNIV* (λ*j*. *M*)]) *auto*
qed
next
show *space* (*K0 p x*) $\neq$ {}
using *x p prob-space.not-empty prob-space-K0*
by *blast*
next
show *K0 p x* $\gg\!=$ (λ*x*. *distr* (*distr* (*lim-sequence* (*Suc-policy p x*) (*K x*)) (Pi_M *UNIV* (λ*j*. *M*)) (*case-nat x*)) (*stream-space M*) *to-stream*) = *K0 p x* $\gg\!=$ (λ*y*. *distr* (*lim-sequence* (*Suc-policy p y*) (*K y*)) (*stream-space M*) *to-stream* $\gg\!=$ ($\lambda\omega$. *return* (*stream-space M*) (*y* ## ω)))
proof (*intro bind-cong refl*, *subst distr-distr*)
show *to-stream* $\in$ Pi_M *UNIV* (λ*j*. *M*) $\rightarrow_M$ *stream-space M*
by *measurable*
next
show $\bigwedge$*a*. *a* $\in$ *space* (*K0 p x*) $\Longrightarrow$
case-nat a $\in$ *lim-sequence* (*Suc-policy p a*) (*K a*) $\rightarrow_M$ Pi_M *UNIV* (λ*j*. *M*)
by *measurable* (*auto simp*: *p x intro*!: *measurable-ident-sets sets-lim-sequence intro*: *measurable-space*)
next

show $\bigwedge a.\ a \in space\ (K0\ p\ x) \Longrightarrow$
distr (*lim-sequence* (*Suc-policy p a*) (*K a*)) (*stream-space M*) (*to-stream* ∘ *case-nat a*) =
distr (*lim-sequence* (*Suc-policy p a*) (*K a*)) (*stream-space M*) *to-stream* ⤜
(λω. *return* (*stream-space M*) (*a ## ω*))
proof (*subst bind-return-distr′*, *goal-cases*)
case (*1 a*)
then show *?case* **by** (*simp add*: *p space-stream-space-M-ne x*)
next
case (*2 a*)
then show *?case* **using** *p x* **by** (*auto simp*: *sets-lim-sequence cong*: *measurable-cong-sets intro*!: *distr-cong*)[*1*]
next
case (*3 a*)
then show *?case*
using *p x*
by (*subst distr-distr*) (*auto simp*: *to-stream-nat-case intro*!: *measurable-compose*[*OF - measurable-to-stream*]
sets-lim-sequence distr-cong measurable-ident-sets)
qed
qed
qed

end
end

theory *MDP-disc*
imports
MDP-cont
HOL−Library.Omega-Words-Fun
begin

6 Markov Decision Processes with Discrete State Spaces

lemma (**in** *prob-space*) *integral-stream-space*:
fixes *f* :: *′a stream* ⇒ (*′b* :: {*banach*, *second-countable-topology*,*real-normed-vector*})
assumes *int-f*: *integrable* (*stream-space M*) *f*
assumes [*measurable*]: *f* ∈ *borel-measurable* (*stream-space M*)
shows (∫ *X*. *f X* ∂*stream-space M*) = (∫ *x*. (∫ *X*. *f* (*x ## X*) ∂*stream-space M*) ∂*M*)
proof −
interpret *S*: *sequence-space M* **..**
interpret *P*: *pair-sigma-finite M* Π$_M$ *i*::*nat*∈*UNIV*. *M* **..**

interpret *P′*: *pair-sigma-finite* Π$_M$ *i*::*nat*∈*UNIV*. *M M* **..**

obtain i **where** *has-bochner-integral* (*stream-space* M) f i
using *int-f*
using *integrable.cases* **by** *blast*
have *integrable* $S.S$ ($\lambda X.$ f (*to-stream* X))
using *int-f*
by (*metis integrable-distr measurable-to-stream stream-space-eq-distr*)
hence *integrable* (*distr* ($M \bigotimes_M Pi_M$ *UNIV* ($\lambda i.$ M)) (Pi_M *UNIV* ($\lambda i.$ M)) ($\lambda(x,$ $y).$ *case-nat* x y)) ($\lambda X.$ f (*to-stream* X))
by (*auto simp*: $S.PiM$-*iter*)
moreover have *integrable* (*distr* ($M \bigotimes_M Pi_M$ *UNIV* ($\lambda i.$ M)) (Pi_M *UNIV* ($\lambda i.$ M)) ($\lambda(x,$ $y).$ *case-nat* x y)) ($\lambda X.$ f (*to-stream* X)) $\longleftrightarrow$
integrable ($M \bigotimes_M S.S$) ($\lambda X.$ f (*to-stream* (($\lambda(s,$ $\omega).$ *case-nat* s ω) X)))
by (*auto simp*: *integrable-distr-eq*)
ultimately have *integrable* ($M \bigotimes_M S.S$) ($\lambda X.$ f (*to-stream* (($\lambda(s,$ $\omega).$ *case-nat* s ω) X)))
by *auto*
hence *integrable* ($M \bigotimes_M$ (Pi_M *UNIV* ($\lambda i.$ M)))
($\lambda X.$ f (*to-stream* (($\lambda(s,$ $\omega).$ *case-nat* s ω) X)))
by *auto*
moreover have *integrable* ($M \bigotimes_M$ (Pi_M *UNIV* ($\lambda i.$ M)))
($\lambda X.$ f (*to-stream* (($\lambda(s,$ $\omega).$ *case-nat* s ω) X))) =
integrable ($M \bigotimes_M Pi_M$ *UNIV* ($\lambda i.$ M)) ($\lambda(x,$ $X).$ f (*to-stream* (*case-nat* x X)))
by (*auto intro*!: *Bochner-Integration.integrable-cong*)
ultimately have $*$: *integrable* ($M \bigotimes_M Pi_M$ *UNIV* ($\lambda i.$ M)) ($\lambda(x,$ $X).$ f (*to-stream* (*case-nat* x X)))
by *auto*
have ($\int X.$ f X ∂*stream-space* M) = ($\int X.$ f (*to-stream* X) $\partial S.S$)
by (*subst stream-space-eq-distr*) (*simp add*: *integral-distr*)
also have $\ldots$ = ($\int X.$ f (*to-stream* (($\lambda(s,$ $\omega).$ *case-nat* s ω) X)) ∂($M \bigotimes_M S.S$))
by (*subst* $S.PiM$-*iter*[*symmetric*]) (*simp add*: *integral-distr*)
also have $\ldots$ = ($\int x.$ $\int X.$ f (*to-stream* (($\lambda(s,$ $\omega).$ *case-nat* s ω) (x, X)))$\partial S.S$ ∂M)
using $*$
by (*auto simp*: *pair-sigma-finite.integral-fst P.pair-sigma-finite-axioms case-prod-unfold*)
also have $\ldots$ = ($\int x.$ $\int X.$ f (x ## *to-stream* X) $\partial S.S$ ∂M)
by (*auto intro*!: *integral-cong simp*: *to-stream-nat-case*)
also have $\ldots$ = ($\int x.$ $\int X.$ f (x ## X) ∂*distr* (Pi_M *UNIV* ($\lambda i.$ M)) (*stream-space* M) *to-stream* ∂M)
by (*subst Bochner-Integration.integral-cong*[*OF refl*]) (*auto simp*: *integral-distr*)
also have $\ldots$ = ($\int x.$ $\int X.$ f (x ## X) ∂*stream-space* M ∂M)
using *stream-space-eq-distr* **by** *metis*

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

lemma *prefix-cons*:
Omega-Words-Fun.prefix (*Suc n*) *seq* = *seq 0*#*Omega-Words-Fun.prefix n* (λ*n*. *seq* (*Suc n*))
by (*metis map-upt-Suc subsequence-def*)

lemma *restrict-Suc*: *restrict y* {*0*..<*Suc i*} (*Suc n*) = (*restrict* (λ*n*. *y* (*Suc n*)) {*0*..<*i*}) *n*
by *auto*

lemma *prefix-restrict*: *Omega-Words-Fun.prefix i* (*restrict y* {*0*..<*i*}) = *Omega-Words-Fun.prefix i y*
proof (*induction i arbitrary*: *y*)
case (*Suc i*)
then show *?case*
unfolding *restrict-Suc prefix-cons*
by *fastforce+*
qed *simp*

lemma *prefix-measurable*[*measurable*]:
Omega-Words-Fun.prefix i ∈ *Pi*$_M$ {*0*..<*i*}
(λ*-*. *count-space* (*UNIV* :: (*'s* ::*countable* × *'a*::*countable*) *set*)) →$_M$ *count-space UNIV*
proof (*induction i*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc i*)
have *aux*: (λ*w*. (*restrict w* {*0*..<*i*}, *w i*)) ∈ *Pi*$_M$ {*0*..<*Suc i*} (λ*-*. *count-space UNIV*) →$_M$
Pi$_M$ {*0*..<*i*} (λ*-*. *count-space UNIV*) ⨂$_M$ (*count-space UNIV*)
by *auto*
have *aux'*: (λ(*w*,*wi*). *Omega-Words-Fun.prefix i* (*restrict w* {*0*..<*i*})@[*wi*]) ∈ *Pi*$_M$ {*0*..<*i*}
(λ*-*. *count-space* (*UNIV* :: (*'s* × *'a*) *set*)) ⨂$_M$ (*count-space UNIV*) →$_M$ *count-space UNIV*
using *Suc.IH* **by** *auto*
have *f-eq*: ⋀*w*. *Omega-Words-Fun.prefix i* (*restrict w* {*0*..<*i*}) @ [*w i*] =
(λ(*w*,*wi*). *Omega-Words-Fun.prefix i w* @[*wi*]) ((*restrict w* {*0*..<*i*}), *w i*)
by *auto*
have (λ*w*:: *nat* ⇒ *'s* × *'a*. *Omega-Words-Fun.prefix i* (*restrict w* {*0*..<*i*}) @ [*w i*]) ∈ *Pi*$_M$ {*0*..<*Suc i*} (λ*-*. *count-space UNIV*) →$_M$ *count-space UNIV*
using *aux aux'*[*unfolded prefix-restrict*]
by (*subst f-eq*) *auto*

thus *?case*
unfolding *prefix-restrict*[*of - i*]
by *auto*
qed

no-notation *Omega-Words-Fun.build* (**infixr** ‹##› *65*)

locale *discrete-MDP* =
fixes *A* :: *'s*::*countable* ⇒ *'a*::*countable set* — enabled actions
and *K* :: *'s* × *'a* ⇒ *'s pmf* — MDP kernel, transition probabilities
assumes
A-ne: ⋀*s*. *A s* ≠ {} — set of enabled actions is nonempty
begin

6.1 Policies

Type synonym for decision rules.

type-synonym (*'c*, *'d*) *dec* = *'c* ⇒ *'d pmf*

definition *is-dec* :: (*'s*, *'a*) *dec* ⇒ *bool* **where**
is-dec d ≡ ∀ *s*. *d s* ⊆ *A s*

lemma *is-decI*[*intro*]:
(⋀*s*. *set-pmf* (*d s*) ⊆ *A s*) ⟹ *is-dec d*
unfolding *is-dec-def*
by *auto*

abbreviation *D*$_R$ ≡ {*d*. *is-dec d*}

definition *is-dec-det* :: (*'s* ⇒ *'a*) ⇒ *bool* **where**
is-dec-det d ≡ ∀ *s*. *d s* ∈ *A s*

abbreviation *D*$_D$ ≡ {*d*. *is-dec-det d*}

definition *mk-dec-det d s* = *return-pmf* (*d s*)

lemma *is-dec-mk-dec-det-iff* [*simp*]: *is-dec* (*mk-dec-det d*) ⟷ *is-dec-det d*
by (*simp add*: *is-dec-def is-dec-det-def mk-dec-det-def*)

lemma *D-det-to-MR*[*intro*]: *is-dec-det d* ⟹ *is-dec* (*mk-dec-det d*)
by *simp*

Due to the assumption *A ?s* ≠ {}, a deterministic decision rule always exists. It immediately follows via *is-dec* (*mk-dec-det ?d*) = *is-dec-det ?d* that a randomized decision rule also exists.

lemma *SOME-is-dec-det*: *is-dec-det* (λ*s*. *SOME a*. *a* ∈ *A s*)
using *A-ne* **by** (*simp add*: *is-dec-det-def some-in-eq*)

lemma *ex-dec-det* [*simp*]: $\exists\, d.$ *is-dec-det d*
using *SOME-is-dec-det* **by** *blast*

lemma *D-det-ne* [*simp*]: $D_D \neq \{\}$
by *simp*

lemma D_R*-ne* [*simp*]: $D_R \neq \{\}$
using *D-det-ne D-det-to-MR* **by** *blast*

lemma *ex-dec*[*intro*, *simp*]: $\exists\, d.$ *is-dec d*
using *ex-dec-det* **by** *blast*

Type synonym for policies.

type-synonym $('c, 'd)$ *pol* $= ('c \times 'd)$ *list* $\Rightarrow ('c, 'd)$ *dec*

A policy assigns a decision rule to each observed past.

definition *is-policy* :: $('s, 'a)$ *pol* $\Rightarrow$ *bool* **where**
is-policy p $\equiv \forall\, hs.$ *is-dec* (*p hs*)

abbreviation $\Pi_{H\,R} \equiv \{p.$ *is-policy p*$\}$

Deterministic policies

definition *is-deterministic p* $\equiv$ *is-policy p* $\wedge$ $(\forall\, h\ s.\ \exists\, a.\ p\ h\ s =$ *return-pmf a*)

definition *mk-det p h s* $\equiv$ *return-pmf* (*p h s*)

abbreviation $\Pi_{H\,D} \equiv \{p.\ \forall\, h.\ p\ h \in D_D\}$

Markovian policies

definition *is-markovian p* $\equiv$ *is-policy p* $\wedge$ $(\forall\, h\ h'.$ *length h* $=$ *length* $h' \longrightarrow p\ h = p\ h')$

definition *mk-markovian* :: $(nat \Rightarrow ('s, 'a)$ *dec*$) \Rightarrow ('s, 'a)$ *pol* **where**
mk-markovian p $\equiv (\lambda h.\ p$ (*length h*))

lemma *is-markovian-mk-iff*[*simp*]: *is-markovian* (*mk-markovian p*) $\longleftrightarrow$ $(\forall\, n.$ *is-dec* (*p n*))
unfolding *is-markovian-def mk-markovian-def is-policy-def*
by (*metis* (*mono-tags*, *opaque-lifting*) *Ex-list-of-length*)

lemma *is-markovian-mk*[*intro*]: $\forall\, n.$ *is-dec* (*p n*) $\Longrightarrow$ *is-markovian* (*mk-markovian p*)
unfolding *is-markovian-def mk-markovian-def is-policy-def*
by *auto*

lemma *mk-markovian-nil* [*simp*]: *mk-markovian p* [] $=$ *p 0*

unfolding *mk-markovian-def* **by** *auto*

definition *mk-markovian-det* $p \equiv (\lambda h\ s.\ \textit{return-pmf}\ (p\ (\textit{length}\ h)\ s))$

abbreviation $\Pi_{MD} \equiv \{p.\ \forall\, n{::}\textit{nat}.\ p\ n \in D_D\}$
abbreviation $\Pi_{MR} \equiv \{p.\ \forall\, n.\ p\ n \in D_R\}$

lemma Π_{MR}*-imp-policies*[*intro*]: $p \in \Pi_{MR} \Longrightarrow \textit{mk-markovian}\ p \in \Pi_{HR}$
unfolding *is-policy-def mk-markovian-def* **by** *auto*

lemma Π_{MD}*-MR-iff*[*simp*]: $(\lambda n.\ \textit{mk-dec-det}\ (p\ n)) \in \Pi_{MR} \longleftrightarrow p \in \Pi_{MD}$
by *auto*

lemma Π_{MD}*-to-MR*[*intro*]: $p \in \Pi_{MD} \Longrightarrow (\lambda n.\ \textit{mk-dec-det}\ (p\ n)) \in \Pi_{MR}$
by *simp*

lemma *p-n-π-MD*[*intro*]: $p \in \Pi_{MD} \Longrightarrow p\ n \in D_D$
by *auto*

lemma *p-n-π-MR*[*intro*]: $p \in \Pi_{MR} \Longrightarrow p\ n \in D_R$
by *auto*

lemma Π_{MD}*-ne*[*simp*]: $\Pi_{MD} \neq \{\}$
by (*auto simp*: *someI-ex*[*OF ex-dec-det*] *intro*: *exI*[*of* - $\lambda n.$ (*SOME d. is-dec-det d*)])

lemma Π_{MR}*-ne*[*simp*]: $\Pi_{MR} \neq \{\}$
using Π_{MD}*-ne* **by** *fast*

lemma *policies-ne*[*simp*, *intro*]: $\Pi_{HR} \neq \{\}$
using Π_{MR}*-ne is-policy-def* **by** *auto*

Stationary policies

definition *is-stationary* $p \equiv \textit{is-policy}\ p \wedge (\forall\, h\ h'.\ p\ h = p\ h')$

lemma *is-stationary-const-iff*[*simp*]: *is-stationary* $(\lambda\text{-}.\ d) = \textit{is-dec}\ d$
unfolding *is-stationary-def is-policy-def* **by** *simp*

lemma *is-stationary-const*[*intro*]: *is-dec* $d \Longrightarrow$ *is-stationary* $(\lambda\text{-}.\ d)$
by *simp*

abbreviation *mk-stationary* $p \equiv \textit{mk-markovian}\ (\lambda\text{-}.\ p)$
abbreviation *mk-stationary-det* $d \equiv \textit{mk-markovian}\ (\lambda\text{-}.\ \textit{mk-dec-det}\ d)$

6.1.1 Successor Policy

After taking the first step in the MDP, we will know which state and which action got selected during the initial epoch. To obtain a policy that acts as if the current epoch was the initial one, we prepend the observed state-action pair to the history. The result is again a policy, i.e. it satisfies *is-policy*.

definition π-*Suc* :: ('*s*, '*a*) *pol* ⇒ '*s* × '*a* ⇒ ('*s*, '*a*) *pol*
  where
    π-*Suc p sa h* = *p* (*sa*#*h*)

lemma *is-policy-*π*-Suc* [*intro*]: *is-policy p* ⟹ *is-policy* (π-*Suc p sa*)
  unfolding *is-policy-def* π-*Suc-def* **by** *force*

lemma *Suc-mk-markovian*[*simp*]: π-*Suc* (*mk-markovian p*) *x* = *mk-markovian* (λ*n*. *p* (*Suc n*))
  unfolding π-*Suc-def mk-markovian-def* **by** *auto*

6.2 Stream Space of the MDP

6.2.1 Initial State-Action Distribution

If we fix a decision rule *d* and an initial distribution of states *S0*, we obtain a distribution over state-action pairs in the following way: First, the initial state *s* is sampled from *S0*, then an action *a* is selected from *d s*.

definition *K0 d S0* = *do* {
  s ← *S0*;
  a ← *d s*;
  return-pmf (*s*,*a*)
}

notation *K0* (‹K_0›)

lemma *K0-iff*: *K0 d S0* = *S0* ⤜ (λ*s*. *map-pmf* (λ*a*. (*s*,*a*)) (*d s*))
  by (*simp add*: *K0-def map-pmf-def*)

lemma *vimage-pair*[*simp*]: *Pair x* −‘ {*p*} = (*if x* = *fst p then* {*snd p*} *else* {})
  by *auto*

lemma *pmf-K0* [*simp*]: *pmf* (*K0 d S0*) (*s*,*a*) = *pmf S0 s* ∗ *pmf* (*d s*) *a*
  unfolding *K0-iff pmf-bind*
 by (*subst integral-measure-pmf*[**where** *A* = {*s*}]) (*auto simp*: *pmf-map pmf.rep-eq split*: *if-splits*)

lemma *set-pmf-K0*: *set-pmf* (*K0 p S0*) = {(*s*,*a*). *s* ∈ *S0* ∧ *a* ∈ *p s*}

by (*auto simp add*: *K0-def*)

lemma *fst-K0*[*simp*]: *map-pmf fst* (*K0 p S0*) = *S0*
unfolding *K0-def*
by (*simp add*: *map-bind-pmf map-pmf-comp bind-return-pmf'*)

abbreviation *S* ≡ *stream-space* (*count-space UNIV*)

