

Markov Decision Processes with Rewards

Maximilian Schäffeler and Mohammad Abdulaziz

February 6, 2026

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 | 3 |
| 1.1 | Definition | 3 |
| 1.2 | Supremum Norm | 6 |
| 1.3 | Complete Space | 8 |
| 1.4 | Order Instance | 12 |
| 1.5 | Miscellaneous | 12 |
| 1.6 | Bounded Functions and Vectors | 14 |
| 2 | Bounded Linear Functions | 15 |
| 2.1 | Composition | 15 |
| 2.2 | Power | 15 |
| 2.3 | Geometric Sum | 17 |
| 2.4 | Inverses | 20 |
| 2.5 | Norm | 22 |
| 2.6 | Miscellaneous | 26 |
| 3 | Auxiliary Lemmas | 29 |
| 3.1 | Summability | 29 |
| 3.2 | Infinite sums | 30 |
| 3.3 | Bounded Functions | 30 |
| 3.4 | Push-Forward of a Bounded Function | 30 |
| 3.5 | Boundedness | 31 |

| | | |
|----------|--|-----------|
| 3.6 | Probability Theory | 31 |
| 3.7 | Argmax | 34 |
| 3.8 | Contraction Mappings | 36 |
| 3.9 | Limits | 38 |
| 3.10 | Supremum | 39 |
| 4 | Least argmax | 41 |
| 5 | Discrete-Time Markov Decision Processes with Arbitrary State Spaces | 42 |
| 5.1 | Definition and Basic Properties | 44 |
| 5.2 | Policies | 46 |
| 5.3 | Successor Policy | 48 |
| 5.4 | Single-Step Distribution | 49 |
| 5.5 | Initial State-Action Distribution | 51 |
| 5.6 | Sequence Space of the MDP | 52 |
| 5.7 | Measurability of the Sequence Space | 53 |
| 5.8 | Iteration Rule | 60 |
| 5.9 | Stream Space of the MDP | 65 |
| 6 | Markov Decision Processes with Discrete State Spaces | 67 |
| 6.1 | Policies | 70 |
| 6.1.1 | Successor Policy | 73 |
| 6.2 | Stream Space of the MDP | 73 |
| 6.2.1 | Initial State-Action Distribution | 73 |
| 6.2.2 | Decomposition of the Stream Space | 75 |
| 6.2.3 | A Denotational View on the Stochastic Process | 77 |
| 6.2.4 | State Process | 78 |
| 6.2.5 | The Conditional Distribution of Actions | 79 |
| 6.2.6 | Action Process | 83 |
| 6.3 | Restriction to Markovian Policies | 83 |
| 6.4 | MDPs without Initial Distribution | 84 |
| 7 | Markov Decision Processes with Rewards | 89 |
| 7.1 | Util | 89 |
| 7.1.1 | Basic Properties of rewards | 89 |
| 7.1.2 | Infinite discounted sums | 90 |
| 7.2 | Total Reward for Single Traces | 90 |
| 7.3 | Expected Finite-Horizon Discounted Reward | 90 |
| 7.4 | Expected Total Discounted Reward | 91 |
| 7.5 | Reward of a Decision Rule | 91 |
| 7.6 | Transition Probability Matrix for MDPs | 92 |
| 7.7 | The Bellman Operator | 94 |
| 7.7.1 | Bellman Operator for Single Actions | 94 |
| 7.8 | Optimality Equations | 96 |
| 7.8.1 | Equivalences involving \mathcal{L}_b | 96 |
| 7.9 | Monotonicity | 98 |
| 7.10 | Optimal Reward | 103 |
| 7.11 | Properties of Solutions of the Optimality Equations | 107 |
| 7.12 | Solutions to the Optimality Equation | 109 |

| | | |
|--------|--|-----|
| 7.12.1 | \mathcal{L}_b and L are Contraction Mappings | 109 |
| 7.12.2 | Existence of a Fixpoint of \mathcal{L}_b | 111 |
| 7.13 | Existence of Optimal Policies | 111 |
| 7.13.1 | Conserving Decision Rules are Optimal | 112 |
| 7.13.2 | Deterministic Decision Rules are Optimal | 113 |
| 7.13.3 | Optimal Decision Rules for Finite Action Spaces | 114 |
| 7.13.4 | Existence of Epsilon-Optimal Policies | 115 |
| 7.14 | More Restrictive MDP Locales | 119 |

1 Bounded Functions

theory *Bounded-Functions*

imports

HOL.Topological-Spaces

HOL-Analysis.Uniform-Limit

HOL-Probability.Probability

begin

1.1 Definition

definition $bfun = \{f. \text{bounded } (range\ f)\}$

typedef (**overloaded**) $('a, 'b) \text{ bfun } (\langle(- \Rightarrow_b -)\rangle [22] 21) =$

$bfun::('a \Rightarrow 'b :: \text{metric-space}) \text{ set}$

morphisms $\text{apply-bfun } Bfun$

by $(\text{fastforce simp: bounded-def bfun-def})$

declare $[[\text{coercion } \text{apply-bfun} :: ('a \Rightarrow_b ('b :: \text{metric-space})) \Rightarrow 'a \Rightarrow 'b]]$

setup-lifting $\text{type-definition-bfun}$

lemma $\text{bounded-apply-bfun}[\text{intro}, \text{simp}]: \text{bounded } ((\text{apply-bfun } x) \text{ ' } X)$
using apply-bfun **by** $(\text{fastforce simp: bfun-def bounded-def})$

lemma $\text{apply-bfun-bdd-above}[\text{simp}, \text{intro}]:$

fixes $f :: 'c \Rightarrow_b \text{real}$

shows $\text{bdd-above } (f \text{ ' } X)$

by $(\text{auto intro: bounded-imp-bdd-above})$

lemma $\text{bfun-eqI}[\text{intro}]: (\bigwedge x. \text{apply-bfun } f\ x = \text{apply-bfun } g\ x) \Longrightarrow f = g$

by transfer auto

lemma $\text{bfun-eqD}[\text{dest}]: f = g \Longrightarrow (\bigwedge x. \text{apply-bfun } f\ x = \text{apply-bfun } g\ x)$

by auto

lemma $\text{bfunE}:$

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

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

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

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

instantiation *bfun* :: $(\text{type}, \text{metric-space}) \text{ metric-space}$
begin

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

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

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

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

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

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

```

instance
proof
  fix f g h :: 'a  $\Rightarrow_b$  'b
  show  $dist\ f\ g = 0 \iff f = g$ 
  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  $\lambda I\ X. Pi\ I\ X \cap bfun$ 
  by auto

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

1.2 Supremum Norm

instantiation *bfun* :: $(\text{type}, \text{real-normed-vector}) \text{ real-vector}$
begin

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

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

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

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

lemma *bfun-normI[intro]*: $(\bigwedge x. \text{norm } (f x) \leq b) \Longrightarrow f \in \text{bfun}$
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** $\lambda-. 0$

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'*: $f \in \text{bfun} \implies (\lambda x. a * f x) \in \text{bfun}$ **for** $f :: 'a \Rightarrow$
real

using *scaleR-cont[of f a]* **by** *auto*

lemma *bfun-norm-le-SUP-norm*:

$f \in \text{bfun} \implies \text{norm} (f x) \leq (\text{SUP } x. \text{norm} (f x))$

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* :: $('a, 'b) \text{bfun} \Rightarrow \text{real}$

where *norm-bfun* $f = \text{dist } f 0$

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

instance

proof

fix $a :: \text{real}$

fix $f g :: ('a, 'b) \text{bfun}$

show $\text{dist } f g = \text{norm} (f - g)$

unfolding *norm-bfun-def*

by *transfer (simp add: dist-norm)*

show $\text{norm } (f + g) \leq \text{norm } f + \text{norm } g$
unfolding *norm-bfun-def*
by *transfer*
(auto intro!: cSUP-least norm-triangle-le add-mono bfun-norm-le-SUP-norm
simp: dist-norm)
show $\text{norm } (a *_R f) = |a| * \text{norm } f$
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'*: $\text{norm } f = (\bigsqcup x. \text{norm } ((f :: 'a \Rightarrow_b 'b :: \text{real-normed-vector}) x))$
by *(subst norm-conv-dist, simp add: dist-bfun.rep-eq)*

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

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

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

1.3 Complete Space

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

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

lemma *complete-bfun*:
assumes *cauchy-f*: $\text{Cauchy } (f :: \text{nat} \Rightarrow ('a, 'b :: \{\text{complete-space, real-normed-vector}\}) \text{ bfun})$
shows *convergent f*

proof –
let $?f = \lambda x. \text{lim } (\lambda n. f n x)$

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

ric-CauchyD)+

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

have *lim-f-bfun*: $?f \in \text{bfun}$
proof –
have $\exists b. \forall x. \text{norm} (\text{lim} (\lambda n. f\ n\ x)) \leq b$
proof –
obtain $N\ b$ **where** *dist-N*: $\text{dist} (f\ n\ x) (f\ m\ x) < b$ **if** $n \geq N\ m$
 $\geq N$ **for** $x\ m\ n$
using *metric-CauchyD*[*OF cauchy-f zero-less-numeral*] *dist-fun-lt-imp-dist-val-lt*
by *metis*
have *aux*: $\text{norm} (\text{lim} (\lambda n. f\ n\ x)) \leq b + \text{norm} (f\ N\ x)$ **for** x
proof–
from *conv-fx*[*unfolded convergent-LIMSEQ-iff*]
have *tendsto-f-N*: $(\lambda n. f\ (n + N)\ x) \longrightarrow ?f\ x$
by (*auto dest*: *LIMSEQ-ignore-initial-segment*)
hence *tendsto-f-dist*: $(\lambda n. \text{dist} (f\ (n + N)\ x) (f\ N\ x)) \longrightarrow$
 $\text{dist} (?f\ x) (f\ N\ x)$
by (*auto intro*: *tendsto-intros*)
have $\text{dist} (f\ (n + N)\ x) (f\ N\ x) \leq b$ **for** n
by (*auto intro!*: *less-imp-le simp: dist-N*)
hence $\text{dist} (?f\ x) (f\ N\ x) \leq b$
using *lim-le*[*OF convergentI*[*OF tendsto-f-dist*]]
by (*auto simp*: *limI*[*OF tendsto-f-dist, symmetric*])
thus $\text{norm} (?f\ x) \leq b + \text{norm} (f\ N\ x)$
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*: $\text{apply-bfun} (Bfun\ ?f) = ?f$
using *bfun.Bfun-inverse* **by** *blast*

have $f \longrightarrow Bfun\ ?f$
proof –
have $\bigwedge e. e > 0 \implies \exists N. \forall n \geq N. \text{dist} (Bfun\ ?f) (f\ n) < e$
proof –
fix $e :: \text{real}$
assume $e > 0$
hence $\exists N. \forall n \geq N. \forall m \geq N. \text{dist} (f\ n) (f\ m) < 0.5 * e$ (**is**
 $\exists N. \forall n \geq N. \forall m \geq N. ?P\ n\ m\ N\ e$)

```

    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* ($\lambda s. \text{norm} (\text{apply-bfun } v \ s)$))

proof –

obtain *b* **where** $\forall s. \text{norm} (v \ s) \leq b$

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* $v \in \text{borel-measurable } S$

assumes ($v :: 'a \Rightarrow 'b :: \{\text{second-countable-topology, banach}\}$) $\in \text{bfun}$

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 ($v :: 'a \Rightarrow 'c :: \{\text{euclidean-space}\}$) $\in \text{bfun}$

shows ($\lambda S. \int x. v \ x \ \partial(S :: 'a \ \text{pmf})$) $\in \text{bfun}$

proof –

obtain *b* **where** *bH*: $\forall x. \text{norm} (v \ x) \leq b$

using *bfun-norm-le-SUP-norm assms* **by** *fast*

have ($\int x. \text{norm} (v \ x) \ \partial S$) $\leq b$ **for** *S* :: *'a pmf*

using $\langle v \in \text{bfun} \rangle$ *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 $\forall S :: 'a \ \text{pmf}. \text{norm} (\int x. (v \ x) \ \partial S) \leq b$

using *integral-norm-bound order-trans* **by** *blast*

thus *?thesis*

unfolding *bfun-def*

by (*auto intro: boundedI*)

qed

lemma *scale-bfun[intro!]*: $f \in \text{bfun} \implies (\lambda x. (k :: \text{real}) * f \ x) \in \text{bfun}$

using *scaleR-cont[of f k]* **by** *auto*

lemma *bfun-spec[intro]*: $f \in \text{bfun} \implies (\lambda x. f \ (g \ x)) \in \text{bfun}$

unfolding *bfun-def bounded-def* **by** *auto*

lemma *apply-bfun-bfun[simp]*: $\text{apply-bfun } f \in \text{bfun}$

using *apply-bfun* .

lemma *bfun-integral-bound*[*intro*]: $(v :: 'a \Rightarrow 'c :: \{euclidean-space\}) \in bfun \implies$
 $(\lambda S. \int x. v x \partial((F S) :: 'a pmf)) \in bfun$
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*]: $(\bigwedge x. apply-bfun f x \leq apply-bfun g x) \implies f \leq g$

unfolding *less-eq-bfun-def*

by *auto*

lemma *less-eq-bfunD*[*dest*]: $f \leq g \implies (\bigwedge x. apply-bfun f x \leq apply-bfun g x)$

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)  $\implies$  f  $\in$  bfun
  by (simp add: bfun-def)

lemma finite-bfun[simp]: ( $\lambda(i::\text{finite}). f\ i$ )  $\in$  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 (( $\lambda b. (F\ b)\ x$ ) ' S)
proof -
  obtain b where  $\forall x \in S. \text{norm}\ (F\ x) \leq b$ 
    by (meson assms bounded-pos image-eqI)
  thus bounded (( $\lambda b. (F\ b)\ x$ ) ' 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 ( $\lambda n. F\ n\ x$ )  $\longrightarrow$  y\ x
proof -
  have aux: ( $\lambda n. \text{dist}\ (F\ n)\ y$ )  $\longrightarrow$  0
    using h
    using tendsto-dist-iff by blast
  have  $\bigwedge n. \text{dist}\ (F\ n\ x)\ (y\ x) \leq \text{dist}\ (F\ n)\ y$ 
    unfolding dist-bfun-def
    using Bounded-Continuous-Function.bounded-dist-le-SUP-dist by
fastforce
  hence  $\bigwedge n. \text{norm}\ (\text{dist}\ (F\ n\ x)\ (y\ x)) \leq \text{norm}(\text{dist}\ (F\ n)\ y)$ 
    by auto
  hence ( $\lambda n. \text{dist}\ (F\ n\ x)\ (y\ x)$ )  $\longrightarrow$  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]*: $(\$) x \in \text{bfun}$
using *finite-bfun*.

lemma *norm-bfun-le-norm-vec*: $\text{norm} (\text{bfun.Bfun } ((\$) (x :: \text{real}^c :: \text{finite}))) \leq \text{norm } x$

proof –

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

qed

lemma *bounded-linear-bfun-nth*: $\text{bounded-linear } f \implies \text{bounded-linear } (\lambda v. \text{bfun.Bfun } ((\$) (f v)))$

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*:

$\text{norm} (\text{vec-lambda } (\text{apply-bfun } (x :: 'd::\text{finite} \Rightarrow_b \text{real}))) \leq \text{norm } x * \text{card } (\text{UNIV} :: 'd \text{ set})$

proof –

have $\text{norm} (\text{vec-lambda } (\text{apply-bfun } x)) \leq (\sum i \in \text{UNIV} . |(\text{apply-bfun } x i)|)$

using *L2-set-le-sum-abs*

unfolding *norm-vec-def L2-set-def*

by *auto*

also have $\dots \leq (\text{card } (\text{UNIV} :: 'd \text{ set}) * (\bigsqcup xa. |\text{apply-bfun } x xa|))$

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:  $F \circ_L G \circ_L H = F \circ_L (G \circ_L H)$ 
  using blinfun-apply-inject by fastforce

```

```

lemma blinfun-compose-diff-right:  $f \circ_L (g - h) = (f \circ_L g) - (f \circ_L h)$ 
  by (auto intro!: blinfun-eqI simp: blinfun.bilinear-simps)

```

2.2 Power

overloading

```

  blinfunpow  $\equiv$  compow ::  $\text{nat} \Rightarrow ('a::\text{real-normed-vector} \Rightarrow_L 'a) \Rightarrow ('a$ 
 $\Rightarrow_L 'a)$ 

```

begin

```

primrec blinfunpow ::  $\text{nat} \Rightarrow ('a::\text{real-normed-vector} \Rightarrow_L 'a) \Rightarrow ('a$ 
 $\Rightarrow_L 'a)$ 

```

where

```

  blinfunpow 0 f = id-blinfun
  | blinfunpow (Suc n) f = f o_L blinfunpow n f

```

end

```

lemma bounded-pow-blinfun[intro]:
  assumes bounded ( $\text{range } (F::\text{nat} \Rightarrow 'a::\text{real-normed-vector} \Rightarrow_L 'a)$ )
  shows bounded ( $\text{range } (\lambda t. (F t) \wedge (Suc n))$ )
  using assms proof –
  assume bounded ( $\text{range } F$ )
  then obtain b where bh:  $\bigwedge x. \text{norm } (F x) \leq b$ 
  by (auto simp: bounded-iff)
  hence  $\text{norm } ((F x) \wedge (Suc n)) \leq b \wedge (Suc n)$  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*: $(a *_{\mathbb{R}} (F :: 'a :: \text{real-normed-vector} \Rightarrow_L 'a)) \widehat{\sim} t = a \widehat{\sim} t *_{\mathbb{R}} F \widehat{\sim} t$

by (*induction t*) (*auto intro: blinfun-eqI simp: blinfun.scaleR-left blinfun.scaleR-right*)

lemma *summable-inv-Q*:

fixes $Q :: 'a :: \text{banach} \Rightarrow_L 'a$

assumes *onorm-le*: $\text{norm } (id\text{-blinfun} - Q) < 1$

shows *summable* $(\lambda n. (id\text{-blinfun} - Q) \widehat{\sim} n)$

using *onorm-le norm-blinfun-compose*

by (*force intro!: summable-ratio-test*)

lemma *blinfunpow-assoc*: $(F :: 'a :: \text{real-normed-vector} \Rightarrow_L 'a) \widehat{\sim} (Suc\ n) = (F \widehat{\sim} n) \circ_L F$

by (*induction n*) (*auto simp: blinfun-compose-assoc[symmetric]*)

lemma *norm-blinfunpow-le*: $\text{norm } ((f :: 'b :: \text{real-normed-vector} \Rightarrow_L 'b) \widehat{\sim} n) \leq \text{norm } f \widehat{\sim} n$

by (*induction n*) (*auto simp: norm-blinfun-id-le intro!: order.trans[OF norm-blinfun-compose] mult-left-mono*)

lemma *blinfunpow-nonneg*:

assumes $\bigwedge v. 0 \leq v \implies 0 \leq \text{blinfun-apply } (f :: ('b :: \{\text{ord, real-normed-vector}\} \Rightarrow_L 'b)) v$

shows $0 \leq v \implies 0 \leq (f \widehat{\sim} n) v$

by (*induction n*) (*auto simp: assms*)

lemma *blinfunpow-mono*:

assumes $\bigwedge u v. u \leq v \implies (f :: 'b :: \{\text{ord, real-normed-vector}\} \Rightarrow_L 'b) u \leq f v$

shows $u \leq v \implies (f \widehat{\sim} n) u \leq (f \widehat{\sim} n) v$

by (*induction n*) (*auto simp: assms*)

lemma *banach-blinfun*:

fixes $C :: 'b :: \{\text{real-normed-vector, complete-space}\} \Rightarrow_L 'b$

assumes *norm C < 1*

shows $\exists! v. C v = v \bigwedge v. (\lambda n. (C \widehat{\sim} n) v) \longrightarrow (THE v. C v = v)$

using *assms*

proof –

obtain v **where** $C v = v \bigwedge v'. C v' = v' \longrightarrow v' = v$

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** $(\forall v u. \text{norm } (C (v - u)) \leq l * \text{dist } v u) \ 0 \leq l \ l < 1$

by (*metis assms dist-norm norm-blinfun norm-imp-pos-and-ge*)

hence $1: \text{dist } (C v) (C u) \leq l * \text{dist } v u$ **for** $v u$

by (*simp add: blinfun.diff-right dist-norm*)

```

have 2:  $\text{dist } ((C \hat{\sim} n) v0) v \leq l \wedge n * \text{dist } v0 v$  for  $n v0$ 
  using  $\langle 0 \leq l \rangle$ 
  by (induction  $n$ ) (auto simp: mult.assoc
    intro!: mult-mono' order.trans[OF 1[of - v , unfolded \langle C v =
 $v \rangle]]$ )
  have ( $\lambda n. l \wedge n$ )  $\longrightarrow 0$ 
    by (simp add: LIMSEQ-realpow-zero \langle 0 \leq l \rangle \langle l < 1 \rangle)
  hence  $k: \bigwedge v0. (\lambda n. l \wedge n * \text{dist } v0 v) \longrightarrow 0$ 
    by (auto simp add: tendsto-mult-left-zero)
  have ( $\lambda n. \text{dist } ((C \hat{\sim} n) v0) v$ )  $\longrightarrow 0$  for  $v0$ 
    using  $k$  2 order-trans abs-ge-self
    by (subst Limits.tendsto-0-le[where ?K = 1, where ?f = (\lambda n. l
 $\wedge n * \text{dist } v0 v)]$ 
      (fastforce intro: eventuallyI)+
    hence  $\bigwedge v0. (\lambda n. (C \hat{\sim} n) v0) \longrightarrow v$ 
      using tendsto-dist-iff
      by blast
  thus ( $\lambda n. (C \hat{\sim} n) v0$ )  $\longrightarrow (THE v. C v = v)$  for  $v0$ 
    using theI'[of \lambda x. C x = x] \langle C v = v \rangle \langle \forall v'. C v' = v' \longrightarrow v' = v \rangle
    by blast
next
  show  $\text{norm } C < 1 \implies \exists !v. \text{blinfun-apply } C v = v$ 
    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  $Q :: 'a :: \text{banach} \Rightarrow_L 'a$ 
  assumes onorm-le:  $\text{norm } (id\text{-blinfun} - Q) < 1$ 
  shows ( $Q \circ_L (\sum i. (id\text{-blinfun} - Q) \hat{\sim} i)$ ) = id-blinfun
    and ( $\sum i. (id\text{-blinfun} - Q) \hat{\sim} i$ )  $\circ_L Q = id\text{-blinfun}$ 
proof -
  obtain  $b$  where  $bh: b < 1$   $\text{norm } (id\text{-blinfun} - Q) < b$ 
    using onorm-le dense
    by blast
  have  $0 < b$ 
    using le-less-trans[OF norm-ge-zero bh(2)] .
  have norm-le-aux:  $\text{norm } ((id\text{-blinfun} - Q) \hat{\sim} \text{Suc } n) \leq b \wedge (\text{Suc } n)$ 
for  $n$ 
  proof (induction  $n$ )
    case 0
    thus ?case
      using  $bh$ 
      by simp
  next
  case ( $\text{Suc } n$ )
  thus ?case

