Verified Algorithms for Solving Markov Decision Processes

Maximilian Schäffeler and Mohammad Abdulaziz

December 28, 2021

Abstract

We present a formalization of algorithms for solving Markov Decision Processes (MDPs) with formal guarantees on the optimality of their solutions. In particular we build on our analysis of the Bellman operator for discounted infinite horizon MDPs. From the iterator rule on the Bellman operator we directly derive executable value iteration and policy iteration algorithms to iteratively solve finite MDPs. We also prove correct optimized versions of value iteration that use matrix splittings to improve the convergence rate. In particular, we formally verify Gauss-Seidel value iteration and modified policy iteration. The algorithms are evaluated on two standard examples from the literature, namely, inventory management and gridworld. Our formalization covers most of chapter 6 in Puterman's book [1].

Contents

theory *Value-Iteration*
  imports *MDP−Rewards.MDP-reward*
begin

context *MDP-att-$\mathcal{L}$*
begin

1 Value Iteration

In the previous sections we derived that repeated application of $\mathcal{L}_b$ to any bounded function from states to the reals converges to the optimal value of the MDP ν_b-*opt*.

We can turn this procedure into an algorithm that computes not only an approximation of ν_b-*opt* but also a policy that is arbitrarily close to optimal.

Most of the proofs rely on the assumption that the supremum in $\mathcal{L}_b$ can always be attained.

The following lemma shows that the relation we use to prove termination of the value iteration algorithm decreases in each step. In essence, the distance of the estimate to the optimal value decreases by a factor of at least l per iteration.

lemma *vi-rel-dec*:
  assumes $l \neq 0$ $\mathcal{L}_b$ $v \neq \nu_b$-*opt*
  shows $\lceil log\ (1\ /\ l)\ (dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt) - c\rceil < \lceil log\ (1\ /\ l)\ (dist\ v\ \nu_b\text{-}opt) - c\rceil$
proof –
  have $log\ (1\ /\ l)\ (dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt) - c \leq log\ (1\ /\ l)\ (l * dist\ v\ \nu_b\text{-}opt) - c$
    using *contraction-*$\mathcal{L}$[*of - ν_b-opt*] *disc-lt-one*
    by (*auto simp*: *assms less-le intro*: *log-le*)
  also have $\ldots = log\ (1\ /\ l)\ l + log\ (1/l)\ (dist\ v\ \nu_b\text{-}opt) - c$
    using *assms disc-lt-one*
    by (*auto simp*: *less-le intro*!: *log-mult*)
  also have $\ldots = -(log\ (1\ /\ l)\ (1/l)) + (log\ (1/l)\ (dist\ v\ \nu_b\text{-}opt)) - c$
    using *assms disc-lt-one*
   by (*subst log-inverse*[*symmetric*]) (*auto simp*: *less-le right-inverse-eq*)
  also have $\ldots = (log\ (1/l)\ (dist\ v\ \nu_b\text{-}opt)) - 1 - c$
    using *assms order.strict-implies-not-eq*[*OF disc-lt-one*]
    by (*auto intro*!: *log-eq-one neq-le-trans*)
  finally have $log\ (1\ /\ l)\ (dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt) - c \leq log\ (1\ /\ l)\ (dist\ v\ \nu_b\text{-}opt) - 1 - c$ **.**
  thus *?thesis*
    by *linarith*
qed

lemma *dist-$\mathcal{L}_b$-lt-dist-opt*: $dist\ v\ (\mathcal{L}_b\ v) \leq 2 * dist\ v\ \nu_b\text{-}opt$
proof –
  have *le1*: $dist\ v\ (\mathcal{L}_b\ v) \leq dist\ v\ \nu_b\text{-}opt + dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt$
    by (*simp add*: *dist-triangle dist-commute*)
  have *le2*: $dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt \leq l * dist\ v\ \nu_b\text{-}opt$
    using $\mathcal{L}_b$-*opt contraction-*$\mathcal{L}$
    by *metis*
  show *?thesis*
    using *mult-right-mono*[*of l 1*] *disc-lt-one*
    by (*fastforce intro*!: *order.trans*[*OF le2*] *order.trans*[*OF le1*])
qed

abbreviation *term-measure* $\equiv (\lambda(eps,\ v).$

  if $v = \nu_b$*-opt* $\vee$ $l = 0$
  then 0
  else nat (*ceiling* (*log* (*1/l*) (*dist v* ν_b*-opt*) $-$ *log* (*1/l*) (*eps* $*$ (*1*$-$*l*) / (*8* $*$ *l*)))))

function *value-iteration* :: *real* $\Rightarrow$ (*'s* $\Rightarrow_b$ *real*) $\Rightarrow$ (*'s* $\Rightarrow_b$ *real*) **where**
 value-iteration eps v =
 (*if 2* $*$ *l* $*$ *dist v* ($\mathcal{L}_b$ *v*) $<$ *eps* $*$ (*1*$-$*l*) $\vee$ *eps* $\leq$ *0 then* $\mathcal{L}_b$ *v else value-iteration eps* ($\mathcal{L}_b$ *v*))
 by *auto*

termination
proof (*relation Wellfounded.measure term-measure*, (*simp*; *fail*), *cases l = 0*)
 case *False*
 fix *eps v*
 assume *h*: $\neg$ (*2* $*$ *l* $*$ *dist v* ($\mathcal{L}_b$ *v*) $<$ *eps* $*$ (*1* $-$ *l*) $\vee$ *eps* $\leq$ *0*)
 show ((*eps*, $\mathcal{L}_b$ *v*), *eps*, *v*) $\in$ *Wellfounded.measure term-measure*
 proof $-$
  have *gt-zero*[*simp*]: *l* $\neq$ *0 eps* $>$ *0* **and** *dist-ge*: *eps* $*$ (*1* $-$ *l*) $\leq$ *dist v* ($\mathcal{L}_b$ *v*) $*$ (*2* $*$ *l*)
   using *h*
   by (*auto simp*: *algebra-simps*)
  have *v-not-opt*: *v* $\neq$ ν_b*-opt*
   using *h*
   by *force*
  have *log* (*1* / *l*) (*eps* $*$ (*1* $-$ *l*) / (*8* $*$ *l*)) $<$ *log* (*1* / *l*) (*dist v* ν_b*-opt*)
  proof (*intro log-less*)
   show *1* $<$ *1* / *l*
    by (*auto intro*!: *mult-imp-less-div-pos intro*: *neq-le-trans*)
   show *0* $<$ *eps* $*$ (*1* $-$ *l*) / (*8* $*$ *l*)
    by (*auto intro*!: *mult-imp-less-div-pos intro*: *neq-le-trans*)
   show *eps* $*$ (*1* $-$ *l*) / (*8* $*$ *l*) $<$ *dist v* ν_b*-opt*
    using *dist-pos-lt*[*OF v-not-opt*] *dist-*$\mathcal{L}_b$*-lt-dist-opt*[*of v*] *gt-zero zero-le-disc*
     mult-strict-left-mono[*of dist v* ($\mathcal{L}_b$ *v*) (*4* $*$ *dist v* ν_b*-opt*) *l*]
    by (*intro mult-imp-div-pos-less le-less-trans*[*OF dist-ge*], *argo*+)
  qed
  thus *?thesis*
   using *vi-rel-dec h*
   by *auto*
 qed
qed *auto*

The distance between an estimate for the value and the optimal value can be bounded with respect to the distance between the estimate and the result of applying it to $\mathcal{L}_b$

lemma *contraction-*$\mathcal{L}$*-dist*: (*1* $-$ *l*) $*$ *dist v* ν_b*-opt* $\leq$ *dist v* ($\mathcal{L}_b$ *v*)
 using *contraction-dist contraction-*$\mathcal{L}$ *disc-lt-one zero-le-disc*

by *fastforce*

lemma *dist-$\mathcal{L}_b$-opt-eps*:
  assumes $eps > 0$ $2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1-l)$
  shows $dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt < eps\ /\ 2$
proof −
  have $dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (\mathcal{L}_b\ v)\ /\ (1 - l)$
    using *contraction-$\mathcal{L}$-dist*
    by (*simp add*: *mult.commute pos-le-divide-eq*)
  hence $2 * l * dist\ v\ \nu_b\text{-}opt \leq 2 * l * (dist\ v\ (\mathcal{L}_b\ v)\ /\ (1 - l))$
    using *contraction-$\mathcal{L}$-dist assms mult-le-cancel-left-pos*[*of* $2 * l$]
    by (*fastforce intro*!: *mult-left-mono*[*of - -* $2 * l$])
  hence $2 * l * dist\ v\ \nu_b\text{-}opt < eps$
   by (*auto simp*: *assms*(*2*) *pos-divide-less-eq intro*: *order.strict-trans1*)
  hence $dist\ v\ \nu_b\text{-}opt * l < eps\ /\ 2$
    by *argo*
  hence $l * dist\ v\ \nu_b\text{-}opt < eps\ /\ 2$
    by (*auto simp*: *algebra-simps*)
  thus $dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt < eps\ /\ 2$
    using *contraction-$\mathcal{L}$*[*of v ν_b-opt*]
    by *auto*
qed