We inherit the trace space from MDPs with continuous state-action spaces

interpretation *MDP-cont*: *MDP-cont.discrete-MDP count-space UNIV count-space UNIV A K*
proof *standard*
show (λ*x*. *measure-pmf* (*K x*)) ∈
count-space UNIV ⨂$_M$ *count-space UNIV* →$_M$ *prob-algebra* (*count-space UNIV*)
using *measurable-prob-algebraI*
by (*measurable*, *auto simp*: *prob-space-measure-pmf measurable-pair-measure-countable1*)+
show ∃ δ∈*count-space UNIV* →$_M$ *count-space UNIV*. ∀ *s*. δ *s* ∈ *A s*
by (*auto simp*: *A-ne some-in-eq intro*: *bexI*[*of* - λ*s*. *SOME a*. *a* ∈ *A s*])
qed (*auto simp*: *A-ne*)

lemma *count-space-M*[*simp*]: *MDP-cont.M* = *count-space UNIV*
by (*auto simp*: *pair-measure-countable*)

lemma *space-M*[*simp*]: *space MDP-cont.M* = *UNIV*
by (*auto simp*: *MDP-cont.space-lim-stream*)

We reuse the stream space provided by *MDP-cont.lim-stream*

definition *T* :: ('*s*, '*a*) *pol* ⇒ '*s pmf* ⇒ ('*s* × '*a*) *stream measure*
where *T p* = *MDP-cont.lim-stream* (λ*n* (*h*,*s*). *p* (*Omega-Words-Fun.prefix n h*) *s*)

lemma *sets-T*[*measurable-cong*]:
sets (*T p x*) = *sets S*
by (*auto simp*: *T-def MDP-cont.sets-lim-stream*)

lemma *space-stream-space-ne*[*simp*]: *space S* ≠ {}
by (*auto simp*: *space-stream-space*)

lemma *space-T*[*simp*]: *space* (*T p S0*) = *space S*
by (*simp add*: *MDP-cont.space-lim-stream T-def space-stream-space*)

lemma *is-policy-MDP-cont*[*intro*]:
fixes *p* :: ('*s* × '*a*) *list* ⇒ '*s* ⇒ '*a pmf*
shows *MDP-cont.is-policy* (λ*n* (*h*,*s*). *p* (*Omega-Words-Fun.prefix n h*) *s*)
unfolding *MDP-cont.is-policy-def MDP-cont.is-dec-def*

using *prefix-measurable measurable-pair-swap-iff*
by (*auto simp*: *prob-space-measure-pmf*
intro: *measurable-pair-measure-countable1 measurable-prob-algebraI*)

lemma *prob-space-T*[*intro*, *simp*]: *prob-space* (*T p x*)
by (*auto simp add*: *T-def prob-space-measure-pmf space-prob-algebra*)

lemma *T-subprob*[*simp*]:
T p S0 ∈ *space* (*subprob-algebra S*)
by (*metis prob-space.M-in-subprob prob-space-T sets-T subprob-algebra-cong*)

lemma *T-subprob-space* [*simp*]: *subprob-space* (*T p S0*)
by (*auto intro*: *prob-space-imp-subprob-space*)

lemma *K0-MDP-cont-eq*:
MDP-cont.K0 (λ*x* (*h*,*s*). *measure-pmf* (*p* (*Omega-Words-Fun.prefix x h*) *s*)) (*measure-pmf S0*) =
K0 (*p* []) *S0*
unfolding *MDP-cont.K0-def K0-def MDP-cont.K′-def map-pmf-def*
by (*simp add*: *measure-pmf-bind return-pmf.rep-eq*)

6.2.2 Decomposition of the Stream Space

The distribution of traces/walks the MDP allows should intuitively satisfy the following rule:

1. select the initial state s from *S0*
2. pass it to the decision rule *p* [] to determine a distribution over actions
3. select the action a

- finally pass the state-action pair (s, a) to the kernel K to get a new distribution over states *s0′*

  Then the iteration repeats with the updated policy *π-Suc p* (*s*, *a*).

  The result carries over from ⟦*MDP-cont.is-policy ?p*; *?x* ∈ *space* (*prob-algebra* (*count-space UNIV*))⟧ ⟹ *MDP-cont.lim-stream ?p ?x* = *MDP-cont.K0 ?p ?x* ⤜ (λ*y*. *MDP-cont.lim-stream* (*MDP-cont.Suc-policy ?p y*) (*measure-pmf* (*K y*)) ⤜ (λω. *return* (*stream-space MDP-cont.M*) (*y* ## ω))).

lemma *T-eq*:
shows *T p S0* = *do* {
sa ← *measure-pmf* (*K0* (*p* []) *S0*);
ω ← *T* (*π-Suc p sa*) (*K sa*);
return S (*sa* ## ω)
}

unfolding *T-def*
proof (*subst MDP-cont.lim-stream-eq*)
show *MDP-cont.is-policy* (λ*x xa*. *measure-pmf* (*case xa of* (*h*, *xa*) ⇒ *p* (*Omega-Words-Fun.prefix x h*) *xa*))
by *auto*
qed (*auto simp*: *space-prob-algebra prob-space-measure-pmf* π*-Suc-def MDP-cont.Suc-policy-def*
prefix-cons K0-MDP-cont-eq prod.case-distrib)

lemma *T-eq-distr*:
shows *T p S0* = *measure-pmf* (*K0* (*p* []) *S0*) ⤜ (λ*sa*. *distr* (*T* (π*-Suc p sa*) (*K sa*)) *S* ((##) *sa*))
by (*simp add*: *T-eq*[*symmetric*] *bind-return-distr'*[*symmetric*])

The iteration rule lets us nicely decompose integrals (expected values) over functions on traces of the MDP.

lemma *integral-T*:
fixes *f* :: (′*s* × ′*a*) *stream* ⇒ *real*
assumes *f-bounded*: ⋀*x*. |*f x*| ≤ *B*
assumes *f*: *f* ∈ *borel-measurable S*
shows (∫ *t*. *f t* ∂*T p x*) = ∫ *sa*. ∫ *t′*. *f* (*sa*##*t′*) ∂*T* (π*-Suc p sa*) (*K sa*) ∂*K0* (*p* []) *x*
proof −
note *T-eq-distr*
have (∫ *t*. *f t* ∂*T p x*) = (∫ *t*. *f t* ∂*measure-pmf* (*K*$_0$ (*p* []) *x*) ⤜ (λ*sa*. *distr* (*T* (π*-Suc p sa*) (*K sa*)) (*stream-space* (*count-space UNIV*)) ((##) *sa*)))
using *T-eq-distr* **by** *metis*
also have ... = *measure-pmf.expectation* (*K*$_0$ (*p* []) *x*) (λ*sa*. *LINT t′*|*T* (π*-Suc p sa*) (*K sa*). *f* (*sa* ## *t′*))
proof (*subst integral-bind*[*OF f f-bounded*, **where** *B′* = *1*], *goal-cases*)
case *1*
then show *?case*
by (*auto intro*!: *prob-space-imp-subprob-space prob-space.prob-space-distr*
simp: *space-subprob-algebra*)
next
case *3*
then show *?case*
by (*auto intro*!: *prob-space.emeasure-le-1 prob-space.prob-space-distr*)
next
case *4*
then show *?case*
by (*auto simp*: *f integral-distr intro*: *Bochner-Integration.integral-cong*)
qed *auto*
finally show *?thesis*.
qed

lemma *nn-integral-T*:
assumes *f*: $f \in$ *borel-measurable S*
shows $(\int^+ t.\ f\ t\ \partial T\ p\ x) = (\int^+ sa.\ \int^+ t'.\ f\ (sa \#\# t')\ \partial T\ (\pi\text{-}Suc\ p\ sa)\ (K\ sa)\ \partial K0\ (p\ [])\ x)$
unfolding *T-eq-distr*[*of p*]
by (*subst nn-integral-bind*[*OF f*])
(*auto intro*!: *prob-space-imp-subprob-space prob-space.prob-space-distr*

simp: *f nn-integral-distr space-subprob-algebra*)

6.2.3 A Denotational View on the Stochastic Process

Many definitions on MDPs do not rely on the individual traces but only on the distribution of states and actions at each epoch.

We define this view on the trace space as the repeated iteration of K_0 and K. It coincides with the definition of T.

primrec *Pn* :: ('*s*, '*a*) *pol* ⇒ '*s pmf* ⇒ *nat* ⇒ ('*s* × '*a*) *pmf* **where**
Pn p S0 0 = *K0* (*p* []) *S0*
| *Pn p S0* (*Suc n*) = *K0* (*p* []) *S0* ⤜ (λ*sa*. *Pn* (*π-Suc p sa*) (*K sa*) *n*)

declare *Pn.simps*(*2*)[*simp del*]

lemma *Pn-eq-T*: *measure-pmf* (*Pn p S0 n*) = *distr* (*T p S0*) (*count-space UNIV*) (λ*t*. *t* !! *n*)
proof (*induction n arbitrary*: *p S0*)
case (*0 p S0*)
then show *?case*
unfolding *T-eq*[*of p*]
proof (*subst distr-bind*[**where** *K* = *S*], *goal-cases*)
case *1*
then show *?case*
by (*auto intro*!: *prob-space-imp-subprob-space subprob-space.bind-in-space*)
next
case *4*
then show *?case*
by (*subst bind-cong*[*OF refl*, **where** *g* = *return* (*count-space UNIV*)])
(*auto intro*!: *bind-const*′ *simp*: *distr-bind*[**where** *K* = *S*] *distr-return bind-return*′′ *space-stream-space subprob-space-return-ne*)
qed *auto*
next
case (*Suc n*)
show *?case*
unfolding *T-eq*[*of p*]
proof (*subst distr-bind*[**where** *K* = *S*], *goal-cases*)
case *1*
then show *?case*
by (*auto intro*!: *prob-space-imp-subprob-space subprob-space.bind-in-space*)[*1*]

```
  next
    case 4
    then show ?case
    by (auto simp: Pn.simps(2) measure-pmf-bind Suc bind-return-distr'
distr-distr comp-def intro!: bind-cong)
  qed auto
qed
```

The definition of *Pn* also allows us to easily prove that only enabled actions can occur in the traces of the MDP.

```
lemma Pn-in-A: is-policy p ⟹ (s, a) ∈ Pn p S0 n ⟹ a ∈ A s
proof (induction n arbitrary: S0 p)
  case 0
  then show ?case
    using 0 unfolding is-policy-def is-dec-def
    by (auto simp: K0-def)
next
  case (Suc n)
  then show ?case
    by (auto simp: Pn.simps(2) K0-def)
qed

lemma T-in-A:
  assumes is-policy p
  shows AE t in T p S0. snd (t !! n) ∈ A (fst (t !! n))
proof −
  have aux: AE t in distr (T p S0) (count-space UNIV) (λt. t !! n).
snd t ∈ A (fst t)
    using assms Pn-eq-T[symmetric]
    by (auto simp: Pn-in-A intro!: AE-pmfI cong: AE-cong-simp)
  show ?thesis
    by (auto intro!: AE-distrD[OF - aux])
qed
```

6.2.4 State Process

Alongside *Pn*, we also define the state and action distributions as projections.

```
definition Xn p S0 n = map-pmf fst (Pn p S0 n)

lemma X0 [simp]: Xn p S0 0 = S0
  using fst-K0 Xn-def by auto

lemma Xn-Suc: Xn p S0 (Suc n) = Pn p S0 n ⤜ K
proof (induction n arbitrary: p S0)
  case 0
  then show ?case
    by (simp add: Pn.simps(2) Xn-def map-bind-pmf)
```

next
case (*Suc n*)
then show *?case*
by (*simp add*: *Pn.simps*(*2*) *Xn-def map-bind-pmf bind-assoc-pmf*)
qed

lemma *Pn-markovian-eq-Xn-bind*: *Pn* (*mk-markovian p*) *S0 n* = *K0* (*p n*) (*Xn* (*mk-markovian p*) *S0 n*)
proof (*induction n arbitrary*: *p S0*)
case *0*
then show *?case*
unfolding *Xn-def* **by** *auto*
next
case (*Suc n*)
then show *?case*
unfolding *Xn-def K0-def*
by (*auto intro*!: *bind-pmf-cong simp*: *Pn.simps*(*2*) *map-bind-pmf Suc bind-assoc-pmf*)
qed

lemma *Xn-Suc'*: *Xn p S0* (*Suc n*) = *K0* (*p* []) *S0* ⤜ (λ*sa*. *Xn* (*π-Suc p sa*) (*K sa*) *n*)
unfolding *Xn-def* **by** (*auto simp*: *Pn.simps*(*2*) *map-bind-pmf*)

lemma *set-pmf-X0* [*simp*]: *set-pmf* (*Xn p S0 0*) = *S0*
using *X0* **by** *auto*

lemma *set-pmf-PSuc*: *set-pmf* (*Pn* (*mk-markovian p*) *S0 n*) =
{(*s*, *a*). *s* ∈ *set-pmf* (*Xn* (*mk-markovian p*) *S0 n*) ∧ *a* ∈ *p n s*}
using *set-pmf-K0 Pn-markovian-eq-Xn-bind* **by** *auto*

6.2.5 The Conditional Distribution of Actions

Actions are selected wrt. the whole history of state-action pairs encountered so far. The following definition defines the expected action selection when only the current state is given.

definition *Y-cond-X p S0 n x* = *map-pmf snd* (*cond-pmf* (*Pn p S0 n*) {(*s*,*a*). *s* = *x*})

lemma *prob-K0-X* [*simp*]: *measure-pmf.prob* (*K0 p S0*) {(*s*, *a*). *s* = *x*} = *pmf S0 x*
unfolding *K0-iff*
proof (*subst measure-pmf-bind*, *subst measure-pmf.measure-bind*[*of* - - *count-space UNIV*], *goal-cases*)
case *1*
then show *?case*
by (*simp add*: *measure-pmf-in-subprob-algebra*)
next
case *3*

then show *?case*
by (*subst integral-measure-pmf-real*[*of* {*x*}]) (*auto split*: *if-splits*)
qed *simp*

lemma *prob-Pn-X*[*simp*]: *measure-pmf.prob* (*Pn p S0 n*) {(*s*, *a*). *s* = *x*} = *pmf* (*Xn p S0 n*) *x*
proof (*induction n arbitrary*: *p S0*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
show *?case*
unfolding *Xn-Suc' Pn.simps*(*2*) *measure-pmf-bind*
using *Suc*
by (*simp add*: *measure-pmf.measure-bind*[*of - - count-space UNIV*] *K0-def*
measure-pmf-in-subprob-algebra pmf-bind)
qed

lemma *pmf-Pn-pair*:
assumes *sa* ∈ *set-pmf* (*Pn p S0 n*)
shows *pmf* (*Pn p S0 n*) *sa* = *pmf* (*Y-cond-X p S0 n* (*fst sa*)) (*snd sa*) ∗ *pmf* (*Xn p S0 n*) (*fst sa*)
proof −
have *aux*: *set-pmf* (*Pn p S0 n*) ∩ {(*s*, *a*). *s* = *fst sa*} ≠ {}
using *Xn-def assms* **by** *auto*
have *aux'*: ({(*s*, *a*). *s* = *fst sa*} ∩ *snd* −‘ {*snd sa*}) = {*sa*}
by *auto*
show *?thesis*
using *assms*
unfolding *Y-cond-X-def pmf-map cond-pmf.rep-eq*[*OF aux*]
by (*auto simp*: *Xn-def pmf-eq-0-set-pmf measure-pmf.emeasure-eq-measure aux' measure-pmf-single*)
qed

lemma *pmf-Pn*:
assumes *x* ∈ *set-pmf* (*Xn p S0 n*)
shows *pmf* (*Pn p S0 n*) (*x*,*a*) = *pmf* (*Y-cond-X p S0 n x*) *a* ∗ *pmf* (*Xn p S0 n*) *x*
proof −
have *aux*: *set-pmf* (*Pn p S0 n*) ∩ {(*s*, *a*). *s* = *x*} ≠ {}
using *Xn-def assms* **by** *auto*
have *aux'*: ({(*s*, *a*). *s* = *x*} ∩ *snd* −‘ {*a*}) = {(*x*, *a*)}
by *auto*
show *?thesis*
using *assms*
unfolding *Y-cond-X-def cond-pmf.rep-eq*[*OF aux*] *pmf-map*
by (*auto simp*: *pmf-eq-0-set-pmf measure-pmf.emeasure-eq-measure*

aux′ measure-pmf-single)
qed

lemma *pmf-Y-cond-X*:
assumes $x \in$ *set-pmf* (*Xn p S0 n*)
shows *pmf* (*Y-cond-X p S0 n x*) *a* = *pmf* (*Pn p S0 n*) (*x*,*a*) / *pmf* (*Xn p S0 n*) *x*
proof −
have *aux*: *set-pmf* (*Pn p S0 n*) ∩ {(*s*, *a*). *s* = *x*} ≠ {}
using *Xn-def assms* **by** *auto*
have *aux′*: ({(*s*, *a*). *s* = *x*} ∩ *snd* −‘ {*a*}) = {(*x*, *a*)}
by *auto*
show *?thesis*
using *assms aux′*
unfolding *Y-cond-X-def*
by (*auto simp*: *cond-pmf.rep-eq*[*OF aux*] *pmf-map pmf-eq-0-set-pmf measure-pmf.emeasure-eq-measure*
measure-pmf-single)
qed

lemma *Y-cond-X-0*[*simp*]:
assumes $x \in$ *set-pmf S0*
shows *Y-cond-X p S0 0 x* = *p* [] *x*
by (*auto intro*: *pmf-eqI simp*: *assms pmf-Y-cond-X pmf-eq-0-set-pmf*)

lemma *Y-cond-X-markovian*[*simp*]:
assumes *h*: *x* ∈ *Xn* (*mk-markovian p*) *S0 n*
shows *Y-cond-X* (*mk-markovian p*) *S0 n x* = *p n x*
by (*auto intro*!: *pmf-eqI simp*: *pmf-Y-cond-X h Pn-markovian-eq-Xn-bind pmf-eq-0-set-pmf*)

lemma *Pn-eq-Xn-Y-cond*: *Pn p S0 n* = *Xn p S0 n* ⤜ (λ*x*. *map-pmf* (λ*a*. (*x*, *a*)) (*Y-cond-X p S0 n x*))
proof (*induction n*)
case *0*
then show *?case*
by (*auto simp*: *K0-iff intro*: *bind-pmf-cong*)
next
case (*Suc n*)
show *?case*
proof (*intro pmf-eqI*; *safe*)
fix *a* :: *′s*
fix *b* :: *′a*
have *aux′*: *pmf* (*Xn p S0* (*Suc n*) ⤜ (λ*x*. *map-pmf* (*Pair x*) (*Y-cond-X p S0* (*Suc n*) *x*))) (*a*,*b*)
= *measure-pmf.expectation* (*Pn p S0* (*Suc n*)) (λ*x*.
if fst x = *a then pmf* (*Y-cond-X p S0* (*Suc n*) *a*) *b else 0*)
by (*auto intro*!: *Bochner-Integration.integral-cong*[*OF refl*]

```
      simp: Xn-def bind-map-pmf pmf-map pmf-bind measure-pmf-single)
    also have ... = measure-pmf.expectation (Pn p S0 (Suc n))
     (λx. indicator {(s',a'). s' = a} x * (pmf (Pn p S0 (Suc n)) (a, b)
/ pmf (Xn p S0 (Suc n)) a))
    proof (intro Bochner-Integration.integral-cong-AE AE-pmfI)
      fix y
      assume h: y ∈ set-pmf (Pn p S0 (Suc n))
      hence h': fst y ∈ set-pmf (Xn p S0 (Suc n))
        by (metis mult-eq-0-iff pmf-Pn-pair pmf-eq-0-set-pmf)
      show (if fst y = a then pmf (Y-cond-X p S0 (Suc n) a) b else 0)
=
        indicat-real {(s', a'). s' = a} y *
        (pmf (Pn p S0 (Suc n)) (a, b) / pmf (Xn p S0 (Suc n)) a)
        by (auto simp: case-prod-beta' pmf-Y-cond-X[of fst y p S0 (Suc
n) b, OF h'])
    qed auto
    also have ... = measure-pmf.prob (Pn p S0 (Suc n)) {(s',a'). s'
= a} *
      pmf (Pn p S0 (Suc n)) (a, b) / pmf (Xn p S0 (Suc n)) a
      by auto
    also have ... = pmf (Pn p S0 (Suc n)) (a,b)
      using prob-Pn-X Xn-def pmf-Pn-pair pmf-eq-0-set-pmf by fast-
force
    finally show pmf (Pn p S0 (Suc n)) (a, b) = pmf (Xn p S0 (Suc
n)) ⤜
      (λx. map-pmf (Pair x) (Y-cond-X p S0 (Suc n) x))) (a, b)
      by auto
  qed
qed

lemma Pn-eq-Xn-Y-cond':
   Pn p S0 n = Xn p S0 n ⤜ (λs. Y-cond-X p S0 n s ⤜ (λa.
return-pmf (s,a)))
  by (metis K0-def K0-iff Pn-eq-Xn-Y-cond)

lemma Pn-markovian-Suc: Pn (mk-markovian p) S0 (Suc n) =
  Pn (mk-markovian p) S0 n ⤜ (λsa. K0 (p (Suc n)) (K sa))
proof (induction n arbitrary: S0 p)
  case 0
  then show ?case
    by (auto intro: bind-pmf-cong simp: Pn.simps(2) π-Suc-def)
next
  case (Suc n)
  show ?case
    by (auto simp add: Suc bind-assoc-pmf Pn.simps(2)[of - S0] intro:
bind-pmf-cong)
qed
```