```

```

proof –
  have norm ((id-blinfun - Q)  $\widehat{\sim}$  Suc (Suc n))  $\leq$  norm (id-blinfun
- Q) * norm((id-blinfun - Q)  $\widehat{\sim}$  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 oL (∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i)) = (id-blinfun - (id-blinfun
- Q)  $\widehat{\sim}$  (Suc n)) for n
  by (induction n) (auto simp: bounded-bilinear.diff-left bounded-bilinear.add-right

      bounded-bilinear-blinfun-compose)
hence  $\bigwedge n$ . norm (id-blinfun - (Q oL (∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i)))
 $\leq$  b  $\widehat{\sim}$  Suc n
  using norm-le-aux
  by auto
hence l2: (λn. (id-blinfun - (Q oL (∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i))))
 $\longrightarrow$  0
  using ⟨0 < b⟩ bh
  by (subst Lim-transform-bound[where g=λn. b  $\widehat{\sim}$  Suc n]) (auto
intro!: tendsto-eq-intros)
have summable (λn. (id-blinfun - Q)  $\widehat{\sim}$  n)
  using onorm-le norm-blinfun-compose
  by (force intro!: summable-ratio-test)
hence (λn. ∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i)  $\longrightarrow$  (∑ i. (id-blinfun
- Q)  $\widehat{\sim}$  i)
  using summable-LIMSEQ'
  by blast
hence (λn. Q oL (∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i))  $\longrightarrow$  (Q oL (∑ i.
(id-blinfun - Q)  $\widehat{\sim}$  i))
  using bounded-bilinear-blinfun-compose
  by (subst Limits.bounded-bilinear.tendsto[where prod = (oL)] auto

hence (λn. id-blinfun - (Q oL (∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i)))
 $\longrightarrow$ 
  id-blinfun - (Q oL (∑ i. (id-blinfun - Q)  $\widehat{\sim}$  i))
  by (subst Limits.tendsto-diff) auto
thus (Q oL (∑ i. (id-blinfun - Q)  $\widehat{\sim}$  i)) = id-blinfun
  using LIMSEQ-unique l2 by fastforce

have ((∑ i≤n. (id-blinfun - Q)  $\widehat{\sim}$  i) oL Q) = (id-blinfun - (id-blinfun
- Q)  $\widehat{\sim}$  (Suc n)) for n
proof (induction n)
  case (Suc n)
  have sum (( $\widehat{\sim}$ ) (id-blinfun - Q)) {..Suc n} oL Q =
    (sum (( $\widehat{\sim}$ ) (id-blinfun - Q)) {..n} oL Q) + ((id-blinfun - Q)

```

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

hence $(\lambda n. id\text{-blinfun} - ((\sum i \leq n. (id\text{-blinfun} - Q) \sim i) \text{ } o_L \text{ } Q))$
 \longrightarrow
 $id\text{-blinfun} - ((\sum i. (id\text{-blinfun} - Q) \sim i) \text{ } o_L \text{ } Q)$
by (*subst Limits.tendsto-diff*) *auto*
thus $((\sum i. (id\text{-blinfun} - Q) \sim i) \text{ } o_L \text{ } Q) = id\text{-blinfun}$
using *LIMSEQ-unique l2* **by** *fastforce*
qed

lemma *inv-norm-le:*
fixes $Q :: 'a :: \text{banach} \Rightarrow_L 'a$
assumes $\text{norm } Q < 1$
shows $(id\text{-blinfun} - Q) \text{ } o_L (\sum i. Q \sim i) = id\text{-blinfun}$
 $(\sum i. Q \sim i) \text{ } o_L (id\text{-blinfun} - Q) = id\text{-blinfun}$

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

lemma *inv-norm-le'*:
fixes $Q :: 'a :: \text{banach} \Rightarrow_L 'a$
assumes $\text{norm } Q < 1$
shows $(\text{id-blinfun} - Q) ((\sum i. Q \hat{\sim} i) x) = x$
 $(\sum i. Q \hat{\sim} i) ((\text{id-blinfun} - Q) x) = x$
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 $\text{is-inverse}_L X Y \longleftrightarrow X \circ_L Y = \text{id-blinfun} \wedge Y \circ_L X = \text{id-blinfun}$

abbreviation $\text{invertible}_L X \equiv \exists X'. \text{is-inverse}_L X X'$

lemma *is-inverse_L-I[intro]*:
assumes $X \circ_L Y = \text{id-blinfun}$ $Y \circ_L X = \text{id-blinfun}$
shows $\text{is-inverse}_L X Y$
using *assms*
unfolding *is-inverse_L-def*
by *auto*

lemma *is-inverse_L-D[dest]*:
assumes $\text{is-inverse}_L X Y$
shows $X \circ_L Y = \text{id-blinfun}$ $Y \circ_L X = \text{id-blinfun}$
using *assms*
unfolding *is-inverse_L-def*
by *auto*

lemma *invertible_L-D[dest]*:
assumes $\text{invertible}_L f$
obtains g **where** $f \circ_L g = \text{id-blinfun}$ $g \circ_L f = \text{id-blinfun}$
using *assms*
by *auto*

lemma *invertible_L-I[intro]*:
assumes $f \circ_L g = \text{id-blinfun}$ $g \circ_L f = \text{id-blinfun}$
shows $\text{invertible}_L f$
using *assms*
by *auto*

lemma *is-inverse_L-comm*: $\text{is-inverse}_L X Y \longleftrightarrow \text{is-inverse}_L Y X$
by *auto*

lemma *is-inverse_L-unique*: $\text{is-inverse}_L f g \Longrightarrow \text{is-inverse}_L f h \Longrightarrow g$

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

lemma *is-inverse_L-ex1*: $is_inverse_L\ f\ g \implies \exists!h. is_inverse_L\ f\ h$
using *is-inverse_L-unique*
by *auto*

lemma *is-inverse_L-ex1'*: $\exists x. is_inverse_L\ f\ x \implies \exists!x. is_inverse_L\ f\ x$
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]*: $invertible_L\ X \implies (inv_L\ X)\ o_L\ X = id_blinfun$
using *is-inverse_L-ex1' inv_L-eq*
by *blast*

lemma *inv-app2[simp]*: $invertible_L\ X \implies X\ o_L\ (inv_L\ X) = id_blinfun$
using *is-inverse_L-ex1' inv_L-eq*
by *blast*

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

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

lemma *inv_L-inv_L[simp]*: $invertible_L\ X \implies inv_L\ (inv_L\ X) = X$
by (*metis inv_L-eq is-inverse_L-comm*)

lemma *inv_L-cancel-iff*:

assumes $\text{invertible}_L f$
shows $f x = y \iff x = \text{inv}_L f y$
by (*auto simp add: assms*)

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

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

lemma *is-inverse_L-compose*:
assumes $\text{invertible}_L f \text{ invertible}_L g$
shows $\text{is-inverse}_L (f \circ_L g) (\text{inv}_L g \circ_L \text{inv}_L f)$
by (*auto intro!: blinfun-eqI is-inverse_L-I[of - inv_L g \circ_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*: $\text{invertible}_L f \implies \text{invertible}_L g \implies \text{invertible}_L (f \circ_L g)$
using *is-inverse_L-compose*
by *blast*

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

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

2.5 Norm

lemma *bounded-range-subset*:
 $\text{bounded } (\text{range } f :: \text{real set}) \implies \text{bounded } (f ' X')$
by (*auto simp: bounded-iff*)

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

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

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

lemma *bfun-pos-zero[simp]*: $bfun\text{-}pos\ f = 0 \longleftrightarrow f = 0$
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*: $bfun\text{-}nonneg\ x - bfun\text{-}nonneg\ (-x) = x$
by (*auto simp: bfun-nonneg.rep-eq*)

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

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

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

lemma *bfun-pos-eq-nonneg*: $bfun\text{-}pos\ n = bfun\text{-}nonneg\ n + bfun\text{-}nonneg\ (-n)$
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 $\bigwedge v :: 'c \Rightarrow_b real. v \geq 0 \implies f\ v \geq 0$
shows $norm\ (f\ n) \leq norm\ (f\ (bfun\text{-}pos\ n))$

proof –

have *: $|f\ n\ i| \leq |f\ (bfun\text{-}pos\ n)\ i|$ **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 $norm\ (f\ n) \leq norm\ ((f\ (bfun\text{-}pos\ n)))$

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]*: $norm\ (bfun\text{-}pos\ f) = norm\ f$

proof –

have $norm\ (bfun\text{-}pos\ f) = (\bigsqcup i. |bfun\text{-}pos\ f\ i|)$

by (*auto simp add: norm-bfun-def'*)

also have ... = ($\bigsqcup i. |f i|$)
 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 $f :: ('c \Rightarrow_b \text{real}) \Rightarrow_L ('d \Rightarrow_b \text{real})$
 assumes $\bigwedge v :: 'c \Rightarrow_b \text{real}. v \geq 0 \implies f v \geq 0$
 shows $\text{norm } f = (\bigsqcup v \in \{v. v \geq 0\}. \text{norm } (f v) / \text{norm } v)$
 unfolding norm-blinfun.rep-eq onorm-def
 proof (rule antisym, rule cSUP-mono)
 have *: $\text{norm } (\text{blinfun-apply } f v) / \text{norm } v \leq \text{norm } f$ for v
 using norm-blinfun[of f]
 by (cases $v = 0$) (auto simp: pos-divide-le-eq)
 thus bdd-above $((\lambda v. \text{norm } (f v) / \text{norm } v) ' \{v. 0 \leq v\})$
 by (auto intro!: bounded-imp-bdd-above boundedI)
 show $\exists m \in \{v. 0 \leq v\}. \text{norm } (f n) / \text{norm } n \leq \text{norm } (f m) / \text{norm } m$
 for n
 using blinfun-mono-norm-pos[OF assms]
 by (cases $\text{norm } (\text{bfun-pos } n) = 0$)
 (auto intro!: frac-le exI[of - bfun-pos n] simp: less-eq-bfun-def
 bfun-pos.rep-eq)
 show $(\bigsqcup v \in \{v. 0 \leq v\}. \text{norm } (f v) / \text{norm } v) \leq (\bigsqcup x. \text{norm } (f x) / \text{norm } x)$
 using *
 by (auto intro!: cSUP-mono bounded-imp-bdd-above boundedI)
 qed auto

lemma norm-blinfun-normalized-le: $\text{norm } (\text{blinfun-apply } f v) / \text{norm } v \leq \text{norm } f$
 by (simp add: blinfun.bounded-linear-right le-onorm norm-blinfun.rep-eq)

lemma norm-blinfun-mono-eq-nonneg':
 fixes $f :: ('c \Rightarrow_b \text{real}) \Rightarrow_L ('d \Rightarrow_b \text{real})$
 assumes $\bigwedge v :: 'c \Rightarrow_b \text{real}. 0 \leq v \implies 0 \leq f v$
 shows $\text{norm } f = (\bigsqcup x \in \{x. \text{norm } x = 1 \wedge x \geq 0\}. \text{norm } (f x))$
 proof (subst norm-blinfun-mono-eq-nonneg[OF assms])
 show $(\bigsqcup v \in \{v. 0 \leq v\}. \text{norm } (f v) / \text{norm } v) =$
 $(\bigsqcup x \in \{x. \text{norm } x = 1 \wedge 0 \leq x\}. \text{norm } (f x))$
 proof (rule antisym, rule cSUP-mono)
 show $\{v :: 'c \Rightarrow_b \text{real}. 0 \leq v\} \neq \{\}$ by auto
 show bdd-above $((\lambda x. \text{norm } (f x)) ' \{x. \text{norm } x = 1 \wedge 0 \leq x\})$
 by (fastforce intro: order.trans[OF norm-blinfun[of f]] bounded-imp-bdd-above
 boundedI)
 show $\exists m \in \{x. \text{norm } x = 1 \wedge 0 \leq x\}. \text{norm } (f n) / \text{norm } n \leq \text{norm } (f m)$
 if $n \in \{v. 0 \leq v\}$ for n
 proof (cases $\text{norm } (\text{bfun-pos } n) = 0$)
 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 ( $\bigsqcup x \in \{x. \text{norm } x = 1 \wedge 0 \leq x\}. \text{norm } (f x) \leq (\bigsqcup v \in \{v. 0 \leq v\}. \text{norm } (f v) / \text{norm } v)$ )
  proof (rule cSUP-mono)
    show  $\{x::'c \Rightarrow_b \text{real}. \text{norm } x = 1 \wedge 0 \leq x\} \neq \{\}$ 
    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  $f :: ('c \Rightarrow_b \text{real}) \Rightarrow_L ('d \Rightarrow_b \text{real})$ 
  assumes  $\bigwedge v :: 'c \Rightarrow_b \text{real}. v \geq 0 \implies f v \geq 0$ 
  assumes  $\text{norm } x = 1 \wedge 0 \leq x$ 
  shows  $\text{norm } (f x) \leq \text{norm } (f 1)$ 
proof -
  have  $** : 0 \leq 1 - x$ 
  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)
    show bdd-above ( $\text{range } (\lambda x. \text{norm } (\text{apply-bfun } (\text{blinfun-apply } f 1) x))$ )
    using order.trans abs-le-norm-bfun norm-blinfun
    by (fastforce intro!: bounded-imp-bdd-above boundedI)
    show  $\exists m \in \text{UNIV}. \text{norm } (f x n) \leq \text{norm } (f 1 m)$  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  $f :: ('c \Rightarrow_b \text{real}) \Rightarrow_L ('d \Rightarrow_b \text{real})$ 
  assumes  $\bigwedge v :: 'c \Rightarrow_b \text{real}. v \geq 0 \implies f v \geq 0$ 
  shows  $\text{norm } f = \text{norm } (f 1)$ 
proof -
  have ( $\bigsqcup x \in \{x. \text{norm } x = 1 \wedge 0 \leq x\}. \text{norm } (f x) = \text{norm } (f 1)$ )
  proof (rule antisym, rule cSUP-least)
    show  $\{x::'c \Rightarrow_b \text{real}. \text{norm } x = 1 \wedge 0 \leq x\} \neq \{\}$ 
    by (auto intro!: exI[of - 1])
  qed

```

```

next
  show  $\bigwedge x. x \in \{x. \text{norm } x = 1 \wedge 0 \leq x\} \implies \text{norm } (f x) \leq \text{norm } (f 1)$ 
  by (simp add: assms norm-blinfun-mono-le-norm-one)
next
  show  $\text{norm } (f 1) \leq (\bigsqcup_{x \in \{x. \text{norm } x = 1 \wedge 0 \leq x\}}. \text{norm } (f x))$ 
  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  $(\lambda x. \text{apply-bfun } x \ i)$ 
  using norm-le-norm-bfun
  by (fastforce intro: bounded-linear-intro[of - 1])

```

```

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

```

```

lemma bounded-apply-blinfun:
  assumes bounded  $((F :: 'c \Rightarrow 'd::\text{real-normed-vector} \Rightarrow_L 'b::\text{real-normed-vector}) \text{ ' } S)$ 
  shows bounded  $((\lambda b. \text{blinfun-apply } (F \ b) \ x) \text{ ' } S)$ 
proof -
  obtain b where  $\forall x \in S. \text{norm } (F \ x) \leq b$ 
  by (meson  $\langle \text{bounded } (F \text{ ' } S) \rangle$  bounded-pos image-eqI)
  thus bounded  $((\lambda b. (F \ b) \ x) \text{ ' } S)$ 
  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:  $(\lambda n. X \ n) \longrightarrow L \implies (\lambda n. \text{blinfun-apply } (X \ n) \ u) \longrightarrow L \ u$ 
  using blinfun.bounded-bilinear-axioms
  by (auto simp: convergent-LIMSEQ-iff intro: Limits.bounded-bilinear.tendsto)

```

```

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

```

```

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

```

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

lemma *nonneg-blinfun-mono[dest]*: *nonneg-blinfun* $Q \implies u \leq v \implies Q u \leq Q v$
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 $\forall v \geq 0. \text{blinfun-apply } (f :: ('c \Rightarrow_b \text{real}) \Rightarrow_L ('c \Rightarrow_b \text{real})) v = \text{blinfun-apply } g v$
shows $f = g$
proof (*rule blinfun-eqI*)
fix $v :: 'c \Rightarrow_b \text{real}$
define $v1$ **where** $v1 = Bfun (\lambda x. \max (v x) 0)$
define $v2$ **where** $v2 = Bfun (\lambda x. - \min (v x) 0)$
have *in-bfun[simp]*: $(\lambda x. \max (v x) 0) \in bfun (\lambda x. - \min (v x) 0) \in bfun$
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*: $0 \leq (1 :: 'c \Rightarrow_b \text{real})$
by (*simp add: less-eq-bfunI*)

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

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

lemma *blinfun-apply-mono*: $\text{nonneg-blinfun } X \implies 0 \leq v \implies \text{blinfun-le } X \ Y \implies X \ v \leq Y \ v$

by (*simp add: blinfun.diff-left blinfun-le-def nonneg-blinfun-def*)

lemma *nonneg-blinfun-scaleR[intro]*: $\text{nonneg-blinfun } B \implies 0 \leq c \implies \text{nonneg-blinfun } (c *_{\mathbb{R}} B)$

by (*simp add: nonneg-blinfun-def scaleR-blinfun.rep-eq scaleR-nonneg-nonneg*)

lemma *nonneg-blinfun-compose[intro]*: $\text{nonneg-blinfun } B \implies \text{nonneg-blinfun } C \implies \text{nonneg-blinfun } (C \circ_L B)$

by (*simp add: nonneg-blinfun-def*)

lemma *matrix-le-norm-mono*:

assumes $\text{nonneg-blinfun } (C :: ('c \Rightarrow_b \text{real}) \Rightarrow_L ('c \Rightarrow_b \text{real}))$

and $\text{nonneg-blinfun } (D - C)$

shows $\text{norm } C \leq \text{norm } D$

proof –

have $\text{nonneg-blinfun } D$

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* $\langle \text{nonneg-blinfun } D \rangle$

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* $\langle \text{nonneg-blinfun } D \rangle$ *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*: $Y \subseteq X \implies \text{bounded } (f \text{ ` } X) \implies \text{bounded } (f \text{ ` } Y)$

by (*auto simp: bounded-def*)

lemma *bounded-subset-range*: $\text{bounded } (\text{range } f) \implies \text{bounded } (f \text{ ` } Y)$

using *bounded-subset subset-UNIV* **by** *metis*

lift-definition *bfun-if* :: $('b \Rightarrow \text{bool}) \Rightarrow ('b \Rightarrow_b 'c :: \text{metric-space}) \Rightarrow ('b \Rightarrow_b 'c) \Rightarrow ('b \Rightarrow_b 'c)$ **is** $\lambda b \ u \ v \ s. \text{ if } b \ s \text{ then } u \ s \text{ else } v \ s$

using *bounded-subset-range*

by (*auto simp: bfun-def*)

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

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

lemma *bfun-if-zero-le*: $0 \leq v \implies \text{bfun-if } b \ 0 \ v \leq v$
by (*metis (no-types, lifting) bfun-if.rep-eq le-less less-eq-bfun-def*)

lemma *bfun-if-eq*: $(\bigwedge i. P \ i \implies \text{apply-bfun } v \ i = \text{apply-bfun } u \ i) \implies (\bigwedge i. \neg P \ i \implies v \ i = \text{apply-bfun } w \ i) \implies \text{bfun-if } P \ u \ w = v$
by (*auto simp: bfun-if.rep-eq*)

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

lemma *summable-blinfun-apply*:
assumes *summable* ($f :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector} \Rightarrow_L 'a$)
shows *summable* ($\lambda n. f \ n \ v$)
using *assms tendsto-blinfun-apply*
unfolding *summable-def sums-def blinfun.sum-left[symmetric]*
by *auto*

lemma *blinfun-apply-suminf*:
assumes *summable* ($f :: \text{nat} \Rightarrow 'a :: \text{real-normed-vector} \Rightarrow_L 'a$)
shows $(\sum k. \text{blinfun-apply } (f \ k) \ v) = (\sum k. f \ k) \ v$
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 $c :: \text{real}$
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 ($f :: nat \Rightarrow 'x :: real\text{-normed-vector}$) \implies *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 $(\sum i < n. f\ i)\ x = (\sum i < n. apply\ bfun\ (f\ i)\ x)$
by (*induction n*) *auto*

3.4 Push-Forward of a Bounded Function

lemma *integrable-bfun-prob-space* [*simp*]:
integrable (*measure-pmf* P) ($\lambda t. apply\ bfun\ f\ (F\ t) :: real$)

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**
 $\lambda c\ f\ s. measure\ pmf.\ expectation\ (c\ s)\ f$
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  $\bigwedge s. (\int s'. norm (x s') \partial d s) \leq norm x$ 
    using measure-pmf.prob-space-axioms norm-le-norm-bfun[of x]
    by (auto intro!: prob-space.integral-le-const)
  hence aux:  $\bigwedge s. norm (\int s'. x s' \partial d s) \leq norm x$ 
    using integral-norm-bound order-trans by blast
  have  $norm (push-exp d x) = (\bigsqcup s. norm (\int s'. x s' \partial d s))$ 
    unfolding norm-bfun-def'
    by auto
  also have  $\dots \leq norm x$ 
    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  $onorm (push-exp d) \leq 1$ 
    using norm-push-exp-le-norm
    by (auto intro!: onorm-bound)
next
  show  $1 \leq onorm (push-exp d)$ 
    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)  $\implies$  bounded (abs ' X')
  by (auto simp: bounded-iff)

lemma bounded-abs-range[intro]:
  bounded (range f :: real set)  $\implies$  bounded (range ( $\lambda x. abs (f x)$ ))
  by (auto simp: bounded-iff)

```

3.6 Probability Theory

```

lemma integral-measure-pmf-bind:
  assumes  $(\bigwedge x. |(f :: 'b \Rightarrow real) x| \leq B)$ 