The estimates above allow to give a bound on the error of *value-iteration*.

declare *value-iteration.simps*[*simp del*]

lemma *value-iteration-error*:
  assumes $eps > 0$
  shows *dist* (*value-iteration eps v*) ν_b*-opt* $< eps\ /\ 2$
  using *assms dist-$\mathcal{L}_b$-opt-eps value-iteration.simps*
  by (*induction eps v rule*: *value-iteration.induct*) *auto*

After the value iteration terminates, one can easily obtain a stationary deterministic epsilon-optimal policy.

Such a policy does not exist in general, attainment of the supremum in $\mathcal{L}_b$ is required.

definition *find-policy* (v :: $'s \Rightarrow_b real$) s = *arg-max-on* ($\lambda a.\ L_a\ a\ v\ s$) ($A\ s$)

definition *vi-policy eps v* = *find-policy* (*value-iteration eps v*)

We formalize the attainment of the supremum using a predicate *has-arg-max*.

abbreviation $vi\ u\ n \equiv (\mathcal{L}_b\ \widehat{\ }\widehat{\ }n)\ u$

lemma *$\mathcal{L}_b$-iter-mono*:
  assumes $u \leq v$ **shows** $vi\ u\ n \leq vi\ v\ n$
  using *assms $\mathcal{L}_b$-mono*

by (*induction n*) *auto*

lemma
assumes *vi v* (*Suc n*) $\leq$ *vi v n*
shows *vi v* (*Suc n* + *m*) $\leq$ *vi v* (*n* + *m*)
proof –
have *vi v* (*Suc n* + *m*) = *vi* (*vi v* (*Suc n*)) *m*
by (*simp add*: *Groups.add-ac*(*2*) *funpow-add funpow-swap1*)
also have ... $\leq$ *vi* (*vi v n*) *m*
using $\mathcal{L}_b$-*iter-mono*[*OF assms*]
by *auto*
also have ... = *vi v* (*n* + *m*)
by (*simp add*: *add.commute funpow-add*)
finally show *?thesis* **.**
qed

lemma
assumes *vi v n* $\leq$ *vi v* (*Suc n*)
shows *vi v* (*n* + *m*) $\leq$ *vi v* (*Suc n* + *m*)
proof –
have *vi v* (*n* + *m*) $\leq$ *vi* (*vi v n*) *m*
by (*simp add*: *Groups.add-ac*(*2*) *funpow-add funpow-swap1*)
also have ... $\leq$ *vi v* (*Suc n* + *m*)
using $\mathcal{L}_b$-*iter-mono*[*OF assms*]
by (*auto simp only*: *add.commute funpow-add o-apply*)
finally show *?thesis* **.**
qed

lemma *vi v* $\longrightarrow$ ν_b-*opt*
using $\mathcal{L}_b$-*lim***.**

lemma (λ*n. dist* (*vi v* (*Suc n*)) (*vi v n*)) $\longrightarrow$ *0*
using *thm-6-3-1-b-aux*[*of v*]
by (*auto simp only*: *dist-commute*[*of* (($\mathcal{L}_b$ ^^ *Suc* -) *v*)])

end

context *MDP-att-*$\mathcal{L}$
begin

The error of the resulting policy is bounded by the distance from its value to the value computed by the value iteration plus the error in the value iteration itself. We show that both are less than *eps* / (*2*::′*b*) when the algorithm terminates.

lemma *find-policy-error-bound*:
  assumes *eps* $>$ *0* *2* $*$ *l* $*$ *dist v* ($\mathcal{L}_b$ *v*) $<$ *eps* $*$ (*1*$-$*l*)
  shows *dist* (ν_b (*mk-stationary-det* (*find-policy* ($\mathcal{L}_b$ *v*)))) ν_b*-opt* $<$ *eps*
proof $-$
  let *?d* = *mk-dec-det* (*find-policy* ($\mathcal{L}_b$ *v*))
  let *?p* = *mk-stationary ?d*

  have *L-eq-*$\mathcal{L}_b$: *L* (*mk-dec-det* (*find-policy v*)) *v* = $\mathcal{L}_b$ *v* **for** *v*
    unfolding *find-policy-def*
  proof (*intro antisym*)
    show *L* (*mk-dec-det* (λ*s. arg-max-on* (λ*a.* L_a *a v s*) (*A s*))) *v* $\leq$ $\mathcal{L}_b$ *v*
      using *Sup-att has-arg-max-arg-max abs-L-le*
    unfolding $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-eq-SUP-det less-eq-bfun-def arg-max-on-def is-dec-det-def max-L-ex-def*
      by (*auto intro*!: *cSUP-upper bounded-imp-bdd-above boundedI*[*of* - r_M + *l* $*$ *norm v*])
  next
    show $\mathcal{L}_b$ *v* $\leq$ *L* (*mk-dec-det* (λ*s. arg-max-on* (λ*a.* L_a *a v s*) (*A s*))) *v*
      unfolding *less-eq-bfun-def* $\mathcal{L}_b$*.rep-eq* $\mathcal{L}$*-eq-SUP-det*
      using *Sup-att ex-dec-det*
        by (*auto intro*!: *cSUP-least app-arg-max-ge simp*: *L-eq-*L_a*-det max-L-ex-def is-dec-det-def*)
  qed
  have *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) = *dist* (*L ?d* (ν_b *?p*)) ($\mathcal{L}_b$ *v*)
    using *L-*ν*-fix*
    by *force*
  also have ... $\leq$ *dist* (*L ?d* (ν_b *?p*)) ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) + *dist* ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) ($\mathcal{L}_b$ *v*)
    using *dist-triangle*
    by *blast*
  also have ... = *dist* (*L ?d* (ν_b *?p*)) (*L ?d* ($\mathcal{L}_b$ *v*)) + *dist* ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) ($\mathcal{L}_b$ *v*)
    by (*auto simp*: *L-eq-*$\mathcal{L}_b$)
  also have ... $\leq$ *l* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) + *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*
    using *contraction-*$\mathcal{L}$ *contraction-L*
    by (*fastforce intro*!: *add-mono*)
  finally have *aux*: *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $\leq$ *l* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) + *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v* **.**
  hence *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $-$ *l* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $\leq$ *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*
    by *auto*
  hence *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $*$ (*1* $-$ *l*) $\leq$ *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*
    by *argo*
  hence *2* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $*$ (*1*$-$*l*) $\leq$ *2* $*$ (*l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*)
    using *zero-le-disc mult-left-mono*
    by *auto*

```
  also have ... ≤ eps * (1−l)
    using assms
   by (auto intro!: mult-left-mono simp: dist-commute pos-divide-le-eq)
  finally have 2 * dist (ν_b ?p) (ℒ_b v) * (1 − l) ≤ eps * (1 − l) .
  hence 2 * dist (ν_b ?p) (ℒ_b v) ≤ eps
    using disc-lt-one mult-right-le-imp-le
    by auto
  moreover have 2 * dist (ℒ_b v) ν_b-opt < eps
    using dist-ℒ_b-opt-eps assms
    by fastforce
 moreover have dist (ν_b ?p) ν_b-opt ≤ dist (ν_b ?p) (ℒ_b v) + dist (ℒ_b
v) ν_b-opt
    using dist-triangle
    by blast
  ultimately show ?thesis
    by auto
qed

lemma vi-policy-opt:
  assumes 0 < eps
  shows dist (ν_b (mk-stationary-det (vi-policy eps v))) ν_b-opt < eps
  unfolding vi-policy-def
  using assms
proof (induction eps v rule: value-iteration.induct)
  case (1 v)
  then show ?case
    using find-policy-error-bound
    by (subst value-iteration.simps) auto
qed

lemma lemma-6-3-1-d:
  assumes eps > 0
  assumes 2 * l * dist (vi v (Suc n)) (vi v n) < eps * (1−l)
  shows dist (vi v (Suc n)) ν_b-opt < eps / 2
  using dist-ℒ_b-opt-eps assms
  by (simp add: dist-commute)

end

context MDP-act begin

definition find-policy′ (v :: ′s ⇒_b real) s = arb-act (opt-acts v s)

definition vi-policy′ eps v = find-policy′ (value-iteration eps v)

lemma find-policy′-error-bound:
  assumes eps > 0 2 * l * dist v (ℒ_b v) < eps * (1−l)
   shows dist (ν_b (mk-stationary-det (find-policy′ (ℒ_b v)))) ν_b-opt <
eps
```