6.2.6 Action Process

The distribution of actions.

definition *Yn p S0 n = map-pmf snd* (*Pn p S0 n*)

lemma *Y0*: *Yn p S0 0 = S0* $\gg\!=$ *p* []
by (*simp add*: *Yn-def K0-iff map-bind-pmf map-pmf-comp*)

For markovian policies, the decision rules at each epoch are independent of each other, hence we may express *Yn* solely in terms of *Xn* and the current decision rule.

lemma *Yn-markovian*: *Yn* (*mk-markovian p*) *S0 n = Xn* (*mk-markovian p*) *S0 n* $\gg\!=$ *p n*
proof (*induction n arbitrary*: *p S0*)
case *0*
then show *?case*
by (*auto simp*: *Y0*)
next
case (*Suc n*)
then show *?case*
by (*simp add*: *Xn-def Yn-def map-bind-pmf Suc Pn.simps*(*2*) *bind-assoc-pmf*)
qed

6.3 Restriction to Markovian Policies

abbreviation *as-markovian p S0 n x* ≡
if x ∈ (*Xn p S0 n*) *then Y-cond-X p S0 n x else return-pmf* (*SOME a. a* ∈ *A x*)

For states which cannot occur we choose an arbitrary enabled action, as in this case we cannot make any statements about *Y-cond-X* (a distribution conditioned on an event with probability 0).

lemma *is-*Π_{MR}*-as-markovian*:
assumes *p*: *is-policy p*
shows *as-markovian p S0* ∈ Π_{MR}
proof −
have *aux*: $\bigwedge$*hs s. s* ∈ *set-pmf* (*Xn p S0 hs*) ⟹ *set-pmf* ((*Pn p S0 hs*)) ∩ {(*s*′, *a*). *s*′ = *s*} ≠ {}
by (*simp add*: *measure-pmf-zero-iff*[*symmetric*] *pmf-eq-0-set-pmf*)
thus *?thesis*
using *assms A-ne Pn-in-A* **by** (*auto simp*: *is-dec-def some-in-eq Y-cond-X-def*)
qed

lemma *is-policy-as-markovian*: *is-policy p* ⟹ *is-policy* (*mk-markovian* (*as-markovian p S0*))

using *is-*Π_{MR}*-as-markovian* Π_{MR}*-imp-policies* **by** *auto*

theorem *Pn-as-markovian-eq*: *Pn* (*mk-markovian* (*as-markovian p S0*)) *S0* = *Pn p S0*
proof
fix *n* **show** *Pn* (*mk-markovian* (*as-markovian p S0*)) *S0 n* = *Pn p S0 n*
proof (*induction n*)
case *0*
thus *?case*
by (*auto intro*!: *map-pmf-cong bind-pmf-cong simp*: *K0-def*)
next
case (*Suc n*)
have ⋀*x*. *x* ∈ *Xn p S0* (*Suc n*) ⟹
Y-cond-X (*mk-markovian* (*as-markovian p S0*)) *S0* (*Suc n*) *x* = *Y-cond-X p S0* (*Suc n*) *x*
by (*auto simp*: *Suc.IH Xn-Suc*)
moreover have *Xn* (*mk-markovian* (*as-markovian p S0*)) *S0* (*Suc n*) = *Xn p S0* (*Suc n*)
by (*simp add*: *Xn-Suc Suc.IH*)
ultimately show *Pn* (*mk-markovian* (*as-markovian p S0*)) *S0* (*Suc n*) = *Pn p S0* (*Suc n*)
by (*auto intro*: *bind-pmf-cong simp*: *Pn-eq-Xn-Y-cond*)
qed
qed

6.4 MDPs without Initial Distribution

From now on, we assume a known, deterministic initial state. All results from the previous discussion carry over as we are now in the special case where we the initial state is of the form *return-pmf s*.

definition $\mathcal{T}$ *p s* ≡ *T p* (*return-pmf s*)

lemma $\mathcal{T}$*-eq-return-distr*: $\mathcal{T}$ *p s* =
measure-pmf (*p* [] *s*) ⤜ (λ*a*. *distr* (*T* (*π-Suc p* (*s*,*a*)) (*K* (*s*,*a*))) *S* ((##) (*s*,*a*)))
unfolding $\mathcal{T}$*-def*
by (*subst T-eq-distr*) (*fastforce intro*!: *bind-distr subprob-space.subprob-space-distr*

simp: *K0-iff map-pmf-rep-eq space-subprob-algebra bind-return-pmf*)+

lemma $\mathcal{T}$*-eq-return*:
shows $\mathcal{T}$ *p s* = *do* {
y ← *measure-pmf* (*p* [] *s*);
ω ← *T* (*π-Suc p* (*s*,*y*)) (*K* (*s*,*y*));
return S ((*s*,*y*) ## *ω*)
}

by (*auto simp*: $\mathcal{T}$*-eq-return-distr bind-return-distr' prob-space.not-empty intro*!: *bind-cong*)

lemma $\mathcal{T}$*-return*:
shows *T p S0* = *measure-pmf S0* $\gg\!=$ $\mathcal{T}$ *p*
proof −
have *T p S0* = *measure-pmf S0* $\gg\!=$ (λ*x*. *measure-pmf* (*map-pmf* (*Pair x*) (*p* [] *x*))) $\gg\!=$
(λ*sa*. *distr* (*T* (π*-Suc p sa*) (*K sa*)) (*stream-space* (*count-space UNIV*)) ((##) *sa*))
unfolding *T-eq-distr*[*of p*] *K0-iff measure-pmf-bind*
by *auto*
also have ... = *measure-pmf S0* $\gg\!=$
(λ*x*. *distr* (*measure-pmf* (*p* [] *x*)) (*count-space UNIV*) (*Pair x*) $\gg\!=$
(λ*sa*. *distr* (*T* (π*-Suc p sa*) (*K sa*)) (*stream-space* (*count-space UNIV*)) ((##) *sa*)))
using *measurable-measure-pmf*
by (*subst bind-assoc*[**where** *N* = *count-space UNIV*, **where** *R* = *S*])
(*fastforce intro*!: *prob-space-imp-subprob-space prob-space.prob-space-distr*
simp: *space-subprob-algebra prob-space-measure-pmf map-pmf-rep-eq*)+
also have ... = *measure-pmf S0* $\gg\!=$ $\mathcal{T}$ *p*
by (*subst bind-distr*[**where** *K* = *S*])
(*auto intro*!: *prob-space-imp-subprob-space prob-space.prob-space-distr bind-cong*
simp: *space-subprob-algebra* $\mathcal{T}$*-eq-return-distr*)
finally show *?thesis*.
qed

lemma $\mathcal{T}$*-return-eq*:
$\mathcal{T}$ *p s* = *do* {
a ← *measure-pmf* (*p* [] *s*);
s' ← *measure-pmf* (*K* (*s*,*a*));
w ← *T* (π*-Suc p* (*s*,*a*)) (*return-pmf s'*);
return S ((*s*,*a*)##*w*)
}
unfolding $\mathcal{T}$*-eq-return*
unfolding $\mathcal{T}$*-return*
by (*subst bind-assoc*[*of* - - *S* - *S*]) (*auto simp add*: $\mathcal{T}$*-def* $\mathcal{T}$*-return*[*symmetric*])

lemma $\mathcal{T}$*-eq*:
shows $\mathcal{T}$ *p s* = *do* {
a ← *measure-pmf* (*p* [] *s*);
s' ← *measure-pmf* (*K* (*s*,*a*));
w ← $\mathcal{T}$ (π*-Suc p* (*s*,*a*)) *s'*;
return S ((*s*,*a*)##*w*)
}
by (*subst* $\mathcal{T}$*-return-eq*) (*auto simp add*: $\mathcal{T}$*-def*)

lemma $\mathcal{T}$*-prob-space*[*intro*]: *prob-space* ($\mathcal{T}$ *p s*)
by (*metis* $\mathcal{T}$*-def prob-space-T*)

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

lemma *measurable-ident-Suc'*[*measurable*]:
($\lambda x.\ x$) $\in$ $\mathcal{T}$ (π*-Suc p sa*) *s'* $\rightarrow_M$ *S*
by (*simp add*: $\mathcal{T}$*-def*)

lemma *nn-integral-*$\mathcal{T}$:
fixes *f* :: ('*s* × '*a*) *stream* ⇒ *real*
assumes *f*[*measurable*]: *f* ∈ *borel-measurable S*
shows ($\int^+ t.\ f\ t\ \partial\mathcal{T}\ p\ s$)
= $\int^+ a.\ \int^+ s'.\ \int^+ t'.\ f\ ((s,a)\#\#t')\ \partial\mathcal{T}\ (\pi\text{-}Suc\ p\ (s,a))\ s'\ \partial K\ (s,a)\ \partial p\ []\ s$
proof −
have ($\int^+ t.\ f\ t\ \partial\mathcal{T}\ p\ s$) =
$\int^+ x.\ \int^+ y.\ (f\ y)\ \partial measure\text{-}pmf\ (K\ (s,\ x))\ \gg\!=\ (\lambda s'.\ \mathcal{T}\ (\pi\text{-}Suc\ p\ (s,\ x))\ s'\ \gg\!=\ (\lambda w.\ return\ S\ ((s,\ x)\ \#\#\ w)))\ \partial(p\ []\ s)$
unfolding $\mathcal{T}$*-eq*[*of p*]
by (*subst nn-integral-bind*[*of - S*])
(*auto intro*!: *measure-pmf.bind-in-space subprob-space.bind-in-space simp*: $\mathcal{T}$*-prob-space prob-space-imp-subprob-space*)
also have ... = $\int^+ x.\ \int^+ xa.\ \int^+ y.\ (f\ y)\ \partial\mathcal{T}\ (\pi\text{-}Suc\ p\ (s,\ x))\ xa\ \gg\!=\ (\lambda w.\ return\ S\ ((s,\ x)\ \#\#\ w))$
$\partial measure\text{-}pmf\ (K\ (s,\ x))\ \partial(p\ []\ s)$
by (*subst nn-integral-bind*[*of - S*])
(*auto intro*!: *subprob-space.bind-in-space simp*: $\mathcal{T}$*-prob-space prob-space-imp-subprob-space*)
also have ... = $\int^+ x.\ \int^+ xa.\ \int^+ y.\ (f\ y)\ \partial distr\ (\mathcal{T}\ (\pi\text{-}Suc\ p\ (s,\ x))\ xa)\ S\ ((\#\#)\ (s,\ x))\ \partial measure\text{-}pmf\ (K\ (s,\ x))\ \partial(p\ []\ s)$
by (*auto simp add*: *bind-return-distr'* $\mathcal{T}$*-prob-space prob-space.not-empty*)
also have ... = $\int^+ x.\ \int^+ xa.\ \int^+ xa.\ (f\ ((s,\ x)\ \#\#\ xa))\ \partial\mathcal{T}\ (\pi\text{-}Suc\ p\ (s,\ x))\ xa\ \partial measure\text{-}pmf\ (K\ (s,\ x))\ \partial(p\ []\ s)$
by (*auto simp*: *nn-integral-distr*)
finally show *?thesis*.
qed

lemma *integral-*$\mathcal{T}$:
fixes *f* :: ('*s* × '*a*) *stream* ⇒ *real*
assumes *f-bounded*: $\bigwedge x.\ |f\ x| \le B$
assumes *f*[*measurable*]: *f* ∈ *borel-measurable S*
shows ($\int t.\ f\ t\ \partial\mathcal{T}\ p\ s$)
= $\int a.\ \int s'.\ \int t'.\ f\ ((s,a)\#\#t')\ \partial\mathcal{T}\ (\pi\text{-}Suc\ p\ (s,a))\ s'\ \partial K\ (s,a)\ \partial p\ []\ s$
unfolding $\mathcal{T}$*-def integral-T*[*OF f-bounded f*] *K0-iff bind-return-pmf*

unfolding $\mathcal{T}$-*return*[*of* π-*Suc p* -] *integral-map-pmf*
using $\mathcal{T}$-*return*[*of* π-*Suc p* -, *symmetric*]
by (*subst integral-bind*[*OF* - *f-bounded*, **where** $B' = 1$, **where** $K = S$])
(*auto simp*: $\mathcal{T}$-*def intro*: *prob-space.emeasure-le-1*)

lemma *integrable-*$\mathcal{T}$*-bounded*[*intro*]:
fixes $f :: ('s \times 'a)$ *stream* $\Rightarrow$ $'d :: \{$*second-countable-topology*,*banach*$\}$
assumes f[*measurable*]: $f \in$ *borel-measurable S*
assumes *b*: *bounded* (*range f*)
shows *integrable* ($\mathcal{T}$ *p s*) *f*
using *b*
by (*auto simp*: *prob-space.finite-measure* $\mathcal{T}$*-prob-space bounded-iff intro*!: *finite-measure.integrable-const-bound*)

definition $Pn'\ p\ s = Pn\ p$ (*return-pmf s*)
definition $Xn'\ p\ s = Xn\ p$ (*return-pmf s*)
definition $Yn'\ p\ s = Yn\ p$ (*return-pmf s*)
definition $K0'\ d\ s \equiv$ *map-pmf* ($\lambda a.\ (s,\ a)$) (*d s*)

definition *K-st d s* $\equiv$ *d s* $\gg\!=$ ($\lambda a.$ *K* (*s*,*a*))

lemma *pmf-K-st*: *pmf* (*K-st d s*) *t* $= \int a.$ *pmf* (*K*(*s*, *a*)) *t* ∂*d s*
unfolding *K-st-def* **by** (*auto simp*: *pmf-bind*)

K-st defines the distribution over the successor states for a given decision rule and state. It is mostly useful for markovian policies, as the information which action was selected is lost.

lemma *P0'*[*simp*]: $Pn'\ p\ s\ 0 = K0'\ (p\ [])\ s$
by (*simp add*: *Pn'-def K0'-def K0-iff bind-return-pmf*)

lemma *X0'*[*simp*]: $Xn'\ p\ s\ 0 =$ *return-pmf s*
using *X0 Xn'-def* **by** *auto*

lemma *Pn-return-pmf*: *S0* $\gg\!=$ ($\lambda s'.$ *Pn p* (*return-pmf s'*) *n*) = *Pn p S0 n*
by (*induction n arbitrary*: *p S0*)
(*auto intro*: *bind-pmf-cong simp add*: *Pn.simps*(*2*) *K0-def bind-assoc-pmf bind-return-pmf*)

lemma *PSuc'*: $Pn'\ p\ s$ (*Suc n*) = *K0'* (*p* []) *s* $\gg\!=$ ($\lambda sa.$ *K sa* $\gg\!=$ ($\lambda s'.$ Pn' (π-*Suc p sa*) s' *n*))
unfolding *Pn'-def*
by (*auto intro*!: *bind-pmf-cong*
simp: *Pn.simps*(*2*) *Pn-return-pmf K0-iff K0'-def bind-return-pmf map-bind-pmf bind-map-pmf*)

lemma *PSuc'-markovian*:
Pn' (*mk-markovian p*) *s* (*Suc n*) = *K-st* (*p 0*) *s* $\gg\!=$ ($\lambda s'.$ Pn'

(*mk-markovian* (*p* ∘ *Suc*)) *s'* *n*)
unfolding *PSuc'*
by (*auto simp*: *bind-map-pmf bind-assoc-pmf comp-def K0'-def K-st-def intro*!: *bind-pmf-cong*)

lemma *Xn'-Suc*: *Xn' p s* (*Suc n*) = *Pn' p s n* ⤜ *K*
by (*auto simp*: *Xn-Suc Xn'-def Pn'-def*)

lemma *Xn'-Pn'*: *Xn' p s n* = *map-pmf fst* (*Pn' p s n*)
by (*simp add*: *Xn-def Xn'-def Pn'-def*)

lemma *Suc-Xn'*: *Xn' p s* (*Suc n*) = *p* [] *s* ⤜ (λ*a*. *K* (*s*,*a*) ⤜ (λ*s'*. *Xn'* (*π-Suc p* (*s*,*a*)) *s'* *n*))
by (*auto simp*: *Xn'-Pn' map-bind-pmf bind-map-pmf PSuc' K0'-def*)

lemma *Suc-Xn'-markovian*:
Xn' (*mk-markovian p*) *s* (*Suc n*) = *K-st* (*p* 0) *s* ⤜ (λ*s'*. *Xn'* (*mk-markovian* (λ*n*. *p* (*Suc n*))) *s'* *n*)
by (*auto simp*: *K-st-def bind-assoc-pmf Suc-Xn'*)

lemma *Xn'-split*: *Xn'* (*mk-markovian p*) *s* (*n* + *m*) =
Xn' (*mk-markovian p*) *s n* ⤜ (λ*s*. *Xn'* (*mk-markovian* (λ*i*. *p* (*i* + *n*))) *s m*)
by (*induction n arbitrary*: *p s*) (*auto intro*!: *bind-pmf-cong simp*: *bind-assoc-pmf bind-return-pmf Suc-Xn'*)

lemma *Yn'-markovian*: *Yn'* (*mk-markovian p*) *s n* = *Xn'* (*mk-markovian p*) *s n* ⤜ *p n*
unfolding *Yn'-def Xn'-def Yn-markovian* **by** *simp*

lemma *Pn'-markovian-eq-Xn'-bind*: *Pn'* (*mk-markovian p*) *s n* = *Xn'* (*mk-markovian p*) *s n* ⤜ *K0'* (*p n*)
unfolding *Xn'-def Pn'-def K0'-def K0-iff Pn-markovian-eq-Xn-bind*
by *simp*

lemma *Pn'-eq-T*: *measure-pmf* (*Pn' p s n*) = *distr* (*T p s*) (*count-space UNIV*) (λ*t*. *t* !! *n*)
by (*auto simp*: *T-def Pn'-def Pn-eq-T*)

end
end

theory *MDP-reward*
imports
Bounded-Functions
MDP-reward-Util
Blinfun-Util
MDP-disc

begin

7 Markov Decision Processes with Rewards

locale *MDP-reward* = *discrete-MDP A K*
  for
    A **and**
    K :: *'s* ::*countable* × *'a* ::*countable* ⇒ *'s pmf* +
  fixes
    r :: (*'s* × *'a*) ⇒ *real* **and**
    l :: *real*
  assumes
    zero-le-disc [*simp*]: *0* ≤ *l* **and**
    r-bounded: *bounded* (*range r*)
begin

This extension to the basic MDPs is formalized with another locale. It assumes the existence of a reward function r which takes a state-action pair to a real number. We assume that the function is bounded *r-bounded*.

Furthermore, we fix a discounting factor l, where $0 \leq l \wedge l < 1$.