```

shows $(\int x. f x \partial((\text{measure-pmf } M) \gg (\lambda x. \text{measure-pmf } (N x))))$
 $= (\int x. \int y. f y \partial N x \partial M)$
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* $((w :: 'c \Rightarrow \text{real}) \text{ ' } W)$
and $W \neq \{\}$
and $\text{measure-pmf.expectation } p \ w = (\bigsqcup p \in \{p. \text{set-pmf } p \subseteq W\}.$
*measure-pmf.expectation } p \ w)
shows $\exists x \in W. \text{measure-pmf.expectation } p \ w = w \ x$
proof –
have *abs-w-le-sup*: $y \in W \implies |w \ y| \leq (\bigsqcup x \in W. |w \ x|)$ **for** *y*
using *assms bounded-abs[OF assms(2)]*
by (*auto intro!: cSUP-upper bounded-imp-bdd-above simp: image-image*)
have *False* **if** $x \in \text{set-pmf } p \ w \ x < \bigsqcup (w \text{ ' } W)$ **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* $= \lambda s. (\text{if } x = s \text{ then SOME } x'. x' \in W \wedge w \ x < w \ x' \text{ else } s)$
have $\text{measure-pmf.expectation } p \ w < \text{measure-pmf.expectation } p$
 $(\lambda x a. w \ (?s \ x a))$
proof (*intro measure-pmf.integral-less-AE[where A = {x}]*)
show *integrable* $(\text{measure-pmf } p) \ w$
using *assms abs-w-le-sup*
by (*fastforce simp: AE-measure-pmf-iff*
intro!: measure-pmf.integrable-const-bound)
show *integrable* $(\text{measure-pmf } p) (\lambda x a. w \ (?s \ x a))$
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* $(\text{measure-pmf } p) \ \{x\} \neq 0$
by (*simp add: emeasure-pmf-single-eq-zero-iff $\langle x \in p \rangle$*)
show $\{x\} \in \text{measure-pmf.events } p$
by *auto*
show *AE* $x a \in \{x\}$ *in* *p*. $w \ x a \neq w \ (?s \ x a)$ *AE* $x a$ *in* *p*. $w \ x a \leq w$
 $(?s \ x a)$
using *someI[where P = $\lambda x'. (x' \in W) \wedge (w \ x < w \ x')$*
 ex-gr
by (*fastforce intro!: AE-pmfI*)+
qed
hence $\text{measure-pmf.expectation } p \ w < \bigsqcup ((\lambda p. \text{measure-pmf.expectation}$
 $p \ w) \text{ ' } \{p. \text{set-pmf } p \subseteq W\})$
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:  $x \in \text{set-pmf } p \implies w x = \bigsqcup (w \text{ ' } W)$  for  $x$ 
  using assms
  by (fastforce intro: antisym simp: bounded-imp-bdd-above cSUP-upper)
hence  $w (\text{SOME } x. x \in \text{set-pmf } p) = \bigsqcup (w \text{ ' } W)$ 
  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  $\text{set-pmf } p \subseteq W$  integrable (measure-pmf p) w bounded ((w
:: 'c ⇒ real) ' W)
shows  $\text{measure-pmf.expectation } p w \leq \bigsqcup (w \text{ ' } W)$ 
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*: $\text{finite } X \implies X \neq \{\} \implies \exists x. \text{is-arg-max } (f :: 'c \Rightarrow \text{real}) (\lambda x. x \in X) x$

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* ($X :: 'c \text{ set}$) $X \neq \{\}$

shows $s \in X \implies (f :: 'c \Rightarrow \text{real}) s \leq f (\text{arg-max-on } (f :: 'c \Rightarrow \text{real}) X)$

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* ($X :: 'c \text{ set}$) $X \neq \{\}$

shows $(\text{arg-max-on } (f :: 'c \Rightarrow \text{real}) X) \in X$

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* ($X :: 'c \text{ set}$) $X \neq \{\}$

shows $(f :: 'c \Rightarrow \text{real}) (\text{arg-max-on } f X) = \text{Max } (f \text{ ` } X)$

using *assms*

by (*auto intro!: Max-eqI[symmetric] finite-arg-max-le arg-max-on-in*)

lemma *arg-max-SUP*: $\text{is-arg-max } (f :: 'b \Rightarrow \text{real}) (\lambda x. x \in X) m \implies f m = (\bigsqcup (f \text{ ` } X))$

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* $f X \equiv \exists x. \text{is-arg-max } f (\lambda x. x \in X) x$

lemma *has-max* $((f :: 'b \Rightarrow \text{real}) \text{ ` } X) \iff \text{has-arg-max } f X$

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* \implies *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* \implies (*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* \implies $x \in X$
 $\implies f x \leq f$ (*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* \implies 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)* ($f :: 'c \Rightarrow \text{real}$) $x = \bigsqcup$ (*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** $f :: 'c \Rightarrow 'b :: \text{linorder}$
assumes $P x \wedge y. (P y \implies f x \geq f y)$
shows *is-arg-max f P x*
using *assms* **by** (*auto simp: is-arg-max-linorder*)

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

lemma *is-arg-max-cong*:
assumes $\bigwedge x. P x \implies f x = g x$
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 $\bigwedge x. P x \implies f x = g x$
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* $\wedge x. P x \implies f x = g x$

shows *is-arg-max g P x*

using *is-arg-max-cong assms* **by force**

3.8 Contraction Mappings

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

lemma *banach'*:

fixes *C :: 'b :: complete-space* $\Rightarrow 'b$

assumes *is-contraction C*

shows $\exists! v. C v = v \wedge v. (\lambda n. (C \overset{\sim}{\sim} n) v) \longrightarrow (THE v. C v = v)$

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) ≤ l * dist v u* $0 \leq l < 1$

for *v u*

using *assms is-contraction-def* **by blast**

have $*$: $\bigwedge n v0. \text{dist } ((C \overset{\sim}{\sim} n) v0) v \leq l \wedge n * \text{dist } v0 v$

proof –

fix *n v0*

show $\text{dist } ((C \overset{\sim}{\sim} n) v0) v \leq l \wedge n * \text{dist } v0 v$

proof (*induction n*)

case (*Suc n*)

thus $\text{dist } ((C \overset{\sim}{\sim} \text{Suc } n) v0) v \leq l \wedge \text{Suc } n * \text{dist } v0 v$

using $\langle 0 \leq l \rangle$

by (*subst C(1)[symmetric]*) (*auto simp: algebra-simps intro!*:

order-trans[OF cont(1)] mult-left-mono)

qed *simp*

qed

have $(\lambda n. l \wedge n) \longrightarrow 0$

by (*simp add: LIMSEQ-realpow-zero* $\langle 0 \leq l \rangle \langle l < 1 \rangle$)

hence $\bigwedge v0. (\lambda n. l \wedge n * \text{dist } v0 v) \longrightarrow 0$

by (*simp add: tendsto-mult-left-zero*)

hence $(\lambda n. \text{dist } ((C \overset{\sim}{\sim} n) v0) v) \longrightarrow 0$ **for** *v0*

using $*$ *order-trans abs-ge-self*

by (*subst Limits.tendsto-0-le[of* $(\lambda n. l \wedge n * \text{dist } v0 v) - - 1]$)

(fastforce intro!: eventuallyI)+

hence $\bigwedge v0. (\lambda n. (C \overset{\sim}{\sim} n) v0) \longrightarrow v$

using *tendsto-dist-iff* **by blast**

thus $\bigwedge v0. (\lambda n. (C \overset{\sim}{\sim} n) v0) \longrightarrow (THE v. C v = v)$

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

```

by *metric*
 also have ... = $\text{dist } v (C v) + \text{dist } (C v) ((C \rightsquigarrow N) (C v))$
 by (*metis funpow-simps-right(2) o-def*)
 also have ... $\leq \text{dist } v (C v) + (\sum i \leq N. \text{dist } ((C \rightsquigarrow i) (C v)))$
 (($C \rightsquigarrow \text{Suc } i$) ($C v$))
 using *Suc.IH add-le-cancel-left* by *blast*
 also have ... $\leq \text{dist } v (C v) + (\sum i \leq N. \text{dist } ((C \rightsquigarrow \text{Suc } i) v))$
 (($C \rightsquigarrow (\text{Suc } (\text{Suc } i))$) v)
 by (*simp only: funpow-simps-right(2) o-def*)
 also have ... $\leq (\sum i \leq \text{Suc } N. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow (\text{Suc } i)) v))$
 by (*subst sum.atMost-Suc-shift*) *simp*
 finally show $\text{dist } v ((C \rightsquigarrow \text{Suc } N) v) \leq (\sum i \leq \text{Suc } N. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow \text{Suc } i) v))$.
 qed
 moreover have
 $(\sum i \leq N. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow \text{Suc } i) v)) \leq (\sum i. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow (\text{Suc } i)) v))$
 using *summable-C*
 by (*auto intro: sum-le-suminf*)
 ultimately have $\text{dist } v ((C \rightsquigarrow N) v) \leq (\sum i. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow (\text{Suc } i)) v))$
 by *linarith*
 thus $\text{dist } v v\text{-fix} \leq (\sum i. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow \text{Suc } i) v)) + e$
 using * by *fastforce*
 qed
 hence *le-suminf*: $\text{dist } v v\text{-fix} \leq (\sum i. \text{dist } ((C \rightsquigarrow i) v) ((C \rightsquigarrow \text{Suc } i) v))$
 using *field-le-epsilon* by *blast*
 have $\text{dist } v v\text{-fix} \leq (\sum i. c \hat{=} i * \text{dist } v (C v))$
 using *dist-contr-le-pow summable-C assms summable-powser-const*
 by (*auto intro!: order-trans[OF le-suminf] suminf-le*)
 hence $\text{dist } v v\text{-fix} \leq \text{dist } v (C v) / (1 - c)$
 using *sum-disc-lim*
 by (*metis sum-disc-lim abs-of-nonneg assms(2) assms(3)*)
 hence $(1 - c) * \text{dist } v v\text{-fix} \leq \text{dist } v (C v)$
 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
 $(f :: \text{nat} \Rightarrow 'b \Rightarrow_b \text{real}) \longrightarrow x (g :: \text{nat} \Rightarrow 'b \Rightarrow_b \text{real}) \longrightarrow x$
 eventually $(\lambda n. f n \leq h n)$ sequentially eventually $(\lambda n. h n \leq g n)$
 sequentially

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

using *assms(3) assms(4)*
proof *eventually-elim*
case (*elim n*)
hence $\text{dist } (h \ n \ a) \ (g \ n \ a) \leq \text{dist } (f \ n \ a) \ (g \ n \ a)$ **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* $(\lambda n. \text{dist } (h \ n) \ x \leq \text{dist } (h \ n) \ (g \ n) +$
 $\text{dist } (g \ n) \ x)$ *sequentially*
by (*simp add: dist-triangle*)
ultimately **have** 2: *eventually* $(\lambda n. \text{dist } (h \ n) \ x \leq \text{dist } (f \ n) \ (g \ n)$
 $+ \text{dist } (g \ n) \ x)$ *sequentially*
using *eventually-elim2* **by** *fastforce*
have $(\lambda n. \text{dist } (h \ n) \ x) \longrightarrow 0$
proof (*subst tendsto-iff, safe*)
fix *e :: real*
assume $e > 0$
hence 3: $\forall_F \ x a$ *in* *sequentially*. $\text{dist } (f \ x a) \ (g \ x a) + \text{dist } (g \ x a) \ x$
 $< e$
using 1
by (*auto simp: tendsto-iff*)
show $\forall_F \ x a$ *in* *sequentially*. $\text{dist } (\text{dist } (h \ x a) \ x) \ 0 < e$
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 $(\bigsqcup c \in X. (B :: 'a \Rightarrow \text{real}) \ c + A' \ c) \leq (\bigsqcup b \in X. B \ b) +$

$(\bigsqcup a \in X. A' a)$
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* $(B \text{ ' } X)$ *bounded* $(A' \text{ ' } X)$
and *sup-le*: $(\bigsqcup a \in X. (A' :: 'a \Rightarrow \text{real}) a) \leq (\bigsqcup b \in X. B b)$
shows $(\bigsqcup b \in X. B b) - (\bigsqcup a \in X. (A' :: 'a \Rightarrow \text{real}) a) \leq (\bigsqcup c \in X. B c - A' c)$
proof $-$
have *bounded* $((\lambda x. (B x - A' x)) \text{ ' } X)$
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 = $\bigsqcup (B \text{ ' } X) - e$, where ?X = $B \text{ ' } X$]*) *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 $\langle z \in X \rangle$ *bdd bounded-has-Sup(1)* **by** *fastforce*
thus $(\bigsqcup b \in X. B b) - (\bigsqcup a \in X. A' a) - e \leq (\bigsqcup c \in X. B c - A' c)$
if *e*: $e > 0$ **for** *e*
using \langle *bounded* $((\lambda x. B x - A' x) \text{ ' } X)\rangle$ *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 $A' :: 'a \Rightarrow \text{real}$
assumes $X \neq \{\}$ *bounded* $(B \text{ ' } X)$ *bounded* $(A' \text{ ' } X)$ $(\bigsqcup a \in X. A' a) \leq (\bigsqcup b \in X. B b)$
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* $(f \text{ ' } X) \Longrightarrow (\bigsqcup x \in X. (l :: \text{real}) * f x) = (l * (\bigsqcup x \in X. f x))$
proof $-$
assume $X \neq \{\}$ *bounded* $(f \text{ ' } X)$ $0 \leq l$
have $(\bigsqcup x \in X. \text{ereal } l * \text{ereal } (f x)) = (l * (\bigsqcup x \in X. \text{ereal } (f x)))$

by (simp add: Sup-ereal-mult-left' <math>0 \leq b \wedge X \neq \{\}>)

obtain b **where** $\forall a \in X. |f a| \leq b$

using $\langle \text{bounded } (f \text{ ' } X) \rangle \text{ bounded-real}$ **by** *auto*

have $\forall a \in X. |\text{ereal } (f a)| \leq b$

by (simp add: $\langle \forall a \in X. |f a| \leq b \rangle$)

hence *sup-leb*: $(\bigsqcup a \in X. |\text{ereal } (f a)|) \leq b$

by (simp add: SUP-least)

have $(\bigsqcup a \in X. \text{ereal } (f a)) \leq (\bigsqcup a \in X. |\text{ereal } (f a)|)$

by (auto intro: Complete-Lattices.SUP-mono')

moreover have $-(\bigsqcup a \in X. \text{ereal } (f a)) \leq (\bigsqcup a \in X. |\text{ereal } (f a)|)$

using <math>\langle X \neq \{\}>

by (auto intro!: Inf-less-eq cSUP-upper2 simp add: ereal-INF-uminus-eq[symmetric])

ultimately have $|(\bigsqcup a \in X. \text{ereal } (f a))| \leq (\bigsqcup a \in X. |\text{ereal } (f a)|)$

by (auto intro: ereal-abs-leI)

hence $|\bigsqcup a \in X. \text{ereal } (f a)| \leq b$

using *sup-leb* **by** *auto*

hence $|\bigsqcup a \in X. \text{ereal } (f a)| \neq \infty$

by *auto*

hence $(\bigsqcup x \in X. \text{ereal } (f x)) = \text{ereal } (\bigsqcup x \in X. (f x))$

using *ereal-SUP* **by** *metis*

hence $(\bigsqcup x \in X. \text{ereal } (l * f x)) = \text{ereal } (l * (\bigsqcup x \in X. f x))$

using $\langle \bigsqcup x \in X. \text{ereal } l * \text{ereal } (f x) = \text{ereal } l * (\bigsqcup x \in X. \text{ereal } (f x)) \rangle$ **by** *auto*

hence $\text{ereal } (\bigsqcup x \in X. l * f x) = \text{ereal } (l * (\bigsqcup x \in X. f x))$

by (simp add: ereal-SUP)

thus *?thesis*

by *auto*

qed

lemma *abs-cSUP-le[intro]*:

$X \neq \{\} \implies \text{bounded } (F \text{ ' } X) \implies |\bigsqcup x \in X. (F x) :: \text{real}| \leq (\bigsqcup x \in X. |F x|)$

by (auto intro!: cSup-abs-le cSUP-upper2 bounded-imp-bdd-above simp: image-image[symmetric])

4 Least argmax

definition *least-arg-max* $f P = (\text{LEAST } x. \text{is-arg-max } f P x)$

lemma *least-arg-max-prop*: $\exists x :: 'a :: \text{wellorder}. P x \implies \text{finite } \{x. P x\} \implies P (\text{least-arg-max } (f :: - \Rightarrow \text{real}) P)$

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*: $\text{is-arg-max } (f :: - \Rightarrow - :: \text{linorder}) P x \implies \text{is-arg-max } f P y \implies f x = f y$

by (auto simp: *is-arg-max-def not-less dual-order.eq-iff*)

```

lemma least-arg-max-apply:
  assumes is-arg-max ( $f :: - \Rightarrow - :: \text{linorder}$ )  $P (x:::\text{wellorder})$ 
  shows  $f (\text{least-arg-max } f P) = f x$ 
proof –
  have is-arg-max  $f P (\text{least-arg-max } f P)$ 
    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:  $\text{finite } \{x . P x\} \Longrightarrow \text{is-arg-max } (f :: - \Rightarrow - :: \text{linorder}) P x \Longrightarrow f x = \text{Max } (f ' \{x. P x\})$ 
  by (auto simp: is-arg-max-def intro!: Max-eqI[symmetric])

lemma apply-arg-max-eq-max':  $\text{finite } X \Longrightarrow \text{is-arg-max } (f :: - \Rightarrow - :: \text{linorder}) (\lambda x. x \in X) x \Longrightarrow (\text{MAX } x \in X. f x) = f x$ 
  by (auto simp: is-arg-max-linorder intro!: Max-eqI)

lemma least-arg-max-is-arg-max:  $P \neq \{\}$   $\Longrightarrow \text{finite } P \Longrightarrow \text{is-arg-max } f (\lambda x:::\text{wellorder}. x \in P) (\text{least-arg-max } (f :: - \Rightarrow \text{real}) (\lambda x. x \in P))$ 
  unfolding least-arg-max-def
  apply (rule LeastI-ex)
  using finite-is-arg-max
  by auto

lemma is-arg-max-const: is-arg-max ( $f :: - \Rightarrow - :: \text{linorder}$ ) ( $\lambda y. y = c$ )  $x \longleftrightarrow x = c$ 
  unfolding is-arg-max-def
  by auto

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

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

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

```

proof (rule measure-eqI-PiM-infinite[OF * - A])
  fix J :: nat set and F'
  assume J: finite J  $\wedge$  i. i  $\in$  J  $\implies$  F' i  $\in$  sets (M i)

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

  show emeasure P (prod-emb UNIV M J (PiE J F')) = emeasure Q
    (prod-emb UNIV M J (PiE J F'))
  unfolding emb-eq by (rule eq) fact
qed

```

```

lemma distr-cong-simp:
  M = K  $\implies$  sets N = sets L  $\implies$  ( $\wedge$ x. x  $\in$  space M =simp=> f x =
  g x)  $\implies$  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  $\Rightarrow$  'a set
  and K :: 's  $\times$  'a  $\Rightarrow$  's measure

  assumes A-s:  $\wedge$ s. A s  $\in$  sets Ma

  assumes A-ne:  $\wedge$ s. A s  $\neq$  {}

  assumes ex-pol:  $\exists$   $\delta \in$  Ms  $\rightarrow_M$  Ma.  $\forall$  s.  $\delta$  s  $\in$  A s

  assumes K[measurable]: K  $\in$  Ms  $\otimes_M$  Ma  $\rightarrow_M$  prob-algebra Ms
begin

  lemma space-prodI[intro]: x  $\in$  space A'  $\implies$  y  $\in$  space B  $\implies$  (x,y)  $\in$ 
  space (A'  $\otimes_M$  B)
  by (auto simp add: space-pair-measure)

```

abbreviation $M \equiv Ms \otimes_M Ma$

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

lemma $\text{space-ma}[\text{intro}]$: $s \in \text{space } Ms \implies a \in \text{space } Ma \implies (s,a) \in \text{space } M$

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

lemma $\text{space-x0}[\text{simp}]$: $x0 \in \text{space } (\text{prob-algebra } Ms) \implies \text{space } x0 = \text{space } Ms$

by (*metis (mono-tags, lifting) space-prob-algebra mem-Collect-eq sets-eq-imp-space-eq*)

lemma $A\text{-subs-Ma}$: $A\ s \subseteq \text{space } Ma$

by (*simp add: A-s sets.sets-into-space*)

lemma space-Ma-A-subset : $s \in \text{space } Ms \implies \text{space } (Ma-A\ s) \subseteq A\ s$

by (*simp add: space-restrict-space*)

lemma $K\text{-restrict} [\text{measurable}]$: $K \in (Ms \otimes_M Ma-A\ s) \rightarrow_M \text{prob-algebra } Ms$

by *measurable (metis measurable-id measurable-pair-iff measurable-restrict-space2-iff)*

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

by *measurable*

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

by *measurable*

lemma $\text{space-K}[\text{simp}]$: $sa \in \text{space } M \implies \text{space } (K\ sa) = \text{space } Ms$

using K **unfolding** *prob-algebra-def measurable-restrict-space2-iff*

by (*auto dest: subprob-measurableD*)

lemma $\text{space-K2}[\text{simp}]$: $s \in \text{space } Ms \implies a \in \text{space } Ma \implies \text{space } (K\ (s, a)) = \text{space } Ms$

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

lemma space-K-E : $s' \in \text{space } (K\ (s,a)) \implies s \in \text{space } Ms \implies a \in \text{space } Ma \implies s' \in \text{space } Ms$

by *auto*

lemma sets-K : $sa \in \text{space } M \implies \text{sets } (K\ sa) = \text{sets } Ms$

using K **unfolding** *prob-algebra-def unfolding measurable-restrict-space2-iff*

by (*auto dest: subprob-measurableD*)

lemma $\text{sets-K}'[\text{measurable-cong}]$: $s \in \text{space } Ms \implies a \in \text{space } Ma \implies \text{sets } (K\ (s,a)) = \text{sets } Ms$

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

lemma *prob-space-K[intro]*: $sa \in \text{space } M \implies \text{prob-space } (K \text{ } sa)$
using *measurable-space[OF K]* **by** (*simp add: space-prob-algebra*)

lemma *prob-space-K2[intro]*: $s \in \text{space } Ms \implies a \in \text{space } Ma \implies$
 $\text{prob-space } (K \text{ } (s,a))$
using *prob-space-K* **by** (*simp add: space-pair-measure*)

lemma *K-in-space [intro]*: $m \in \text{space } M \implies K \text{ } m \in \text{space } (\text{prob-algebra } Ms)$
by (*meson K measurable-space*)

5.2 Policies

type-synonym (*'c, 'd*) *pol* = $\text{nat} \Rightarrow ((\text{nat} \Rightarrow 'c \times 'd) \times 'c) \Rightarrow 'd$
measure

abbreviation *H i* $\equiv \text{Pi}_M \{0..<i\} (\lambda-. M)$

abbreviation *Hs i* $\equiv H \text{ } i \otimes_M Ms$

lemma *space-H1*: $j < (i :: \text{nat}) \implies \omega \in \text{space } (H \text{ } i) \implies \omega \text{ } j \in \text{space } M$
using *PiE-def*
by (*auto simp: space-PiM*)

lemma *space-case-nat[intro]*:
assumes $\omega \in \text{space } (H \text{ } i) \text{ } s \in \text{space } Ms$
shows *case-nat* $s \text{ } (\text{fst} \circ \omega) \text{ } i \in \text{space } Ms$
using *assms*
by (*cases i*) (*auto intro!: space-H1 measurable-space[OF measurable-fst]*)

lemma *undefined-in-H0*: $(\lambda-. \text{undefined}) \in \text{space } (H \text{ } (0 :: \text{nat}))$
by *auto*

lemma *sets-K-Suc[measurable-cong]*: $h \in \text{space } (H \text{ } (\text{Suc } n)) \implies \text{sets } (K \text{ } (h \text{ } n)) = \text{sets } Ms$
using *sets-K space-H1* **by** *blast*

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

definition *is-dec0* $d \equiv d \in Ms \rightarrow_M \text{prob-algebra } Ma$

definition *is-dec* $(t :: \text{nat}) \text{ } d \equiv (d \in Hs \text{ } t \rightarrow_M \text{prob-algebra } Ma)$

lemma *is-dec0* $d \implies \text{is-dec } t \text{ } (\lambda(-,s). d \text{ } s)$