proof −
let *?d* = *mk-dec-det* (*find-policy′* ($\mathcal{L}_b$ *v*))
let *?p* = *mk-stationary ?d*
have *L-eq-*$\mathcal{L}_b$: *L* (*mk-dec-det* (*find-policy′ v*)) *v* = $\mathcal{L}_b$ *v* **for** *v*
unfolding *find-policy′-def*
by (*metis* ν*-improving-imp-*$\mathcal{L}_b$ ν*-improving-opt-acts*)
have *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) = *dist* (*L ?d* (ν_b *?p*)) ($\mathcal{L}_b$ *v*)
using *L-*ν*-fix*
by *force*
also have ... ≤ *dist* (*L ?d* (ν_b *?p*)) ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) + *dist* ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) ($\mathcal{L}_b$ *v*)
using *dist-triangle*
by *blast*
also have ... = *dist* (*L ?d* (ν_b *?p*)) (*L ?d* ($\mathcal{L}_b$ *v*)) + *dist* ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) ($\mathcal{L}_b$ *v*)
by (*auto simp*: *L-eq-*$\mathcal{L}_b$)
also have ... ≤ *l* ∗ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) + *l* ∗ *dist* ($\mathcal{L}_b$ *v*) *v*
using *contraction-*$\mathcal{L}$ *contraction-L*
by (*fastforce intro*!: *add-mono*)
finally have *aux*: *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) ≤ *l* ∗ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) + *l* ∗ *dist* ($\mathcal{L}_b$ *v*) *v* **.**
hence *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) − *l* ∗ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) ≤ *l* ∗ *dist* ($\mathcal{L}_b$ *v*) *v*
by *auto*
hence *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) ∗ (*1* − *l*) ≤ *l* ∗ *dist* ($\mathcal{L}_b$ *v*) *v*
by *argo*
hence *2* ∗ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) ∗ (*1*−*l*) ≤ *2* ∗ (*l* ∗ *dist* ($\mathcal{L}_b$ *v*) *v*)
using *zero-le-disc mult-left-mono*
by *auto*
also have ... ≤ *eps* ∗ (*1*−*l*)
using *assms*
by (*auto intro*!: *mult-left-mono simp*: *dist-commute pos-divide-le-eq*)
finally have *2* ∗ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) ∗ (*1* − *l*) ≤ *eps* ∗ (*1* − *l*)**.**
hence *2* ∗ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) ≤ *eps*
using *disc-lt-one mult-right-le-imp-le*
by *auto*
moreover have *2* ∗ *dist* ($\mathcal{L}_b$ *v*) ν_b*-opt* < *eps*
using *dist-*$\mathcal{L}_b$*-opt-eps assms*
by *fastforce*
moreover have *dist* (ν_b *?p*) ν_b*-opt* ≤ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) + *dist* ($\mathcal{L}_b$ *v*) ν_b*-opt*
using *dist-triangle*
by *blast*
ultimately show *?thesis*
by *auto*
qed

lemma *vi-policy′-opt*:
assumes *eps* > *0 l* > *0*

shows *dist* (ν_b (*mk-stationary-det* (*vi-policy′ eps v*))) ν_b*-opt* < *eps*
unfolding *vi-policy′-def*
using *assms*
proof (*induction eps v rule*: *value-iteration.induct*)
case (*1 v*)
then show *?case*
using *find-policy′-error-bound*
by (*subst value-iteration.simps*) *auto*
qed

end
end

theory *Policy-Iteration*
imports *MDP−Rewards.MDP-reward*

begin

2 Policy Iteration

The Policy Iteration algorithms provides another way to find optimal policies under the expected total reward criterion. It differs from Value Iteration in that it continuously improves an initial guess for an optimal decision rule. Its execution can be subdivided into two alternating steps: policy evaluation and policy improvement.

Policy evaluation means the calculation of the value of the current decision rule.

During the improvement phase, we choose the decision rule with the maximum value for L, while we prefer to keep the old action selection in case of ties.

context *MDP-att-*$\mathcal{L}$ **begin**
definition *policy-eval d* = ν_b (*mk-stationary-det d*)
end

context *MDP-act*
begin

definition *policy-improvement d v s* = (
if is-arg-max ($\lambda a.\ L_a$ *a* (*apply-bfun v*) *s*) ($\lambda a.\ a \in A\ s$) (*d s*)
then d s
else arb-act (*opt-acts v s*))

definition *policy-step d* = *policy-improvement d* (*policy-eval d*)

function *policy-iteration* :: $('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow 'a)$ **where**
policy-iteration d = (
let $d' =$ *policy-step d in*
if $d = d' \vee \neg$*is-dec-det d then d else policy-iteration* d')
by *auto*

The policy iteration algorithm as stated above does require that the supremum in $\mathcal{L}_b$ is always attained.

Each policy improvement returns a valid decision rule.

lemma *is-dec-det-pi*: *is-dec-det* (*policy-improvement d v*)
unfolding *policy-improvement-def is-dec-det-def is-arg-max-def*
by (*auto simp*: *some-opt-acts-in-A*)

lemma *policy-improvement-is-dec-det*: $d \in D_D \Longrightarrow$ *policy-improvement* $d\ v \in D_D$
unfolding *policy-improvement-def is-dec-det-def*
using *some-opt-acts-in-A*
by *auto*

lemma *policy-improvement-improving*:
assumes $d \in D_D$
shows ν*-improving v* (*mk-dec-det* (*policy-improvement d v*))
proof –
have $\mathcal{L}_b$ *v x* = *L* (*mk-dec-det* (*policy-improvement d v*)) *v x* **for** *x*
using *is-opt-act-some*
by (*fastforce simp*: *thm-6-2-10-a-aux′* *L-eq-*L_a*-det is-opt-act-def policy-improvement-def*
arg-max-SUP[*symmetric*, *of* - - (*policy-improvement d v x*)])
thus *?thesis*
using *policy-improvement-is-dec-det assms*
by (*auto simp*: ν*-improving-alt*)
qed

lemma *eval-policy-step-L*:
assumes *is-dec-det d*
shows *L* (*mk-dec-det* (*policy-step d*)) (*policy-eval d*) = $\mathcal{L}_b$ (*policy-eval d*)
unfolding *policy-step-def*
using *assms*
by (*auto simp*: ν*-improving-imp-*$\mathcal{L}_b$[*OF policy-improvement-improving*])

The sequence of policies generated by policy iteration has monotonically increasing discounted reward.