7.1 Util

7.1.1 Basic Properties of rewards

lemma *r-bfun*: *r* ∈ *bfun*
  using *r-bounded*
  by *auto*

lemma *r-bounded'*: *bounded* (*r* ‘ *X*)
  by (*auto intro*: *r-bounded bounded-subset*)

definition $r_M = (\bigsqcup sa.\ |r\ sa|)$

lemma *abs-r-le-r*$_M$: |*r sa*| ≤ r_M
  using *bounded-norm-le-SUP-norm r-bounded* r_M*-def* **by** *fastforce*

lemma *abs-*r_M*-eq-*r_M [*simp*]: |r_M| = r_M
  using *abs-r-le-*r_M **by** *fastforce*

lemma r_M*-nonneg*: *0* ≤ r_M
  using *abs-*r_M*-eq-*r_M **by** *linarith*

lemma *measurable-r-nth* [*measurable*]: (λ*t*. *r* (*t* !! *i*)) ∈ *borel-measurable S*
  by *measurable*

lemma *integrable-r-nth* [*simp*]: *integrable* ($\mathcal{T}$ *p s*) ($\lambda t.$ *r* (*t* !! *i*))
by (*fastforce simp*: *bounded-iff intro*: *abs-r-le-r*$_M$)

lemma *expectation-abs-r-le*: *measure-pmf.expectation d* ($\lambda a.$ |*r* (*s*, *a*)|)
$\leq$ *r*$_M$
using *abs-r-le-r*$_M$
by (*fastforce intro*!: *measure-pmf.integral-le-const measure-pmf.integrable-const-bound*)

lemma *abs-exp-r-le*: |*measure-pmf.expectation d r*| $\leq$ *r*$_M$
using *abs-r-le-r*$_M$
by (*fastforce intro*!: *measure-pmf.integral-le-const order.trans*[*OF integral-abs-bound*] *measure-pmf.integrable-const-bound*)

7.1.2 Infinite disounted sums

lemma *abs-disc-eq*[*simp*]: |*l* ^ *i* $*$ *x*| = *l* ^ *i* $*$ |*x*|
by (*auto simp*: *abs-mult*)

lemma *norm-l-pow-eq*[*simp*]: *norm* (*l*^*t* $*_R$ *F*) = *l*^*t* $*$ *norm F*
by *auto*

7.2 Total Reward for Single Traces

abbreviation ν*-trace-fin t N* $\equiv \sum i < N.$ *l* ^ *i* $*$ *r* (*t* !! *i*)
abbreviation ν*-trace t* $\equiv \sum i.$ *l* ^ *i* $*$ *r* (*t* !! *i*)

lemma *abs-*ν*-trace-fin-le*: |ν*-trace-fin t N*| $\leq$ ($\sum i < N.$ *l*^*i* $*$ *r*$_M$)
by (*auto intro*!: *sum-mono order.trans*[*OF sum-abs*] *mult-left-mono abs-r-le-r*$_M$)

lemma *measurable-suminf-reward*[*measurable*]: ν*-trace* $\in$ *borel-measurable S*
by *measurable*

lemma *integrable-*ν*-trace-fin*: *integrable* ($\mathcal{T}$ *p s*) ($\lambda t.$ ν*-trace-fin t N*)
by (*fastforce simp*: *bounded-iff intro*: *abs-*ν*-trace-fin-le*)

context
fixes *p* :: (*'s*, *'a*) *pol*
begin

7.3 Expected Finite-Horizon Discounted Reward

definition ν*-fin n s* = $\int$ *t.* ν*-trace-fin t n* $\partial\mathcal{T}$ *p s*

lemma *abs-*ν*-fin-le*: |ν*-fin N s*| $\leq$ ($\sum i$<*N.* *l*^*i* $*$ *r*$_M$)
unfolding ν*-fin-def*
using *abs-*ν*-trace-fin-le*

by (*fastforce intro*!: *prob-space.integral-le-const order-trans*[*OF integral-abs-bound*])

lemma *ν-fin-bfun*: $(\lambda s.\ \nu\text{-}fin\ N\ s) \in bfun$
by (*auto intro*!: *abs-ν-fin-le*)

lift-definition ν_b*-fin* :: *nat* ⇒ *'s* ⇒$_b$ *real* **is** *ν-fin*
using *ν-fin-bfun* **.**

lemma *ν-fin-Suc*[*simp*]: *ν-fin* (*Suc n*) *s* = *ν-fin n s* + *l* ^ *n* ∗ $\int$ *t*. *r* (*t* !! *n*) $\partial\mathcal{T}$ *p s*
by (*simp add*: *ν-fin-def*)

lemma *ν-fin-zero*[*simp*]: *ν-fin 0 s* = *0*
by (*simp add*: *ν-fin-def*)

lemma *ν-fin-eq-Pn*: *ν-fin n s* = ($\sum$ *i*<*n*. *l*^*i* ∗ *measure-pmf.expectation* (*Pn' p s i*) *r*)
by (*induction n*) (*auto simp*: *Pn'-eq-*$\mathcal{T}$ *integral-distr*)
end

7.4 Expected Total Discounted Reward

definition *ν p s* = *lim* (λ*n*. *ν-fin p n s*)

lemmas *ν-eq-lim* = *ν-def*

lemma *ν-eq-Pn*: *ν p s* = ($\sum$ *i*. *l*^*i* ∗ *measure-pmf.expectation* (*Pn' p s i*) *r*)
by (*simp add*: *ν-fin-eq-Pn ν-eq-lim suminf-eq-lim*)

7.5 Reward of a Decision Rule

context
fixes *d* :: (*'s*, *'a*) *dec*
begin
abbreviation *r-dec s* ≡ $\int$ *a*. *r* (*s*, *a*) ∂*d s*

lemma *abs-r-dec-le*: |*r-dec s*| ≤ r_M
using *expectation-abs-r-le integral-abs-bound order-trans* **by** *fast*

lemma *r-dec-eq-r-K0*: *r-dec s* = *measure-pmf.expectation* (*K0' d s*) *r*
by (*simp add*: *K0'-def*)

lemma *r-dec-bfun*: *r-dec* ∈ *bfun*
using *abs-r-dec-le* **by** (*auto intro*!: *bfun-normI*)

lift-definition *r-dec*$_b$ :: *'s* ⇒$_b$ *real* **is** *r-dec*
using *r-dec-bfun* **.**

declare *r-dec$_b$.rep-eq*[*simp*] *bfun.Bfun-inverse*[*simp*]

lemma *norm-r-dec-le*: *norm r-dec$_b$* $\leq r_M$
by (*simp add*: *abs-r-dec-le norm-bound*)
end

lemma *r-dec-det* [*simp*]: *r-dec* (*mk-dec-det d*) *s* = *r* (*s*, *d s*)
unfolding *mk-dec-det-def* **by** *auto*

7.6 Transition Probability Matrix for MDPs

context
fixes *p* :: *nat* ⇒ ('*s*, '*a*) *dec*
begin
definition $\mathcal{P}_X$ *n* = *push-exp* (λ*s*. *Xn'* (*mk-markovian p*) *s n*)

lemma $\mathcal{P}_X$*-0*[*simp*]: $\mathcal{P}_X$ *0* = *id*
by (*simp add*: $\mathcal{P}_X$*-def*)

lemma $\mathcal{P}_X$*-bounded-linear*[*simp*]: *bounded-linear* ($\mathcal{P}_X$ *t*)
unfolding $\mathcal{P}_X$*-def* **by** *simp*

lemma *norm-*$\mathcal{P}_X$ [*simp*]: *onorm* ($\mathcal{P}_X$ *t*) = *1*
unfolding $\mathcal{P}_X$*-def* **by** *simp*

lemma *norm-*$\mathcal{P}_X$*-apply*[*simp*]: *norm* ($\mathcal{P}_X$ *n x*) ≤ *norm x*
using *onorm*[*OF* $\mathcal{P}_X$*-bounded-linear*] **by** *simp*

lemma $\mathcal{P}_X$*-bound-r*: *norm* ($\mathcal{P}_X$ *t* (*r-dec$_b$* (*p t*))) ≤ r_M
using *norm-*$\mathcal{P}_X$*-apply norm-r-dec-le order.trans* **by** *blast*

lemma $\mathcal{P}_X$*-bounded-r*: *bounded* (*range* (λ*t*. ($\mathcal{P}_X$ *t* (*r-dec$_b$* (*p t*)))))
using $\mathcal{P}_X$*-bound-r* **by** (*auto intro*!: *boundedI*)

end

lemma *ν-fin-elem*: *ν-fin* (*mk-markovian p*) *n s* = (∑ *i*<*n*. *l*^*i* ∗ $\mathcal{P}_X$ *p i* (*r-dec$_b$* (*p i*)) *s*)
unfolding $\mathcal{P}_X$*-def ν-fin-eq-Pn Pn'-markovian-eq-Xn'-bind measure-pmf-bind*
using *measure-pmf-in-subprob-algebra abs-r-le-*r_M
by (*subst integral-bind*) (*auto simp*: *r-dec-eq-r-K0*)

lemma *ν$_b$-fin-eq-*$\mathcal{P}_X$: *ν$_b$-fin* (*mk-markovian p*) *n* = (∑ *i*<*n*. *l*^*i* ∗$_R$ $\mathcal{P}_X$ *p i* (*r-dec$_b$* (*p i*)))
by (*auto simp*: *ν-fin-elem sum-apply-bfun ν$_b$-fin.rep-eq*)

lemma *ν-fin-eq-*$\mathcal{P}_X$: *ν-fin* (*mk-markovian p*) *n* = (∑ *i*<*n*. *l*^*i* ∗$_R$ $\mathcal{P}_X$ *p i* (*r-dec$_b$* (*p i*)))
by (*metis ν$_b$-fin.rep-eq ν$_b$-fin-eq-*$\mathcal{P}_X$)

$\mathcal{P}_1$ d v defines for each state the expected value of v after taking a single step in the MDP according to the decision rule d.

context
fixes d :: $('s, 'a)$ *dec*
begin
lift-definition $\mathcal{P}_1$ :: $('s \Rightarrow_b real) \Rightarrow_L ('s \Rightarrow_b real)$ **is** *push-exp* (*K-st* d)
using *push-exp-bounded-linear* **.**

lemma $\mathcal{P}_1$*-bfun-one* [*simp*]:$\mathcal{P}_1$ *1* = *1*
by (*auto simp*: $\mathcal{P}_1$*.rep-eq*)

lemma $\mathcal{P}_1$*-pow-bfun-one* [*simp*]: ($\mathcal{P}_1$ ^^ t) *1* = *1*
by (*induction t*) *auto*

lemma $\mathcal{P}_1$*-pow*: *blinfun-apply* ($\mathcal{P}_1$ ^^ n) = *blinfun-apply* $\mathcal{P}_1$ ^^ n
by (*induction n*) *auto*

lemma *norm-*$\mathcal{P}_1$ [*simp*]: *norm* $\mathcal{P}_1$ = *1*
by (*simp add*: *norm-blinfun.rep-eq* $\mathcal{P}_1$*.rep-eq*)
end

lemma $\mathcal{P}_X$*-Suc*: $\mathcal{P}_X$ p (*Suc* n) v = $\mathcal{P}_1$ (p *0*) (($\mathcal{P}_X$ ($\lambda n.$ p (*Suc* n))) n) v)
unfolding $\mathcal{P}_X$*-def* $\mathcal{P}_1$*.rep-eq*
by (*fastforce intro*!: *abs-le-norm-bfun integral-bind*[**where** K = *count-space UNIV*]
simp: *measure-pmf-in-subprob-algebra measure-pmf-bind Suc-Xn'-markovian*)

lemma $\mathcal{P}_X$*-Suc'*: $\mathcal{P}_X$ p (*Suc* n) v = $\mathcal{P}_X$ p n ($\mathcal{P}_1$ (p n) v)
proof (*induction n arbitrary*: p)
case *0*
thus *?case*
by (*simp add*: $\mathcal{P}_X$*-Suc*)
next
case (*Suc n*)
thus *?case*
by (*metis* $\mathcal{P}_X$*-Suc*)
qed

lemma $\mathcal{P}_X$*-const*: $\mathcal{P}_X$ (λ-. d) n = $\mathcal{P}_1$ d ^^ n
by (*induction n*) (*auto simp add*: $\mathcal{P}_1$*-pow* $\mathcal{P}_X$*-Suc*)

lemma $\mathcal{P}_X$*-sconst*: $\mathcal{P}_X$ (λ-. p) n = $\mathcal{P}_1$ p ^^n
using $\mathcal{P}_X$*-const***.**

lemma *norm-P-n*[*simp*]: *onorm* ($\mathcal{P}_1$ d ^^ n) = *1*
using *norm-*$\mathcal{P}_X$[*of* λ-. d] **by** (*auto simp*: $\mathcal{P}_X$*-sconst*)

lemma *norm-$\mathcal{P}_1$-pow* [*simp*]: *norm* ($\mathcal{P}_1$ *d* ^^ *t*) = *1*
by (*simp add*: *norm-blinfun.rep-eq*)

lemma *$\mathcal{P}_X$-Suc-n-elem*: $\mathcal{P}_X$ *p n* ($\mathcal{P}_1$ (*p n*) *v*) = $\mathcal{P}_X$ *p* (*Suc n*) *v*
using *$\mathcal{P}_X$-Suc' $\mathcal{P}_1$.rep-eq* **by** *auto*

lemma *$\mathcal{P}_1$-eq-$\mathcal{P}_X$-one*: *blinfun-apply* ($\mathcal{P}_1$ (*p 0*)) = $\mathcal{P}_X$ *p 1*
by (*auto simp*: *$\mathcal{P}_X$-Suc' $\mathcal{P}_1$.rep-eq*)

lemma *$\mathcal{P}_1$-pos*: *0* ≤ *u* ⟹ *0* ≤ $\mathcal{P}_1$ *d u*
by (*auto simp*: *$\mathcal{P}_1$.rep-eq less-eq-bfun-def*)

lemma *$\mathcal{P}_1$-nonneg*: *nonneg-blinfun* ($\mathcal{P}_1$ *d*)
by (*simp add*: *$\mathcal{P}_1$-pos nonneg-blinfun-def*)

lemma *$\mathcal{P}_1$-n-pos*: *0* ≤ *u* ⟹ *0* ≤ ($\mathcal{P}_1$ *d* ^^ *n*) *u*
by (*induction n*) (*auto simp*: *$\mathcal{P}_1$.rep-eq less-eq-bfun-def*)

lemma *$\mathcal{P}_1$-n-nonneg*: *nonneg-blinfun* ($\mathcal{P}_1$ *d* ^^ *n*)
by (*simp add*: *$\mathcal{P}_1$-n-pos nonneg-blinfun-def*)

lemma *$\mathcal{P}_1$-n-disc-pos*: *0* ≤ *u* ⟹ *0* ≤ (*l*^*n* *$_R$ $\mathcal{P}_1$ *d* ^^*n*) *u*
by (*auto simp*: *$\mathcal{P}_1$-n-pos scaleR-nonneg-nonneg blinfun.scaleR-left*)

lemma *$\mathcal{P}_1$-sum-pos*: *0* ≤ *u* ⟹ *0* ≤ (∑ *t*≤*n*. *l*^*t* *$_R$ ($\mathcal{P}_1$ *d* ^^ *t*)) *u*
using *$\mathcal{P}_1$-n-pos $\mathcal{P}_1$-pos*
by (*induction n*) (*auto simp*: *blinfun.add-left blinfun.scaleR-left scaleR-nonneg-nonneg*)

lemma *$\mathcal{P}_1$-sum-ge*:
assumes *0* ≤ *u*
shows *u* ≤ (∑ *t*≤*n*. *l*^*t* *$_R$ $\mathcal{P}_1$ *d* ^^*t*) *u*
using *$\mathcal{P}_1$-n-disc-pos*[*OF assms*, *of Suc* -]
by (*induction n*) (*auto intro*: *add-increasing2 simp add*: *blinfun.add-left*)

7.7 The Bellman Operator

definition *L d v* ≡ *r-dec$_b$ d* + *l* *$_R$ $\mathcal{P}_1$ *d v*

lemma *norm-L-le*: *norm* (*L d v*) ≤ *r_M* + *l* ∗ *norm v*
using *norm-blinfun*[*of $\mathcal{P}_1$ d*] *norm-$\mathcal{P}_1$ norm-r-dec-le*
by (*auto intro*!: *norm-add-rule-thm mult-left-mono simp*: *L-def*)

lemma *abs-L-le*: |*L d v s*| ≤ *r_M* + *l* ∗ *norm v*
using *order.trans*[*OF norm-le-norm-bfun norm-L-le*] **by** *auto*

7.7.1 Bellman Operator for Single Actions

abbreviation *L_a a v s* ≡ *r* (*s*, *a*) + *l* ∗ *measure-pmf.expectation* (*K* (*s*,*a*)) *v*

lemma L_a*-le*:
fixes $v :: 's \Rightarrow_b real$
shows $|L_a\ a\ v\ s| \leq r_M + l * norm\ v$
using *abs-r-le-*r_M
by (*fastforce intro*: *order-trans*[*OF abs-triangle-ineq*] *order-trans*[*OF integral-abs-bound*]
add-mono mult-mono measure-pmf.integral-le-const abs-le-norm-bfun

simp: *abs-mult*)

lemma L_a*-bounded*:
bounded (*range* ($\lambda a.\ L_a\ a$ (*apply-bfun v*) *s*))
using L_a*-le* **by** (*auto intro*!: *boundedI*)

lemma L_a*-int*:
fixes $d :: 'a\ pmf$ **and** $v :: 's \Rightarrow_b real$
shows $(\int a.\ L_a\ a\ v\ s\ \partial d) = (\int a.\ r\ (s,\ a)\ \partial d) + l * \int a.\ \int s'.\ v\ s'\ \partial K\ (s,\ a)\ \partial d$
proof (*subst Bochner-Integration.integral-add*)
show *integrable d* ($\lambda a.\ r\ (s,\ a)$)
using *abs-r-le-*r_M **by** (*fastforce intro*!: *bounded-integrable simp*: *bounded-iff*)
show *integrable d* ($\lambda a.\ l * \int s'.\ v\ s'\ \partial K\ (s,\ a)$)
by (*intro bounded-integrable*)
(*auto intro*!: *mult-mono order-trans*[*OF integral-abs-bound*] *boundedI*[*of - l * norm v*]
measure-pmf.integral-le-const simp: *abs-le-norm-bfun abs-mult*)
qed *auto*

lemma *L-eq-*L_a: $L\ d\ v\ s = measure\text{-}pmf.expectation\ (d\ s)\ (\lambda a.\ L_a\ a\ v\ s)$
unfolding L_a*-int L-def K-st-def* $\mathcal{P}_1$*.rep-eq*
by (*auto simp*: *measure-pmf-bind integral-measure-pmf-bind*[**where** $B = norm\ v$] *abs-le-norm-bfun*)

lemma *L-eq-*L_a*-det*: $L\ (mk\text{-}dec\text{-}det\ d)\ v\ s = L_a\ (d\ s)\ v\ s$
by (*auto simp*: *L-eq-*L_a *mk-dec-det-def*)

lemma L_a*-eq-L*: *measure-pmf.expectation p* ($\lambda a.\ L_a\ a$ (*apply-bfun v*) *s*) =
$L\ (\lambda t.\ if\ t = s\ then\ p\ else\ return\text{-}pmf\ (SOME\ a.\ a \in A\ t))\ v\ s$
unfolding *L-eq-*L_a **by** *auto*

lemma *L-le*: $L\ d\ v\ s \leq r_M + l * norm\ v$
unfolding *L-def*
using *norm-*$\mathcal{P}_1$ *norm-blinfun*[*of* ($\mathcal{P}_1$ *d*)] *abs-r-dec-le*
by (*fastforce intro*: *order-trans*[*OF le-norm-bfun*] *add-mono mult-left-mono dest*: *abs-le-D1*)

lemma L_a*-le'*: $L_a\ a\ (apply\text{-}bfun\ v)\ s \leq r_M + l * norm\ v$
using L_a*-le abs-le-D1* **by** *blast*

7.8 Optimality Equations

definition $\mathcal{L}\ (v :: 's \Rightarrow_b real)\ s = (\bigsqcup d \in D_R.\ L\ d\ v\ s)$

lemma $\mathcal{L}$*-bfun*: $\mathcal{L}\ v \in bfun$
unfolding $\mathcal{L}$*-def* **using** *abs-L-le ex-dec* **by** (*fastforce intro*!: *cSup-abs-le bfun-normI*)

lift-definition $\mathcal{L}_b :: ('s \Rightarrow_b real) \Rightarrow 's \Rightarrow_b real$ **is** $\mathcal{L}$
using $\mathcal{L}$*-bfun* **.**

lemma *L-bounded*[*simp*, *intro*]: *bounded* (*range* ($\lambda p.\ L\ p\ v\ s$))
using *abs-L-le* **by** (*auto intro*!: *boundedI*)

lemma *L-bounded'*[*simp*, *intro*]: *bounded* (($\lambda p.\ L\ p\ v\ s$) ' X)
by (*auto intro*: *bounded-subset*)

lemma *L-bdd-above*[*simp*, *intro*]: *bdd-above* (($\lambda p.\ L\ p\ v\ s$) ' X)
by (*auto intro*: *bounded-imp-bdd-above*)

lemma *L-le-*$\mathcal{L}_b$: *is-dec d* $\Longrightarrow L\ d\ v \leq \mathcal{L}_b\ v$
by (*fastforce simp*: $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-def intro*!: *cSUP-upper*)