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* \Rightarrow *bool* **where**
is-policy *p* $\equiv \forall i. \text{is-dec } i (p \ i)$

abbreviation *p0* :: ('s, 'a) *pol* \Rightarrow 's \Rightarrow 'a *measure* **where**
p0 *p* *s* $\equiv p \ (0 :: \text{nat}) \ (\lambda-. \text{undefined}, s)$

context

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

lemma *is-policyD[measurable]*: $p \ i \in \text{Hs } i \rightarrow_M \text{prob-algebra } Ma$
using *p* **unfolding** *is-policy-def is-dec-def* **by** *auto*

lemma *space-policy[simp]*: $hs \in \text{space } (Hs \ i) \Longrightarrow \text{space } (p \ i \ hs) = \text{space } Ma$
using *K is-policyD* **unfolding** *prob-algebra-def measurable-restrict-space2-iff*
by (*auto dest: subprob-measurableD*)

lemma *space-policy'[simp]*: $h \in \text{space } (H \ i) \Longrightarrow s \in \text{space } Ms \Longrightarrow \text{space } (p \ i \ (h,s)) = \text{space } Ma$
using *space-policy*
by (*simp add: space-pair-measure*)

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

lemma *sets-policy[simp]*: $hs \in \text{space } (Hs \ i) \Longrightarrow \text{sets } (p \ i \ hs) = \text{sets } Ma$
using *p K is-policyD*
unfolding *prob-algebra-def measurable-restrict-space2-iff*
by (*auto dest: subprob-measurableD*)

lemma *sets-policy'[measurable-cong, simp]*:
 $h \in \text{space } (H \ i) \Longrightarrow s \in \text{space } Ms \Longrightarrow \text{sets } (p \ i \ (h,s)) = \text{sets } Ma$
using *sets-policy*
by (*auto simp: space-pair-measure*)

lemma *sets-policy''[measurable-cong, simp]*:
 $h \in \text{space } ((Pi_M \ \{\} \ (\lambda-. \ M))) \Longrightarrow s \in \text{space } Ms \Longrightarrow \text{sets } (p \ 0 \ (h,s)) = \text{sets } Ma$
using *sets-policy*
by (*auto simp: space-pair-measure*)

lemma *policy-prob-space*: $hs \in \text{space } (Hs \ i) \implies \text{prob-space } (p \ i \ hs)$

proof –

assume h : $hs \in \text{space } (Hs \ i)$

hence $p \ i \ hs \in \text{space } (\text{prob-algebra } Ma)$

using p **by** (*auto intro: measurable-space*)

thus $\text{prob-space } (p \ i \ hs)$

unfolding *prob-algebra-def* **by** (*simp add: space-restrict-space*)

qed

lemma *policy-prob-space'*: $h \in \text{space } (H \ i) \implies s \in \text{space } Ms \implies \text{prob-space } (p \ i \ (h,s))$

using *policy-prob-space* **by** (*simp add: space-pair-measure*)

lemma *prob-space-p0*: $x \in \text{space } Ms \implies \text{prob-space } (p0 \ p \ x)$

by (*simp add: policy-prob-space'*)

lemma *p0-sets[measurable-cong]*: $x \in \text{space } Ms \implies \text{sets } (p \ 0 \ (\lambda \cdot \text{undefined}, x)) = \text{sets } Ma$

using *subprob-measurableD(2) measurable-prob-algebraD* **by** *simp*

lemma *space-p0[simp]*: $s \in \text{space } Ms \implies \text{space } (p0 \ p \ s) = \text{space } Ma$

by *auto*

lemma *return-policy-prob-algebra [measurable]*:

$h \in \text{space } (H \ n) \implies x \in \text{space } Ms \implies (\lambda a. \text{return } M \ (x, a)) \in p \ n$
 $(h, x) \rightarrow_M \text{prob-algebra } M$

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* $p \ sa = (\lambda i \ (h, s). p \ (Suc \ i) \ (\lambda i'. \text{case-nat } sa \ h \ i', s))$

lemma *p-as-Suc-policy*: $p \ (Suc \ i) \ (h, s) = \text{Suc-policy } p \ ((h \ 0)) \ i \ (\lambda i. h \ (Suc \ i), s)$

proof –

have $*$: $\text{case-nat } (f \ 0) \ (\lambda i. f \ (Suc \ i)) = f$ **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 : $sa \in \text{space } M$ **and** p : *is-policy* p

shows *is-policy* (*Suc-policy* *p sa*)
proof –
have *: ($\lambda x. \text{case-nat } sa \text{ (fst } x) \in Pi_M \{0..<i\} (\lambda-. M) \otimes_M Ms$
 $\rightarrow_M Pi_M \{0..<Suc\ i\} (\lambda-. M)$ **for** *i*
using *s space-H1*
by (*intro measurable-PiM-single*) (*fastforce simp: space-PiM*
space-pair-measure split: nat.splits)
have $\bigwedge i. p \ i \in Pi_M \{0..<i\} (\lambda-. M) \otimes_M Ms \rightarrow_M \text{prob-algebra } Ma$
using *is-policyD p by blast*
hence $\bigwedge i. \text{Suc-policy } p \ sa \ i \in Pi_M \{0..<i\} (\lambda-. M) \otimes_M Ms \rightarrow_M$
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 ($\lambda x. \text{Suc-policy } p \text{ (fst (fst } x)) \ n \text{ (snd (fst } x), \text{snd } x) \in$
 $(M \otimes_M Pi_M \{0..<n\} (\lambda-. M)) \otimes_M Ms \rightarrow_M \text{prob-algebra } Ma$
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 $K' :: ('s, 'a) \text{pol} \Rightarrow 's \text{measure} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow ('s \times 'a))$
 $\Rightarrow ('s \times 'a) \text{measure}$

where

$$K' \ p \ s0 \ n \ \omega = \text{do } \{$$

$$s \leftarrow \text{case-nat } s0 \ (K \circ \omega) \ n;$$

$$a \leftarrow p \ n \ (\omega, s);$$

$$\text{return } M \ (s, a)$$

$$\}$$

lemma *prob-space-K'*:
assumes *p: is-policy p* **and** *x: x0 ∈ space (prob-algebra Ms)* **and** *h:*
 $h \in \text{space } (H \ n)$
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 ∈ space (prob-algebra Ms)
  shows K' p x i ∈ H i →M prob-algebra M
proof -
  fix i
  show K' p x i ∈ PiM {0..<i} (λ-. M) →M prob-algebra M
  unfolding K'-def
  proof (intro measurable-bind-prob-space2[where N = Ms])
    show (λa. case i of 0 ⇒ x | Suc x ⇒ (K ∘ a) x) ∈ PiM {0..<i}
    (λ-. M) →M prob-algebra Ms
    using x by measurable auto
  next
  show (λ(ω, y). p i (ω, y) ≫ (λa. return M (y, a))) ∈
    PiM {0..<i} (λ-. M) ⊗M Ms →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 $K0 p s0 = K' p s0 0$ (λ -. undefined)

```

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

```

```

lemma K0-prob[measurable]: is-policy p ⇒ K0 p ∈ prob-algebra Ms
→M prob-algebra M
unfolding K0-def'
by measurable

```

```

lemma prob-space-K0: is-policy p ⇒ x0 ∈ space (prob-algebra Ms)
⇒ prob-space (K0 p x0)
by (simp add: K0-def prob-space-K')

```

```

lemma space-K0[simp]: is-policy p ⇒ s ∈ space (prob-algebra Ms)
⇒ 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* $p \ s \in \text{space} \ (\text{prob-algebra} \ Ms)$
shows *sets* $(K0 \ p \ s) = \text{sets} \ M$
using *assms* **by** $(\text{meson} \ K0\text{-prob} \ \text{measurable-prob-algebraD} \ \text{sub-prob-measurableD}(2))$

lemma *K0-return-eq-p0*:

assumes *is-policy* $p \ s \in \text{space} \ Ms$
shows $K0 \ p \ (\text{return} \ Ms \ s) = p0 \ p \ s \gg\gg (\lambda a. \text{return} \ M \ (s,a))$
unfolding *K0-def'*
using *assms*
by $(\text{subst} \ \text{bind-return}[\text{where} \ N = M]) \ (\text{auto} \ \text{intro}!: \ \text{measurable-prob-algebraD})$

lemma *M-ne-policy[intro]*: *is-policy* $p \implies s \in \text{space} \ (\text{prob-algebra} \ Ms)$
 $\implies \text{space} \ M \neq \{\}$
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* $p \implies x \in \text{space} \ (\text{prob-algebra} \ Ms) \implies \text{Ionescu-Tulcea}$
 $(K' \ p \ x) \ (\lambda-. \ M)$
unfolding *Ionescu-Tulcea-def* **using** *measurable-K'* *prob-space-K'*
by $(\text{fast} \ \text{dest}: \ \text{measurable-prob-algebraD})$

definition *lim-sequence* :: $('s, 'a) \ \text{pol} \ \Rightarrow 's \ \text{measure} \ \Rightarrow (\text{nat} \ \Rightarrow ('s \times 'a)) \ \text{measure}$

where

lim-sequence $p \ x = \text{projective-family.lim} \ UNIV \ (\text{Ionescu-Tulcea.CI} \ (K' \ p \ x) \ (\lambda-. \ M)) \ (\lambda-. \ M)$

lemma

assumes $x: x \in \text{space} \ (\text{prob-algebra} \ Ms)$ **and** $p: \text{is-policy} \ p$
shows *space-lim-sequence*: $\text{space} \ (\text{lim-sequence} \ p \ x) = \text{space} \ (\prod_{M \ i \in UNIV. \ M})$
and *sets-lim-sequence[measurable-cong]*: $\text{sets} \ (\text{lim-sequence} \ p \ x) = \text{sets} \ (\prod_{M \ i \in UNIV. \ M})$
and *emeasure-lim-sequence-emb*: $\bigwedge J \ X. \ \text{finite} \ J \ \implies X \in \text{sets} \ (\prod_{M \ j \in J. \ M}) \ \implies$
 $\text{emeasure} \ (\text{lim-sequence} \ p \ x) \ (\text{prod-emb} \ UNIV \ (\lambda-. \ M) \ J \ X) =$
 $\text{emeasure} \ (\text{Ionescu-Tulcea.CI} \ (K' \ p \ x) \ (\lambda-. \ M) \ J) \ X$
and *emeasure-lim-sequence-emb-I0o*: $\bigwedge n \ X. \ X \in \text{sets} \ (\prod_{M \ i \in \{0..<n\}. \ M}) \ \implies$
 $\text{emeasure} \ (\text{lim-sequence} \ p \ x) \ (\text{prod-emb} \ UNIV \ (\lambda-. \ M) \ \{0..<n\} \ X) =$
 $\text{emeasure} \ (\text{Ionescu-Tulcea.C} \ (K' \ p \ x) \ (\lambda-. \ M) \ 0 \ n \ (\lambda x. \ \text{undefined})) \ X$

proof –

interpret *Ionescu-Tulcea* $K' \ p \ x \ \lambda-. \ M$

```

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

  { fix J :: nat set and X assume finite J X ∈ sets (ΠM 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 (ΠM 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) {} (ΠE 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 Measurability of the Sequence Space

lemma *lim-sequence[measurable]*:

```

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

```

```

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

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

  have C: ( $\lambda x. \text{Ionescu-Tulcea.C } (K' \ p \ x) \ (\lambda \cdot. \ M) \ 0 \ n \ (\lambda x. \ \text{undefined}))$ 
   $\in \text{prob-algebra } Ms \rightarrow_M \text{subprob-algebra } (Pi_M \ \{0..<n\} \ (\lambda \cdot. \ M))$ 
  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  $(\lambda w. \text{Ionescu-Tulcea.eP } (K' \ p \ (\text{fst } w)) \ (\lambda \cdot. \ M) \ n \ (\text{snd } w))$ 
     $\in \text{prob-algebra } Ms \otimes_M Pi_M \ \{0..<n\} \ (\lambda \cdot. \ M) \rightarrow_M \text{subprob-algebra}$ 
     $(Pi_M \ \{0..<\text{Suc } n\} \ (\lambda \cdot. \ M))$ 
    proof (subst measurable-cong)
      fix w assume  $w \in \text{space } (\text{prob-algebra } Ms \otimes_M Pi_M \ \{0..<n\} \ (\lambda \cdot. \ M))$ 
      then show  $\text{Ionescu-Tulcea.eP } (K' \ p \ (\text{fst } w)) \ (\lambda \cdot. \ M) \ n \ (\text{snd } w)$ 
      =

```

$distr (K' p (fst w) n (snd w)) (\Pi_M i \in \{0..<Suc n\}. M) (fun-upd (snd w) n)$
by (*auto simp: p space-pair-measure Ionescu-Tulcea.eP-def[OF IT-K'] split: prod.split*)
next
show ($\lambda w. distr (K' p (fst w) n (snd w)) (\Pi_M i \in \{0..<Suc n\}. M) (fun-upd (snd w) n)$)
 $\in prob-algebra Ms \otimes_M Pi_M \{0..<n\} (\lambda-. M) \rightarrow_M subprob-algebra (Pi_M \{0..<Suc n\} (\lambda-. M))$
proof (*rule measurable-distr2[where M = M]*)
show ($\lambda(x, y). (snd x)(n := y) \in (prob-algebra Ms \otimes_M Pi_M \{0..<n\} (\lambda-. M)) \otimes_M M \rightarrow_M Pi_M \{0..<Suc n\} (\lambda i. M)$)
proof (*rule measurable-PiM-single'*)
fix i **assume** $i \in \{0..<Suc n\}$
then show ($\lambda\omega. (case \omega of (x, y) \Rightarrow (snd x)(n := y)) i \in (prob-algebra Ms \otimes_M Pi_M \{0..<n\} (\lambda-. M)) \otimes_M M \rightarrow_M M$)
unfolding *split-beta'* **by** (*cases i = n auto*)
next
show ($\lambda\omega. case \omega of (x, y) \Rightarrow (snd x)(n := y) \in space ((prob-algebra Ms \otimes_M Pi_M \{0..<n\} (\lambda-. M)) \otimes_M M) \rightarrow \{0..<Suc n\} \rightarrow_E space M$)
by (*auto simp: space-pair-measure space-PiM less-Suc-eq PiE-iff*)
qed
next
show ($\lambda x. K' p (fst x) n (snd x) \in prob-algebra Ms \otimes_M Pi_M \{0..<n\} (\lambda-. M) \rightarrow_M subprob-algebra M$)
unfolding *K'-def comp-def*
using p
by (*auto intro!: measurable-prob-algebraD*)
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 $*$: ($\lambda x. Ionescu-Tulcea.CI (K' p x) (\lambda-. M) J \in prob-algebra Ms \rightarrow_M subprob-algebra (Pi_M J (\lambda-. M))$)
using *measurable-distr[OF measurable-restrict-subset[OF J-le-n], of (\lambda-. 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 ($\lambda a. emeasure (lim-sequence p a) X \in borel-measurable (prob-algebra Ms) \longleftrightarrow (\lambda a. emeasure (Ionescu-Tulcea.CI (K' p a) (\lambda-. M) J) (Pi_E J F) \in$

borel-measurable (prob-algebra Ms)
unfolding X **using** J Pi - F **by** (intro p measurable-cong emea-
 sure-lim-sequence-emb) auto
also have ...
using * measurable-emeasure-subprob-algebra Pi - F (2) **by** auto
finally show $(\lambda a. \text{emeasure } (\text{lim-sequence } p \ a) \ X) \in \text{borel-measurable}$
 (prob-algebra Ms) .
qed

lemma *lim-sequence-aux*[measurable]:

assumes p : is-policy p
assumes f : $\bigwedge x. x \in \text{space } M \implies \text{is-policy } (f \ x)$
assumes f' : $\bigwedge n. (\lambda x. f \ (fst \ (fst \ x)) \ n \ (snd \ (fst \ x), \ snd \ x)) \in$
 $(M \otimes_M Pi_M \ \{0..<n\} \ (\lambda-. \ M)) \otimes_M Ms \rightarrow_M \text{prob-algebra } Ma$
assumes gm : $g \in M \rightarrow_M \text{prob-algebra } Ms$
shows $(\lambda x. \text{lim-sequence } (f \ x) \ (g \ x)) \in M \rightarrow_M \text{prob-algebra } (Pi_M$
 $UNIV \ (\lambda-. \ M))$
proof (intro measurable-prob-algebra-generated[OF sets- Pi M Int-stable-prod-algebra
 prod-algebra-sets-into-space])
fix a **assume** a [simp]: $a \in \text{space } M$
have g : $\bigwedge x. x \in \text{space } M \implies g \ x \in \text{space } (\text{prob-algebra } Ms)$
by (meson gm measurable-space)
interpret Ionescu-Tulcea K' $(f \ a) \ (g \ a) \ \lambda-. \ M$
using IT- K' p
using f [OF $\langle a \in \text{space } M \rangle$] g **by** measurable
have p' : is-policy $(f \ a)$
using $\langle a \in \text{space } M \rangle$ $p \ f$ **by** auto
have sp :
 $\text{space } (\text{lim-sequence } (f \ a) \ (g \ a)) = \text{prod-emb } UNIV \ (\lambda-. \ M) \ \{\}$ $(\Pi_E$
 $j \in \{\}. \ \text{space } M)$
 $\text{space } (CI \ \{\}) = \{\} \rightarrow_E \text{space } M$
using $g \ a \ p'$ **by** (auto simp: space-lim-sequence p' space- Pi M
 prod-emb-def PF.space- P)
have $\text{emeasure } (\text{lim-sequence } (f \ a) \ (g \ a)) \ (\text{space } (\text{lim-sequence } (f \ a)$
 $(g \ a))) = 1$
unfolding sp
using $g \ a \ p'$ sets.top[of $(Pi_M \ \{\}) \ (\lambda-. \ M)$], unfolded space- Pi M-empty]

 $PF.\text{prob-space-}P$ [THEN prob-space.emeasure-space-1, of $\{\}$]
by (subst emeaure-lim-sequence-emb) (auto simp: emeaure-lim-sequence-emb
 sp)
thus prob-space $(\text{lim-sequence } (f \ a) \ (g \ a))$
by (auto intro: prob-spaceI)
show sets $(\text{lim-sequence } (f \ a) \ (g \ a)) = \text{sets } (Pi_M \ UNIV \ (\lambda i. \ M))$
by (simp add: lim-sequence-def)
next
fix X :: $(nat \implies 's \times 'a)$ set **assume** $X \in \text{prod-algebra } UNIV \ (\lambda i. \ M)$
then obtain J :: nat set **and** F **where** J : $J \neq \{\}$ finite $J \ F \in J$

\rightarrow sets M
and $X: X = \text{prod-emb UNIV } (\lambda\cdot. M) J (Pi_E J F)$
unfolding *prod-algebra-def* **by** *auto*
then have $Pi\text{-}F: \text{finite } J Pi_E J F \in \text{sets } (Pi_M J (\lambda\cdot. M))$
by (*auto intro: sets-PiM-I-finite*)

define n **where** $n = (\text{LEAST } n. \forall i \geq n. i \notin J)$
have $J\text{-}le\text{-}n: J \subseteq \{0..<n\}$
unfolding *n-def*
by (*rule LeastI2[of - Suc (Max J)]*) (*auto simp: ‹finite J› Suc-le-eq not-le[symmetric]*)

have $g: \bigwedge x. x \in \text{space } M \implies g x \in \text{space } (\text{prob-algebra } Ms)$
by (*meson gm measurable-space*)

have $C: (\lambda x. \text{Ionescu-Tulcea.C } (K' (f x) (g x)) (\lambda\cdot. M) 0 n (\lambda x. \text{undefined})) \in$
 $M \rightarrow_M \text{subprob-algebra } (Pi_M \{0..<n\}) (\lambda\cdot. M)$
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 ($\text{Suc } n$)
then show *?case*
proof (*subst measurable-cong[OF Ionescu-Tulcea.C.simps(2)]*) (*OF IT-K'*)
show $(\lambda w. \text{Ionescu-Tulcea.C } (K' (f w) (g w)) (\lambda\cdot. M) 0 n (\lambda x. \text{undefined})) \ggg \text{Ionescu-Tulcea.eP } (K' (f w) (g w)) (\lambda\cdot. M) (0 + n)$
 $\in M \rightarrow_M \text{subprob-algebra } (Pi_M \{0..<\text{Suc } n\}) (\lambda\cdot. M)$
if $h: (\lambda x. \text{Ionescu-Tulcea.C } (K' (f x) (g x)) (\lambda\cdot. M) 0 n (\lambda x. \text{undefined})) \in M \rightarrow_M \text{subprob-algebra } (Pi_M \{0..<n\}) (\lambda\cdot. M)$
proof (*rule measurable-bind'[OF h]*)
show $(\lambda(x, y). \text{Ionescu-Tulcea.eP } (K' (f x) (g x)) (\lambda\cdot. M) (0 + n) y) \in M \otimes_M Pi_M \{0..<n\} (\lambda\cdot. M) \rightarrow_M \text{subprob-algebra } (Pi_M \{0..<\text{Suc } n\}) (\lambda\cdot. M)$
proof (*subst measurable-cong*)
fix $n :: \text{nat}$ **and** w **assume** $w \in \text{space } (M \otimes_M Pi_M \{0..<n\}) (\lambda\cdot. M)$
then show (*case w of* $(x, a) \Rightarrow \text{Ionescu-Tulcea.eP } (K' (f x) (g x)) (\lambda\cdot. M) (0 + n) a =$
(case w of $(x, a) \Rightarrow \text{distr } (K' (f x) (g x) n a) (\prod_M i \in \{0..<\text{Suc } n\}. M) (\text{fun-upd } a n)$
by (*auto simp: IT-K' Ionescu-Tulcea.eP-def f g space-ma p space-pair-measure*)
 $\text{Ionescu-Tulcea.eP-def}$ [*OF IT-K'*])
next

```

fix n
  have ( $\lambda w. \text{distr } (K' (f (fst w)) (g (fst w))) n (snd w)$ ) ( $Pi_M$ 
    {0..<Suc n} ( $\lambda i. M$ )) ( $\text{fun-upd } (snd w) n$ )
     $\in M \otimes_M Pi_M \{0..<n\} (\lambda-. M) \rightarrow_M \text{subprob-algebra } (Pi_M$ 
    {0..<Suc n} ( $\lambda-. M$ ))
    proof (intro measurable-distr2[where  $M=M$ ] measurable-
    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 ( $\lambda w. \text{case } w \text{ of } (x, a) \Rightarrow \text{distr } ((K' (f x)) (g x)) n a$ )
    ( $Pi_M \{0..<Suc n\} (\lambda i. M)$ ) ( $\text{fun-upd } a n$ )  $\in M \otimes_M Pi_M \{0..<n\}$ 
    ( $\lambda-. M$ )  $\rightarrow_M \text{subprob-algebra } (Pi_M \{0..<Suc n\} (\lambda-. M))$ 
    by measurable
    qed
    qed
    qed (auto simp: f g)
  qed

have p':  $\bigwedge a. a \in \text{space } M \implies \text{is-policy } (f a)$ 
  using f by auto
have ( $\lambda a. \text{emeasure } (\text{lim-sequence } (f a) (g a)) X$ )  $\in \text{borel-measurable}$ 
  M  $\longleftrightarrow$ 
  ( $\lambda a. \text{emeasure } (\text{Ionescu-Tulcea.CI } (K' (f a)) (g a)) (\lambda-. M) J$ ) ( $Pi_E$ 
  J F)  $\in \text{borel-measurable } M$ 