lemma *policy-eval-mon*:
assumes *is-dec-det d*
shows *policy-eval d* $\leq$ *policy-eval* (*policy-step d*)
proof –

let $?d' = mk\text{-}dec\text{-}det\ (policy\text{-}step\ d)$
let $?dp = mk\text{-}stationary\text{-}det\ d$
let $?P = \sum t.\ l\ \widehat{}\ t *_R \mathcal{P}_1\ ?d'\ \widehat{}\widehat{}\ t$

have $L\ (mk\text{-}dec\text{-}det\ d)\ (policy\text{-}eval\ d) \leq L\ ?d'\ (policy\text{-}eval\ d)$
using *assms*
by (*auto simp*: $L\text{-}le\text{-}\mathcal{L}_b$ *eval-policy-step-L*)
hence $policy\text{-}eval\ d \leq L\ ?d'\ (policy\text{-}eval\ d)$
using *L-ν-fix policy-eval-def*
by *auto*
hence $\nu_b\ ?dp \leq r\text{-}dec_b\ ?d' + l *_R \mathcal{P}_1\ ?d'\ (\nu_b\ ?dp)$
unfolding *policy-eval-def L-def*
by *auto*
hence $(id\text{-}blinfun - l *_R \mathcal{P}_1\ ?d')\ (\nu_b\ ?dp) \leq r\text{-}dec_b\ ?d'$
by (*simp add*: *blinfun.diff-left diff-le-eq scaleR-blinfun.rep-eq*)
hence $?P\ ((id\text{-}blinfun - l *_R \mathcal{P}_1\ ?d')\ (\nu_b\ ?dp)) \leq ?P\ (r\text{-}dec_b\ ?d')$
using *lemma-6-1-2-b*
by *auto*
hence $\nu_b\ ?dp \leq ?P\ (r\text{-}dec_b\ ?d')$
using *inv-norm-le'(2)*[*OF norm-*$\mathcal{P}_1$*-l-less*] *blincomp-scaleR-right suminf-cong*
by (*metis* (*mono-tags*, *lifting*))
thus *?thesis*
unfolding *policy-eval-def*
by (*auto simp*: *ν-stationary*)
qed

If policy iteration terminates, i.e. $d = policy\text{-}step\ d$, then it does so with optimal value.

lemma *policy-step-eq-imp-opt*:
assumes $is\text{-}dec\text{-}det\ d\ d = policy\text{-}step\ d$
shows $\nu_b\ (mk\text{-}stationary\ (mk\text{-}dec\text{-}det\ d)) = \nu_b\text{-}opt$
proof −
have $policy\text{-}eval\ d = \mathcal{L}_b\ (policy\text{-}eval\ d)$
unfolding *policy-eval-def*
using *L-ν-fix assms eval-policy-step-L*[*unfolded policy-eval-def*]
by *fastforce*
thus *?thesis*
unfolding *policy-eval-def*
using $\mathcal{L}$*-fix-imp-opt*
by *blast*
qed

end

We prove termination of policy iteration only if both the state and action sets are finite.

locale $MDP\text{-}PI\text{-}finite = MDP\text{-}act\ A\ K\ r\ l\ arb\text{-}act$
for

A **and**
K :: *'s* ::*countable* × *'a* ::*countable* ⇒ *'s pmf* **and** *r l arb-act* +
assumes *fin-states*: *finite* (*UNIV* :: *'s set*) **and** *fin-actions*: ⋀*s*. *finite* (*A s*)
begin

If the state and action sets are both finite, then so is the set of deterministic decision rules D_D

lemma *finite-*D_D[*simp*]: *finite* D_D
proof −
let *?set* = {*d*. ∀ *x* :: *'s*. (*x* ∈ *UNIV* ⟶ *d x* ∈ (⋃*s*. *A s*)) ∧ (*x* ∉ *UNIV* ⟶ *d x* = *undefined*)}
have *finite* (⋃*s*. *A s*)
using *fin-actions fin-states* **by** *blast*
hence *finite ?set*
using *fin-states*
by (*fastforce intro*: *finite-set-of-finite-funs*)
moreover have D_D ⊆ *?set*
unfolding *is-dec-det-def*
by *auto*
ultimately show *?thesis*
using *finite-subset*
by *auto*
qed

lemma *finite-rel*: *finite* {(*u*, *v*). *is-dec-det u* ∧ *is-dec-det v* ∧ ν_b (*mk-stationary-det u*) >
ν_b (*mk-stationary-det v*)}
proof−
have *aux*: *finite* {(*u*, *v*). *is-dec-det u* ∧ *is-dec-det v*}
by *auto*
show *?thesis*
by (*auto intro*: *finite-subset*[*OF - aux*])
qed

This auxiliary lemma shows that policy iteration terminates if no improvement to the value of the policy could be made, as then the policy remains unchanged.

lemma *eval-eq-imp-policy-eq*:
assumes *policy-eval d* = *policy-eval* (*policy-step d*) *is-dec-det d*
shows *d* = *policy-step d*
proof −
have *policy-eval d s* = *policy-eval* (*policy-step d*) *s* **for** *s*
using *assms*
by *auto*
have *policy-eval d* = *L* (*mk-dec-det d*) (*policy-eval* (*policy-step d*))
unfolding *policy-eval-def*
using *L-ν-fix*
by (*auto simp*: *assms*(*1*)[*symmetric*, *unfolded policy-eval-def*])

hence *policy-eval* $d = \mathcal{L}_b$ (*policy-eval* d)
by (*metis* *L-ν-fix* *policy-eval-def* *assms* *eval-policy-step-L*)
hence L (*mk-dec-det* d) (*policy-eval* d) $s = \mathcal{L}_b$ (*policy-eval* d) s **for** s
using ‹*policy-eval* $d = L$ (*mk-dec-det* d) (*policy-eval* (*policy-step* d))› *assms*(*1*) **by** *auto*
hence *is-arg-max* ($\lambda a.$ L_a a (ν_b (*mk-stationary* (*mk-dec-det* d))) s) ($\lambda a.$ $a \in A$ s) (d s) **for** s
unfolding *L-eq-L_a-det*
unfolding *policy-eval-def* $\mathcal{L}_b$*.rep-eq* *$\mathcal{L}$-eq-SUP-det* *SUP-step-det-eq*
using *assms*(*2*) *is-dec-det-def* L_a*-le*
by (*auto simp del*: ν_b*.rep-eq* *simp*: ν_b*.rep-eq*[*symmetric*]
intro!: *SUP-is-arg-max* *boundedI*[*of* - r_M + l $*$ *norm* -] *bounded-imp-bdd-above*)
thus *?thesis*
unfolding *policy-eval-def* *policy-step-def* *policy-improvement-def*
by *auto*
qed

We are now ready to prove termination in the context of finite state-action spaces. Intuitively, the algorithm terminates as there are only finitely many decision rules, and in each recursive call the value of the decision rule increases.

termination *policy-iteration*
proof (*relation* $\{(u, v).$ $u \in D_D \wedge v \in D_D \wedge \nu_b$ (*mk-stationary-det* u) $> \nu_b$ (*mk-stationary-det* v)$\}$)
show *wf* $\{(u, v).$ $u \in D_D \wedge v \in D_D \wedge \nu_b$ (*mk-stationary-det* v) $< \nu_b$ (*mk-stationary-det* u)$\}$
using *finite-rel*
by (*auto intro*!: *finite-acyclic-wf* *acyclicI-order*)
next
fix d x
assume h: $x =$ *policy-step* d $\neg$ ($d = x \vee \neg$ *is-dec-det* d)
have *is-dec-det* $d \Longrightarrow \nu_b$ (*mk-stationary-det* d) $\leq \nu_b$ (*mk-stationary-det* (*policy-step* d))
using *policy-eval-mon*
by (*simp add*: *policy-eval-def*)
hence *is-dec-det* $d \Longrightarrow d \neq$ *policy-step* $d \Longrightarrow$
ν_b (*mk-stationary-det* d) $< \nu_b$ (*mk-stationary-det* (*policy-step* d))
using *eval-eq-imp-policy-eq* *policy-eval-def*
by (*force intro*!: *order.not-eq-order-implies-strict*)
thus $(x, d) \in \{(u, v).$ $u \in D_D \wedge v \in D_D \wedge \nu_b$ (*mk-stationary-det* v) $< \nu_b$ (*mk-stationary-det* u)$\}$
using *is-dec-det-pi* *policy-step-def* h
by *auto*
qed

The termination proof gives us access to the induction rule/simplification lemmas associated with the *policy-iteration* definition.