7.8.1 Equivalences involving $\mathcal{L}_b$

lemma *SUP-step-MR-eq*:
$\mathcal{L}\ v\ s = (\bigsqcup pa \in \{pa.\ set\text{-}pmf\ pa \subseteq A\ s\}.\ (\int a.\ L_a\ a\ v\ s\ \partial measure\text{-}pmf\ pa))$
unfolding $\mathcal{L}$*-def*
proof (*intro antisym*)
show $(\bigsqcup d \in D_R.\ L\ d\ v\ s) \leq (\bigsqcup pa \in \{pa.\ set\text{-}pmf\ pa \subseteq A\ s\}.\ \int a.\ L_a\ a\ v\ s\ \partial measure\text{-}pmf\ pa)$
proof (*rule cSUP-mono*)
show $D_R \neq \{\}$
using D_R*-ne* **.**
next show *bdd-above* (($\lambda pa.\ \int a.\ L_a\ a\ v\ s\ \partial measure\text{-}pmf\ pa$) ' $\{pa.\ set\text{-}pmf\ pa \subseteq A\ s\}$)
using L_a*-bounded* L_a*-le*
by (*auto intro*!: *order-trans*[*OF integral-abs-bound*]
bounded-imp-bdd-above boundedI[**where** $B = r_M + l * norm\ v$]
measure-pmf.integral-le-const bounded-integrable)
next show $\exists m \in \{pa.\ set\text{-}pmf\ pa \subseteq A\ s\}.\ L\ n\ v\ s \leq \int a.\ L_a\ a\ v\ s\ \partial measure\text{-}pmf\ m$ **if** $n \in D_R$ **for** n
using *that*
by (*fastforce simp*: *L-eq-*L_a L_a*-int is-dec-def*)

qed
next
have *aux*: $\{pa.\ set\text{-}pmf\ pa \subseteq A\ s\} \neq \{\}$
using D_R*-ne is-dec-def* **by** *auto*
show $(\bigsqcup pa \in \{pa.\ set\text{-}pmf\ pa \subseteq A\ s\}.\ \int a.\ L_a\ a\ v\ s\ \partial measure\text{-}pmf\ pa) \leq (\bigsqcup d \in D_R.\ L\ d\ v\ s)$
proof (*intro cSUP-least*[*OF aux*] *cSUP-upper2*)
fix *n*
assume *h*: $n \in \{pa.\ set\text{-}pmf\ pa \subseteq A\ s\}$
let $?p = (\lambda s'.\ if\ s = s'\ then\ n\ else\ SOME\ a.\ set\text{-}pmf\ a \subseteq A\ s')$
have *aux*: $\exists a.\ set\text{-}pmf\ a \subseteq A\ sa$ **for** *sa*
using *ex-dec is-dec-def* **by** *blast*
show $?p \in D_R$
unfolding *is-dec-def* **using** *h someI-ex*[*OF aux*] **by** *auto*
thus $(\int a.\ L_a\ a\ v\ s\ \partial n) \leq L\ ?p\ v\ s$
by (*auto simp*: *L-eq-*L_a)
show *bdd-above* $((\lambda d.\ L\ d\ v\ s)\ `\ D_R)$
by (*fastforce intro*!: *bounded-imp-bdd-above simp*: *bounded-def*)
next
qed
qed

lemma $\mathcal{L}_b$*-eq-SUP-*L_a: $\mathcal{L}_b\ v\ s = (\bigsqcup p \in \{p.\ set\text{-}pmf\ p \subseteq A\ s\}.\ \int a.\ L_a\ a\ v\ s\ \partial measure\text{-}pmf\ p)$
using *SUP-step-MR-eq* $\mathcal{L}_b$*.rep-eq* **by** *presburger*

lemma *SUP-step-det-eq*: $(\bigsqcup d \in D_D.\ L\ (mk\text{-}dec\text{-}det\ d)\ v\ s) = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
proof (*intro antisym cSUP-mono*)
show *bdd-above* $((\lambda a.\ L_a\ a\ v\ s)\ `\ A\ s)$
using L_a*-bounded* **by** (*fastforce intro*!: *bounded-imp-bdd-above simp*: *bounded-def*)
show *bdd-above* $((\lambda d.\ L\ (mk\text{-}dec\text{-}det\ d)\ v\ s)\ `\ D_D)$
by (*auto intro*!: *bounded-imp-bdd-above boundedI abs-L-le*)
show $\exists m \in A\ s.\ L\ (mk\text{-}dec\text{-}det\ n)\ v\ s \leq L_a\ m\ v\ s$ **if** $n \in D_D$ **for** *n*
using *that is-dec-det-def* **by** (*auto simp*: *L-eq-*L_a*-det intro*: *bexI*[*of - n s*])
show $\exists m \in D_D.\ L_a\ n\ v\ s \leq L\ (mk\text{-}dec\text{-}det\ m)\ v\ s$ **if** $n \in A\ s$ **for** *n*
using *that A-ne*
by (*fastforce simp*: *L-eq-*L_a*-det is-dec-det-def some-in-eq*
intro!: *bexI*[*of - λs'. if s = s' then - else SOME a. a ∈ A s'*])
qed (*auto simp*: *A-ne*)

lemma *integrable-*L_a: *integrable* (*measure-pmf x*) $(\lambda a.\ L_a\ a\ (apply\text{-}bfun\ v)\ s)$
proof (*intro Bochner-Integration.integrable-add integrable-mult-right*)
show *integrable* (*measure-pmf x*) $(\lambda x.\ r\ (s,\ x))$
using *abs-r-le-*r_M
by (*auto intro*: *measure-pmf.integrable-const-bound*[*of - *r_M])

next
show *integrable* (*measure-pmf x*) (λ*x*. *measure-pmf.expectation* (*K* (*s*, *x*)) *v*)
by (*auto intro*!: *bounded-integrable boundedI order.trans*[*OF integral-abs-bound*]
measure-pmf.integral-le-const abs-le-norm-bfun)
qed

lemma *SUP-L_a-eq-det*:
fixes *v* :: *'s* ⇒$_b$ *real*
shows (⨆ *p*∈{*p*. *set-pmf p* ⊆ *A s*}. ∫ *a*. *L$_a$ a v s* ∂*measure-pmf p*) = (⨆ *a*∈*A s*. *L$_a$ a v s*)
proof (*intro antisym*)
show (⨆ *pa*∈{*pa*. *set-pmf pa* ⊆ *A s*}. *measure-pmf.expectation pa* (λ*a*. *L$_a$ a v s*))
≤ (⨆ *a*∈*A s*. *L$_a$ a v s*)
using *ex-dec is-dec-def integrable-L$_a$ A-ne L$_a$-bounded*
by (*fastforce intro*: *bounded-range-subset intro*!: *cSUP-least lemma-4-3-1*)
show (⨆ *a*∈*A s*. *L$_a$ a v s*) ≤ (⨆ *p*∈{*p*. *set-pmf p* ⊆ *A s*}. ∫ *a*. *L$_a$ a v s* ∂*measure-pmf p*)
unfolding *SUP-step-MR-eq*[*symmetric*] *SUP-step-det-eq*[*symmetric*] *$\mathcal{L}$-def*
using *ex-dec-det* **by** (*fastforce intro*!: *cSUP-mono*)
qed

lemma *$\mathcal{L}$-eq-SUP-det*: $\mathcal{L}$ *v s* = (⨆ *d* ∈ *D$_D$*. *L* (*mk-dec-det d*) *v s*)
using *SUP-step-MR-eq SUP-step-det-eq SUP-L$_a$-eq-det* **by** *auto*

lemma *$\mathcal{L}_b$-eq-SUP-det*: $\mathcal{L}_b$ *v s* = (⨆ *d* ∈ *D$_D$*. *L* (*mk-dec-det d*) *v s*)
using *$\mathcal{L}$-eq-SUP-det* **unfolding** *$\mathcal{L}_b$.rep-eq* **by** *auto*

7.9 Monotonicity

lemma *$\mathcal{P}_X$-mono*[*intro*]: *a* ≤ *b* ⟹ $\mathcal{P}_X$ *p n a* ≤ $\mathcal{P}_X$ *p n b*
by (*fastforce simp*: *$\mathcal{P}_X$-def intro*: *integral-mono*)

lemma *$\mathcal{P}_1$-mono*[*intro*]: *a* ≤ *b* ⟹ $\mathcal{P}_1$ *p a* ≤ $\mathcal{P}_1$ *p b*
using *$\mathcal{P}_1$-nonneg* **by** *auto*

lemma *L-mono*[*intro*]: *u* ≤ *v* ⟹ *L d u* ≤ *L d v*
unfolding *L-def* **by** (*auto intro*: *scaleR-left-mono*)

lemma *$\mathcal{L}_b$-mono*[*intro*]: *u* ≤ *v* ⟹ $\mathcal{L}_b$ *u* ≤ $\mathcal{L}_b$ *v*
using *ex-dec L-mono*[*of u v*]
by (*fastforce intro*!: *cSUP-mono simp*: *$\mathcal{L}_b$.rep-eq $\mathcal{L}$-def*)

lemma *step-mono*:
assumes $\mathcal{L}_b$ *v* ≤ *v d* ∈ *D$_R$*
shows *L d v* ≤ *v*

using *assms L-le-*$\mathcal{L}_b$ *order.trans* **by** *blast*

lemma *step-mono-elem-det*:
assumes $v \leq \mathcal{L}_b\ v\ e > 0$
shows $\exists\, d{\in}D_D.\ v \leq L\ (mk\text{-}dec\text{-}det\ d)\ v + e *_R 1$
proof −
have $v\ s \leq (\bigsqcup a{\in}A\ s.\ L_a\ a\ v\ s)$ **for** s
using *SUP-step-det-eq* $\mathcal{L}_b$*-eq-SUP-det assms*(*1*) **by** *fastforce*
hence $\exists\, a{\in}A\ s.\ v\ s - e < L_a\ a\ v\ s$ **for** s
using *A-ne* L_a*-le*$'$
by (*subst less-cSUP-iff*[*symmetric*]) (*fastforce simp*: *assms add-strict-increasing algebra-simps intro*!: *bdd-above.I2*)+
hence *aux*: $\exists\, a{\in}A\ s.\ v\ s \leq L_a\ a\ v\ s + e$ **for** s
by (*auto simp*: *diff-less-eq intro*: *less-imp-le*)
then obtain d **where** *is-dec-det* $d\ v\ s \leq L\ (mk\text{-}dec\text{-}det\ d)\ v\ s + e$
for s
by (*metis L-eq-*L_a*-det is-dec-det-def*)
thus *?thesis*
by *fastforce*
qed

lemma *step-mono-elem*:
assumes $v \leq \mathcal{L}_b\ v\ e > 0$
shows $\exists\, d{\in}D_R.\ v \leq L\ d\ v + e *_R 1$
using *assms step-mono-elem-det* **by** *blast*

lemma $\mathcal{P}_X$*-L-le*:
assumes $\mathcal{L}_b\ v \leq v\ p \in \Pi_{M\,R}$
shows $\mathcal{P}_X\ p\ n\ (L\ (p\ n)\ v) \leq \mathcal{P}_X\ p\ n\ v$
using *assms step-mono* **by** *auto*

end

locale *MDP-reward-disc* = *MDP-reward A K r l*
for
A **and**
$K :: 's ::countable \times 'a ::countable \Rightarrow 's\ pmf$ **and**
r l +
assumes
disc-lt-one [*simp*]: $l < 1$
begin

definition *is-opt-act v s* = *is-arg-max* $(\lambda a.\ L_a\ a\ v\ s)$ $(\lambda a.\ a \in A\ s)$
abbreviation *opt-acts v s* $\equiv \{a.\ is\text{-}opt\text{-}act\ v\ s\ a\}$

lemma *summable-disc* [*intro*, *simp*]: *summable* $(\lambda i.\ l\ \widehat{}\ i * x)$
by (*simp add*: *mult.commute*)

lemma *summable-r-disc*[*intro*, *simp*]:

summable (λ*i*. |*l* ^ *i* ∗ *r* (*sa i*)|)
summable (λ*i*. *l* ^ *i* ∗ |*r* (*sa i*)|)
summable (λ*i*. *l* ^ *i* ∗ *r* (*sa i*))
proof −
show *summable* (λ*i*. |*l* ^ *i* ∗ *r* (*sa i*)|)
using *abs-r-le-r*$_M$
by (*fastforce intro*!: *mult-left-mono summable-comparison-test*′[*OF summable-disc*])
thus *summable* (λ*i*. *l* ^ *i* ∗ *r* (*sa i*)) *summable* (λ*i*. *l* ^ *i* ∗ |*r* (*sa i*)|)
by (*auto intro*: *summable-rabs-cancel*)
qed

lemma *summable-norm-disc-I*[*intro*]:
assumes *summable* (λ*t*. (*l*^*t* ∗ *norm F*))
shows *summable* (λ*t*. *norm* (*l*^*t* ∗$_R$ *F*))
using *assms* **by** *auto*

lemma *summable-norm-disc-I*′[*intro*]:
assumes *summable* (λ*t*. (*l*^*t* ∗ *norm* (*F t*)))
shows *summable* (λ*t*. *norm* (*l*^*t* ∗$_R$ *F t*))
using *assms* **by** *auto*

lemma *summable-discI* [*intro*]:
assumes *bounded* (*range F*)
shows *summable* (λ*t*. *l*^*t* ∗ *norm* (*F t*))
proof −
obtain *b* **where** *norm* (*F x*) ≤ *b* **for** *x*
using *assms* **by** (*auto simp*: *bounded-iff*)
thus *?thesis*
using *Abel-lemma*[*of l 1 F b*] **by** (*auto simp*: *mult.commute*)
qed

lemma *summable-disc-reward* [*intro*]:
assumes *bounded* (*range* (*F* :: *nat* ⇒ ′*b* :: *banach*))
shows *summable* (λ*t*. *l*^*t* ∗$_R$ (*F t*))
using *assms* **by** (*auto intro*: *summable-norm-cancel*)

lemma *summable-norm-bfun-disc*: *summable* (λ*t*. *l*^*t* ∗ *norm* (*apply-bfun f t*))
using *norm-le-norm-bfun*
by (*auto simp*: *mult.commute*[*of l*^-] *intro*!: *Abel-lemma*[*of - 1 - norm f*])

lemma *summable-bfun-disc* [*simp*]: *summable* (λ*t*. *l*^*t* ∗ (*apply-bfun f t*))
proof −
have *norm* (*l*^*t* ∗ *apply-bfun f t*) = *l*^*t* ∗ *norm* (*apply-bfun f t*) **for** *t*
by (*auto simp*: *abs-mult*)
hence *summable* (λ*t*. *norm* (*l*^*t* ∗ (*apply-bfun f t*)))

by (*auto simp only*: *abs-mult*)
thus *?thesis*
by (*auto intro*: *summable-norm-cancel*)
qed

lemma *norm-bfun-disc-le*: *norm f* $\leq B \Longrightarrow (\sum x.\ l\hat{}x * norm\ (apply\text{-}bfun\ f\ x)) \leq (\sum x.\ l\hat{}x * B)$
by (*fastforce intro*!: *suminf-le mult-left-mono norm-le-norm-bfun intro*: *order.trans*)

lemma *norm-bfun-disc-le'*: *norm f* $\leq B \Longrightarrow (\sum x.\ l\hat{}x * (apply\text{-}bfun\ f\ x)) \leq (\sum x.\ l\hat{}x * B)$
by (*auto simp*: *mult-left-mono intro*!: *suminf-le order.trans*[*OF - norm-bfun-disc-le*])

lemma *sum-disc-lim-l*: $(\sum x.\ l\hat{}x * B) = B\ /(1-l)$
by (*simp add*: *suminf-mult2*[*symmetric*] *summable-geometric suminf-geometric*[*of l*])

lemma *sum-disc-bound*: $(\sum x.\ l\hat{}x * apply\text{-}bfun\ f\ x) \leq (norm\ f)\ /(1-l)$
using *norm-bfun-disc-le' sum-disc-lim* **by** *auto*

lemma *sum-disc-bound'*:
fixes *f* :: *nat* $\Rightarrow$ *'b* $\Rightarrow_b$ *real*
assumes *h*: $\forall n.\ norm\ (f\ n) \leq B$
shows *norm* $(\sum x.\ l\hat{}x *_R f\ x) \leq B\ /(1-l)$
proof −
have *norm* $(\sum x.\ l\hat{}x *_R f\ x) \leq\ (\sum x.\ norm\ (l\hat{}x *_R f\ x))$
using *h*
by (*fastforce intro*!: *boundedI summable-norm*)
also have $\ldots \leq (\sum x.\ l\hat{}x * B)$
using *h*
by (*auto intro*!: *suminf-le boundedI simp*: *mult-mono'*)
also have $\ldots = B\ /(1-l)$
by (*simp add*: *sum-disc-lim*)
finally show *norm* $(\sum x.\ l\hat{}x *_R f\ x) \leq B\ /(1-l)$ **.**
qed

lemma *abs-ν-trace-le*: $|\nu\text{-}trace\ t| \leq (\sum i.\ l\ \hat{}\ i * r_M)$
by (*auto intro*!: *abs-r-le-r*$_M$ *mult-left-mono order-trans*[*OF summable-rabs*] *suminf-le*)

lemma *integrable-ν-trace*: *integrable* $(\mathcal{T}\ p\ s)$ *ν-trace*
by (*fastforce simp*: *bounded-iff intro*: *abs-ν-trace-le*)

context
fixes *p* :: (*'s*, *'a*) *pol*
begin

lemma *ν-eq-ν-trace*: $\nu\ p\ s = \int t.\ \nu\text{-}trace\ t\ \partial\mathcal{T}\ p\ s$
proof −
have $(\lambda n.\ \nu\text{-}fin\ p\ n\ s) \longrightarrow \int t.\ \nu\text{-}trace\ t\ \partial\mathcal{T}\ p\ s$
unfolding *ν-fin-def*
proof(*intro integral-dominated-convergence*)
show *AE x in* $\mathcal{T}$ *p s.* $\nu\text{-}trace\text{-}fin\ x \longrightarrow \nu\text{-}trace\ x$
using *summable-LIMSEQ* **by** *blast*
next
have $(\sum i{<}N.\ l\ \widehat{}\ i * r_M) \leq (\sum N.\ l\ \widehat{}\ N * r_M)$ **for** *N*
by (*auto intro*: *sum-le-suminf simp*: r_M*-nonneg*)
thus *AE x in* $\mathcal{T}$ *p s. norm* $(\nu\text{-}trace\text{-}fin\ x\ N) \leq (\sum N.\ l\ \widehat{}\ N * r_M)$
for *N*
using *order-trans*[*OF abs-ν-trace-fin-le*] **by** *fastforce*
qed *auto*
thus *?thesis*
using *ν-eq-lim limI* **by** *fastforce*
qed

lemma *abs-ν-le*: $|\nu\ p\ s| \leq (\sum i.\ l\widehat{}i * r_M)$
unfolding *ν-eq-Pn*
using *abs-exp-r-le*
by (*fastforce intro*!: *order.trans*[*OF summable-rabs*] *suminf-le summable-comparison-test'*[*OF summable-disc*] *mult-left-mono*)

lemma *ν-le*: $\nu\ p\ s \leq (\sum i.\ l\widehat{}i * r_M)$
by (*auto intro*: *abs-ν-le abs-le-D1*)

lemma *ν-bfun*: $\nu\ p \in bfun$
by (*auto intro*!: *abs-ν-le*)

lift-definition $\nu_b :: 's \Rightarrow_b real$ **is** $\nu\ p$
using *ν-bfun* **by** *blast*

lemma *norm-ν-le*: *norm* $\nu_b \leq r_M\ /\ (1{-}l)$
using *abs-ν-le sum-disc-lim*
by (*auto simp*: ν_b*.rep-eq norm-bfun-def' intro*: *cSUP-least*)
end

lemma *ν-as-markovian*: ν (*mk-markovian* (*as-markovian p* (*return-pmf s*))) $s = \nu\ p\ s$
by (*auto simp*: *ν-eq-Pn Pn-as-markovian-eq Pn'-def*)

lemma ν_b*-as-markovian*: ν_b (*mk-markovian* (*as-markovian p* (*return-pmf s*))) $s = \nu_b\ p\ s$
using *ν-as-markovian* **by** (*auto simp*: ν_b*.rep-eq*)

7.10 Optimal Reward

definition ν-*MD* $s \equiv \bigsqcup p \in \Pi_{MD}.\ \nu\ (mk\text{-}markovian\text{-}det\ p)\ s$
definition ν-*opt* $s \equiv \bigsqcup p \in \Pi_{HR}.\ \nu\ p\ s$

lemma ν-*opt-bfun*: ν-*opt* $\in$ *bfun*
using *abs-*ν*-le policies-ne*
by (*fastforce simp*: ν*-opt-def intro*!: *order-trans*[*OF cSup-abs-le*] *bfun-normI*)

lift-definition ν_b-*opt* :: $'s \Rightarrow_b$ *real* **is** ν-*opt*
using ν*-opt-bfun* .