```

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 = (λw. distr (Ionescu-Tulcea.C (K' (f w) (g w)) (λ-. M) 0 n (λx. undefined)) (Pi_M J (λ-. M)) (λf. 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 ($\lambda a. \text{emeasure } (\text{lim-sequence } (f a) (g a)) X \in \text{borel-measurable } M$)
using *calculation by blast*
qed

lemma *lim-sequence-Suc-return[measurable]*:
assumes p : *is-policy p*
assumes s : $s \in \text{space } Ms$
shows ($\lambda x. \text{lim-sequence } (\text{Suc-policy } p (s, \text{snd } x)) (\text{return } Ms (fst x)) \in$
 $M \rightarrow_M \text{prob-algebra } (Pi_M \text{ UNIV } (\lambda-. M))$)
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 $(\lambda x. \text{lim-sequence } (\text{Suc-policy } p \ x) \ (K \ x)) \in M \rightarrow_M \text{prob-algebra}$
($Pi_M \ UNIV \ (\lambda-. \ M)$)

using *assms*

by (*fastforce intro!*: *lim-sequence-aux*)

5.8 Iteration Rule

lemma *step-C*:

assumes $x: x \in \text{space } (\text{prob-algebra } Ms)$ and $p: \text{is-policy } p$

shows $\text{Ionescu-Tulcea.C } (K' \ p \ x) \ (\lambda-. \ M) \ 0 \ 1 \ (\lambda-. \ \text{undefined}) \gg=$

$\text{Ionescu-Tulcea.C } (K' \ p \ x) \ (\lambda-. \ M) \ 1 \ n =$

$K0 \ p \ x \gg= (\lambda a. \text{Ionescu-Tulcea.C } (K' \ p \ x) \ (\lambda-. \ M) \ 1 \ n \ (\text{case-nat}$
 $a \ (\lambda-. \ \text{undefined})))$

proof –

interpret *Ionescu-Tulcea* $K' \ p \ x \ \lambda-. \ M$

using *IT-K'[OF p x]* .

have [*simp*]: $\text{space } (K0 \ p \ x) \neq \{\}$

using *space-K0[OF p x]* x by *auto*

have [*simp*]: $((\lambda-. \ \text{undefined})(0 := x::('s \times 'a))) = \text{case-nat } x \ (\lambda-. \ \text{undefined})$ for x

by (*auto simp: fun-eq-iff split: nat.split*)

have $C \ 0 \ 1 \ (\lambda-. \ \text{undefined}) \gg= C \ 1 \ n = eP \ 0 \ (\lambda-. \ \text{undefined}) \gg= C$
 $1 \ n$

using *measurable-eP[of 0]* *measurable-C[of 1 n, measurable del]*

by (*simp add: bind-return[where N=Pi_M {0} (\lambda-. M)]*)

also have $\dots = K0 \ p \ x \gg= (\lambda a. C \ 1 \ n \ (\text{case-nat } a \ (\lambda-. \ \text{undefined})))$

unfolding *eP-def*

proof (*subst bind-distr[where K = Pi_M {0..<Suc n} (\lambda-. 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 ≫ (λa. 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 ≫ (λy. distr (lim-sequence (Suc-policy p y) (K y)) (ΠM
-∈UNIV. M) (case-nat y))
  (is - = ?B p x)
proof (rule measure-eqI-PiM-infinite)
  show sets (lim-sequence p x) = sets (ΠM j∈UNIV. M)
    using x p by (rule sets-lim-sequence)
  have [simp]: space (K' p x 0 (λn. undefined)) ≠ {}
    using p
    using IT-K' Ionescu-Tulcea.non-empty Ionescu-Tulcea.space-P x
by fastforce
  show sets (?B p x) = sets (PiM UNIV (λj. 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 (PiE J F') = PF.emb UNIV {0..<Suc
n} (PiE {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 (PiE J F')) =*
emeasure (C 0 (Suc n) ?U) (PiE {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 $C\ 0\ (Suc\ n)\ ?U = K0\ p\ x \gg (\lambda y. C\ 1\ n\ (case\ nat\ y$
?U))
using *split-C[of ?U 0 Suc 0 n] step-C[OF x p] by simp*
also have *emeasure (K0 p x >> (\lambda y. C 1 n (case-nat y ?U))) (PiE*
{0..<Suc n} F) =
 $(\int^+ y. C\ 1\ n\ (case\ nat\ y\ ?U)\ (PiE\ \{0..<Suc\ n\}\ F)\ \partial K0\ p\ x)$
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 - measur-*
able-C]])
(auto cong: measurable-cong-sets intro!: measurable-PiM-single'
split: nat.split-asm)
also have $\dots = (\int^+ y. distr\ (lim\ sequence\ (Suc\ policy\ p\ y)\ (K\ y))$
 $(PiM\ UNIV\ (\lambda j. M))\ (case\ nat\ y)\ (PF.emb\ UNIV\ J\ (PiE\ J\ F'))$
 $\partial K0\ p\ x)$
proof (*intro nn-integral-cong*)
fix y **assume** $y \in space\ (K0\ p\ x)$
then have $y: y \in space\ M$
using $x\ p$ *space-K0 by blast*
then interpret $y: Ionescu-Tulcea\ K'\ (Suc\ policy\ p\ y)\ (K\ y)\ \lambda\cdot. M$
using p **by** (*auto intro!: IT-K'*)
have $fst\ y \in space\ Ms$
by (*meson measurable-fst measurable-space y*)
let $?y = case\ nat\ y$
have [*simp*]: $?y\ ?U \in space\ (PiM\ \{0\}\ (\lambda i. M))$
using y **by** (*auto simp: space-PiM PiE-iff extensional-def split:*
nat.split)
have $yM[measurable]: ?y \in PiM\ \{0..<m\}\ (\lambda\cdot. M) \rightarrow_M PiM$
 $\{0..<Suc\ m\}\ (\lambda i. M)$ **for** m
using y
by (*auto intro: measurable-PiM-single' simp: space-PiM PiE-iff*
extensional-def split: nat.split)
have $y': ?y\ ?U \in space\ (PiM\ \{0..<1\}\ (\lambda i. M))$
by (*simp add: space-PiM PiE-def y extensional-def split: nat.split*)

have $eq1: ?y - ' PiE\ \{0..<Suc\ n\}\ F \cap space\ (PiM\ \{0..<n\}\ (\lambda\cdot.$

$M)) =$
 (if $y \in F$ 0 then $Pi_E \{0..<n\} (F \circ Suc)$ 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 $eq2: ?y - ' PF.emb UNIV \{0..<Suc\ n\} (Pi_E \{0..<Suc\ n\} F)$
 $\cap space (Pi_M UNIV (\lambda-. M)) =$
 (if $y \in F$ 0 then $PF.emb UNIV \{0..<n\} (Pi_E \{0..<n\} (F \circ Suc))$
 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 $fst\ y \in space\ Ms$
using y **by** (*meson measurable-fst measurable-space*)
have $C\ 1\ n\ (?y\ ?U) = distr (y.C\ 0\ n\ ?U) (\Pi_M\ i \in \{0..<Suc\ n\}.$
 $M)\ ?y$
proof (*induction n*)
case ($Suc\ m$)

have $C\ 1\ (Suc\ m)\ (?y\ ?U) = distr (y.C\ 0\ m\ ?U) (Pi_M \{0..<Suc$
 $m\} (\lambda i. M))\ ?y \gg eP (Suc\ m)$
using Suc **by** *simp*
also have $\dots = y.C\ 0\ m\ ?U \gg (\lambda x. eP (Suc\ m) (?y\ x))$
by (*auto intro!: bind-distr[where $K = Pi_M \{0..<Suc\} (Suc\ m)$]*
 $(\lambda-. M)]\ simp: y\ y.space-C\ y.sets-C\ cong: measurable-cong-sets$)
also have $\dots = y.C\ 0\ m\ ?U \gg (\lambda x. distr (y.eP\ m\ x) (Pi_M$
 $\{0..<Suc\} (Suc\ m)\} (\lambda i. M))\ ?y$
proof (*intro bind-cong refl*)
fix ω' **assume** $\omega': \omega' \in space (y.C\ 0\ m\ ?U)$
moreover have $K'\ p\ x\ (Suc\ m)\ (?y\ \omega') = K' (Suc-policy\ p\ y)$
 $(K\ y)\ m\ \omega'$
unfolding $K'-def\ Suc-policy-def$
by (*auto split: nat.splits*)
ultimately show $eP (Suc\ m)\ (?y\ \omega') = distr (y.eP\ m\ \omega') (Pi_M$
 $\{0..<Suc\} (Suc\ m)\} (\lambda i. M))\ ?y$
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 $\dots = distr (y.C\ 0\ m\ ?U \gg y.eP\ m) (Pi_M \{0..<Suc$
 $(Suc\ m)\} (\lambda i. M))\ ?y$
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 $\langle \text{simp add: PiM-empty distr-return} \rangle$)
then have $C\ 1\ n$ (case-nat $y\ ?U$) ($Pi_E\ \{0..<Suc\ n\}\ F$) =
 $(\text{distr}\ (y.C\ 0\ n\ ?U)\ (\prod_M\ i \in \{0..<Suc\ n\}. M)\ ?y)\ (Pi_E\ \{0..<Suc\ n\}\ F)$ **by** *simp*
also have $\dots = ?I * y.C\ 0\ n\ ?U\ (Pi_E\ \{0..<n\}\ (F \circ Suc))$
by (*subst emeasure-distr*) (*auto simp: y.sets-C y.space-C eq1 cong: measurable-cong-sets*)
also have
 $\dots = ?I * \text{lim-sequence}\ (Suc\text{-policy}\ p\ y)\ (K\ y)\ (PF.\text{emb}\ UNIV\ \{0..<n\}\ (Pi_E\ \{0..<n\}\ (F \circ Suc)))$
using $y\ \text{sets-PiM-I-finite}$
by (*subst emeasure-lim-sequence-emb-I0o*) (*auto simp add: p sets-PiM-I-finite*)
also have $\dots = \text{distr}\ (\text{lim-sequence}\ (Suc\text{-policy}\ p\ y)\ (K\ y))\ (Pi_M\ UNIV\ (\lambda j. M))\ ?y$
 $(PF.\text{emb}\ UNIV\ \{0..<Suc\ n\}\ (Pi_E\ \{0..<Suc\ n\}\ F))$
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 $y\ (\lambda-. \text{undefined})$)) ($Pi_E\ \{0..<Suc\ n\}\ F$) =
 $\text{emeasure}\ (\text{distr}\ (\text{lim-sequence}\ (Suc\text{-policy}\ p\ y)\ (K\ y))\ (Pi_M\ UNIV\ (\lambda j. M))\ (\text{case-nat}\ y))$
 $(y.PF.\text{emb}\ UNIV\ J\ (Pi_E\ J\ F'))$
unfolding *emb-eq* .
qed
also have $\dots = \text{emeasure}\ (K0\ p\ x \gg (\lambda y. \text{distr}\ (\text{lim-sequence}\ (Suc\text{-policy}\ p\ y)\ (K\ y))\ (Pi_M\ UNIV\ (\lambda j. M))\ (\text{case-nat}\ y)))\ (PF.\text{emb}\ UNIV\ J\ (Pi_E\ J\ F'))$
using $J\ \text{sets-K0}[OF\ \langle \text{is-policy}\ p \rangle\ \langle x \in \text{space}\ (\text{prob-algebra}\ Ms) \rangle]\ p$
by (*subst emeasure-bind[where N=PiM UNIV (lambda-. 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 (lambda-. M)]
finally show *emeasure* ($\text{lim-sequence}\ p\ x$) ($PF.\text{emb}\ UNIV\ J\ (Pi_E\ J\ F')$) =

$\text{emeasure } (K0 \ p \ x \gg= (\lambda y. \text{distr } (\text{lim-sequence } (\text{Suc-policy } p \ y) \ (K \ y)) \ (Pi_M \ UNIV \ (\lambda j. \ M)) \ (\text{case-nat } y))) \ (PF.\text{emb } UNIV \ J \ (Pi_E \ J \ F'))$.
qed

5.9 Stream Space of the MDP

definition $\text{lim-stream} :: ('s, 'a) \text{pol} \Rightarrow 's \text{ measure} \Rightarrow ('s \times 'a) \text{ stream measure}$

where

$\text{lim-stream } p \ x = \text{distr } (\text{lim-sequence } p \ x) \ (\text{stream-space } M) \ \text{to-stream}$

lemma space-lim-stream : $\text{space } (\text{lim-stream } p \ x) = \text{streams } (\text{space } M)$
unfolding lim-stream-def **by** $(\text{simp add: space-stream-space})$

lemma $\text{sets-lim-stream[measurable-cong]}$: $\text{sets } (\text{lim-stream } p \ x) = \text{sets } (\text{stream-space } M)$

unfolding lim-stream-def **by** simp

lemma $\text{lim-stream[measurable]}$:

assumes $\text{is-policy } p$

shows $\text{lim-stream } p \in \text{prob-algebra } Ms \rightarrow_M \text{prob-algebra } (\text{stream-space } M)$

unfolding $\text{lim-stream-def[abs-def]}$

using assms

by $(\text{auto intro: measurable-distr-prob-space2[OF lim-sequence]})$

lemma $\text{lim-stream-Suc[measurable]}$:

assumes p : $\text{is-policy } p$

shows $(\lambda a. \text{lim-stream } (\text{Suc-policy } p \ a) \ (K \ a)) \in M \rightarrow_M \text{prob-algebra } (\text{stream-space } M)$

unfolding $\text{lim-stream-def[abs-def]}$

using p

by $(\text{auto intro: measurable-distr-prob-space2[OF lim-sequence-Suc-K]})$

lemma $\text{space-stream-space-M-ne}$: $x \in \text{space } M \implies \text{space } (\text{stream-space } M) \neq \{\}$

using $\text{sconst-streams[of } x \ \text{space } M]$ **by** $(\text{auto simp: space-stream-space})$

lemma $\text{prob-space-lim-stream[intro]}$:

assumes $\text{is-policy } p \ x \in \text{space } (\text{prob-algebra } Ms)$

shows $\text{prob-space } (\text{lim-stream } p \ x)$

by $(\text{metis } (\text{no-types, lifting}) \ \text{space-prob-algebra measurable-space assms } \text{lim-stream mem-Collect-eq})$

lemma prob-space-step :

assumes $\text{is-policy } p \ x \in \text{space } M$

shows $\text{prob-space } (\text{lim-stream } (\text{Suc-policy } p \ x) \ (K \ x))$

by $(\text{auto simp: assms } K\text{-in-space is-policy-Suc-policy})$

```

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

```

```

show  $\bigwedge a. a \in \text{space } (K0\ p\ x) \implies$ 
   $\text{distr } (\text{lim-sequence } (\text{Suc-policy } p\ a) (K\ a)) (\text{stream-space } M)$ 
   $(\text{to-stream } \circ \text{case-nat } a) =$ 
   $\text{distr } (\text{lim-sequence } (\text{Suc-policy } p\ a) (K\ a)) (\text{stream-space } M)$ 
   $\text{to-stream } \gg$ 
   $(\lambda\omega. \text{return } (\text{stream-space } M) (a\ \#\#\ \omega))$ 
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  $\Rightarrow$  ('b :: {banach, second-countable-topology, real-normed-vector})
assumes int-f: integrable (stream-space M) f
assumes [measurable]: f  $\in$  borel-measurable (stream-space M)
shows  $(\int X. f\ X\ \partial \text{stream-space } M) = (\int x. (\int X. f\ (x\ \#\#\ X)$ 
 $\partial \text{stream-space } M)\ \partial M)$ 
proof -
interpret S: sequence-space M ..
interpret P: pair-sigma-finite M  $\Pi_M\ i::\text{nat} \in \text{UNIV}. M$  ..

interpret P': pair-sigma-finite  $\Pi_M\ i::\text{nat} \in \text{UNIV}. M\ M$  ..

```

obtain i **where** *has-bochner-integral (stream-space M) f i*
using *int-f*
using *integrable.cases by blast*
have *integrable S.S (λX. f (to-stream X))*
using *int-f*
by (*metis integrable-distr measurable-to-stream stream-space-eq-distr*)
hence *integrable (distr (M ⊗_M Pi_M UNIV (λi. M)) (Pi_M UNIV*
(λi. M))
(λ(x, y). case-nat x y)) (λX. f (to-stream X))
by (*auto simp: S.PiM-iter*)
moreover have *integrable (distr (M ⊗_M Pi_M UNIV (λi. M))*
(Pi_M UNIV (λi. M)) (λ(x, y). case-nat x y))
(λX. f (to-stream X)) ↔
integrable (M ⊗_M S.S) (λX. f (to-stream ((λ(s, ω). case-nat s ω)
X)))
by (*auto simp: integrable-distr-eq*)
ultimately have *integrable (M ⊗_M S.S) (λX. f (to-stream ((λ(s,*
ω). case-nat s ω) X)))
by *auto*
hence *integrable (M ⊗_M (Pi_M UNIV (λi. M)))*
(λX. f (to-stream ((λ(s, ω). case-nat s ω) X)))
by *auto*
moreover have *integrable (M ⊗_M (Pi_M UNIV (λi. M)))*
(λX. f (to-stream ((λ(s, ω). case-nat s ω) X))) =
integrable (M ⊗_M Pi_M UNIV (λi. M)) (λ(x, X). f (to-stream
(case-nat x X)))
by (*auto intro!: Bochner-Integration.integrable-cong*)
ultimately have **: integrable (M ⊗_M Pi_M UNIV (λi. M)) (λ(x,*
X). f (to-stream (case-nat x X)))
by *auto*
have $(\int X. f X \partial \text{stream-space } M) = (\int X. f \text{ (to-stream } X) \partial S.S)$
by (*subst stream-space-eq-distr (simp add: integral-distr)*)
also have $\dots = (\int X. f \text{ (to-stream ((}\lambda(s, \omega). \text{ case-nat } s \omega) X)) \partial (M$
 $\otimes_M S.S))$
by (*subst S.PiM-iter[symmetric] (simp add: integral-distr)*)
also have $\dots = (\int x. \int X. f \text{ (to-stream ((}\lambda(s, \omega). \text{ case-nat } s \omega) (x,$
 $X))) \partial S.S \partial M)$
using ***
by (*auto simp: pair-sigma-finite.integral-fst P.pair-sigma-finite-axioms*
case-prod-unfold)
also have $\dots = (\int x. \int X. f (x \#\# \text{ to-stream } X) \partial S.S \partial M)$
by (*auto intro!: integral-cong simp: to-stream-nat-case*)
also have $\dots = (\int x. \int X. f (x \#\# X) \partial \text{distr } (Pi_M UNIV (\lambda i.$
 $M)) \text{ (stream-space } M) \text{ to-stream } \partial M)$
by (*subst Bochner-Integration.integral-cong[OF refl] (auto simp:*
integral-distr))
also have $\dots = (\int x. \int X. f (x \#\# X) \partial \text{stream-space } M \partial M)$
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 (λn. seq (Suc n))
by (*metis map-upt-Suc subsequence-def*)

lemma *restrict-Suc*: *restrict y {0..<Suc i} (Suc n) = (restrict (λn. 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*: *(λw. (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 *(λw:: 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  $\Rightarrow$  'a::countable set — enabled actions
  and K :: 's  $\times$  'a  $\Rightarrow$  's pmf — MDP kernel, transition probabilities
  assumes
    A-ne:  $\bigwedge s. A\ s \neq \{\}$  — set of enabled actions is nonempty
begin

```

6.1 Policies

Type synonym for decision rules.

```

type-synonym ('c, 'd) dec = 'c  $\Rightarrow$  'd pmf

```

```

definition is-dec :: ('s, 'a) dec  $\Rightarrow$  bool where
  is-dec d  $\equiv \forall s. d\ s \subseteq A\ s$ 

```

```

lemma is-decI[intro]:
  ( $\bigwedge s. set-pmf\ (d\ s) \subseteq A\ s$ )  $\Longrightarrow$  is-dec d
  unfolding is-dec-def
  by auto

```

```

abbreviation DR  $\equiv \{d. is-dec\ d\}$ 

```

```

definition is-dec-det :: ('s  $\Rightarrow$  'a)  $\Rightarrow$  bool where
  is-dec-det d  $\equiv \forall s. d\ s \in A\ s$ 

```

```

abbreviation DD  $\equiv \{d. is-dec-det\ d\}$ 

```

```

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

```

```

lemma is-dec-mk-dec-det-iff [simp]: is-dec (mk-dec-det d)  $\longleftrightarrow$  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  $\Longrightarrow$  is-dec (mk-dec-det d)
  by simp

```

Due to the assumption $A\ ?s \neq \{\}$, 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 ( $\lambda s. SOME\ a. a \in A\ s$ )
  using A-ne by (simp add: is-dec-det-def some-in-eq)

```

lemma *ex-dec-det* [simp]: $\exists d. \text{is-dec-det } d$
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*]: $\exists d. \text{is-dec } d$
using *ex-dec-det* **by** *blast*

Type synonym for policies.

type-synonym ('c, 'd) *pol* = ('c × 'd) *list* ⇒ ('c, 'd) *dec*

A policy assigns a decision rule to each observed past.