Thus we can prove that the algorithm finds an optimal policy.

lemma *is-dec-det-pi'*: $d \in D_D \Longrightarrow$ *is-dec-det* (*policy-iteration d*)
using *is-dec-det-pi*
by (*induction d rule*: *policy-iteration.induct*) (*auto simp*: *Let-def policy-step-def*)

lemma *pi-pi*[*simp*]: $d \in D_D \Longrightarrow$ *policy-step* (*policy-iteration d*) = *policy-iteration d*
using *is-dec-det-pi*
by (*induction d rule*: *policy-iteration.induct*) (*auto simp*: *policy-step-def Let-def*)

lemma *policy-iteration-correct*:
$d \in D_D \Longrightarrow \nu_b$ (*mk-stationary-det* (*policy-iteration d*)) = ν_b-*opt*
by (*induction d rule*: *policy-iteration.induct*)
(*fastforce intro*!: *policy-step-eq-imp-opt is-dec-det-pi' simp del*: *policy-iteration.simps*)
end

context *MDP-finite-type* **begin**

The following proofs concern code generation, i.e. how to represent $\mathcal{P}_1$ as a matrix.

sublocale *MDP-att-$\mathcal{L}$*
by (*auto simp*: *A-ne finite-is-arg-max MDP-att-$\mathcal{L}$-def MDP-att-$\mathcal{L}$-axioms-def max-L-ex-def*
has-arg-max-def MDP-reward-axioms)

definition *fun-to-matrix f* = *matrix* (λv. (χ *j*. *f* (*vec-nth v*) *j*))
definition *Ek-mat d* = *fun-to-matrix* (λv. (($\mathcal{P}_1$ *d*) (*Bfun v*)))
definition *nu-inv-mat d* = *fun-to-matrix* ((λv. ((*id-blinfun* $-$ *l* $*_R$ $\mathcal{P}_1$ *d*) (*Bfun v*))))
definition *nu-mat d* = *fun-to-matrix* (λv. (($\sum$ *i*. (*l* $*_R$ $\mathcal{P}_1$ *d*) ^^ *i*) (*Bfun v*)))

lemma *apply-nu-inv-mat*:
(*id-blinfun* $-$ *l* $*_R$ $\mathcal{P}_1$ *d*) *v* = *Bfun* (λi. ((*nu-inv-mat d*) $*v$ (*vec-lambda v*)) \$ *i*)
proof $-$
have *eq-onpI*: *P x* $\Longrightarrow$ *eq-onp P x x* **for** *P x*
by(*simp add*: *eq-onp-def*)

have *Real-Vector-Spaces.linear* (λv. *vec-lambda* (((*id-blinfun* $-$ *l* $*_R$ $\mathcal{P}_1$ *d*) (*bfun.Bfun* ((\$) *v*)))))
by (*auto simp del*: *real-scaleR-def intro*: *linearI*
simp: *scaleR-vec-def eq-onpI plus-vec-def vec-lambda-inverse plus-bfun.abs-eq*[*symmetric*]
scaleR-bfun.abs-eq[*symmetric*] *blinfun.scaleR-right blinfun.add-right*)
thus *?thesis*

unfolding *Ek-mat-def fun-to-matrix-def nu-inv-mat-def*
by (*auto simp*: *apply-bfun-inverse vec-lambda-inverse*)
qed

lemma *bounded-linear-vec-lambda*: *bounded-linear* (λx. *vec-lambda* (x :: $'s \Rightarrow_b$ *real*))
proof (*intro bounded-linear-intro*)
fix x :: $'s \Rightarrow_b$ *real*
have *sqrt* ($\sum$ $i \in$ *UNIV* . (*apply-bfun* x i)^2) $\leq$ ($\sum$ $i \in$ *UNIV* . $|$(*apply-bfun* x i)$|$)
using *L2-set-le-sum-abs*
unfolding *L2-set-def*
by *auto*
also have ($\sum$ $i \in$ *UNIV* . $|$(*apply-bfun* x i)$|$) $\leq$ (*card* (*UNIV* :: $'s$ *set*) $*$ ($\bigsqcup$*xa*. $|$*apply-bfun* x *xa*$|$))
by (*auto intro*!: *cSup-upper sum-bounded-above*)
finally show *norm* (*vec-lambda* (*apply-bfun* x)) $\leq$ *norm* $x * CARD('s)$
unfolding *norm-vec-def norm-bfun-def dist-bfun-def L2-set-def*
by (*auto simp add*: *mult.commute*)
qed (*auto simp*: *plus-vec-def scaleR-vec-def*)

lemma *bounded-linear-vec-lambda-blinfun*:
fixes f :: ($'s \Rightarrow_b$ *real*) $\Rightarrow_L$ ($'s \Rightarrow_b$ *real*)
shows *bounded-linear* (λv. *vec-lambda* (*apply-bfun* (*blinfun-apply* f (*bfun.Bfun* ((\$) v)))))
using *blinfun.bounded-linear-right*
by (*fastforce intro*: *bounded-linear-compose*[*OF bounded-linear-vec-lambda*]

bounded-linear-bfun-nth bounded-linear-compose[*of f*])

lemma *invertible-nu-inv-max*: *invertible* (*nu-inv-mat* d)
unfolding *nu-inv-mat-def fun-to-matrix-def*
by (*auto simp*: *matrix-invertible inv-norm-le$'$ vec-lambda-inverse apply-bfun-inverse*
bounded-linear.linear[*OF bounded-linear-vec-lambda-blinfun*]
intro!: *exI*[*of* - λv. (χ j. (λv. ($\sum i$. ($l *_R \mathcal{P}_1$ d) $\widehat{\ }\widehat{\ }$ i) (*Bfun* v)) (*vec-nth* v) j)])

end

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

locale *MDP-ord* = *MDP-finite-type* A K r l
for A **and**
K :: $'s$ :: {*finite*, *wellorder*} $\times$ $'a$ :: {*finite*, *wellorder*} $\Rightarrow$ $'s$ *pmf*
and r l
begin

lemma *$\mathcal{L}$-fin-eq-det*: $\mathcal{L}\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
  by (*simp add*: *SUP-step-det-eq* *$\mathcal{L}$-eq-SUP-det*)

lemma *$\mathcal{L}_b$-fin-eq-det*: $\mathcal{L}_b\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
  by (*simp add*: *SUP-step-det-eq* *$\mathcal{L}_b$.rep-eq* *$\mathcal{L}$-eq-SUP-det*)

sublocale *MDP-PI-finite* $A\ K\ r\ l\ \lambda X.\ Least\ (\lambda x.\ x \in X)$
  by *unfold-locales* (*auto intro*: *LeastI*)

end
end

theory *Modified-Policy-Iteration*
  imports
    Policy-Iteration
    Value-Iteration
begin

3 Modified Policy Iteration

locale *MDP-MPI* = *MDP-finite-type* $A\ K\ r\ l$ + *MDP-act* $A\ K\ r\ l$ *arb-act*
  for A **and** K :: $'s$ :: *finite* $\times$ $'a$ :: *finite* $\Rightarrow$ $'s$ *pmf* **and** $r\ l$ *arb-act*
begin

3.1 The Advantage Function B

definition $B\ v\ s = (\bigsqcup d \in D_R.\ (r\text{-}dec\ d\ s + (l *_R \mathcal{P}_1\ d - id\text{-}blinfun)\ v\ s))$

The function B denotes the advantage of choosing the optimal action vs. the current value estimate

lemma *B-eq-$\mathcal{L}$*: $B\ v\ s = \mathcal{L}\ v\ s - v\ s$
proof –
  have $*$: $B\ v\ s = (\bigsqcup d \in D_R.\ L\ d\ v\ s - v\ s)$
    unfolding *B-def L-def*
    by (*auto simp add*: *blinfun.bilinear-simps add-diff-eq*)
  show *?thesis*
    unfolding $*$
  proof (*rule antisym*)
    show $(\bigsqcup d{\in}D_R.\ L\ d\ v\ s - v\ s) \le \mathcal{L}\ v\ s - v\ s$
      unfolding *$\mathcal{L}$-def*
      using *ex-dec*
      by (*fastforce intro*!: *cSUP-upper cSUP-least*)
  next
    have *bdd-above* $((\lambda d.\ L\ d\ v\ s - v\ s)\ `\ D_R)$
    by (*auto intro*!: *bounded-const bounded-minus-comp bounded-imp-bdd-above*)
    thus $\mathcal{L}\ v\ s - v\ s \le (\bigsqcup d \in D_R.\ L\ d\ v\ s - v\ s)$

unfolding $\mathcal{L}$*-def diff-le-eq*
by (*intro cSUP-least*) (*auto intro*: *cSUP-upper2 simp*: *diff-le-eq*[*symmetric*])
qed
qed

B is a bounded function.