lemma ν_b-*opt-eq*: ν_b-*opt* $s = (\bigsqcup p \in \Pi_{HR}.\ \nu_b\ p\ s)$
using ν_b*.rep-eq* ν_b*-opt.rep-eq* ν*-opt-def* **by** *presburger*

lemma ν-*le-*ν-*opt* [*intro*]:
assumes *is-policy p*
shows $\nu\ p\ s \leq \nu$-*opt s*
unfolding ν*-opt-def* **using** *abs-*ν*-le assms*
by (*force intro*: *cSUP-upper intro*!: *bounded-imp-bdd-above boundedI*)

lemma ν_b-*le-opt* [*intro*]: $p \in \Pi_{HR} \Longrightarrow \nu_b\ p \leq \nu_b$-*opt*
using ν*-le* **by** (*fastforce simp*: ν_b*.rep-eq* ν_b*-opt.rep-eq*)

lemma ν_b-*le-opt-MD* [*intro*]: $p \in \Pi_{MD} \Longrightarrow \nu_b\ (mk\text{-}markovian\text{-}det\ p) \leq \nu_b$-*opt*
by (*auto simp*: *mk-markovian-det-def is-dec-det-def is-dec-def is-policy-def*)

lemma ν_b-*le-opt-DD* [*intro*]: *is-dec-det d* $\Longrightarrow \nu_b\ (mk\text{-}stationary\text{-}det\ d) \leq \nu_b$-*opt*
by (*auto simp add*: *is-policy-def mk-markovian-def*)

lemma ν_b-*le-opt-DR* [*intro*]: *is-dec d* $\Longrightarrow \nu_b\ (mk\text{-}stationary\ d) \leq \nu_b$-*opt*
by (*auto simp add*: *is-policy-def mk-markovian-def*)

lemma ν_b-*opt-eq-MR*: ν_b-*opt* $s = (\bigsqcup p \in \Pi_{MR}.\ \nu_b\ (mk\text{-}markovian\ p)\ s)$
proof (*rule antisym*)
show ν_b-*opt* $s \leq (\bigsqcup p \in \Pi_{MR}.\ \nu_b\ (mk\text{-}markovian\ p)\ s)$
unfolding ν_b*-opt-eq*
proof (*rule cSUP-mono*)
show $\Pi_{HR} \neq \{\}$
using *policies-ne* **by** *simp*
show *bdd-above* $((\lambda p.\ \nu_b\ (mk\text{-}markovian\ p)\ s)\ `\ \Pi_{MR})$
by (*auto intro*!: *boundedI bounded-imp-bdd-above abs-*ν*-le simp*: ν_b*.rep-eq*)
show $n \in \Pi_{HR} \Longrightarrow \exists\, m \in \Pi_{MR}.\ \nu_b\ n\ s \leq \nu_b\ (mk\text{-}markovian\ m)\ s$
for n

using *is-*Π_{MR}*-as-markovian* **by** (*subst* ν_b*-as-markovian*[*symmetric*]) *fastforce*
qed
show ($\bigsqcup p{\in}\Pi_{MR}.\ \nu_b\ (mk\text{-}markovian\ p)\ s) \leq \nu_b\text{-}opt\ s$
using Π_{MR}*-ne* Π_{MR}*-imp-policies*
by (*auto intro*!: *cSUP-mono bounded-imp-bdd-above boundedI abs-*ν*-le simp*: ν_b*-opt-eq* ν_b*.rep-eq*)
qed

lemma *summable-norm-disc-reward'*[*simp*]: *summable* ($\lambda t.\ l\hat{}t * norm\ (\mathcal{P}_X\ p\ t\ (r\text{-}dec_b\ (p\ t))))$
using $\mathcal{P}_X$*-bounded-r* **by** *auto*

lemma *summable-disc-reward-*$\mathcal{P}_X$ [*simp*]: *summable* ($\lambda t.\ l\hat{}t *_R \mathcal{P}_X\ p\ t\ (r\text{-}dec_b\ (p\ t))$)
using *summable-disc-reward* $\mathcal{P}_X$*-bounded-r* **by** *blast*

lemma *disc-reward-tendsto*:
($\lambda n.\ \sum t{<}n.\ l\hat{}t *_R \mathcal{P}_X\ p\ t\ (r\text{-}dec_b\ (p\ t))) \longrightarrow (\sum t.\ l\hat{}t *_R \mathcal{P}_X\ p\ t\ (r\text{-}dec_b\ (p\ t)))$
by (*simp add*: *summable-LIMSEQ*)

lemma ν*-eq-*$\mathcal{P}_X$: $\nu\ (mk\text{-}markovian\ p) = (\sum i.\ l\hat{}i *_R \mathcal{P}_X\ p\ i\ (r\text{-}dec_b\ (p\ i)))$
proof −
have $\nu\ (mk\text{-}markovian\ p)\ s = (\sum i.\ l\hat{}i * \mathcal{P}_X\ p\ i\ (r\text{-}dec_b\ (p\ i))\ s)$
for *s*
unfolding ν_b*.rep-eq* $\mathcal{P}_X$*-def* ν*-eq-Pn Pn'-markovian-eq-Xn'-bind measure-pmf-bind*
using *measure-pmf-in-subprob-algebra abs-r-le-r*$_M$
by (*subst integral-bind*) (*auto simp*: *r-dec-eq-r-K0*)
thus *?thesis*
by (*auto simp*: *suminf-apply-bfun*)
qed

lemma ν_b*-eq-*$\mathcal{P}_X$: $\nu_b\ (mk\text{-}markovian\ p) = (\sum i.\ l\hat{}i *_R \mathcal{P}_X\ p\ i\ (r\text{-}dec_b\ (p\ i)))$
by (*auto simp*: ν*-eq-*$\mathcal{P}_X$ ν_b*.rep-eq*)

lemma ν_b*-fin-tendsto-*ν_b: (ν_b*-fin* (*mk-markovian p*)) $\longrightarrow \nu_b\ (mk\text{-}markovian\ p)$
using *disc-reward-tendsto* ν_b*-eq-*$\mathcal{P}_X$ ν_b*-fin-eq-*$\mathcal{P}_X$
by *presburger*

lemma *norm-*$\mathcal{P}_1$*-l-less*: $norm\ (l *_R \mathcal{P}_1\ d) < 1$
by *auto*
lemma *disc-*$\mathcal{P}_1$*-tendsto*: ($\lambda n.\ (\sum t{\leq}n.\ l\hat{}t *_R \mathcal{P}_1\ d\ \widehat{\ }\widehat{\ }t)) \longrightarrow (\sum t.\ l\hat{}t *_R \mathcal{P}_1\ d\ \widehat{\ }\widehat{\ }t)$
by (*fastforce simp*: *bounded-iff intro*: *summable-LIMSEQ'*)

lemma *disc-$\mathcal{P}_1$-lim*: *lim* ($\lambda n.$ ($\sum t{\leq}n.$ $l \frown t *_R \mathcal{P}_1\ d \frown\frown t$)) = ($\sum t.$ $l \frown t *_R \mathcal{P}_1\ d \frown\frown t$)
using *limI disc-$\mathcal{P}_1$-tendsto*
by *blast*

lemma *convergent-disc-$\mathcal{P}_1$*: *convergent* ($\lambda n.$ ($\sum t{\leq}n.$ $l \frown t *_R \mathcal{P}_1\ d \frown\frown t$))
using *convergentI disc-$\mathcal{P}_1$-tendsto*
by *blast*

lemma *$\mathcal{P}_1$-suminf-ge*:
assumes $0 \leq u$ **shows** $u \leq (\sum t.\ l \frown t *_R \mathcal{P}_1\ d \frown\frown t)\ u$
proof −
have *aux*: $\bigwedge x.$ ($\lambda n.$ ($\sum t{\leq}n.$ $l \frown t *_R \mathcal{P}_1\ d \frown\frown t$) $u\ x$) $\longrightarrow$ ($\sum t.$ $l \frown t *_R \mathcal{P}_1\ d \frown\frown t$) $u\ x$
using *bfun-tendsto-apply-bfun disc-$\mathcal{P}_1$-lim lim-blinfun-apply*[*OF convergent-disc-$\mathcal{P}_1$*]
by *fastforce*
have $\bigwedge n.$ $u \leq (\sum t{\leq}n.$ $l \frown t *_R \mathcal{P}_1\ d \frown\frown t)\ u$
using *$\mathcal{P}_1$-sum-ge*[*OF assms*] **by** *auto*
thus *?thesis*
by (*auto intro*!: *LIMSEQ-le-const*[*OF aux*])
qed

lemma *$\mathcal{P}_1$-suminf-pos*:
assumes $0 \leq u$
shows $0 \leq (\sum t.\ l \frown t *_R \mathcal{P}_1\ d \frown\frown t)\ u$
using *$\mathcal{P}_1$-suminf-ge*[*of u*] *assms order.trans* **by** *auto*

lemma *lemma-6-1-2-b*:
assumes $v \leq u$
shows ($\sum t.\ l \frown t *_R \mathcal{P}_1\ d \frown\frown t)\ v \leq (\sum t.\ l \frown t *_R \mathcal{P}_1\ d \frown\frown t)\ u$
proof −
have $0 \leq (\sum n.\ l \frown n *_R \mathcal{P}_1\ d \frown\frown n)\ (u - v)$
using *$\mathcal{P}_1$-suminf-pos assms* **by** *simp*
thus *?thesis*
by (*simp add*: *blinfun.diff-right*)
qed

lemma *ν-stationary*: ν_b (*mk-stationary d*) = ($\sum t.\ l \frown t *_R$ ($\mathcal{P}_1\ d \frown\frown t$)) (*r-dec$_b$ d*)
proof −
have ν_b (*mk-stationary d*) = ($\sum t.$ ($l \frown t *_R$ ($\mathcal{P}_1\ d \frown\frown t$)) (*r-dec$_b$ d*))
by (*simp add*: *ν_b-eq-$\mathcal{P}_X$ scaleR-blinfun.rep-eq $\mathcal{P}_X$-sconst*)
also have ... = ($\sum t.$ ($l \frown t *_R$ ($\mathcal{P}_1\ d \frown\frown t$))) (*r-dec$_b$ d*)
by (*subst bounded-linear.suminf*[**where** $f = \lambda x.$ *blinfun-apply x* (*r-dec$_b$ d*)])
(*auto intro*!: *bounded-linear.suminf boundedI*)
finally show *?thesis* **.**

qed

lemma *ν-stationary-inv*: ν_b (*mk-stationary d*) = inv_L (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ *d*) (*r-dec*$_b$ *d*)
by (*auto simp*: *ν-stationary* inv_L*-inf-sum blincomp-scaleR-right*)

The value of a markovian policy can be expressed in terms of L.

lemma *ν-step*: ν_b (*mk-markovian p*) = *L* (*p 0*) (ν_b (*mk-markovian* (λ*n*. *p* (*Suc n*))))
proof −
have *s*: *summable* (λ*t*. *l*^*t* $*_R$ ($\mathcal{P}_X$ *p* (*Suc t*) (*r-dec*$_b$ (*p* (*Suc t*)))))
using $\mathcal{P}_X$*-bound-r* **by** (*auto intro*!: *boundedI*[*of* - r_M])
have
ν_b (*mk-markovian p*) = *r-dec*$_b$ (*p 0*) + (∑ *t*. *l* ^ (*Suc t*) $*_R$ $\mathcal{P}_X$ *p* (*Suc t*) (*r-dec*$_b$ (*p* (*Suc t*))))
by (*subst suminf-split-head*) (*auto simp*: ν_b*-eq-*$\mathcal{P}_X$)
also have
. . . = *r-dec*$_b$ (*p 0*) + *l* $*_R$ (∑ *t*. $\mathcal{P}_1$ (*p 0*) (*l*^*t* $*_R$ $\mathcal{P}_X$ (λ*n*. *p* (*Suc n*)) *t* (*r-dec*$_b$ (*p* (*Suc t*))))))
using *suminf-scaleR-right*[*OF s*] **by** (*auto simp*: $\mathcal{P}_X$*-Suc blinfun.scaleR-right*)
also have
. . . = *L* (*p 0*) (ν_b (*mk-markovian* (λ*n*. *p* (*Suc n*))))
using *blinfun.bounded-linear-right bounded-linear.suminf*[*of blinfun-apply* ($\mathcal{P}_1$ (*p 0*))]
by (*fastforce simp add*: ν_b*-eq-*$\mathcal{P}_X$ *L-def*)
finally show *?thesis* **.**
qed

lemma *L-ν-fix*: ν_b (*mk-stationary d*) = *L d* (ν_b (*mk-stationary d*))
using *ν-step* **.**

lemma *L-fix-ν*:
assumes *L p v* = *v*
shows *v* = ν_b (*mk-stationary p*)
proof −
have *r-dec*$_b$ *p* = (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ *p*) *v*
using *assms* **by** (*auto simp*: *eq-diff-eq L-def blinfun.diff-left blinfun.scaleR-left*)
hence *v* = (∑ *t*. (*l* $*_R$ $\mathcal{P}_1$ *p*)^^*t*) (*r-dec*$_b$ *p*)
using *inv-norm-le*′(*2*)[*OF norm-*$\mathcal{P}_1$*-l-less*] **by** *auto*
thus *v* = ν_b (*mk-stationary p*)
by (*auto simp*: *ν-stationary blincomp-scaleR-right*)
qed

lemma *L-ν-fix-iff*: *L d v* = *v* ⟷ *v* = ν_b (*mk-stationary d*)
using *L-fix-ν L-ν-fix* **by** *auto*

7.11 Properties of Solutions of the Optimality Equations

abbreviation $\mathcal{P}_d\ p\ n\ v \equiv l \widehat{}\ n *_R \mathcal{P}_X\ p\ n\ v$

lemma $\mathcal{P}_d$*-lim*: $(\lambda n.\ (\mathcal{P}_d\ p\ n\ v)) \longrightarrow 0$
proof −
  have $(\lambda n.\ l \widehat{}\ n * norm\ v) \longrightarrow 0$
    by (*auto intro*!: *tendsto-eq-intros*)
  moreover have $norm\ (\mathcal{P}_d\ p\ n\ v) \leq l \widehat{}\ n * norm\ v$ **for** $p\ n$
    by (*simp add*: *mult-mono'*)
  ultimately have $(\lambda n.\ norm\ (\mathcal{P}_d\ p\ n\ v)) \longrightarrow 0$ **for** p
    by (*auto simp*: *Lim-transform-bound*[**where** $g = \lambda n.\ (l \widehat{}\ n * norm\ v)$])
  thus $(\lambda n.\ (\mathcal{P}_d\ p\ n\ v)) \longrightarrow 0$ **for** p
    using *tendsto-norm-zero-cancel* **by** *fast*
qed

lemma $\mathcal{L}$*-dec-ge-opt*:
  assumes $\mathcal{L}_b\ v \leq v$
  shows ν_b*-opt* $\leq v$
proof −
  have $\nu_b\ (mk\text{-}markovian\ p) \leq v$ **if** $p \in \Pi_{M\,R}$ **for** p
  proof −
    let *?p* = *mk-markovian p*
    have *aux*: ν_b*-fin ?p n* + $l \widehat{}\ n *_R \mathcal{P}_X\ p\ n\ v \leq v$ **for** n
    proof (*induction n*)
      case (*Suc n*)
     have $\mathcal{P}_X\ p\ n\ (r\text{-}dec_b\ (p\ n)) + l *_R (\mathcal{P}_X\ p\ (Suc\ n)\ v) \leq \mathcal{P}_X\ p\ n\ v$
       using $\mathcal{P}_X$*-L-le assms that* **by** (*simp add*: $\mathcal{P}_X$*-Suc-n-elem L-def linear-simps*)
      hence ν_b*-fin ?p* $(n + 1) + l \widehat{}\ (n + 1) *_R (\mathcal{P}_X\ p\ (n + 1)\ v) \leq \nu_b$*-fin ?p n* + $l \widehat{}\ n *_R (\mathcal{P}_X\ p\ n\ v)$
        by (*auto simp del*: *scaleR-scaleR intro*: *scaleR-left-mono simp*: ν_b*-fin-eq-*$\mathcal{P}_X$
        mult.commute[*of l*] *scaleR-add-right*[*symmetric*] *scaleR-scaleR*[*symmetric*])
      also have $\ldots \leq v$
        using *Suc.IH* **by** (*auto simp*: ν_b*-fin-eq-*$\mathcal{P}_X$)
      finally show *?case*
        by *auto*
    qed (*auto simp*: ν_b*-fin-eq-*$\mathcal{P}_X$)
    have *1*: $(\lambda n.\ (\nu_b$*-fin ?p n* + $\mathcal{P}_d\ p\ n\ v)\ s) \longrightarrow \nu_b\ ?p\ s$ **for** s
    using *bfun-tendsto-apply-bfun Limits.tendsto-add*[*OF* ν_b*-fin-tendsto-*ν_b $\mathcal{P}_d$*-lim*] **by** *fastforce*
    have $\nu_b\ ?p\ s \leq v\ s$ **for** s
      using *that aux assms* **by** (*fastforce intro*!: *lim-mono*[*OF - 1, of -*

- $\lambda n.\ v\ s$])
  thus *?thesis*
    using *that* **by** *blast*
 qed
 thus *?thesis*
   using *policies-ne* **by** (*fastforce simp*: *is-policy-def* ν_b*-opt-eq-MR intro*!: *cSUP-least*)
qed

lemma $\mathcal{L}$*-inc-le-opt*:
 assumes $v \leq \mathcal{L}_b\ v$
 shows $v \leq \nu_b$*-opt*
proof −
 have *le-elem*: $v\ s \leq \nu_b$*-opt* $s + (e/(1-l))$ **if** $e > 0$ **for** $s\ e$
 proof −
  obtain d **where** $d \in D_R$ **and** *hd*: $v \leq L\ d\ v + e *_R 1$
    using *assms step-mono-elem* ‹$e > 0$› **by** *blast*
  let *?Pinf* $= (\sum i.\ l \widehat{}\ i *_R \mathcal{P}_1\ d \widehat{}\widehat{}\ i)$
  have $v \leq$ *r-dec*$_b\ d + l *_R (\mathcal{P}_1\ d)\ v + e *_R 1$
    using *hd L-def* **by** *fastforce*
  hence (*id-blinfun* $- l *_R \mathcal{P}_1\ d)\ v \leq$ *r-dec*$_b\ d + e *_R 1$
    by (*auto simp*: *blinfun.diff-left blinfun.scaleR-left algebra-simps*)
  hence *?Pinf* ((*id-blinfun* $- l *_R \mathcal{P}_1\ d)\ v) \leq$ *?Pinf* (*r-dec*$_b\ d + e *_R 1$)
    using *lemma-6-1-2-b* $\mathcal{P}_1$*-def hd* **by** *auto*
  hence $v \leq$ *?Pinf* (*r-dec*$_b\ d + e *_R 1$)
      using *inv-norm-le'*(*2*)[*of* $l *_R \mathcal{P}_1\ d$] **by** (*auto simp*: *blinfun-comp-scaleR-right*)
  also have $\ldots = \nu_b$ (*mk-stationary d*) $+ e *_R$ *?Pinf 1*
    by (*simp add*: ν*-stationary blinfun.add-right blinfun.scaleR-right*)
  also have $\ldots = \nu_b$ (*mk-stationary d*) $+ e *_R (\sum i.\ (l \widehat{}\ i *_R ((\mathcal{P}_1\ d \widehat{}\widehat{}\ i))))\ 1$)
    using *convergent-disc-*$\mathcal{P}_1$
   by (*auto simp*: *summable-iff-convergent' bounded-linear.suminf*[*of* $\lambda x.$ *blinfun-apply x 1*])
  also have $\ldots = \nu_b$ (*mk-stationary d*) $+ e *_R (\sum i.\ (l \widehat{}\ i *_R 1))$
    by (*auto simp*: *scaleR-blinfun.rep-eq*)
  also have $\ldots \leq (\nu_b$ (*mk-stationary d*) $+ (e\ /\ (1-l)) *_R 1)$
    by (*auto simp*: *bounded-linear.suminf*[*symmetric*, **where** $f = \lambda x.\ x *_R 1$] *suminf-geometric bounded-linear-scaleR-left summable-geometric*)
  finally have $v\ s \leq (\nu_b$ (*mk-stationary d*) $+ (e/(1-l)) *_R 1)\ s$
    by *auto*
  thus $v\ s \leq \nu_b$*-opt* $s + (e/(1-l))$
    using ‹$d \in D_R$› ν_b*-le-opt*
     by (*auto simp*: *is-policy-def mk-markovian-def less-eq-bfun-def intro*: *order-trans*)
 qed
 have $v\ s \leq \nu_b$*-opt* $s + e$ **if** $e > 0$ **for** $s\ e$

proof −
have $e * (1 - l) > 0$
by (*simp add*: ‹$0 < e$›)
thus $v\ s \leq \nu_b$-*opt* $s + e$
using *disc-lt-one that le-elem* **by** (*fastforce split*: *if-splits*)
qed
thus *?thesis*
by (*fastforce intro*: *field-le-epsilon*)
qed
lemma $\mathcal{L}$-*fix-imp-opt*:
assumes $v = \mathcal{L}_b\ v$
shows $v = \nu_b$-*opt*
using *assms dual-order.antisym*[*OF* $\mathcal{L}$-*dec-ge-opt* $\mathcal{L}$-*inc-le-opt*] **by** *auto*

lemma *bounded-P*: *bounded* ($\mathcal{P}_1$ ‘ X)
by (*auto simp*: *bounded-iff*)