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

abbreviation $\Pi_{HR} \equiv \{p. \text{is-policy } p\}$

Deterministic policies

definition *is-deterministic* *p* ≡ *is-policy* *p* ∧ (∀ *h s*. ∃ *a*. *p* *h s* = *return-pmf a*)

definition *mk-det* *p h s* ≡ *return-pmf* (*p h s*)

abbreviation $\Pi_{HD} \equiv \{p. \forall h. p \ h \in D_D\}$

Markovian policies

definition *is-markovian* *p* ≡ *is-policy* *p* ∧ (∀ *h h'*. *length h* = *length h'* → *p h* = *p h'*)

definition *mk-markovian* :: (*nat* ⇒ ('s, 'a) *dec*) ⇒ ('s, 'a) *pol* **where**
mk-markovian *p* ≡ (λ*h*. *p* (*length h*))

lemma *is-markovian-mk-iff*[*simp*]: *is-markovian* (*mk-markovian* *p*) ↔ (∀ *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*]: ∀ *n*. *is-dec* (*p n*) ⇒ *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* $p \equiv (\lambda h s. \text{return-pmf } (p \text{ (length } h) s))$

abbreviation $\Pi_{MD} \equiv \{p. \forall n::\text{nat}. p \ n \in D_D\}$

abbreviation $\Pi_{MR} \equiv \{p. \forall n. p \ n \in D_R\}$

lemma $\Pi_{MR}\text{-imp-policies[intro]}$: $p \in \Pi_{MR} \implies \text{mk-markovian } p \in \Pi_{HR}$

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

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

by auto

lemma $\Pi_{MD}\text{-to-MR[intro]}$: $p \in \Pi_{MD} \implies (\lambda n. \text{mk-dec-det } (p \ n)) \in \Pi_{MR}$

by simp

lemma $p\text{-n-}\pi\text{-MD[intro]}$: $p \in \Pi_{MD} \implies p \ n \in D_D$

by auto

lemma $p\text{-n-}\pi\text{-MR[intro]}$: $p \in \Pi_{MR} \implies p \ n \in D_R$

by auto

lemma $\Pi_{MD}\text{-ne[simp]}$: $\Pi_{MD} \neq \{\}$

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

lemma $\Pi_{MR}\text{-ne[simp]}$: $\Pi_{MR} \neq \{\}$

using $\Pi_{MD}\text{-ne}$ **by fast**

lemma $\text{policies-ne[simp, intro]}$: $\Pi_{HR} \neq \{\}$

using $\Pi_{MR}\text{-ne is-policy-def}$ **by auto**

Stationary policies

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

lemma $\text{is-stationary-const-iff[simp]}$: $\text{is-stationary } (\lambda-. d) = \text{is-dec } d$

unfolding *is-stationary-def is-policy-def* **by simp**

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

by simp

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

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

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 $\pi\text{-Suc} :: ('s, 'a) \text{ pol} \Rightarrow 's \times 'a \Rightarrow ('s, 'a) \text{ pol}$

where

$$\pi\text{-Suc } p \text{ sa } h = p \text{ (sa\#h)}$$

lemma *is-policy- π -Suc* [intro]: *is-policy* $p \implies$ *is-policy* ($\pi\text{-Suc } p \text{ sa}$)

unfolding *is-policy-def* $\pi\text{-Suc-def}$ **by** *force*

lemma *Suc-mk-markovian*[simp]: $\pi\text{-Suc} \text{ (mk-markovian } p) x = \text{mk-markovian} (\lambda n. p \text{ (Suc } n))$

unfolding $\pi\text{-Suc-def}$ *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 $K0 \ d \ S0 = \text{do } \{$

$s \leftarrow S0;$

$a \leftarrow d \ s;$

return-pmf (s, a)

$\}$

notation $K0 \ (\langle K0 \rangle)$

lemma *K0-iff*: $K0 \ d \ S0 = S0 \gg= (\lambda s. \text{map-pmf} (\lambda a. (s, a)) (d \ s))$

by (*simp add: K0-def map-pmf-def*)

lemma *vimage-pair*[simp]: $\text{Pair } x \text{ - ' } \{p\} = (\text{if } x = \text{fst } p \text{ then } \{\text{snd } p\} \text{ else } \{\})$

by *auto*

lemma *pmf-K0* [simp]: $\text{pmf} (K0 \ d \ S0) (s, a) = \text{pmf } S0 \ s * \text{pmf} (d \ s) \ a$

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*: $\text{set-pmf} (K0 \ p \ S0) = \{(s, a). s \in S0 \wedge a \in p \ s\}$

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

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

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

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

interpretation *MDP-cont*: $MDP\text{-cont.}discrete\text{-MDP } \text{count-space } UNIV$
 $\text{count-space } UNIV \text{ } A \text{ } K$

proof *standard*

show $(\lambda x. \text{measure-pmf } (K \text{ } x)) \in$
 $\text{count-space } UNIV \otimes_M \text{count-space } UNIV \rightarrow_M \text{prob-algebra}$
 $(\text{count-space } UNIV)$

using *measurable-prob-algebraI*

by (*measurable, auto simp: prob-space-measure-pmf measurable-pair-measure-countable1*)

show $\exists \delta \in \text{count-space } UNIV \rightarrow_M \text{count-space } UNIV. \forall s. \delta \text{ } s \in A \text{ } s$

by (*auto simp: A-ne some-in-eq intro: bexI[of - $\lambda s. \text{SOME } a. a \in A \text{ } s]$*)

qed (*auto simp: A-ne*)

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

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

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

definition $T :: ('s, 'a) \text{pol} \Rightarrow 's \text{ pmf} \Rightarrow ('s \times 'a) \text{stream measure}$
where $T \text{ } p = MDP\text{-cont.}lim\text{-stream } (\lambda n \text{ } (h, s). p \text{ } (\text{Omega-}Words\text{-}Fun.prefix \text{ } n \text{ } h) \text{ } s)$

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

lemma *space-stream-space-ne[simp]*: $\text{space } S \neq \{\}$
by (*auto simp: space-stream-space*)

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

lemma *is-policy-MDP-cont[intro]*:
fixes $p :: ('s \times 'a) \text{list} \Rightarrow 's \Rightarrow 'a \text{ pmf}$
shows $MDP\text{-cont.}is\text{-policy } (\lambda n \text{ } (h, s). p \text{ } (\text{Omega-}Words\text{-}Fun.prefix \text{ } n \text{ } h) \text{ } s)$
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 (λx (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 $\llbracket \text{MDP-cont.is-policy } ?p; ?x \in \text{space (prob-algebra (count-space UNIV))} \rrbracket \implies \text{MDP-cont.lim-stream } ?p ?x = \text{MDP-cont.K0 } ?p ?x \ggg (\lambda y. \text{MDP-cont.lim-stream (MDP-cont.Suc-policy } ?p y) (\text{measure-pmf (K } y)) \ggg (\lambda \omega. \text{return (stream-space MDP-cont.M) (y \#\# \omega)}))$.

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* ($\lambda x xa. \text{measure-pmf } (\text{case } xa \text{ of } (h, xa))$
 $\Rightarrow p$ (*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 $T p S0 = \text{measure-pmf } (K0 (p [])) S0 \ggg (\lambda sa. \text{distr } (T$
 $(\pi\text{-Suc } p sa) (K sa)) S ((\#\#) sa))$
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 $f :: ('s \times 'a) \text{ stream} \Rightarrow \text{real}$
assumes *f-bounded*: $\bigwedge x. |f x| \leq B$
assumes $f: f \in \text{borel-measurable } S$
shows $(\int t. f t \partial T p x) = \int sa. \int t'. f (sa\#\#t') \partial T (\pi\text{-Suc } p sa)$
 $(K sa) \partial K0 (p []) x$
proof –
note *T-eq-distr*
have $(\int t. f t \partial T p x) = (\int t. f t \partial \text{measure-pmf } (K0 (p [])) x) \ggg$
 $(\lambda sa. \text{distr } (T (\pi\text{-Suc } p sa) (K sa)) (\text{stream-space } (\text{count-space } UNIV))$
 $((\#\#) sa)))$
using *T-eq-distr by metis*
also have $\dots = \text{measure-pmf.expectation } (K0 (p [])) x (\lambda sa. \text{LINT}$
 $t' | T (\pi\text{-Suc } p sa) (K sa). f (sa \#\# t'))$
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.emmeasure-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 $f: f \in \text{borel-measurable } S$
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 $T\text{-eq-distr}[of p]$
by $(\text{subst } nn\text{-integral-bind}[OF f])$
 $(\text{auto intro!}: \text{prob-space-imp-subprob-space } \text{prob-space.} \text{prob-space-distr simp}: f \text{ nn-integral-distr } \text{space-subprob-algebra})$

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 $Pn :: ('s, 'a) \text{pol} \Rightarrow 's \text{ pmf} \Rightarrow \text{nat} \Rightarrow ('s \times 'a) \text{ pmf}$ **where**
 $Pn p S0 0 = K0 (p \[]) S0$
 $| Pn p S0 (\text{Suc } n) = K0 (p \[]) S0 \gg (\lambda sa. Pn (\pi\text{-Suc } p sa) (K sa) n)$

declare $Pn.\text{simps}(2)[\text{simp del}]$

lemma $Pn\text{-eq-T}$: $\text{measure-pmf } (Pn p S0 n) = \text{distr } (T p S0) (\text{count-space } UNIV) (\lambda t. t !! n)$

proof $(\text{induction } n \text{ arbitrary}: p S0)$

case $(0 p S0)$

then show $?case$

unfolding $T\text{-eq}[of p]$

proof $(\text{subst } \text{distr-bind}[\text{where } K = S], \text{goal-cases})$

case 1

then show $?case$

by $(\text{auto intro!}: \text{prob-space-imp-subprob-space } \text{subprob-space.} \text{bind-in-space})$

next

case 4

then show $?case$

by $(\text{subst } \text{bind-cong}[OF \text{ refl}, \text{ where } g = \text{return } (\text{count-space } UNIV)])$

$(\text{auto intro!}: \text{bind-const' } \text{simp}: \text{distr-bind}[\text{where } K = S] \text{ distr-return } \text{bind-return'' } \text{space-stream-space } \text{subprob-space-return-ne})$

qed *auto*

next

case $(\text{Suc } n)$

show $?case$

unfolding $T\text{-eq}[of p]$

proof $(\text{subst } \text{distr-bind}[\text{where } K = S], \text{goal-cases})$

case 1

then show $?case$

by $(\text{auto intro!}: \text{prob-space-imp-subprob-space } \text{subprob-space.} \text{bind-in-space})[1]$

```

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 $p \implies (s, a) \in Pn\ p\ S0\ n \implies a \in A\ s$*

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 $Xn\ p\ S0\ n = \text{map-pmf}\ \text{fst}\ (Pn\ p\ S0\ n)$

lemma $X0\ [simp]: Xn\ p\ S0\ 0 = S0$

using fst-K0 Xn-def **by** auto

lemma $Xn\text{-Suc}: Xn\ p\ S0\ (Suc\ n) = Pn\ p\ S0\ n \ggg K$

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*: $Pn (mk\text{-markovian } p) S0\ n = K0 (p\ n) (Xn (mk\text{-markovian } p) S0\ n)$

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'*: $Xn\ p\ S0\ (Suc\ n) = K0\ (p\ [])\ S0 \ggg (\lambda sa. Xn\ (\pi\text{-Suc } p\ sa)\ (K\ sa)\ n)$

unfolding *Xn-def* **by** (*auto simp: Pn.simps(2) map-bind-pmf*)

lemma *set-pmf-X0* [*simp*]: $set\text{-pmf } (Xn\ p\ S0\ 0) = S0$

using *X0* **by** *auto*

lemma *set-pmf-PSuc*: $set\text{-pmf } (Pn\ (mk\text{-markovian } p) S0\ n) = \{(s, a). s \in set\text{-pmf } (Xn\ (mk\text{-markovian } p) S0\ n) \wedge a \in p\ n\ s\}$

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* $p\ S0\ n\ x = map\text{-pmf } snd\ (cond\text{-pmf } (Pn\ p\ S0\ n)\ \{(s, a). s = x\})$

lemma *prob-K0-X* [*simp*]: $measure\text{-pmf}.prob\ (K0\ p\ S0)\ \{(s, a). s = x\} = pmf\ S0\ x$

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.emmeasure-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.emmeasure-eq-measure*)

aux' *measure-pmf-single*)
qed

lemma *pmf-Y-cond-X*:

assumes $x \in \text{set-pmf } (Xn \text{ } p \text{ } S0 \text{ } n)$
shows $\text{pmf } (Y\text{-cond-}X \text{ } p \text{ } S0 \text{ } n \text{ } x) \text{ } a = \text{pmf } (Pn \text{ } p \text{ } S0 \text{ } n) \text{ } (x, a) / \text{pmf } (Xn \text{ } p \text{ } S0 \text{ } n) \text{ } x$

proof –

have *aux*: $\text{set-pmf } (Pn \text{ } p \text{ } S0 \text{ } n) \cap \{(s, a). s = x\} \neq \{\}$
using *Xn-def* *assms* **by** *auto*
have *aux'*: $(\{(s, a). s = x\} \cap \text{snd} -' \{a\}) = \{(x, a)\}$
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.emmeasure-eq-measure
measure-pmf-single)

qed

lemma *Y-cond-X-0[simp]*:

assumes $x \in \text{set-pmf } S0$
shows $Y\text{-cond-}X \text{ } p \text{ } S0 \text{ } 0 \text{ } x = p \text{ } \square \text{ } x$
by (*auto intro: pmf-eqI simp: assms pmf-Y-cond-X pmf-eq-0-set-pmf*)

lemma *Y-cond-X-markovian[simp]*:

assumes $h: x \in Xn \text{ } (mk\text{-markovian } p) \text{ } S0 \text{ } n$
shows $Y\text{-cond-}X \text{ } (mk\text{-markovian } p) \text{ } S0 \text{ } n \text{ } x = p \text{ } n \text{ } x$
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*: $Pn \text{ } p \text{ } S0 \text{ } n = Xn \text{ } p \text{ } S0 \text{ } n \ggg (\lambda x. \text{map-pmf } (\lambda a. (x, a) \text{ } (Y\text{-cond-}X \text{ } p \text{ } S0 \text{ } n \text{ } x)))$

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'*: $\text{pmf } (Xn \text{ } p \text{ } S0 \text{ } (Suc \text{ } n)) \ggg (\lambda x. \text{map-pmf } (Pair \text{ } x) \text{ } (Y\text{-cond-}X \text{ } p \text{ } S0 \text{ } (Suc \text{ } n) \text{ } x)) \text{ } (a, b)$

$= \text{measure-pmf.expectation } (Pn \text{ } p \text{ } S0 \text{ } (Suc \text{ } n)) \text{ } (\lambda x.$

$\text{if } fst \text{ } x = a \text{ then } \text{pmf } (Y\text{-cond-}X \text{ } p \text{ } S0 \text{ } (Suc \text{ } n) \text{ } a) \text{ } b \text{ else } 0)$

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 $Y_n p S0 n = \text{map-pmf snd } (Pn p S0 n)$

lemma $Y0$: $Y_n p S0 0 = S0 \ggg p \square$
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 Y_n solely in terms of X_n and the current decision rule.

lemma Y_n -markovian: $Y_n (\text{mk-markovian } p) S0 n = X_n (\text{mk-markovian } p) S0 n \ggg p n$

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 $\text{as-markovian } p S0 n x \equiv$

if $x \in (X_n p S0 n)$ *then* $Y\text{-cond-}X p S0 n x$ *else* $\text{return-pmf } (SOME a. a \in A x)$

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

lemma $\text{is-}\Pi_{MR}\text{-as-markovian}$:

assumes p : $\text{is-policy } p$

shows $\text{as-markovian } p S0 \in \Pi_{MR}$

proof –

have aux : $\bigwedge hs s. s \in \text{set-pmf } (X_n p S0 hs) \implies \text{set-pmf } ((Pn p S0 hs) \cap \{(s', a). s' = s\}) \neq \{\}$

by (*simp add: measure-pmf-zero-iff[symmetric] pmf-eq-0-set-pmf*)

thus ?thesis

using $\text{assms } A\text{-ne } Pn\text{-in-}A$ **by** (*auto simp: is-dec-def some-in-eq Y-cond-X-def*)

qed

lemma $\text{is-policy-as-markovian}$: $\text{is-policy } p \implies \text{is-policy } (\text{mk-markovian } (\text{as-markovian } p S0))$

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 $\bigwedge x. x \in Xn\ p\ S0\ (Suc\ n) \implies$

$Y\text{-cond-}X\ (mk\text{-markovian}\ (as\text{-markovian}\ p\ S0))\ S0\ (Suc\ n)\ x =$
 $Y\text{-cond-}X\ p\ S0\ (Suc\ n)\ x$

by (*auto simp: Suc.IH Xn-Suc*)

moreover **have** $Xn\ (mk\text{-markovian}\ (as\text{-markovian}\ p\ S0))\ S0\ (Suc\ n) = Xn\ p\ S0\ (Suc\ n)$

by (*simp add: Xn-Suc Suc.IH*)

ultimately **show** $Pn\ (mk\text{-markovian}\ (as\text{-markovian}\ p\ S0))\ S0\ (Suc\ n) = Pn\ p\ S0\ (Suc\ n)$

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 the initial state is of the form *return-pmf s*.

definition $\mathcal{T}\ p\ s \equiv T\ p\ (return\text{-pmf}\ s)$

lemma *\mathcal{T} -eq-return-distr: $\mathcal{T}\ p\ s =$*

measure-pmf (p \square s) $\gg=$ ($\lambda a. distr\ (T\ (\pi\text{-Suc}\ p\ (s,a))\ (K\ (s,a)))\ S\ ((\#\#)\ (s,a))$)

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 $\mathcal{T}\ p\ s = do\ \{$

$y \leftarrow measure\text{-pmf}\ (p\ \square\ s);$

$\omega \leftarrow T\ (\pi\text{-Suc}\ p\ (s,y))\ (K\ (s,y));$

$return\ S\ ((s,y)\ \#\#\ \omega)$

$\}$

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

lemma \mathcal{T} -return:

shows $T\ p\ S0 = \text{measure-pmf}\ S0 \ggg T\ p$

proof –

have $T\ p\ S0 = \text{measure-pmf}\ S0 \ggg (\lambda x. \text{measure-pmf}\ (\text{map-pmf}\ (\text{Pair}\ x)\ (p\ \square\ x))) \ggg$
 $(\lambda sa. \text{distr}\ (T\ (\pi\text{-Suc}\ p\ sa)\ (K\ sa))\ (\text{stream-space}\ (\text{count-space}\ UNIV))\ ((\#\#)\ sa))$

unfolding $T\text{-eq-distr}[of\ p]\ K0\text{-iff}\ \text{measure-pmf-bind}$

by *auto*

also have $\dots = \text{measure-pmf}\ S0 \ggg$

$(\lambda x. \text{distr}\ (\text{measure-pmf}\ (p\ \square\ x))\ (\text{count-space}\ UNIV)\ (\text{Pair}\ x)) \ggg$

$(\lambda sa. \text{distr}\ (T\ (\pi\text{-Suc}\ p\ sa)\ (K\ sa))\ (\text{stream-space}\ (\text{count-space}\ UNIV))\ ((\#\#)\ sa))$

$UNIV))\ ((\#\#)\ sa))$

using *measurable-measure-pmf*

by (*subst bind-assoc[where $N = \text{count-space}\ UNIV$, 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 $\dots = \text{measure-pmf}\ S0 \ggg T\ p$

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:

$T\ p\ s = \text{do}\ \{\$

$a \leftarrow \text{measure-pmf}\ (p\ \square\ s);$

$s' \leftarrow \text{measure-pmf}\ (K\ (s,a));$

$w \leftarrow T\ (\pi\text{-Suc}\ p\ (s,a))\ (\text{return-pmf}\ s');$

$\text{return}\ S\ ((s,a)\#\#w)$

$\}$

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 $T\ p\ s = \text{do}\ \{\$

$a \leftarrow \text{measure-pmf}\ (p\ \square\ s);$

$s' \leftarrow \text{measure-pmf}\ (K\ (s,a));$

$w \leftarrow T\ (\pi\text{-Suc}\ p\ (s,a))\ s';$

$\text{return}\ S\ ((s,a)\#\#w)$

$\}$

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*]:
 $(\lambda x. x) \in \mathcal{T} (\pi\text{-Suc } p \text{ } sa) \text{ } s' \rightarrow_M S$
by (*simp* *add*: \mathcal{T} -def)

lemma *nn-integral-T*:
fixes $f :: ('s \times 'a) \text{ stream} \Rightarrow \text{real}$
assumes f [*measurable*]: $f \in \text{borel-measurable } S$
shows $(\int^{+t}. f \text{ } t \text{ } \partial \mathcal{T} \text{ } p \text{ } s)$
 $= \int^{+a}. \int^{+s'}. \int^{+t'}. f \text{ } ((s,a)\#\#t') \text{ } \partial \mathcal{T} (\pi\text{-Suc } p \text{ } (s,a)) \text{ } s' \text{ } \partial K \text{ } (s,a)$
 $\partial p \text{ } \square \text{ } s$

proof –
have $(\int^{+t}. f \text{ } t \text{ } \partial \mathcal{T} \text{ } p \text{ } s) =$
 $\int^{+x}. \int^{+y}. (f \text{ } y) \text{ } \partial \text{measure-pmf} (K \text{ } (s, x)) \gg (\lambda s'. \mathcal{T} (\pi\text{-Suc } p \text{ } (s,$
 $x)) \text{ } s' \gg (\lambda w. \text{return } S ((s, x) \#\# w))) \text{ } \partial(p \text{ } \square \text{ } s)$
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** $\dots = \int^{+x}. \int^{+xa}. \int^{+y}. (f \text{ } y) \text{ } \partial \mathcal{T} (\pi\text{-Suc } p \text{ } (s, x)) \text{ } xa$
 $\gg (\lambda w. \text{return } S ((s, x) \#\# w))$
 $\text{ } \partial \text{measure-pmf} (K \text{ } (s, x)) \text{ } \partial(p \text{ } \square \text{ } s)$
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** $\dots = \int^{+x}. \int^{+xa}. \int^{+y}. (f \text{ } y) \text{ } \partial \text{distr} (\mathcal{T} (\pi\text{-Suc } p \text{ } (s,$
 $x)) \text{ } xa) \text{ } S ((\#\#) (s, x)) \text{ } \partial \text{measure-pmf} (K \text{ } (s, x)) \text{ } \partial(p \text{ } \square \text{ } s)$
by (*auto* *simp* *add*: *bind-return-distr'* \mathcal{T} -*prob-space* *prob-space.not-empty*)
also **have** $\dots = \int^{+x}. \int^{+xa}. \int^{+xa}. (f \text{ } ((s, x) \#\# xa)) \text{ } \partial \mathcal{T} (\pi\text{-Suc}$
 $p \text{ } (s, x)) \text{ } xa \text{ } \partial \text{measure-pmf} (K \text{ } (s, x)) \text{ } \partial(p \text{ } \square \text{ } s)$
by (*auto* *simp*: *nn-integral-distr*)
finally **show** *?thesis*.
qed

lemma *integral-T*:
fixes $f :: ('s \times 'a) \text{ stream} \Rightarrow \text{real}$
assumes f -*bounded*: $\bigwedge x. |f \text{ } x| \leq B$
assumes f [*measurable*]: $f \in \text{borel-measurable } S$
shows $(\int^t. f \text{ } t \text{ } \partial \mathcal{T} \text{ } p \text{ } s)$
 $= \int^a. \int^{s'}. \int^{t'}. f \text{ } ((s,a)\#\#t') \text{ } \partial \mathcal{T} (\pi\text{-Suc } p \text{ } (s,a)) \text{ } s' \text{ } \partial K \text{ } (s,a) \text{ } \partial p$
 $\square \text{ } s$
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 $f :: ('s \times 'a) \text{ stream} \Rightarrow 'd :: \{\text{second-countable-topology, banach}\}$
assumes $f[\text{measurable}]$: $f \in \text{borel-measurable } S$
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 $K0' d s \equiv \text{map-pmf } (\lambda a. (s, a)) (d s)$

definition $K\text{-st } d s \equiv d s \ggg (\lambda a. K (s, a))$

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

$K\text{-st}$ 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 $P0'[\text{simp}]$: $Pn' p s 0 = K0' (p []) s$
by (simp add: Pn' -def $K0'$ -def $K0$ -iff bind-return-pmf)

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

lemma Pn -return-pmf: $S0 \ggg (\lambda s'. Pn p (\text{return-pmf } s') n) = Pn p S0 n$
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 $PSuc'$: $Pn' p s (Suc n) = K0' (p []) s \ggg (\lambda sa. K sa \ggg (\lambda s'. Pn' (\pi\text{-Suc } p sa) s' n))$
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:
 $Pn' (mk\text{-markovian } p) s (Suc n) = K\text{-st } (p 0) s \ggg (\lambda s'. Pn'$

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

lemma $Xn'\text{-Suc}$: $Xn' p s (\text{Suc } n) = Pn' p s n \ggg K$
by (*auto simp: Xn-Suc Xn'-def Pn'-def*)

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

lemma Suc-Xn' : $Xn' p s (\text{Suc } n) = p [] s \ggg (\lambda a. K (s, a) \ggg (\lambda s'. Xn' (\pi\text{-Suc } p (s, a)) s' n))$
by (*auto simp: Xn'-Pn' map-bind-pmf bind-map-pmf PSuc' K0'-def*)

lemma Suc-Xn'-markovian :
 $Xn' (\text{mk-markovian } p) s (\text{Suc } n) = K\text{-st } (p 0) s \ggg (\lambda s'. Xn' (\text{mk-markovian } (\lambda n. p (\text{Suc } n))) s' n)$
by (*auto simp: K-st-def bind-assoc-pmf Suc-Xn'*)

lemma $Xn'\text{-split}$: $Xn' (\text{mk-markovian } p) s (n + m) = Xn' (\text{mk-markovian } p) s n \ggg (\lambda s. Xn' (\text{mk-markovian } (\lambda i. p (i + n))) s m)$
by (*induction n arbitrary: p s*) (*auto intro!: bind-pmf-cong simp: bind-assoc-pmf bind-return-pmf Suc-Xn'*)

lemma $Yn'\text{-markovian}$: $Yn' (\text{mk-markovian } p) s n = Xn' (\text{mk-markovian } p) s n \ggg p n$
unfolding $Yn'\text{-def } Xn'\text{-def } Yn\text{-markovian}$ **by** *simp*

lemma $Pn'\text{-markovian-eq-Xn'-bind}$: $Pn' (\text{mk-markovian } p) s n = Xn' (\text{mk-markovian } p) s n \ggg K0' (p n)$
unfolding $Xn'\text{-def } Pn'\text{-def } K0'\text{-def } K0\text{-iff } Pn\text{-markovian-eq-Xn-bind}$
by *simp*

lemma $Pn'\text{-eq-T}$: $\text{measure-pmf } (Pn' p s n) = \text{distr } (\mathcal{T} p s) (\text{count-space UNIV}) (\lambda t. t !! n)$
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 = (⌊ 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]: (λt. 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 integrable-abs-bound] measure-pmf.integrable-const-bound*)

7.1.2 Infinite discounted sums

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

lemma *norm-l-pow-eq*[*simp*]: $norm$ ($l \hat{^} t *_{\mathbb{R}}$ F) = $l \hat{^} t * norm$ F
by *auto*

7.2 Total Reward for Single Traces

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

abbreviation *ν -trace* $t \equiv \sum i. l \hat{^} i * r$ ($t !! i$)

lemma *abs- ν -trace-fin-le*: $|\nu$ -trace-fin t $N| \leq (\sum i < N. l \hat{^} i * r_M)$
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) ($\lambda t. \nu$ -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 $s = \int t. \nu$ -trace-fin t n $\partial \mathcal{T}$ p s

lemma *abs- ν -fin-le*: $|\nu$ -fin N $s| \leq (\sum i < N. l \hat{^} i * r_M)$
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*: $(\lambda s. \nu\text{-fin } N s) \in \text{bfun}$
by (*auto intro!*: *abs- ν -fin-le*)

lift-definition *ν_b -fin* :: $\text{nat} \Rightarrow 's \Rightarrow_b \text{real}$ **is** *ν -fin*
using *ν -fin-bfun* .

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

lemma *ν -fin-zero*[*simp*]: $\nu\text{-fin } 0 s = 0$
by (*simp add*: *ν -fin-def*)

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

7.4 Expected Total Discounted Reward

definition *ν p s* = *lim* ($\lambda n. \nu\text{-fin } p n s$)

lemmas *ν -eq-lim* = *ν -def*

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

7.5 Reward of a Decision Rule

context

fixes *d* :: $('s, 'a) \text{dec}$

begin

abbreviation *r-dec s* $\equiv \int a. r (s, a) \partial d s$

lemma *abs-r-dec-le*: $|r\text{-dec } s| \leq r_M$
using *expectation-abs-r-le integral-abs-bound order-trans* **by** *fast*

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

lemma *r-dec-bfun*: $r\text{-dec} \in \text{bfun}$
using *abs-r-dec-le* **by** (*auto intro!*: *bfun-normI*)

lift-definition *r-dec_b* :: $'s \Rightarrow_b \text{real}$ **is** *r-dec*
using *r-dec-bfun* .

declare $r\text{-dec}_b.\text{rep-eq}[simp]$ $\text{bfun.Bfun-inverse}[simp]$

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

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

7.6 Transition Probability Matrix for MDPs

context

fixes $p :: \text{nat} \Rightarrow ('s, 'a) \text{dec}$

begin

definition $\mathcal{P}_X n = \text{push-exp } (\lambda s. Xn' (\text{mk-markovian } p) s n)$

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

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

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

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

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

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

end

lemma $\nu\text{-fin-elem}$: $\nu\text{-fin } (\text{mk-markovian } p) n s = (\sum i < n. l\hat{i} * \mathcal{P}_X p i (r\text{-dec}_b (p i)) s)$
unfolding $\mathcal{P}_X\text{-def } \nu\text{-fin-eq-}Pn Pn'\text{-markovian-eq-}Xn'\text{-bind measure-pmf-bind}$
using $\text{measure-pmf-in-subprob-algebra abs-r-le-}r_M$
by ($\text{subst integral-bind}$) ($\text{auto simp: } r\text{-dec-eq-r-K}0$)

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

lemma $\nu\text{-fin-eq-}\mathcal{P}_X$: $\nu\text{-fin } (\text{mk-markovian } p) n = (\sum i < n. l\hat{i} *_R \mathcal{P}_X p i (r\text{-dec}_b (p i)))$
by ($\text{metis } \nu_b\text{-fin.rep-eq } \nu_b\text{-fin-eq-}\mathcal{P}_X$)

$\mathcal{P}_1 d v$ 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 $d :: ('s, 'a) \text{ dec}$

begin

lift-definition $\mathcal{P}_1 :: ('s \Rightarrow_b \text{ real}) \Rightarrow_L ('s \Rightarrow_b \text{ real})$ **is** *push-exp* ($K\text{-st } d$)

using *push-exp-bounded-linear* .

lemma $\mathcal{P}_1\text{-bfun-one}$ [*simp*]: $\mathcal{P}_1 1 = 1$

by (*auto simp: \mathcal{P}_1.rep-eq*)

lemma $\mathcal{P}_1\text{-pow-bfun-one}$ [*simp*]: $(\mathcal{P}_1 \sim t) 1 = 1$

by (*induction t auto*)

lemma $\mathcal{P}_1\text{-pow}$: *blinfun-apply* $(\mathcal{P}_1 \sim n) = \text{blinfun-apply } \mathcal{P}_1 \sim n$

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 $\mathcal{P}_X\text{-Suc}$: $\mathcal{P}_X p (\text{Suc } n) v = \mathcal{P}_1 (p 0) ((\mathcal{P}_X (\lambda n. p (\text{Suc } n)) n) v)$

unfolding $\mathcal{P}_X\text{-def } \mathcal{P}_1.rep-eq$

by (*fastforce intro!: abs-le-norm-bfun integral-bind[where $K = \text{count-space UNIV}$]*)

simp: measure-pmf-in-subprob-algebra measure-pmf-bind Suc-Xn'-markovian)

lemma $\mathcal{P}_X\text{-Suc}'$: $\mathcal{P}_X p (\text{Suc } n) v = \mathcal{P}_X p n (\mathcal{P}_1 (p n) v)$

proof (*induction n arbitrary: p*)

case 0

thus *?case*

by (*simp add: \mathcal{P}_X-Suc*)

next

case $(\text{Suc } n)$

thus *?case*

by (*metis \mathcal{P}_X-Suc*)

qed

lemma $\mathcal{P}_X\text{-const}$: $\mathcal{P}_X (\lambda-. d) n = \mathcal{P}_1 d \sim n$

by (*induction n (auto simp add: \mathcal{P}_1-pow \mathcal{P}_X-Suc)*)

lemma $\mathcal{P}_X\text{-sconst}$: $\mathcal{P}_X (\lambda-. p) n = \mathcal{P}_1 p \sim n$

using $\mathcal{P}_X\text{-const}$.

lemma *norm-P-n*[*simp*]: *onorm* $(\mathcal{P}_1 d \sim n) = 1$

using *norm-\mathcal{P}_X*[*of \lambda-. d*] **by** (*auto simp: \mathcal{P}_X-sconst*)

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

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

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

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

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

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

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

lemma *\mathcal{P}_1 -n-disc-pos*: $0 \leq u \implies 0 \leq (l \hat{\sim} n *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} n) u$
by (*auto simp: \mathcal{P}_1 -n-pos scaleR-nonneg-nonneg blinfun.scaleR-left*)

lemma *\mathcal{P}_1 -sum-pos*: $0 \leq u \implies 0 \leq (\sum t \leq n. l \hat{\sim} t *_{\mathcal{R}} (\mathcal{P}_1 d \hat{\sim} t)) u$
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 $0 \leq u$
shows $u \leq (\sum t \leq n. l \hat{\sim} t *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} t) u$
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 $L d v \equiv r\text{-dec}_b d + l *_{\mathcal{R}} \mathcal{P}_1 d v$

lemma *norm-L-le*: $\text{norm } (L d v) \leq r_M + l * \text{norm } v$
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*: $|L d v s| \leq r_M + l * \text{norm } v$
using *order.trans* [*OF* *norm-le-norm-bfun norm-L-le*] **by** *auto*

7.7.1 Bellman Operator for Single Actions

abbreviation $L_a a v s \equiv r (s, a) + l * \text{measure-pmf.expectation } (K (s, a)) v$

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

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

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

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

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

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

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

lemma L_a -le': $L_a a$ (apply-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 $d \implies L d v \leq \mathcal{L}_b v$
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:
 $\mathcal{L} v s = (\bigsqcup pa \in \{pa. set-pmf pa \subseteq A s\}. (\int a. L_a a v s \partial measure-pmf pa))$
unfolding \mathcal{L} -def
proof (intro antisym)
show $(\bigsqcup d \in D_R. L d v s) \leq (\bigsqcup pa \in \{pa. set-pmf pa \subseteq A s\}. \int a. L_a a v s \partial measure-pmf pa)$
proof (rule cSUP-mono)
show $D_R \neq \{\}$
using D_R -ne .
next show bdd-above (($\lambda pa. \int a. L_a a v s \partial measure-pmf pa$) ' $\{pa. set-pmf pa \subseteq A s\}$)
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 $\exists m \in \{pa. set-pmf pa \subseteq A s\}. L n v s \leq \int a. L_a a v s \partial measure-pmf m$ **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-pmf pa  $\subseteq$  A s}  $\neq$  {}
  using  $D_R$ -ne is-dec-def by auto
show ( $\bigsqcup_{pa \in \{pa. \text{set-pmf } pa \subseteq A s\}} \int a. L_a a v s \partial \text{measure-pmf } pa$ )  $\leq$  ( $\bigsqcup_{d \in D_R} L d v s$ )
proof (intro cSUP-least[OF aux] cSUP-upper2)
  fix n
  assume h:  $n \in \{pa. \text{set-pmf } pa \subseteq A s\}$ 
  let ?p = ( $\lambda s'. \text{if } s = s' \text{ then } n \text{ else SOME } a. \text{set-pmf } a \subseteq A s'$ )
  have aux:  $\exists a. \text{set-pmf } a \subseteq A sa$  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- $L_a$ :  $\mathcal{L}_b v s = (\bigsqcup_{p \in \{p. \text{set-pmf } p \subseteq A s\}} \int a. L_a a v s \partial \text{measure-pmf } p)$ 
  using SUP-step-MR-eq  $\mathcal{L}_b$ .rep-eq by presburger