lift-definition B_b :: $('s \Rightarrow_b real) \Rightarrow 's \Rightarrow_b real$ **is** B
using $\mathcal{L}_b$*.rep-eq*[*symmetric*] *B-eq-*$\mathcal{L}$
by (*auto intro*!: *bfun-normI order.trans*[*OF abs-triangle-ineq4*] *add-mono abs-le-norm-bfun*)

lemma B_b*-eq-*$\mathcal{L}_b$: $B_b\ v = \mathcal{L}_b\ v - v$
by (*auto simp*: $\mathcal{L}_b$*.rep-eq* B_b*.rep-eq* *B-eq-*$\mathcal{L}$)

lemma $\mathcal{L}_b$*-eq-SUP-*L_a: $\mathcal{L}_b\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
using *L-eq-*L_a*-det* $\mathcal{L}_b$*-eq-SUP-det SUP-step-det-eq*
by *auto*

3.2 Optimization of the Value Function over Multiple Steps

definition $U\ m\ v\ s = (\bigsqcup d \in D_R.\ (\nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + ((l *_R \mathcal{P}_1\ d)^{\frown\frown}m)\ v)\ s)$

U expresses the value estimate obtained by optimizing the first m steps and afterwards using the current estimate.

lemma *U-zero* [*simp*]: $U\ 0\ v = v$
unfolding *U-def* $\mathcal{L}$*-def*
by (*auto simp*: ν_b*-fin.rep-eq*)

lemma *U-one-eq-*$\mathcal{L}$: $U\ 1\ v\ s = \mathcal{L}\ v\ s$
unfolding *U-def* $\mathcal{L}$*-def*
by (*auto simp*: ν_b*-fin-eq-*$\mathcal{P}_X$ *L-def blinfun.bilinear-simps*)

lift-definition U_b :: $nat \Rightarrow ('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow_b real)$ **is** U
proof –
fix $n\ v$
have $norm\ (\nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m) \le (\sum i<m.\ l\ \widehat{}\ i * r_M)$ **for** $d\ m$
using *abs-*ν*-fin-le* ν_b*-fin.rep-eq*
by (*auto intro*!: *norm-bound*)
moreover have $norm\ (((l *_R \mathcal{P}_1\ d)^{\frown\frown}m)\ v) \le l\ \widehat{}\ m * norm\ v$ **for** $d\ m$
by (*auto simp*: $\mathcal{P}_X$*-const*[*symmetric*] *blinfun.bilinear-simps blincomp-scaleR-right simp del*: $\mathcal{P}_X$*-sconst*
intro!: *boundedI order.trans*[*OF abs-le-norm-bfun*] *mult-left-mono*)
ultimately have $*$: $norm\ (\nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + ((l *_R \mathcal{P}_1\ d)^{\frown\frown}m)\ v) \le (\sum i<m.\ l\ \widehat{}\ i * r_M) +\ l\ \widehat{}\ m * norm\ v$ **for** $d\ m$
using *norm-triangle-mono* **by** *blast*

show $U\ n\ v \in bfun$
using *ex-dec order.trans*[*OF abs-le-norm-bfun* $*$]
by (*fastforce simp*: *U-def intro*!: *bfun-normI cSup-abs-le*)
qed

lemma U_b*-contraction*: $dist\ (U_b\ m\ v)\ (U_b\ m\ u) \le l\ \hat{}\ m * dist\ v\ u$
proof $-$
have *aux*: $dist\ (U_b\ m\ v\ s)\ (U_b\ m\ u\ s) \le l\ \hat{}\ m * dist\ v\ u$ **if** *le*: $U_b\ m\ u\ s \le U_b\ m\ v\ s$ **for** $s\ v\ u$
proof $-$
let $?U = \lambda m\ v\ d.\ (\nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + ((l *_R \mathcal{P}_1\ d)\ \hat{}\hat{}\ m)\ v)\ s$
have $U_b\ m\ v\ s - U_b\ m\ u\ s \le (\bigsqcup d \in D_R.\ ?U\ m\ v\ d - ?U\ m\ u\ d)$
using *bounded-stationary-*ν_b*-fin bounded-disc-*$\mathcal{P}_1$ *le*
unfolding U_b*.rep-eq U-def*
by (*intro le-SUP-diff'*) (*auto intro*: *bounded-plus-comp*)
also have $\ldots = (\bigsqcup d \in D_R.\ ((l *_R \mathcal{P}_1\ d)\ \hat{}\hat{}\ m)\ (v - u)\ s)$
by (*simp add*: *L-def scale-right-diff-distrib blinfun.bilinear-simps*)
also have $\ldots = (\bigsqcup d \in D_R.\ l\hat{}m * ((\mathcal{P}_1\ d\ \hat{}\hat{}\ m)\ (v - u)\ s))$
by (*simp add*: *blincomp-scaleR-right blinfun.scaleR-left*)
also have $\ldots = l\hat{}m * (\bigsqcup d \in D_R.\ ((\mathcal{P}_1\ d\ \hat{}\hat{}\ m)\ (v - u)\ s))$
using D_R*-ne bounded-P bounded-disc-*$\mathcal{P}_1{}'$
by (*auto intro*: *bounded-SUP-mul*)
also have $\ldots \le l\hat{}m * norm\ (\bigsqcup d \in D_R.\ ((\mathcal{P}_1\ d\ \hat{}\hat{}\ m)\ (v - u)\ s))$
by (*simp add*: *mult-left-mono*)
also have $\ldots \le l\hat{}m * (\bigsqcup d \in D_R.\ norm\ (((\mathcal{P}_1\ d\ \hat{}\hat{}\ m)\ (v - u)\ s)))$
using D_R*-ne ex-dec bounded-norm-comp bounded-disc-*$\mathcal{P}_1{}'$
by (*fastforce intro*!: *mult-left-mono*)
also have $\ldots \le l\hat{}m * (\bigsqcup d \in D_R.\ norm\ ((\mathcal{P}_1\ d\ \hat{}\hat{}\ m)\ ((v - u))))$
using *ex-dec*
by (*fastforce intro*!: *order.trans*[*OF norm-blinfun*] *abs-le-norm-bfun mult-left-mono cSUP-mono*)
also have $\ldots \le l\hat{}m * (\bigsqcup d \in D_R.\ norm\ ((v - u)))$
using *norm-*$\mathcal{P}_X$*-apply*
by (*auto simp*: $\mathcal{P}_X$*-const*[*symmetric*] *cSUP-least mult-left-mono*)
also have $\ldots = l\ \hat{}m * dist\ v\ u$
by (*auto simp*: *dist-norm*)
finally have $U_b\ m\ v\ s - U_b\ m\ u\ s \le l\hat{}m * dist\ v\ u$ **.**
thus *?thesis*
by (*simp add*: *dist-real-def le*)
qed
moreover have $U_b\ m\ v\ s \le U_b\ m\ u\ s \Longrightarrow dist\ (U_b\ m\ v\ s)\ (U_b\ m\ u\ s) \le l\hat{}m * dist\ v\ u$ **for** $u\ v\ s$
by (*simp add*: *aux dist-commute*)
ultimately have $dist\ (U_b\ m\ v\ s)\ (U_b\ m\ u\ s) \le l\hat{}m * dist\ v\ u$ **for** $u\ v\ s$
using *linear*

by *blast*
thus $dist\ (U_b\ m\ v)\ (U_b\ m\ u) \leq l\hat{\ }m * dist\ v\ u$
by (*simp add*: *dist-bound*)
qed

lemma U_b-*conv*:
$\exists!v.\ U_b\ (Suc\ m)\ v = v$
$(\lambda n.\ (U_b\ (Suc\ m)\ \hat{}\hat{}\ n)\ v) \longrightarrow (THE\ v.\ U_b\ (Suc\ m)\ v = v)$
proof −
have ∗: *is-contraction* $(U_b\ (Suc\ m))$
unfolding *is-contraction-def*
using U_b-*contraction*[*of Suc m*] *le-neq-trans*[*OF zero-le-disc*]
by (*cases* $l = 0$)
(*auto intro*!: *power-Suc-less-one intro*: *exI*[*of* - $l\hat{\ }(Suc\ m)$]])
show $\exists!v.\ U_b\ (Suc\ m)\ v = v\ (\lambda n.\ (U_b\ (Suc\ m)\ \hat{}\hat{}\ n)\ v) \longrightarrow$
$(THE\ v.\ U_b\ (Suc\ m)\ v = v)$
using *banach′*[*OF* ∗]
by *auto*
qed