7.12 Solutions to the Optimality Equation

7.12.1 $\mathcal{L}_b$ and L are Contraction Mappings

declare *bounded-apply-blinfun*[*intro*] *bounded-apply-bfun'*[*intro*]

lemma *contraction-*$\mathcal{L}$: *dist* ($\mathcal{L}_b\ v$) ($\mathcal{L}_b\ u$) $\leq l *$ *dist* $v\ u$
proof −
have *dist* ($\mathcal{L}_b\ v\ s$) ($\mathcal{L}_b\ u\ s$) $\leq l *$ *dist* $v\ u$ **if** $\mathcal{L}_b\ u\ s \leq \mathcal{L}_b\ v\ s$ **for** $s\ v\ u$
proof −
have *dist* ($\mathcal{L}_b\ v\ s$) ($\mathcal{L}_b\ u\ s$) $\leq$ ($\bigsqcup d \in D_R.\ L\ d\ v\ s - L\ d\ u\ s$)
using *ex-dec that* **by** (*fastforce intro*!: *le-SUP-diff' simp*: *dist-real-def* $\mathcal{L}_b$.*rep-eq* $\mathcal{L}$-*def*)
also have ... = ($\bigsqcup d \in D_R.\ l * (\mathcal{P}_1\ d\ (v - u)\ s)$)
by (*auto simp*: *L-def right-diff-distrib blinfun.diff-right*)
also have ... = $l *$ ($\bigsqcup d \in D_R.\ \mathcal{P}_1\ d\ (v - u)\ s$)
using D_R-*ne bounded-P* **by** (*fastforce intro*: *bounded-SUP-mul*)
also have ... $\leq l *$ *norm* ($\bigsqcup d \in D_R.\ \mathcal{P}_1\ d\ (v - u)\ s$)
by (*simp add*: *mult-left-mono*)
also have ... $\leq l *$ ($\bigsqcup d \in D_R.$ *norm* (($\mathcal{P}_1\ d\ (v - u)$) s))
proof −
have *bounded* ((λ$x.$ *norm* (($\mathcal{P}_1\ x\ (v - u)$) s)) ‘ D_R)
using *bounded-apply-bfun' bounded-P bounded-apply-blinfun bounded-norm-comp* **by** *metis*
thus *?thesis*
using D_R-*ne ex-dec bounded-norm-comp* **by** (*fastforce intro*!: *mult-left-mono*)
qed
also have ... $\leq l *$ ($\bigsqcup p \in D_R.$ *norm* ($\mathcal{P}_1\ p$ (($v - u$))))
using D_R-*ne abs-le-norm-bfun bounded-P*
by (*fastforce simp*: *bounded-norm-comp intro*!: *bounded-imp-bdd-above*

mult-left-mono cSUP-mono)
also have $\ldots \leq l * (\bigsqcup p \in D_R.\ norm\ ((v - u)))$
using *norm-push-exp-le-norm* D_R*-ne*
by (*fastforce simp*: $\mathcal{P}_1$*.rep-eq intro*!: *mult-left-mono cSUP-mono*)
also have $\ldots = l * dist\ v\ u$
by (*auto simp*: *dist-norm*)
finally show *?thesis* **.**
qed
hence $\mathcal{L}_b\ u\ s \leq \mathcal{L}_b\ v\ s \Longrightarrow dist\ (\mathcal{L}_b\ v\ s)\ (\mathcal{L}_b\ u\ s) \leq l * dist\ v\ u$
$\mathcal{L}_b\ v\ s \leq \mathcal{L}_b\ u\ s \Longrightarrow dist\ (\mathcal{L}_b\ v\ s)\ (\mathcal{L}_b\ u\ s) \leq l * dist\ v\ u$ **for** $u\ v\ s$
by (*fastforce simp*: *dist-commute*)+
thus *?thesis*
using *linear*[*of* $\mathcal{L}_b\ u$ -] **by** (*fastforce intro*: *dist-bound*)
qed

lemma *is-contraction-*$\mathcal{L}$: *is-contraction* $\mathcal{L}_b$
using *contraction-*$\mathcal{L}$ *zero-le-disc disc-lt-one* **unfolding** *is-contraction-def*
by *blast*

lemma *contraction-L*: $dist\ (L\ p\ v)\ (L\ p\ u) \leq l * dist\ v\ u$
proof −
have *aux*: $L\ p\ v\ s - L\ p\ u\ s \leq l * dist\ v\ u$ **if** *lea*: $L\ p\ v\ s \geq L\ p\ u\ s$
for $v\ s\ u$
proof −
have $L\ p\ v\ s - L\ p\ u\ s = (l *_R\ (\mathcal{P}_1\ p\ v - \mathcal{P}_1\ p\ u))\ s$
by (*simp add*: *L-def scale-right-diff-distrib*)
also have $\ldots \leq l * norm\ (\mathcal{P}_1\ p\ (v - u)\ s)$
by (*auto simp*: *blinfun.diff-right intro*!: *mult-left-mono*)
also have $\ldots \leq l * norm\ (\mathcal{P}_1\ p\ (v - u))$
using *abs-le-norm-bfun* **by** (*auto intro*!: *mult-left-mono*)
also have $\ldots \leq l * dist\ v\ u$
by (*simp add*: $\mathcal{P}_1$*.rep-eq mult-left-mono norm-push-exp-le-norm dist-norm*)
finally show *?thesis*
by *auto*
qed
have $dist\ (L\ p\ v\ s)\ (L\ p\ u\ s) \leq l * dist\ v\ u$ **for** $v\ s\ u$
using *aux*[*of v - u*] *aux*[*of u - v*]
by (*cases* $L\ p\ v\ s \geq L\ p\ u\ s$) (*auto simp*: *dist-real-def dist-commute*)
thus $dist\ (L\ p\ v)\ (L\ p\ u) \leq l * dist\ v\ u$
by (*simp add*: *dist-bound*)
qed

lemma *is-contraction-L*: *is-contraction* $(L\ p)$
unfolding *is-contraction-def* **using** *contraction-L disc-lt-one zero-le-disc*
by *blast*

7.12.2 Existence of a Fixpoint of $\mathcal{L}_b$

lemma $\mathcal{L}_b$*-conv*:
$\exists!v.\ \mathcal{L}_b\ v = v\ (\lambda n.\ (\mathcal{L}_b\ \widehat{}\widehat{}\ n)\ v) \longrightarrow (THE\ v.\ \mathcal{L}_b\ v = v)$
using *banach'*[*OF is-contraction-*$\mathcal{L}$] **by** *auto*

lemma $\mathcal{L}_b$*-fix-iff-opt* [*simp*]: $\mathcal{L}_b\ v = v \longleftrightarrow v = \nu_b$*-opt*
using *banach'*(*1*) *is-contraction-*$\mathcal{L}$ $\mathcal{L}$*-fix-imp-opt* **by** *metis*

lemma ν_b*-opt-fix*: ν_b*-opt* $= (THE\ v.\ \mathcal{L}_b\ v = v)$
by *auto*

lemma $\mathcal{L}_b$*-opt* [*simp*]: $\mathcal{L}_b\ \nu_b$*-opt* $= \nu_b$*-opt*
by *auto*

lemma $\mathcal{L}_b$*-lim*: $(\lambda n.\ (\mathcal{L}_b\ \widehat{}\widehat{}\ n)\ v) \longrightarrow \nu_b$*-opt*
using $\mathcal{L}_b$*-conv*(*2*) ν_b*-opt-fix* **by** *presburger*

lemma *thm-6-2-6*: $\nu_b\ p = \nu_b$*-opt* $\longleftrightarrow \mathcal{L}_b\ (\nu_b\ p) = \nu_b\ p$
by *force*

lemma *thm-6-2-6'*: $\nu\ p = \nu$*-opt* $\longleftrightarrow \mathcal{L}_b\ (\nu_b\ p) = \nu_b\ p$
using *thm-6-2-6* ν_b*.rep-eq* ν_b*-opt.rep-eq* **by** *fastforce*

7.13 Existence of Optimal Policies

definition ν*-improving* $v\ d \longleftrightarrow (\forall\, s.\ is\text{-}arg\text{-}max\ (\lambda d.\ (L\ d\ v)\ s)\ (\lambda d.\ d \in D_R)\ d)$

lemma ν*-improving-iff*: ν*-improving* $v\ d \longleftrightarrow d \in D_R \wedge (\forall\, d' \in D_R.\ \forall\, s.\ L\ d'\ v\ s \leq L\ d\ v\ s)$
by (*auto simp*: ν*-improving-def is-arg-max-linorder*)

lemma ν*-improving-D-MR*[*dest*]: ν*-improving* $v\ d \Longrightarrow d \in D_R$
by (*auto simp add*: ν*-improving-iff*)

lemma ν*-improving-ge*: ν*-improving* $v\ d \Longrightarrow d' \in D_R \Longrightarrow L\ d'\ v\ s \leq L\ d\ v\ s$
by (*auto simp*: ν*-improving-iff*)

lemma ν*-improving-imp-*$\mathcal{L}_b$: ν*-improving* $v\ d \Longrightarrow \mathcal{L}_b\ v = L\ d\ v$
by (*fastforce intro*!: *cSup-eq-maximum simp*: ν*-improving-iff* $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-def*)

lemma $\mathcal{L}_b$*-imp-*ν*-improving*:
assumes $d \in D_R\ \mathcal{L}_b\ v = L\ d\ v$
shows ν*-improving* $v\ d$
using *assms L-le-*$\mathcal{L}_b$ **by** (*auto simp*: ν*-improving-iff assms*(*2*)[*symmetric*])

lemma ν*-improving-alt*:

assumes $d \in D_R$
shows ν-*improving* v $d \longleftrightarrow \mathcal{L}_b$ $v = L$ d v
using $\mathcal{L}_b$-*imp-*ν-*improving* ν-*improving-imp-*$\mathcal{L}_b$ *assms* **by** *blast*

definition ν-*conserving* $d = \nu$-*improving* (ν_b-*opt*) d

lemma ν-*conserving-iff*: ν-*conserving* $d \longleftrightarrow d \in D_R \wedge (\forall\, d' \in D_R.$ $\forall\, s.\ L\ d'\ \nu_b$-*opt* $s \leq L\ d\ \nu_b$-*opt* $s)$
by (*auto simp*: ν-*conserving-def* ν-*improving-iff*)

lemma ν-*conserving-ge*: ν-*conserving* $d \Longrightarrow d' \in D_R \Longrightarrow L\ d'\ \nu_b$-*opt* $s \leq L\ d\ \nu_b$-*opt* s
by (*auto simp*: ν-*conserving-iff intro*: ν-*improving-ge*)

lemma ν-*conserving-imp-*$\mathcal{L}_b$ [*simp*]: ν-*conserving* $d \Longrightarrow L\ d\ \nu_b$-*opt* = ν_b-*opt*
using ν-*improving-imp-*$\mathcal{L}_b$ **by** (*fastforce simp*: ν-*conserving-def*)

lemma $\mathcal{L}_b$-*imp-*ν-*conserving*:
assumes $d \in D_R$ $\mathcal{L}_b$ ν_b-*opt* = L d ν_b-*opt*
shows ν-*conserving* d
using $\mathcal{L}_b$-*imp-*ν-*improving assms* **by** (*auto simp*: ν-*conserving-def*)

lemma ν-*conserving-alt*:
assumes $d \in D_R$
shows ν-*conserving* $d \longleftrightarrow \mathcal{L}_b$ ν_b-*opt* = L d ν_b-*opt*
unfolding ν-*conserving-def* **using** ν-*improving-alt assms* **by** *auto*

lemma ν-*conserving-alt'*:
assumes $d \in D_R$
shows ν-*conserving* $d \longleftrightarrow L$ d ν_b-*opt* = ν_b-*opt*
using *assms* ν-*conserving-alt* **by** *auto*

7.13.1 Conserving Decision Rules are Optimal

theorem *ex-improving-imp-conserving*:
assumes $\bigwedge v.\ \exists\, d.\ \nu$-*improving* v (*mk-dec-det* d)
shows $\exists\, d.\ \nu$-*conserving* (*mk-dec-det* d)
by (*simp add*: *assms* ν-*conserving-def*)

theorem *conserving-imp-opt*[*simp*]:
assumes ν-*conserving* (*mk-dec-det* d)
shows ν_b (*mk-stationary-det* d) = ν_b-*opt*
using *L-*ν*-fix-iff* ν-*conserving-imp-*$\mathcal{L}_b$[*OF assms*] **by** *simp*

lemma *conserving-imp-opt'*:
assumes $\exists\, d.\ \nu$-*conserving* (*mk-dec-det* d)
shows $\exists\, d \in D_D.$ (ν_b (*mk-stationary-det* d)) = ν_b-*opt*
using *assms* **by** (*fastforce simp*: ν-*conserving-def*)

theorem *improving-att-imp-det-opt*:
assumes $\bigwedge v.\ \exists\, d.\ \nu\text{-improving}\ v\ (\text{mk-dec-det}\ d)$
shows $\nu_b\text{-opt}\ s = (\bigsqcup d \in D_D.\ \nu_b\ (\text{mk-stationary-det}\ d)\ s)$
proof −
obtain d **where** d: $\nu\text{-conserving}\ (\text{mk-dec-det}\ d)$
using *assms ex-improving-imp-conserving* **by** *auto*
hence $d \in D_D$
using $\nu\text{-conserving-iff}$ *is-dec-mk-dec-det-iff* **by** *blast*
thus *?thesis*
using $\Pi_{MR}\text{-imp-policies}$ $\nu_b\text{-le-opt}$
by (*fastforce intro*!: *cSup-eq-maximum*[**where** $z = \nu_b\text{-opt}\ s$, *symmetric*]
simp: *conserving-imp-opt*[*OF d*] *image-iff*)
qed

lemma $\mathcal{L}_b\text{-sup-att-dec}$:
assumes $d \in D_R\ \mathcal{L}_b\ v = L\ d\ v$
shows $\exists\, d' \in D_D.\ \mathcal{L}_b\ v = L\ (\text{mk-dec-det}\ d')\ v$
proof −
have $\exists\, a \in A\ s.\ L\ d\ v\ s = L_a\ a\ v\ s$ **for** s
unfolding $L\text{-eq-}L_a$
using *assms is-dec-def* $L_a\text{-bounded}$ *A-ne* $\mathcal{L}_b\text{.rep-eq}$ $\mathcal{L}\text{-def}$
by (*intro lemma-4-3-1*′)
(*auto intro*: *bounded-range-subset simp*: *assms*(*2*)[*symmetric*] $L\text{-eq-}L_a$[*symmetric*] *SUP-step-MR-eq*)
then obtain d' **where** d: $d'\ s \in A\ s\ L\ d\ v\ s = L_a\ (d'\ s)\ v\ s$ **for** s
by *metis*
thus *?thesis*
using *assms d*
by (*fastforce simp*: *is-dec-det-def mk-dec-det-def* $L\text{-eq-}L_a$)
qed

lemma $\mathcal{L}_b\text{-sup-att-dec}'$:
assumes $d \in D_R\ \mathcal{L}_b\ v = L\ d\ v$
shows $\exists\, d' \in D_D.\ \nu\text{-improving}\ v\ (\text{mk-dec-det}\ d')$
using $\mathcal{L}_b\text{-sup-att-dec}$ $\nu\text{-improving-alt}$ *assms* **by** *force*

7.13.2 Deterministic Decision Rules are Optimal

lemma *opt-imp-opt-dec-det*:
assumes $p \in \Pi_{HR}\ \nu_b\ p = \nu_b\text{-opt}$
shows $\exists\, d \in D_D.\ \nu_b\ (\text{mk-stationary-det}\ d) = \nu_b\text{-opt}$
proof −
have *aux*: $L\ (\text{as-markovian}\ p\ (\text{return-pmf}\ s)\ 0)\ \nu_b\text{-opt}\ s = \nu_b\text{-opt}\ s$ **for** s
proof −
let *?ps* = *as-markovian p* (*return-pmf s*)

have *markovian-suc-le*: ν_b (*mk-markovian* (λn. *as-markovian p* (*return-pmf s*) (*Suc n*))) $\leq \nu_b$-*opt*
using *is-*$\Pi_{M\,R}$*-as-markovian assms* **by** (*auto simp*: *is-policy-def mk-markovian-def*)
have *aux-le*: $\bigwedge x\ f\ g.\ f \leq g \Longrightarrow$ *apply-bfun f x* $\leq$ *apply-bfun g x*
unfolding *less-eq-bfun-def* **by** *auto*
have ν_b-*opt s* = ν_b (*mk-markovian ?ps*) *s*
using *assms* ν_b-*as-markovian* **by** *metis*
also have ... = *L* (*?ps 0*) (ν_b (*mk-markovian* (λn. *?ps* (*Suc n*)))) *s*
using ν-*step* **by** *blast*
also have ... $\leq$ *L* (*?ps 0*) (ν_b-*opt*) *s*
unfolding *L-def* **using** *markovian-suc-le* $\mathcal{P}_1$-*mono* **by** (*auto intro*!: *mult-left-mono*)
finally have ν_b-*opt s* $\leq$ *L* (*?ps 0*) (ν_b-*opt*) *s* **.**
have *as-markovian p* (*return-pmf s*) *0* $\in$ D_R
using *is-*$\Pi_{M\,R}$*-as-markovian assms* **by** *fast*
have *L* (*?ps 0*) ν_b-*opt* $\leq$ ν_b-*opt*
using ‹*?ps 0* $\in$ D_R› *L-le-*$\mathcal{L}_b$[*of ?ps 0* ν_b-*opt*] **by** *simp*
thus *L* (*?ps 0*) ν_b-*opt s* = ν_b-*opt s*
using ‹ν_b-*opt s* $\leq$ (*L* (*?ps 0*) ν_b-*opt*) *s*› **by** (*auto intro*!: *antisym*)
qed
have *L* (*p* []) *v s* = *L* (*as-markovian p* (*return-pmf s*) *0*) *v s* **for** *v s*
by (*auto simp*: *L-def* $\mathcal{P}_1$*.rep-eq K-st-def*)
hence *L* (*p* []) ν_b-*opt* = ν_b-*opt*
using *aux* **by** *auto*
hence $\exists\, d \in D_D.$ *L* (*mk-dec-det d*) ν_b-*opt* = ν_b-*opt*
using $\mathcal{L}_b$-*sup-att-dec assms*(*1*) $\mathcal{L}_b$-*opt is-policy-def mem-Collect-eq* **by** *metis*
thus *?thesis*
using *conserving-imp-opt*′ ν-*conserving-alt*′ **by** *blast*
qed