```

```

lemma SUP-step-det-eq: ( $\bigsqcup_{d \in D_D} L (\text{mk-dec-det } d) v s$ ) = ( $\bigsqcup_{a \in A s} L_a a v s$ )
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 (( $\lambda d. L (\text{mk-dec-det } d) v s$ ) '  $D_D$ )
    by (auto intro!: bounded-imp-bdd-above boundedI abs-L-le)
  show  $\exists m \in A s. L (\text{mk-dec-det } m) v s \leq L_a m v s$  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  $\exists m \in D_D. L_a n v s \leq L (\text{mk-dec-det } m) v s$  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 -  $\lambda s'. \text{if } s = s' \text{ then } - \text{ else SOME } a. a \in A s'$ ])
qed (auto simp: A-ne)

```

```

lemma integrable- $L_a$ : integrable (measure-pmf x) ( $\lambda a. L_a a$  (apply-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) ($\lambda x. \text{measure-pmf.expectation } (K (s, x)) v$)
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 $v :: 's \Rightarrow_b \text{real}$
shows $(\bigsqcup p \in \{p. \text{set-pmf } p \subseteq A \ s\}. \int a. L_a \ a \ v \ s \ \partial \text{measure-pmf } p) =$
 $(\bigsqcup a \in A \ s. L_a \ a \ v \ s)$
proof (*intro antisym*)
show $(\bigsqcup pa \in \{pa. \text{set-pmf } pa \subseteq A \ s\}. \text{measure-pmf.expectation } pa$
 $(\lambda a. L_a \ a \ v \ s))$
 $\leq (\bigsqcup a \in A \ s. L_a \ a \ v \ s)$
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 $(\bigsqcup a \in A \ s. L_a \ a \ v \ s) \leq (\bigsqcup p \in \{p. \text{set-pmf } p \subseteq A \ s\}. \int a. L_a \ a$
 $v \ s \ \partial \text{measure-pmf } p)$
unfolding *SUP-step-MR-eq[symmetric] SUP-step-det-eq[symmetric]*
L-def
using *ex-dec-det by (fastforce intro!: cSUP-mono)*
qed

lemma *L-eq-SUP-det*: $\mathcal{L} \ v \ s = (\bigsqcup d \in D_D. L \ (mk\text{-dec-det } d) \ v \ s)$
using *SUP-step-MR-eq SUP-step-det-eq SUP-L_a-eq-det by auto*

lemma *L_b-eq-SUP-det*: $\mathcal{L}_b \ v \ s = (\bigsqcup d \in D_D. L \ (mk\text{-dec-det } d) \ v \ s)$
using *L-eq-SUP-det unfolding L_b.rep-eq by auto*

7.9 Monotonicity

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

lemma *P₁-mono[intro]*: $a \leq b \implies \mathcal{P}_1 \ p \ a \leq \mathcal{P}_1 \ p \ b$
using *P₁-nonneg by auto*

lemma *L-mono[intro]*: $u \leq v \implies L \ d \ u \leq L \ d \ v$
unfolding *L-def by (auto intro: scaleR-left-mono)*

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

lemma *step-mono*:
assumes $\mathcal{L}_b \ v \leq v \ d \in D_R$
shows $L \ d \ v \leq v$

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 $\exists d \in D_D. v \leq L (mk\text{-dec-det } d) v + e *_R 1$

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* $d v s \leq L (mk\text{-dec-det } d) v s + e$

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 $\mathcal{L}_b v \leq v p \in \Pi_{MR}$

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* $v s = is\text{-arg-max } (\lambda a. L_a a v s) (\lambda a. a \in A s)$

abbreviation *opt-acts* $v s \equiv \{a. is\text{-opt-act } v s a\}$

lemma *summable-disc* [*intro*, *simp*]: *summable* $(\lambda i. l \wedge i * x)$

by (*simp add: mult commute*)

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

$\text{summable } (\lambda i. |l \hat{^} i * r (sa i)|)$
 $\text{summable } (\lambda i. l \hat{^} i * |r (sa i)|)$
 $\text{summable } (\lambda i. l \hat{^} i * r (sa i))$
proof –
show $\text{summable } (\lambda i. |l \hat{^} i * r (sa i)|)$
using *abs-r-le-r_M*
by (*fastforce intro!*: *mult-left-mono summable-comparison-test'*[*OF summable-disc*])
thus $\text{summable } (\lambda i. l \hat{^} i * r (sa i)) \text{ summable } (\lambda i. l \hat{^} i * |r (sa i)|)$
by (*auto intro: summable-rabs-cancel*)
qed

lemma *summable-norm-disc-I*[*intro*]:
assumes $\text{summable } (\lambda t. (l \hat{^} t * \text{norm } F))$
shows $\text{summable } (\lambda t. \text{norm } (l \hat{^} t *_{R} F))$
using *assms* **by** *auto*

lemma *summable-norm-disc-I'*[*intro*]:
assumes $\text{summable } (\lambda t. (l \hat{^} t * \text{norm } (F t)))$
shows $\text{summable } (\lambda t. \text{norm } (l \hat{^} t *_{R} F t))$
using *assms* **by** *auto*

lemma *summable-discI* [*intro*]:
assumes *bounded (range F)*
shows $\text{summable } (\lambda t. l \hat{^} t * \text{norm } (F t))$
proof –
obtain *b* **where** $\text{norm } (F x) \leq b$ **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 $\text{summable } (\lambda t. l \hat{^} t *_{R} (F t))$
using *assms* **by** (*auto intro: summable-norm-cancel*)

lemma *summable-norm-bfun-disc*: $\text{summable } (\lambda t. l \hat{^} t * \text{norm } (\text{apply-bfun } f t))$
using *norm-le-norm-bfun*
by (*auto simp: mult.commute*[*of l^-*] *intro!*: *Abel-lemma*[*of - 1 - norm f*])

lemma *summable-bfun-disc* [*simp*]: $\text{summable } (\lambda t. l \hat{^} t * (\text{apply-bfun } f t))$
proof –
have $\text{norm } (l \hat{^} t * \text{apply-bfun } f t) = l \hat{^} t * \text{norm } (\text{apply-bfun } f t)$ **for** *t*
by (*auto simp: abs-mult*)
hence $\text{summable } (\lambda t. \text{norm } (l \hat{^} t * (\text{apply-bfun } f t)))$

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

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

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

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

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

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

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

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

context
 fixes $p :: ('s, 'a) \text{pol}$
begin

```

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

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

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

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

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

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

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

```

7.10 Optimal Reward

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

proof (*rule antisym*)

show ν_b -opt $s \leq (\bigsqcup p \in \Pi_{MR}. \nu_b$ (*mk-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-markovian* p) s) ' Π_{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-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* ($\lambda t. l^{\wedge}t * norm$
 $(\mathcal{P}_X p t (r\text{-dec}_b (p t)))$)
using *\mathcal{P}_X -bounded-r* **by** *auto*

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

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

lemma *ν -eq- \mathcal{P}_X* : $\nu (mk\text{-markovian } p) = (\sum i. l^{\wedge}i *_R \mathcal{P}_X p i (r\text{-dec}_b$
 $(p i)))$

proof –

have $\nu (mk\text{-markovian } p) s = (\sum i. l^{\wedge}i *_R \mathcal{P}_X p i (r\text{-dec}_b (p i))) s$
for s

unfolding *ν_b .rep-eq \mathcal{P}_X -def ν -eq- \mathcal{P}_n \mathcal{P}_n' -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* : $\nu_b (mk\text{-markovian } p) = (\sum i. l^{\wedge}i *_R \mathcal{P}_X p i (r\text{-dec}_b$
 $(p i)))$

by (*auto simp: ν -eq- \mathcal{P}_X ν_b .rep-eq*)

lemma *ν_b -fin-tendsto- ν_b* : ($\nu_b\text{-fin } (mk\text{-markovian } p)$) $\longrightarrow \nu_b (mk\text{-markovian}$
 $p)$

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*: ($\lambda n. (\sum t \leq n. l^{\wedge}t *_R \mathcal{P}_1 d \text{ } \sim t)$) $\longrightarrow (\sum t.$
 $l^{\wedge}t *_R \mathcal{P}_1 d \text{ } \sim t)$

by (*fastforce simp: bounded-iff intro: summable-LIMSEQ'*)

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

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

lemma *\mathcal{P}_1 -suminf-ge*:
assumes $0 \leq u$ **shows** $u \leq (\sum t. l \hat{\sim} t *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} t) u$
proof –
have *aux*: $\bigwedge x. (\lambda n. (\sum t \leq n. l \hat{\sim} t *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} t) u x) \longrightarrow (\sum t. l \hat{\sim} t *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} t) u x$
using *bfun-tendsto-apply-bfun disc- \mathcal{P}_1 -lim lim-blinfun-apply[OF convergent-disc- \mathcal{P}_1]*
by *fastforce*
have $\bigwedge n. u \leq (\sum t \leq n. l \hat{\sim} t *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} t) u$
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 $0 \leq (\sum t. l \hat{\sim} t *_{\mathcal{R}} \mathcal{P}_1 d \hat{\sim} t) u$
using *\mathcal{P}_1 -suminf-ge[of u] assms order.trans* **by** *auto*

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

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

qed

lemma ν -stationary-inv: $\nu_b (mk\text{-stationary } d) = inv_L (id\text{-blinfun} - l *_R \mathcal{P}_1 d) (r\text{-dec}_b d)$
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: $\nu_b (mk\text{-markovian } p) = L (p 0) (\nu_b (mk\text{-markovian } (\lambda n. p (Suc n))))$

proof –

have s : summable ($\lambda t. l \hat{=} t *_R (\mathcal{P}_X p (Suc t) (r\text{-dec}_b (p (Suc t))))$)
using \mathcal{P}_X -bound-r **by** (auto intro!: boundedI[of - r_M])
have
 $\nu_b (mk\text{-markovian } p) = r\text{-dec}_b (p 0) + (\sum t. l \hat{=} (Suc t) *_R \mathcal{P}_X p (Suc t) (r\text{-dec}_b (p (Suc t))))$
by (subst suminf-split-head) (auto simp: ν_b -eq- \mathcal{P}_X)
also have
 $\dots = r\text{-dec}_b (p 0) + l *_R (\sum t. \mathcal{P}_1 (p 0) (l \hat{=} t *_R \mathcal{P}_X (\lambda n. p (Suc n)) t (r\text{-dec}_b (p (Suc t))))$
using suminf-scaleR-right[OF s] **by** (auto simp: \mathcal{P}_X -Suc blinfun.scaleR-right)
also have
 $\dots = L (p 0) (\nu_b (mk\text{-markovian } (\lambda n. p (Suc n))))$
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: $\nu_b (mk\text{-stationary } d) = L d (\nu_b (mk\text{-stationary } d))$
using ν -step .

lemma L -fix- ν :

assumes $L p v = v$

shows $v = \nu_b (mk\text{-stationary } p)$

proof –

have $r\text{-dec}_b p = (id\text{-blinfun} - l *_R \mathcal{P}_1 p) v$
using assms **by** (auto simp: eq-diff-eq L-def blinfun.diff-left blinfun.scaleR-left)
hence $v = (\sum t. (l *_R \mathcal{P}_1 p) \hat{=} t) (r\text{-dec}_b p)$
using inv-norm-le'(2)[OF norm- \mathcal{P}_1 -l-less] **by** auto
thus $v = \nu_b (mk\text{-stationary } p)$
by (auto simp: ν -stationary blincomp-scaleR-right)

qed

lemma L - ν -fix-iff: $L d v = v \iff v = \nu_b (mk\text{-stationary } d)$
using L -fix- ν L - ν -fix **by** auto

7.11 Properties of Solutions of the Optimality Equations

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

lemma $\mathcal{P}_d\text{-lim}$: $(\lambda n. (\mathcal{P}_d p n v)) \longrightarrow 0$

proof –

have $(\lambda n. l^{\wedge n} * \text{norm } v) \longrightarrow 0$

by $(\text{auto intro!}; \text{tendsto-eq-intros})$

moreover have $\text{norm } (\mathcal{P}_d p n v) \leq l^{\wedge n} * \text{norm } v$ **for** $p n$

by $(\text{simp add}; \text{mult-mono}')$

ultimately have $(\lambda n. \text{norm } (\mathcal{P}_d p n v)) \longrightarrow 0$ **for** p

by $(\text{auto simp}; \text{Lim-transform-bound}[\text{where } g = \lambda n. (l^{\wedge n} * \text{norm } v)])$

thus $(\lambda n. (\mathcal{P}_d p n v)) \longrightarrow 0$ **for** p

using $\text{tendsto-norm-zero-cancel}$ **by** fast

qed

lemma $\mathcal{L}\text{-dec-ge-opt}$:

assumes $\mathcal{L}_b v \leq v$

shows $\nu_b\text{-opt} \leq v$

proof –

have $\nu_b (\text{mk-markovian } p) \leq v$ **if** $p \in \Pi_{MR}$ **for** p

proof –

let $?p = \text{mk-markovian } p$

have $\text{aux}: \nu_b\text{-fin } ?p n + l^{\wedge n} *_R \mathcal{P}_X p n v \leq v$ **for** n

proof $(\text{induction } n)$

case $(\text{Suc } n)$

have $\mathcal{P}_X p n (\text{r-dec}_b (p n)) + l *_R (\mathcal{P}_X p (\text{Suc } n) v) \leq \mathcal{P}_X p n v$

using $\mathcal{P}_X\text{-L-le assms that}$ **by** $(\text{simp add}; \mathcal{P}_X\text{-Suc-n-elem L-def linear-simps})$

hence $\nu_b\text{-fin } ?p (n + 1) + l^{\wedge(n + 1)} *_R (\mathcal{P}_X p (n + 1) v) \leq \nu_b\text{-fin } ?p n + l^{\wedge n} *_R (\mathcal{P}_X p n v)$

by $(\text{auto simp del}; \text{scaleR-scaleR intro}; \text{scaleR-left-mono simp}; \nu_b\text{-fin-eq-}\mathcal{P}_X)$

$\text{mult.commute}[of l] \text{scaleR-add-right}[\text{symmetric}] \text{scaleR-scaleR}[\text{symmetric}]$

also have $\dots \leq v$

using Suc.IH **by** $(\text{auto simp}; \nu_b\text{-fin-eq-}\mathcal{P}_X)$

finally show $?case$

by auto

qed $(\text{auto simp}; \nu_b\text{-fin-eq-}\mathcal{P}_X)$

have $1: (\lambda n. (\nu_b\text{-fin } ?p n + \mathcal{P}_d p n v) s) \longrightarrow \nu_b ?p s$ **for** s

using $\text{bfun-tendsto-apply-bfun Limits.tendsto-add}[OF \nu_b\text{-fin-tendsto-}\nu_b \mathcal{P}_d\text{-lim}]$ **by** fastforce

have $\nu_b ?p s \leq v$ **for** s

using that aux assms **by** $(\text{fastforce intro!}; \text{lim-mono}[OF - 1, of -$