lemma U_b-*convergent*: *convergent* $(\lambda n.\ (U_b\ (Suc\ m)\ \hat{}\hat{}\ n)\ v)$
by (*intro convergentI*[*OF* U_b-*conv*(*2*)]])

lemma U_b-*mono*:
assumes $v \leq u$
shows $U_b\ m\ v \leq U_b\ m\ u$
proof −
have $U_b\ m\ v\ s \leq U_b\ m\ u\ s$ **for** s
unfolding U_b.*rep-eq U-def*
proof (*intro cSUP-mono*, *goal-cases*)
case *2*
thus *?case*
by (*simp add*: *bounded-imp-bdd-above bounded-disc-*$\mathcal{P}_1$ *bounded-plus-comp bounded-stationary-*ν_b*-fin*)
next
case (*3 n*)
thus *?case*
using *less-eq-bfunD*[*OF* $\mathcal{P}_X$-*mono*[*OF assms*]]
by (*auto simp*: $\mathcal{P}_X$-*const*[*symmetric*] *blincomp-scaleR-right blinfun.scaleR-left intro*!: *mult-left-mono exI*)
qed *auto*
thus *?thesis*
using *assms*
by *auto*
qed

lemma U_b-*le*-$\mathcal{L}_b$: $U_b\ m\ v \leq (\mathcal{L}_b\ \hat{}\hat{}\ m)\ v$
proof −
have $U_b\ m\ v\ s = (\bigsqcup d \in D_R.\ (L\ d\hat{}\hat{}\ m)\ v\ s)$ **for** $m\ v\ s$

by (*auto simp*: *L-iter* U_b.*rep-eq* $\mathcal{L}_b$.*rep-eq* *U-def* $\mathcal{L}$-*def*)
thus *?thesis*
using *L-iter-le*-$\mathcal{L}_b$ *ex-dec*
by (*fastforce intro*!: *cSUP-least*)
qed

lemma *L-iter-le*-U_b:
assumes $d \in D_R$
shows $(L\ d^{\frown\frown}m)\ v \le U_b\ m\ v$
using *assms*
by (*fastforce intro*!: *cSUP-upper bounded-imp-bdd-above*
simp: *L-iter* U_b.*rep-eq* *U-def* *bounded-disc*-$\mathcal{P}_1$ *bounded-plus-comp* *bounded-stationary*-ν_b-*fin*)

lemma *lim*-U_b: $lim\ (\lambda n.\ (U_b\ (Suc\ m)\ ^{\frown\frown}\ n)\ v) = \nu_b\text{-}opt$
proof −
have *le-U*: ν_b-*opt* $\le U_b\ m\ \nu_b$-*opt* **for** m
proof −
obtain d **where** d: ν-*improving* ν_b-*opt* (*mk-dec-det* d) $d \in D_D$
using *ex-improving-det* **by** *auto*
have ν_b-*opt* $= (L\ (mk\text{-}dec\text{-}det\ d)\ ^{\frown\frown}\ m)\ \nu_b$-*opt*
by (*induction* m) (*metis* *L*-ν-*fix-iff* $\mathcal{L}_b$-*opt* ν-*improving-imp*-$\mathcal{L}_b$ d(*1*) *funpow-swap1*)+
thus *?thesis*
using ‹$d \in D_D$›
by (*auto intro*!: *order.trans*[*OF* - *L-iter-le*-U_b])
qed
have $U_b\ m\ \nu_b$-*opt* $\le \nu_b$-*opt* **for** m
using $\mathcal{L}$-*inc-le-opt*
by (*auto intro*!: *order.trans*[*OF* U_b-*le*-$\mathcal{L}_b$] *simp*: *funpow-swap1*)
hence $U_b\ (Suc\ m)\ \nu_b$-*opt* $= \nu_b$-*opt*
using *le-U*
by (*simp add*: *antisym*)
moreover have $(lim\ (\lambda n.\ (U_b\ (Suc\ m)\ ^{\frown\frown}n)\ v)) = U_b\ (Suc\ m)\ (lim\ (\lambda n.\ (U_b\ (Suc\ m)\ ^{\frown\frown}n)\ v))$
using *limI*[*OF* U_b-*conv*(*2*)] *theI'*[*OF* U_b-*conv*(*1*)]
by *auto*
ultimately show *?thesis*
using U_b-*conv*(*1*)
by *metis*
qed

lemma U_b-*tendsto*: $(\lambda n.\ (U_b\ (Suc\ m)\ ^{\frown\frown}\ n)\ v) \longrightarrow \nu_b\text{-}opt$
using *lim*-U_b U_b-*convergent convergent-LIMSEQ-iff*
by *metis*

lemma U_b-*fix-unique*: $U_b\ (Suc\ m)\ v = v \longleftrightarrow v = \nu_b\text{-}opt$

using $theI'[OF\ U_b\text{-}conv(1)]\ U_b\text{-}conv(1)$
by $(auto\ simp\text{: } LIMSEQ\text{-}unique[OF\ U_b\text{-}tendsto\ U_b\text{-}conv(2)[of\ m]])$

lemma $dist\text{-}U_b\text{-}opt$: $dist\ (U_b\ m\ v)\ \nu_b\text{-}opt \leq l\hat{}m * dist\ v\ \nu_b\text{-}opt$
proof −
have $dist\ (U_b\ m\ v)\ \nu_b\text{-}opt = dist\ (U_b\ m\ v)\ (U_b\ m\ \nu_b\text{-}opt)$
by $(metis\ U_b.abs\text{-}eq\ U_b\text{-}fix\text{-}unique\ U\text{-}zero\ apply\text{-}bfun\text{-}inverse\ not0\text{-}implies\text{-}Suc)$
also have $\ldots \leq l\hat{}m * dist\ v\ \nu_b\text{-}opt$
by $(meson\ U_b\text{-}contraction)$
finally show *?thesis* **.**
qed

3.3 Expressing a Single Step of Modified Policy Iteration

The function W equals the value computed by the Modified Policy Iteration Algorithm in a single iteration. The right hand addend in the definition describes the advantage of using the optimal action for the first m steps.

definition $W\ d\ m\ v = v + (\sum i < m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)\ (B_b\ v)$

lemma *W-eq-L-iter*:
assumes $\nu\text{-}improving\ v\ d$
shows $W\ d\ m\ v = (L\ d\hat{}\hat{}m)\ v$
proof −
have $(\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)\ (\mathcal{L}_b\ v) = (\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)$
$(L\ d\ v)$
using $\nu\text{-}improving\text{-}imp\text{-}\mathcal{L}_b$ *assms* **by** *auto*
hence $W\ d\ m\ v = v + ((\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)\ (L\ d\ v)) -$
$(\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)\ v$
by $(auto\ simp\text{: } W\text{-}def\ B_b\text{-}eq\text{-}\mathcal{L}_b\ blinfun.bilinear\text{-}simps\ algebra\text{-}simps$
$)$
also have $\ldots = v + \nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + (\sum i{<}m.\ ((l *_R$
$\mathcal{P}_1\ d)\hat{}\hat{}i)\ ((l *_R \mathcal{P}_1\ d)\ v)) - (\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)\ v$
unfolding *L-def*
by $(auto\ simp\text{: } \nu_b\text{-}fin\text{-}eq\ blinfun.bilinear\text{-}simps\ blinfun.sum\text{-}left$
$scaleR\text{-}right.sum)$
also have $\ldots = v + \nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + (\sum i{<}m.\ ((l *_R$
$\mathcal{P}_1\ d)\hat{}\hat{}Suc\ i)\ v) - (\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}i)\ v$
by $(auto\ simp\ del\text{: } blinfunpow.simps\ simp\text{: } blinfunpow\text{-}assoc)$
also have $\ldots = \nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + (\sum i{<}Suc\ m.\ ((l *_R$
$\mathcal{P}_1\ d)\hat{}\hat{}\ i)\ v) - (\sum i{<}m.\ (l *_R \mathcal{P}_1\ d)\hat{}\hat{}\ i)\ v$
by $(subst\ sum.lessThan\text{-}Suc\text{-}shift)\ auto$
also have $\ldots = \nu_b\text{-}fin\ (mk\text{-}stationary\ d)\ m + ((l *_R \mathcal{P}_1\ d)\hat{}\hat{}m)\ v$
by $(simp\ add\text{: } blinfun.sum\text{-}left)$
also have $\ldots = (L\ d\ \hat{}\hat{}\ m)\ v$
using *L-iter* **by** *auto*