7.13.3 Optimal Decision Rules for Finite Action Spaces

lemma *ex-opt-act*:
assumes $\bigwedge s.$ *finite* (*A s*)
shows $\exists\, a \in A\ s.$ L_a *a* (*v* :: - $\Rightarrow_b$ -) *s* = $\mathcal{L}_b$ *v s*
unfolding $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-eq-SUP-det SUP-step-det-eq*
using *arg-max-on-in*[*OF assms A-ne*]
by (*auto simp*: *cSup-eq-Sup-fin Sup-fin-Max assms A-ne finite-arg-max-eq-Max*[*symmetric*])

lemma *ex-opt-dec-det*:
assumes $\bigwedge s.$ *finite* (*A s*)
shows $\exists\, d \in D_D.$ *L* (*mk-dec-det d*) (*v* :: - $\Rightarrow_b$ -) = $\mathcal{L}_b$ *v*
unfolding *is-dec-det-def mk-dec-det-def*
using *ex-opt-act*[*OF assms*] *someI-ex*
apply (*auto intro*!: *exI*[*of* - ‹$\lambda s.$ *SOME a.* $a \in A\ s \wedge L_a\ a\ v\ s = \mathcal{L}_b$

v s›] *bfun-eqI*)
apply (*smt* (*verit*, *best*) *someI-ex*)
apply (*subst L-eq-*L_a)
apply (*subst expectation-return-pmf*)
by (*smt* (*verit*, *best*) *someI-ex*)

lemma *thm-6-2-10*:
assumes $\bigwedge s$. *finite* (*A s*)
shows $\exists\, d \in D_D.\ \nu_b\text{-}opt = \nu_b$ (*mk-stationary-det d*)
using *assms conserving-imp-opt*′ $\mathcal{L}_b$*-opt L-*ν*-fix-iff ex-opt-dec-det*
by *metis*

7.13.4 Existence of Epsilon-Optimal Policies

lemma *ex-det-eps*:
assumes *0* < *e*
shows $\exists\, d \in D_D.\ \mathcal{L}_b\ v \leq L$ (*mk-dec-det d*) *v* + *e* $*_R$ *1*
proof −
have $\exists\, a \in A\ s.\ \mathcal{L}_b\ v\ s \leq L_a\ a\ v\ s + e$ **for** *s*
proof −
have *bdd-above* ((λ*a*. L_a *a v s*) ‘ *A s*)
using L_a*-le* **by** (*auto intro*!: *boundedI bounded-imp-bdd-above*)
hence $\exists\, a \in A\ s.\ \mathcal{L}_b\ v\ s - e < L_a\ a\ v\ s$
unfolding $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-eq-SUP-det SUP-step-det-eq*
by (*auto simp*: *less-cSUP-iff*[*OF A-ne*, *symmetric*] ‹*0* < *e*›)
thus $\exists\, a \in A\ s.\ \mathcal{L}_b\ v\ s \leq L_a\ a\ v\ s + e$
by *force*
qed
thus *?thesis*
unfolding *mk-dec-det-def is-dec-det-def*
by (*auto simp*: *L-def* $\mathcal{P}_1$*.rep-eq bind-return-pmf K-st-def less-eq-bfun-def*)
metis
qed

lemma *thm-6-2-11*:
assumes *eps* > *0*
shows $\exists\, d \in D_D.\ \nu_b\text{-}opt \leq \nu_b$ (*mk-stationary-det d*) + *eps* $*_R$ *1*
proof −
have (*1*−*l*) ∗ *eps* > *0*
by (*simp add*: *assms*)
then obtain *d* **where** $d \in D_D$ **and** *d*: $\mathcal{L}_b\ \nu_b\text{-}opt \leq L$ (*mk-dec-det d*) ν_b*-opt* + ((*1*−*l*)∗*eps*) $*_R$ *1*
using *ex-det-eps*[*of* - ν_b*-opt*] **by** *auto*
let *?d* = *mk-dec-det d*
let *?lK* = *l* $*_R$ $\mathcal{P}_1$ *?d*
let *?lK-opt* = *l* $*_R$ $\mathcal{P}_1$ *?d* ν_b*-opt*
have ν_b*-opt* ≤ *r-dec*$_b$ *?d* + *?lK-opt* + ((*1*−*l*)∗*eps*) $*_R$ *1*
using *L-def* $\mathcal{L}$*-fix-imp-opt d* **by** *simp*
hence ν_b*-opt* − *?lK-opt* − ((*1*−*l*)∗*eps*) $*_R$ *1* ≤ *r-dec*$_b$ *?d*

by (*simp add*: *cancel-ab-semigroup-add-class.diff-right-commute diff-le-eq*)
hence ($\sum i.$ *?lK* ^^ *i*) (ν_b-*opt* − *?lK-opt* − ((*1*−*l*)∗*eps*) $*_R$ *1*) ≤ ν_b (*mk-stationary ?d*)
using *lemma-6-1-2-b suminf-cong* **by** (*simp add*: *blincomp-scaleR-right* ν*-stationary*)
hence (($\sum i.$ *?lK* ^^ *i*) o_L (*id-blinfun* − *?lK*)) ν_b-*opt* − ($\sum i.$ *?lK* ^^ *i*) (((*1*−*l*)∗*eps*) $*_R$ *1*)
≤ (ν_b (*mk-stationary ?d*))
by (*simp add*: *blinfun.diff-right blinfun.diff-left blinfun.scaleR-left*)
hence *le*: ν_b-*opt* − ($\sum i.$ *?lK* ^^ *i*) (((*1*−*l*)∗*eps*) $*_R$ *1*) ≤ ν_b (*mk-stationary ?d*)
by (*auto simp*: *inv-norm-le'*)
have *s*: *summable* (λ*i*. (*l* $*_R$ $\mathcal{P}_1$ *?d*) ^^ *i*)
using *convergent-disc-*$\mathcal{P}_1$ *summable-iff-convergent'*
by (*simp add*: *blincomp-scaleR-right summable-iff-convergent'*)
have ($\sum i.$ *?lK* ^^ *i*) (((*1*−*l*)∗*eps*) $*_R$ *1*) = *eps* $*_R$ *1*
proof −
have ($\sum i.$ *?lK* ^^ *i*) (((*1*−*l*)∗*eps*) $*_R$ *1*) = ((*1*−*l*)∗*eps*) $*_R$ ($\sum i.$ *?lK*^^*i*) *1*
using *blinfun.scaleR-right* **by** *blast*
also have ... = ((*1*−*l*)∗*eps*) $*_R$ ($\sum i.$ (*?lK*^^*i*) *1*)
using *s* **by** (*auto simp*: *bounded-linear.suminf*[*of* λ*x*. *blinfun-apply x 1*])
also have ... = ((*1*−*l*)∗*eps*) $*_R$ ($\sum i.$ (*l* ^ *i*)) $*_R$ *1*
by (*auto simp*: *blinfun.scaleR-left blincomp-scaleR-right bounded-linear-scaleR-left*
bounded-linear.suminf[*of* λ*x*. *x* $*_R$ *1*])
also have ... = ((*1*−*l*)∗*eps*) $*_R$ (*1* / (*1*−*l*)) $*_R$ *1*
by (*simp add*: *suminf-geometric*)
also have ... = *eps* $*_R$ *1*
using *disc-lt-one* ‹*0* < (*1* − *l*) ∗ *eps*› **by** *auto*
finally show *?thesis* **.**
qed
thus *?thesis*
using ‹*d* ∈ D_D› *diff-le-eq le*
by *auto*
qed

lemma *ex-det-dist-eps*:
assumes *0* < (*e* :: *real*)
shows ∃ *d* ∈ D_D. *dist* ($\mathcal{L}_b$ *v*) (*L* (*mk-dec-det d*) *v*) ≤ *e*
proof −
obtain *d* **where** *d* ∈ D_D *L* (*mk-dec-det d*) *v* ≤ ($\mathcal{L}_b$ *v*)
and *h2*: $\mathcal{L}_b$ *v* ≤ *L* (*mk-dec-det d*) *v* + *e* $*_R$ *1*
using *assms ex-det-eps L-le-*$\mathcal{L}_b$ **by** *blast*
hence *0* ≤ $\mathcal{L}_b$ *v* − *L* (*mk-dec-det d*) *v*
by *simp*
moreover have $\mathcal{L}_b$ *v* − *L* (*mk-dec-det d*) *v* ≤ *e* $*_R$ *1*

```
    using h2 by (simp add: add.commute diff-le-eq)
  ultimately have ∀s. |(ℒb v) s −  L (mk-dec-det d) v s| ≤ e
    unfolding less-eq-bfun-def by auto
  hence dist (ℒb v) (L (mk-dec-det d) v) ≤ e
   unfolding dist-bfun.rep-eq by (auto intro!: cSUP-least simp: dist-real-def)
  thus ?thesis
    using ‹d ∈ DD›
    by auto
qed

lemma less-imp-ex-add-le: (x :: real) < y ⟹ ∃eps>0. x + eps ≤ y
  by (meson field-le-epsilon less-le-not-le nle-le)

lemma νb-opt-le-det: νb-opt s ≤ (⨆d ∈ DD. νb (mk-stationary-det d)
s)
proof (subst le-cSUP-iff, safe)
  fix y
  assume y < νb-opt s
  then obtain eps where 1: y ≤ νb-opt s − eps and eps > 0
    using less-imp-ex-add-le by force
  hence eps / 2 > 0 by auto
  obtain d where d ∈ DD and νb-opt s ≤ νb (mk-stationary-det d)
s + eps / 2
    using thm-6-2-11[OF ‹eps / 2 > 0›] by fastforce
  hence y < νb (mk-stationary-det d) s
    using ‹eps > 0› by (auto simp: diff-less-eq intro: le-less-trans[OF
1])
  thus ∃i∈DD. y < νb (mk-stationary-det i) s
    using ‹d ∈ DD› by blast
next
  show DD = {} ⟹ False
    using D-det-ne by blast
  show bdd-above ((λd. νb (mk-stationary-det d) s) ‘ DD)
     by (auto intro!: bounded-imp-bdd-above boundedI abs-ν-le simp:
νb.rep-eq)
qed

lemma νb-opt-eq-det: νb-opt s = (⨆d ∈ DD. νb (mk-stationary-det
d) s)
  using νb-le-opt-DD D-det-ne
  by (fastforce intro!: antisym[OF νb-opt-le-det] cSUP-least)


lemma lemma-6-3-1-a:
  assumes v0 ∈ bfun
  shows uniform-limit UNIV (λn. ((λv. ℒ (Bfun v)) ^^ n) v0) ν-opt
sequentially
proof −
  have ℒ-Bfun-eq: v0 ∈ bfun ⟹ ((λv. ℒ (Bfun v))^^n) v0 = (ℒb
```

$\widehat{}\widehat{}n$) (*Bfun v0*) **for** *n*
by (*induction n*) (*auto simp*: $\mathcal{L}_b$*.rep-eq apply-bfun-inverse*)
have *uniform-limit UNIV* (*λn.* ($\mathcal{L}_b$ $\widehat{}\widehat{}$ *n*) (*Bfun v0*)) ν_b*-opt sequentially*
by (*intro tendsto-bfun-uniform-limit*[*OF* $\mathcal{L}_b$*-lim*])
hence *uniform-limit UNIV* (*λn.* ($\mathcal{L}_b$ $\widehat{}\widehat{}$ *n*) (*Bfun v0*)) *ν-opt sequentially*
by (*simp add*: *ν-opt-bfun* ν_b*-opt.rep-eq*)
thus *?thesis*
by (*auto simp*: *assms* $\mathcal{L}$*-Bfun-eq*)
qed

lemma *dist-Suc-tendsto-zero*:
assumes (*λn. f n*) $\longrightarrow$ (*y*::-::*real-normed-vector*)
shows (*λn. dist* (*f n*) (*f* (*Suc n*))) $\longrightarrow$ *0*
using *assms tendsto-diff tendsto-norm LIMSEQ-Suc* **by** (*fastforce simp*: *dist-norm*)

lemma *dist-*$\mathcal{L}_b$*-tendsto*: (*λn. dist* (($\mathcal{L}_b$ $\widehat{}\widehat{}$ *n*) *v*) (($\mathcal{L}_b$ $\widehat{}\widehat{}$(*Suc n*)) *v*)) $\longrightarrow$ *0*
using $\mathcal{L}_b$*-lim* **by** (*fast intro*!: *dist-Suc-tendsto-zero*)

definition *max-L-ex s v* ≡ *has-arg-max* (*λa.* L_a *a v s*) (*A s*)

lemma ν_b*-fin-zero*[*simp*]: ν_b*-fin p 0* = *0*
by (*auto simp*: ν_b*-fin.rep-eq*)

lemma ν_b*-fin-Suc*[*simp*]:
ν_b*-fin* (*mk-stationary d*) (*Suc n*) = ν_b*-fin* (*mk-stationary d*) *n* + ((*l* $*_R$ $\mathcal{P}_1$ *d*)$\widehat{}\widehat{}$ *n*) (*r-dec*$_b$ *d*)
by (*auto simp*: $\mathcal{P}_X$*-sconst* ν_b*-fin.rep-eq ν-fin-eq-*$\mathcal{P}_X$ *blincomp-scaleR-right blinfun.scaleR-left*)

lemma ν_b*-fin-eq*: ν_b*-fin* (*mk-stationary d*) *n* = ($\sum$ *i* < *n*. ((*l* $*_R$ $\mathcal{P}_1$ *d*)$\widehat{}\widehat{}$ *i*)) (*r-dec*$_b$ *d*)
by (*induction n*) (*auto simp add*: *plus-blinfun.rep-eq*)

lemma *L-iter*: (*L d* $\widehat{}\widehat{}$ *m*) *v* = ν_b*-fin* (*mk-stationary d*) *m* + ((*l* $*_R$ $\mathcal{P}_1$ *d*)$\widehat{}\widehat{}$ *m*) *v*
proof (*induction m arbitrary*: *v*)
case (*Suc m*)
have (*L d* $\widehat{}\widehat{}$ *Suc m*) *v* = (*L d* $\widehat{}\widehat{}$ *m*) (*L d v*)
by (*simp add*: *funpow-Suc-right del*: *funpow.simps*)
also have ... = ν_b*-fin* (*mk-stationary d*) *m* + ((*l* $*_R$ $\mathcal{P}_1$ *d*) $\widehat{}\widehat{}$ *m*) (*L d v*)
using *Suc* **by** *simp*
also have ... = ν_b*-fin* (*mk-stationary d*) (*Suc m*) + ((*l* $*_R$ $\mathcal{P}_1$ *d*) $\widehat{}\widehat{}$ *Suc m*) *v*
unfolding *L-def*

by (*auto simp*: $\mathcal{P}_1$*-pow blinfun.bilinear-simps blincomp-scaleR-right funpow-swap1*)
finally show *?case* **.**
qed *simp*

lemma *bounded-stationary-*ν_b*-fin*: *bounded* ((*λx*. (*ν*$_b$*-fin* (*mk-stationary x*) *N*) *s*) ‘ *X*)
using *ν*$_b$*-fin.rep-eq abs-ν-fin-le* **by** (*auto intro*!: *boundedI*)

lemma *bounded-disc-*$\mathcal{P}_1$: *bounded* ((*λx*. (((*l* $*_R$ $\mathcal{P}_1$ *x*) ⁀⁀ *m*) *v*) *s*) ‘ *X*)
by (*auto simp*: $\mathcal{P}_X$*-const*[*symmetric*] *blinfun.bilinear-simps blincomp-scaleR-right*

intro!: *boundedI*[*of* - *l* ^ *m* ∗ *norm v*] *mult-left-mono order.trans*[*OF abs-le-norm-bfun*])

lemma *bounded-disc-*$\mathcal{P}_1$*′*: *bounded* ((*λx*. (($\mathcal{P}_1$ *x* ⁀⁀ *m*) *v*) *s*) ‘ *X*)
by (*auto simp*: $\mathcal{P}_X$*-const*[*symmetric*] *intro*!: *boundedI*[*of* - *norm v*] *order.trans*[*OF abs-le-norm-bfun*])

lemma *L-iter-le-*$\mathcal{L}_b$: *is-dec d* ⟹ (*L d* ⁀⁀ *n*) *v* ≤ ($\mathcal{L}_b$ ⁀⁀ *n*) *v*
using *order-trans*[*OF L-mono L-le-*$\mathcal{L}_b$] **by** (*induction n*) *auto*

end

7.14 More Restrictive MDP Locales

locale *MDP-fin-acts* = *discrete-MDP* +
assumes ⋀*s*. *finite* (*A s*)

locale *MDP-att-*$\mathcal{L}$ = *MDP-reward-disc A K r l*
for
A **and**
K :: *′s* ::*countable* × *′a* ::*countable* ⇒ *′s pmf* **and**
r **and** *l* +
assumes *Sup-att*: *max-L-ex* (*s* :: *′s*) *v*
begin
theorem $\mathcal{L}_b$*-eq-argmax-*L_a:
fixes *v* :: *′s* ⇒$_b$ *real*
assumes *is-arg-max* (*λa*. L_a *a v s*) (*λa*. *a* ∈ *A s*) *a*
shows $\mathcal{L}_b$ *v s* = L_a *a v s*
using L_a*-le assms A-ne* $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-eq-SUP-det SUP-step-det-eq*
by (*auto intro*!: *cSUP-upper2 antisym cSUP-least simp*: *is-arg-max-linorder*)

lemma L_a*-le-arg-max*: *a* ∈ *A s* ⟹ L_a *a v s* ≤ L_a (*arg-max-on* (*λa*. L_a *a v s*) (*A s*)) *v s*
using *Sup-att app-arg-max-ge*[*OF Sup-att*[*unfolded max-L-ex-def*]]
by (*simp add*: *arg-max-on-def*)

lemma *arg-max-on-in*: *has-arg-max f Q* $\Longrightarrow$ *arg-max-on f Q* $\in$ *Q*
using *has-arg-max-arg-max* **by** (*auto simp*: *arg-max-on-def*)

lemma $\mathcal{L}_b$*-eq-*L_a*-max*: $\mathcal{L}_b$ *v s* = L_a (*arg-max-on* (λ*a*. L_a *a v s*) (*A s*))
v s
using *app-arg-max-eq-SUP*[*symmetric*] *Sup-att max-L-ex-def*
by (*auto simp*: $\mathcal{L}_b$*-eq-SUP-det SUP-step-det-eq*)

lemma *ex-opt-det*: $\exists$ *d* $\in$ D_D. $\mathcal{L}_b$ *v* = *L* (*mk-dec-det d*) *v*
proof −
define *d* **where** *d* = (λ*s*. *arg-max-on* (λ*a*. L_a *a v s*) (*A s*))
have $\mathcal{L}_b$ *v s* = *L* (*mk-dec-det d*) *v s* **for** *s*
by (*auto simp*: *d-def* $\mathcal{L}_b$*-eq-*L_a*-max L-eq-*L_a*-det*)
moreover have *d* $\in$ D_D
using *Sup-att arg-max-on-in* **by** (*auto simp*: *d-def is-dec-det-def*
max-L-ex-def)
ultimately show *?thesis*
by *auto*
qed

lemma *ex-improving-det*: $\exists$ *d* $\in$ D_D. ν*-improving v* (*mk-dec-det d*)
using ν*-improving-alt ex-opt-det* **by** *auto*
end

locale *MDP-act* = *discrete-MDP A K* **for** *A* :: *'s*::*countable* $\Rightarrow$ *'a*::*countable*
set **and** *K* +
fixes *arb-act* :: *'a set* $\Rightarrow$ *'a*
assumes *arb-act-in*[*simp*]: *X* $\neq$ {} $\Longrightarrow$ *arb-act X* $\in$ *X*

locale *MDP-act-disc* = *MDP-act A K* + *MDP-att-*$\mathcal{L}$ *A K r l*
for *A* :: *'s*::*countable* $\Rightarrow$ *'a*::*countable set* **and** *K r l*
begin

lemma *is-opt-act-some*: *is-opt-act v s* (*arb-act* (*opt-acts v s*))
using *arb-act-in*[*of* {*a*. *is-arg-max* (λ*a*. L_a *a v s*) (λ*a*. *a* $\in$ *A s*) *a*}]
Sup-att has-arg-max-def
unfolding *max-L-ex-def is-opt-act-def* **by** *auto*

lemma *some-opt-acts-in-A*: *arb-act* (*opt-acts v s*) $\in$ *A s*
using *is-opt-act-some* **unfolding** *is-opt-act-def is-arg-max-def* **by**
auto

lemma ν*-improving-opt-acts*: ν*-improving v0* (*mk-dec-det* (λ*s*. *arb-act*
(*opt-acts* (*apply-bfun v0*) *s*)))
using *is-opt-act-def is-opt-act-some some-opt-acts-in-A*
by (*subst* ν*-improving-alt*) (*fastforce simp*: *L-eq-*L_a*-det* $\mathcal{L}_b$*-eq-argmax-*L_a
is-dec-det-def)+

end

locale *MDP-finite-type* = *MDP-reward-disc A K r l*
  for *A* **and** *K* :: *'s* :: *finite* × *'a* :: *finite* ⇒ *'s pmf* **and** *r l*

end

References

[1] J. Hölzl and T. Nipkow. Markov models. *Archive of Formal Proofs*, Jan. 2012. https://isa-afp.org/entries/Markov_Models.html, Formal proof development.

[2] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.