```

-  $\lambda n. v s$ )
  thus ?thesis
  using that by blast
qed
thus ?thesis
  using policies-ne by (fastforce simp: is-policy-def  $\nu_b$ -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  $\langle e > 0 \rangle$  by blast
    let ?Pinf =  $(\sum i. l^i *_R \mathcal{P}_1 d^{\wedge} i)$ 
    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: blin-
comp-scaleR-right)
    also have  $\dots = \nu_b (mk$ -stationary  $d) + e *_R ?Pinf 1$ 
      by (simp add:  $\nu$ -stationary blinfun.add-right blinfun.scaleR-right)
    also have  $\dots = \nu_b (mk$ -stationary  $d) + e *_R (\sum i. (l^i *_R ((\mathcal{P}_1$ 
 $d^{\wedge} i))) 1)$ 
      using convergent-disc- $\mathcal{P}_1$ 
      by (auto simp: summable-iff-convergent' bounded-linear.suminf[of
 $\lambda x. blinfun$ -apply  $x 1$ ])
    also have  $\dots = \nu_b (mk$ -stationary  $d) + e *_R (\sum i. (l^i *_R 1))$ 
      by (auto simp: scaleR-blinfun.rep-eq)
    also have  $\dots \leq (\nu_b (mk$ -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  $\langle d \in D_R \rangle \nu_b$ -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  $v s \leq \nu_b\text{-opt } s + e$ 
    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  $v = \nu_b\text{-opt}$ 
  using assms dual-order.antisym[OF  $\mathcal{L}$ -dec-ge-opt  $\mathcal{L}$ -inc-le-opt] by
auto

```

```

lemma bounded-P: bounded ( $\mathcal{P}_1 \text{ ' } X$ )
  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- $\mathcal{L}$ :  $\text{dist } (\mathcal{L}_b v) (\mathcal{L}_b u) \leq l * \text{dist } v u$ 

```

```

proof –

```

```

  have  $\text{dist } (\mathcal{L}_b v s) (\mathcal{L}_b u s) \leq l * \text{dist } v u$  if  $\mathcal{L}_b u s \leq \mathcal{L}_b v s$  for  $s v$ 
  u

```

```

  proof –

```

```

    have  $\text{dist } (\mathcal{L}_b v s) (\mathcal{L}_b u s) \leq (\bigsqcup d \in D_R. L d v s - L d u s)$ 

```

```

    using ex-dec that by (fastforce intro!: le-SUP-diff' simp: dist-real-def

```

```

 $\mathcal{L}_b$ .rep-eq  $\mathcal{L}$ -def)

```

```

    also have  $\dots = (\bigsqcup d \in D_R. l * (\mathcal{P}_1 d (v - u) s))$ 

```

```

    by (auto simp: L-def right-diff-distrib blinfun.diff-right)

```

```

    also have  $\dots = l * (\bigsqcup d \in D_R. \mathcal{P}_1 d (v - u) s)$ 

```

```

    using  $D_R$ -ne bounded-P by (fastforce intro: bounded-SUP-mul)

```

```

    also have  $\dots \leq l * \text{norm } (\bigsqcup d \in D_R. \mathcal{P}_1 d (v - u) s)$ 

```

```

    by (simp add: mult-left-mono)

```

```

    also have  $\dots \leq l * (\bigsqcup d \in D_R. \text{norm } ((\mathcal{P}_1 d (v - u)) s))$ 

```

```

    proof –

```

```

      have  $\text{bounded } ((\lambda x. \text{norm } ((\mathcal{P}_1 x (v - u)) s)) \text{ ' } D_R)$ 

```

```

      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  $\dots \leq l * (\bigsqcup p \in D_R. \text{norm } (\mathcal{P}_1 p ((v - u))))$ 

```

```

    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 $\dots \leq l * (\bigsqcup p \in D_R. \text{norm } ((v - u)))$
using *norm-push-exp-le-norm D_R-ne*
by (*fastforce simp: P₁.rep-eq intro!: mult-left-mono cSUP-mono*)
also have $\dots = l * \text{dist } v \ u$
by (*auto simp: dist-norm*)
finally show *?thesis* .
qed
hence $\mathcal{L}_b \ u \ s \leq \mathcal{L}_b \ v \ s \implies \text{dist } (\mathcal{L}_b \ v \ s) \ (\mathcal{L}_b \ u \ s) \leq l * \text{dist } v \ u$
 $\mathcal{L}_b \ v \ s \leq \mathcal{L}_b \ u \ s \implies \text{dist } (\mathcal{L}_b \ v \ s) \ (\mathcal{L}_b \ u \ s) \leq l * \text{dist } v \ u$ **for** $u \ v \ s$
by (*fastforce simp: dist-commute*)
thus *?thesis*
using *linear[of L_b u -]* **by** (*fastforce intro: dist-bound*)
qed

lemma *is-contraction-L: is-contraction L_b*
using *contraction-L zero-le-disc disc-lt-one unfolding is-contraction-def*
by *blast*

lemma *contraction-L: dist (L p v) (L p u) ≤ l * dist v u*

proof –

have *aux: L p v s – L p u s ≤ l * dist v u if lea: L p v s ≥ 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 $\dots \leq l * \text{norm } (\mathcal{P}_1 \ p \ (v - u) \ s)$

by (*auto simp: blinfun.diff-right intro!: mult-left-mono*)

also have $\dots \leq l * \text{norm } (\mathcal{P}_1 \ p \ (v - u))$

using *abs-le-norm-bfun* **by** (*auto intro!: mult-left-mono*)

also have $\dots \leq l * \text{dist } v \ u$

by (*simp add: P₁.rep-eq mult-left-mono norm-push-exp-le-norm dist-norm*)

finally show *?thesis*

by *auto*

qed

have $\text{dist } (L \ p \ v \ s) \ (L \ p \ u \ s) \leq l * \text{dist } v \ u$ **for** $v \ s \ u$

using *aux[of v - u] aux[of u - v]*

by (*cases L p v s ≥ L p u s*) (*auto simp: dist-real-def dist-commute*)

thus $\text{dist } (L \ p \ v) \ (L \ p \ u) \leq l * \text{dist } v \ u$

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:

$\exists! v. \mathcal{L}_b v = v$ ($\lambda n. (\mathcal{L}_b \widehat{\sim} n) v$) \longrightarrow (*THE* $v. \mathcal{L}_b v = v$)
using *banach'*[*OF is-contraction-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-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 = ν_b -opt

by *auto*

lemma \mathcal{L}_b -lim: ($\lambda n. (\mathcal{L}_b \widehat{\sim} n) v$) $\longrightarrow \nu_b$ -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 $v d \longleftrightarrow (\forall s. \text{is-arg-max } (\lambda d. (L d v) s) (\lambda d. d \in D_R) d)$

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 $(\nu_b\text{-opt}) d$

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

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

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

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

lemma ν -conserving-alt':
assumes $d \in D_R$
shows ν -conserving $d \longleftrightarrow L d \nu_b\text{-opt} = \nu_b\text{-opt}$
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) = $\nu_b\text{-opt}$
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 $\exists d \in D_D. (\nu_b$ (*mk-stationary-det* d)) = $\nu_b\text{-opt}$
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** $d: \nu\text{-conserving (mk-dec-det } d)$
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-eq-L_a*
using *assms is-dec-def L_a-bounded A-ne \mathcal{L}_b .rep-eq \mathcal{L} -def*
by (*intro lemma-4-3-1'*)
(auto intro: bounded-range-subset simp: assms(2)[symmetric]
L-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-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* ($\lambda n.$ *as-markovian* p (*return-pmf* s) (*Suc* n))) $\leq \nu_b$ -*opt*
using *is- Π_{MR} -as-markovian* *assms* **by** (*auto simp: is-policy-def mk-markovian-def*)
have *aux-le*: $\bigwedge x f g. f \leq g \implies \text{apply-bfun } f \ x \leq \text{apply-bfun } g \ x$
unfolding *less-eq-bfun-def* **by** *auto*
have ν_b -*opt* $s = \nu_b$ (*mk-markovian* $?ps$) s
using *assms* ν_b -*as-markovian* **by** *metis*
also have $\dots = L$ ($?ps$ 0) (ν_b (*mk-markovian* ($\lambda n.$ $?ps$ (*Suc* n))))
 s
using ν -*step* **by** *blast*
also have $\dots \leq L$ ($?ps$ 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- Π_{MR} -as-markovian* *assms* **by** *fast*
have L ($?ps$ 0) ν_b -*opt* $\leq \nu_b$ -*opt*
using $\langle ?ps$ $0 \in D_R \rangle$ *L-le- \mathcal{L}_b [of ?ps 0 ν_b -opt]* **by** *simp*
thus L ($?ps$ 0) ν_b -*opt* $s = \nu_b$ -*opt* s
using $\langle \nu_b$ -*opt* $s \leq (L$ ($?ps$ 0) ν_b -*opt*) $s \rangle$ **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* $= \nu_b$ -*opt*
using *aux* **by** *auto*
hence $\exists d \in D_D. L$ (*mk-dec-det* d) ν_b -*opt* $= \nu_b$ -*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 $\bigwedge s. \text{finite}$ (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 $\bigwedge s. \text{finite}$ (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 - $\langle \lambda s. \text{SOME } a. a \in A$ $s \wedge L_a$ 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 $\bigwedge s. \text{finite } (A\ s)$
shows $\exists d \in D_D. \nu_b\text{-opt} = \nu_b\ (\text{mk-stationary-det } d)$
using *assms conserving-imp-opt' L_b-opt L- ν -fix-iff ex-opt-det-det*
by *metis*

7.13.4 Existence of Epsilon-Optimal Policies

lemma *ex-det-eps*:

assumes $0 < e$
shows $\exists d \in D_D. \mathcal{L}_b\ v \leq L\ (\text{mk-dec-det } d)\ v + e\ *_R\ 1$
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* ($(\lambda a. L_a\ a\ v\ s) \text{ ' } A\ s$)
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 *L_b.rep-eq L-eq-SUP-det SUP-step-det-eq*
by (*auto simp: less-cSUP-iff[OF A-ne, symmetric]* $\langle 0 < e \rangle$)
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 P₁.rep-eq bind-return-pmf K-st-def less-eq-bfun-def*)
metis
qed

lemma *thm-6-2-11*:

assumes $\text{eps} > 0$
shows $\exists d \in D_D. \nu_b\text{-opt} \leq \nu_b\ (\text{mk-stationary-det } d) + \text{eps}\ *_R\ 1$
proof –
have $(1-l) * \text{eps} > 0$
by (*simp add: assms*)
then obtain d **where** $d \in D_D$ **and** $d: \mathcal{L}_b\ \nu_b\text{-opt} \leq L\ (\text{mk-dec-det } d)\ \nu_b\text{-opt} + ((1-l)*\text{eps})\ *_R\ 1$
using *ex-det-eps[of - $\nu_b\text{-opt}$]* **by** *auto*
let $?d = \text{mk-dec-det } d$
let $?lK = l *_R\ P_1\ ?d$
let $?lK\text{-opt} = l *_R\ P_1\ ?d\ \nu_b\text{-opt}$
have $\nu_b\text{-opt} \leq r\text{-dec}_b\ ?d + ?lK\text{-opt} + ((1-l)*\text{eps})\ *_R\ 1$
using *L-def L-fix-imp-opt d* **by** *simp*
hence $\nu_b\text{-opt} - ?lK\text{-opt} - ((1-l)*\text{eps})\ *_R\ 1 \leq r\text{-dec}_b\ ?d$

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

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

using *h2* **by** (*simp add: add commute diff-le-eq*)
ultimately have $\forall s. |(\mathcal{L}_b v) s - L (mk\text{-dec-det } d) v s| \leq e$
unfolding *less-eq-bfun-def* **by** *auto*
hence $dist (\mathcal{L}_b v) (L (mk\text{-dec-det } d) v) \leq e$
unfolding *dist-bfun.rep-eq* **by** (*auto intro!: cSUP-least simp: dist-real-def*)
thus *?thesis*
using $\langle d \in D_D \rangle$
by *auto*
qed

lemma *less-imp-ex-add-le*: $(x :: real) < y \implies \exists eps > 0. x + eps \leq y$
by (*meson field-le-epsilon less-le-not-le nle-le*)

lemma *ν_b -opt-le-det*: $\nu_b\text{-opt } s \leq (\bigsqcup d \in D_D. \nu_b (mk\text{-stationary-det } d) s)$

proof (*subst le-cSUP-iff, safe*)
fix *y*
assume $y < \nu_b\text{-opt } s$
then obtain *eps where 1: $y \leq \nu_b\text{-opt } s - eps$ and $eps > 0$*
using *less-imp-ex-add-le* **by** *force*
hence $eps / 2 > 0$ **by** *auto*
obtain *d where $d \in D_D$ and $\nu_b\text{-opt } s \leq \nu_b (mk\text{-stationary-det } d) s + eps / 2$*
using *thm-6-2-11[OF $\langle eps / 2 > 0 \rangle$]* **by** *fastforce*
hence $y < \nu_b (mk\text{-stationary-det } d) s$
using $\langle eps > 0 \rangle$ **by** (*auto simp: diff-less-eq intro: le-less-trans[OF 1]*)
thus $\exists i \in D_D. y < \nu_b (mk\text{-stationary-det } i) s$
using $\langle d \in D_D \rangle$ **by** *blast*
next
show $D_D = \{\} \implies False$
using *D-det-ne* **by** *blast*
show *bdd-above* ($\lambda d. \nu_b (mk\text{-stationary-det } d) s$) ‘*D_D*)
by (*auto intro!: bounded-imp-bdd-above boundedI abs- ν -le simp: $\nu_b.rep-eq$*)
qed

lemma *ν_b -opt-eq-det*: $\nu_b\text{-opt } s = (\bigsqcup d \in D_D. \nu_b (mk\text{-stationary-det } d) s)$
using *ν_b -le-opt-DD D-det-ne*
by (*fastforce intro!: antisym[OF $\nu_b\text{-opt-le-det}$] cSUP-least*)

lemma *lemma-6-3-1-a*:

assumes $v0 \in bfun$
shows *uniform-limit UNIV* ($\lambda n. ((\lambda v. \mathcal{L} (Bfun v)) \overset{\sim}{\sim} n) v0$) *ν -opt sequentially*
proof –
have *\mathcal{L} -Bfun-eq*: $v0 \in bfun \implies ((\lambda v. \mathcal{L} (Bfun v)) \overset{\sim}{\sim} n) v0 = (\mathcal{L}_b$

\widetilde{n}) (Bfun v0) for n
 by (induction n) (auto simp: \mathcal{L}_b .rep-eq apply-bfun-inverse)
 have uniform-limit UNIV ($\lambda n. (\mathcal{L}_b \widetilde{n})$) (Bfun v0) ν_b -opt sequentially
 by (intro tendsto-bfun-uniform-limit[OF \mathcal{L}_b -lim])
 hence uniform-limit UNIV ($\lambda n. (\mathcal{L}_b \widetilde{n})$) (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 ($\lambda n. f n$) \longrightarrow ($y::\text{real-normed-vector}$)
 shows ($\lambda n. \text{dist } (f n) (f (\text{Suc } n))$) $\longrightarrow 0$
 using assms tendsto-diff tendsto-norm LIMSEQ-Suc by (fastforce simp: dist-norm)

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

definition *max-L-ex s v* \equiv has-arg-max ($\lambda a. L_a a v s$) (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 $\mathcal{P}_1 d$) \widetilde{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 = ($\sum i < n. ((l *_R \mathcal{P}_1 d) \widetilde{i})$) (r-dec_b d)
 by (induction n) (auto simp add: plus-blinfun.rep-eq)

lemma *L-iter*: (L d \widetilde{m}) v = ν_b -fin (mk-stationary d) m + ((l *_R $\mathcal{P}_1 d$) \widetilde{m}) v

proof (induction m arbitrary: v)

case (Suc m)

have (L d $\widetilde{\text{Suc } m}$) v = (L d \widetilde{m}) (L d v)

by (simp add: funpow-Suc-right del: funpow.simps)

also have ... = ν_b -fin (mk-stationary d) m + ((l *_R $\mathcal{P}_1 d$) \widetilde{m}) (L d v)

using Suc by simp

also have ... = ν_b -fin (mk-stationary d) (Suc m) + ((l *_R $\mathcal{P}_1 d$) $\widetilde{\text{Suc } m}$) 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 (($\lambda x. (\nu_b$ -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 (($\lambda x. (((l *_{\mathcal{R}} \mathcal{P}_1 x) \widehat{\sim} m) v) s$) ' X)

by (auto simp: \mathcal{P}_X -const[symmetric] blinfun.bilinear-simps blincomp-scaleR-right

intro!: boundedI[of- $l \widehat{\sim} m * norm v$] mult-left-mono order.trans[OF abs-le-norm-bfun])

lemma bounded-disc- \mathcal{P}_1' : bounded (($\lambda x. ((\mathcal{P}_1 x \widehat{\sim} m) v) s$) ' 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 $d \implies (L d \widehat{\sim} n) v \leq (\mathcal{L}_b \widehat{\sim} n) v$

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 $\bigwedge s. finite (A s)$

locale MDP-att- $\mathcal{L} =$ MDP-reward-disc $A K r l$

for

A and

$K :: 's :: countable \times 'a :: countable \Rightarrow 's$ pmf and

r and l +

assumes Sup-att: max-L-ex ($s :: 's$) v

begin

theorem \mathcal{L}_b -eq-argmax- L_a :

fixes $v :: 's \Rightarrow_b real$

assumes is-arg-max ($\lambda a. L_a a v s$) ($\lambda a. a \in A s$) 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: $a \in A s \implies L_a a v s \leq L_a (arg-max-on (\lambda a. L_a a v s) (A s)) v s$

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*: $has\text{-}arg\text{-}max\ f\ Q \implies arg\text{-}max\text{-}on\ f\ Q \in Q$
using *has-arg-max-arg-max* **by** (*auto simp: arg-max-on-def*)

lemma $\mathcal{L}_b\text{-}eq\text{-}L_a\text{-}max$: $\mathcal{L}_b\ v\ s = L_a\ (arg\text{-}max\text{-}on\ (\lambda a. L_a\ a\ v\ s)\ (A\ s))$
v s
using *app-arg-max-eq-SUP[symmetric] Sup-att max-L-ex-def*
by (*auto simp: \mathcal{L}_b\text{-}eq\text{-}SUP\text{-}det SUP\text{-}step\text{-}det\text{-}eq*)

lemma *ex-opt-det*: $\exists d \in D_D. \mathcal{L}_b\ v = L\ (mk\text{-}dec\text{-}det\ d)\ v$
proof –
define *d* **where** $d = (\lambda s. arg\text{-}max\text{-}on\ (\lambda a. L_a\ a\ v\ s)\ (A\ s))$
have $\mathcal{L}_b\ v\ s = L\ (mk\text{-}dec\text{-}det\ d)\ v\ s$ **for** *s*
by (*auto simp: d-def \mathcal{L}_b\text{-}eq\text{-}L_a\text{-}max L\text{-}eq\text{-}L_a\text{-}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*: $\exists d \in D_D. \nu\text{-improving}\ v\ (mk\text{-}dec\text{-}det\ d)$
using *\nu-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]*: $X \neq \{\} \implies arb\text{-}act\ X \in X$

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*: $is\text{-}opt\text{-}act\ v\ s\ (arb\text{-}act\ (opt\text{-}acts\ v\ s))$
using *arb-act-in[of \{a. is-arg-max (\lambda a. L_a a v s) (\lambda 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*: $arb\text{-}act\ (opt\text{-}acts\ v\ s) \in A\ s$
using *is-opt-act-some* **unfolding** *is-opt-act-def is-arg-max-def* **by**
auto

lemma *\nu-improving-opt-acts*: $\nu\text{-improving}\ v\ 0\ (mk\text{-}dec\text{-}det\ (\lambda s. arb\text{-}act$
*(opt-acts (apply-bfun v0) s)))
using *is-opt-act-def is-opt-act-some some-opt-acts-in-A*
by (*subst \nu-improving-alt*) (*fastforce simp: L\text{-}eq\text{-}L_a\text{-}det \mathcal{L}_b\text{-}eq\text{-}argmax\text{-}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.