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

lemma *W-le-U_b*:
assumes $v \le u$ *ν-improving v d*
shows $W\ d\ m\ v \le U_b\ m\ u$
proof −
have $U_b\ m\ u - W\ d\ m\ v \ge \nu_b$*-fin* (*mk-stationary d*) $m + ((l *_R \mathcal{P}_1\ d)$ ^^ $m)\ u - (\nu_b$*-fin* (*mk-stationary d*) $m + ((l *_R \mathcal{P}_1\ d)$^^$m)\ v)$
using *ν-improving-D-MR assms*(*2*) *bounded-stationary-ν_b-fin bounded-disc-$\mathcal{P}_1$*
by (*fastforce intro*!: *diff-mono bounded-imp-bdd-above cSUP-upper bounded-plus-comp simp*: *U_b.rep-eq U-def L-iter W-eq-L-iter*)
hence ∗: $U_b\ m\ u - W\ d\ m\ v \ge ((l *_R \mathcal{P}_1\ d)$ ^^ $m)\ (u - v)$
by (*auto simp*: *blinfun.diff-right*)
show $W\ d\ m\ v \le U_b\ m\ u$
using *order.trans*[*OF $\mathcal{P}_1$-n-disc-pos*[*unfolded blincomp-scaleR-right*[*symmetric*]] ∗] *assms*
by *auto*
qed

lemma *W-ge-$\mathcal{L}_b$*:
assumes $v \le u$ $0 \le B_b\ u$ *ν-improving u d′*
shows $\mathcal{L}_b\ v \le W\ d'\ (Suc\ m)\ u$
proof −
have $\mathcal{L}_b\ v \le u + B_b\ u$
using *assms*(*1*) *$\mathcal{L}_b$-mono B_b-eq-$\mathcal{L}_b$*
by *auto*
also have $\ldots \le W\ d'\ (Suc\ m)\ u$
using *L-mono ν-improving-imp-$\mathcal{L}_b$ assms*(*3*) *assms*
by (*induction m*) (*auto simp*: *W-eq-L-iter B_b-eq-$\mathcal{L}_b$*)
finally show *?thesis* **.**
qed

lemma *B_b-le*:
assumes *ν-improving v d*
shows $B_b\ v + (l *_R \mathcal{P}_1\ d - id\text{-}blinfun)\ (u - v) \le B_b\ u$
proof −
have $r\text{-}dec_b\ d + (l *_R \mathcal{P}_1\ d - id\text{-}blinfun)\ u \le B_b\ u$
using *L-def L-le-$\mathcal{L}_b$ assms*
by (*auto simp*: *B_b-eq-$\mathcal{L}_b$ $\mathcal{L}_b$.rep-eq $\mathcal{L}$-def blinfun.bilinear-simps*)
moreover have $B_b\ v = r\text{-}dec_b\ d + (l *_R \mathcal{P}_1\ d - id\text{-}blinfun)\ v$
using *assms*
by (*auto simp*: *B_b-eq-$\mathcal{L}_b$ ν-improving-imp-$\mathcal{L}_b$*[*of - d*] *L-def blinfun.bilinear-simps*)
ultimately show *?thesis*
by (*simp add*: *blinfun.diff-right*)
qed

lemma *$\mathcal{L}_b$-W-ge*:

assumes $u \le \mathcal{L}_b\ u$ *ν-improving u d*
shows $W\ d\ m\ u \le \mathcal{L}_b\ (W\ d\ m\ u)$
proof −
have $0 \le ((l *_R \mathcal{P}_1\ d)\ \widehat{\ }\widehat{\ }\ m)\ (B_b\ u)$
by (*metis* B_b-*eq*-$\mathcal{L}_b$ $\mathcal{P}_1$-*n-disc-pos assms*(*1*) *blincomp-scaleR-right diff-ge-0-iff-ge*)
also have $\ldots = ((l *_R \mathcal{P}_1\ d)\widehat{\ }\widehat{\ }0 + (\sum i < m.\ (l *_R \mathcal{P}_1\ d)\widehat{\ }\widehat{\ }(Suc\ i)))\ (B_b\ u) - (\sum i < m.\ (l *_R \mathcal{P}_1\ d)\widehat{\ }\widehat{\ }\ i)\ (B_b\ u)$
by (*subst sum.lessThan-Suc-shift*[*symmetric*]) (*auto simp*: *blinfun.diff-left*[*symmetric*])
also have $\ldots = B_b\ u + ((l *_R \mathcal{P}_1\ d - id\text{-}blinfun)\ o_L\ (\sum i < m.\ (l *_R \mathcal{P}_1\ d)\widehat{\ }\widehat{\ }i))\ (B_b\ u)$
by (*auto simp*: *blinfun.bilinear-simps sum-subtractf*)
also have $\ldots = B_b\ u + (l *_R \mathcal{P}_1\ d - id\text{-}blinfun)\ (W\ d\ m\ u - u)$
by (*auto simp*: *W-def sum.lessThan-Suc*[*unfolded lessThan-Suc-atMost*])
also have $\ldots \le B_b\ (W\ d\ m\ u)$
using B_b-*le assms*(*2*) **by** *blast*
finally have $0 \le B_b\ (W\ d\ m\ u)$ **.**
thus *?thesis* **using** B_b-*eq*-$\mathcal{L}_b$
by *auto*
qed

3.4 Computing the Bellman Operator over Multiple Steps

definition *L-pow* :: $('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow 'a) \Rightarrow nat \Rightarrow ('s \Rightarrow_b real)$
where
L-pow v d m = $(L\ (mk\text{-}dec\text{-}det\ d)\ \widehat{\ }\widehat{\ }\ Suc\ m)\ v$

lemma *sum-telescope′*: $(\sum i{\le}k.\ f\ (Suc\ i) - f\ i\ ) = f\ (Suc\ k) - (f\ 0 :: 'c :: ab\text{-}group\text{-}add)$
using *sum-telescope*[*of* $-f$ *k*]
by *auto*

lemma *L-pow-eq*:
assumes *ν-improving v* (*mk-dec-det d*)
shows *L-pow v d m* $= v + (\sum i \le m.\ ((l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d))\widehat{\ }\widehat{\ }i))\ (B_b\ v)$
proof −
let *?d* = (*mk-dec-det d*)
have $(\sum i \le m.\ ((l *_R \mathcal{P}_1\ ?d)\widehat{\ }\widehat{\ }i))\ (B_b\ v) = (\sum i \le m.\ ((l *_R \mathcal{P}_1\ ?d)\widehat{\ }\widehat{\ }i))\ (L\ ?d\ v) - (\sum i \le m.\ ((l *_R \mathcal{P}_1\ ?d)\widehat{\ }\widehat{\ }i))\ v$
using *assms*
by (*auto simp*: B_b-*eq*-$\mathcal{L}_b$ *blinfun.bilinear-simps ν-improving-imp-*$\mathcal{L}_b$)
also have $\ldots = (\sum i \le m.\ ((l *_R \mathcal{P}_1\ ?d)\widehat{\ }\widehat{\ }i))\ (r\text{-}dec_b\ ?d) + (\sum i \le m.\ ((l *_R \mathcal{P}_1\ ?d)\widehat{\ }\widehat{\ }i))\ ((l *_R \mathcal{P}_1\ ?d)\ v) - (\sum i \le m.\ ((l *_R \mathcal{P}_1\ ?d)\widehat{\ }\widehat{\ }i))\ v$
by (*simp add*: *L-def blinfun.bilinear-simps*)

also have ... $= (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i))\ (r\text{-}dec_b\ ?d) + (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}Suc\ i))\ v - (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i))\ v$
by (*auto simp*: *blinfun.sum-left blinfunpow-assoc simp del*: *blinfunpow.simps*)
also have ... $= (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i))\ (r\text{-}dec_b\ ?d) + (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}Suc\ i) - (l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i)\ v$
by (*simp add*: *blinfun.diff-left sum-subtractf*)
also have ... $= (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i))\ (r\text{-}dec_b\ ?d) + ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}Suc\ m)\ v - v$
by (*subst sum-telescope'*) (*auto simp*: *blinfun.bilinear-simps*)
finally have $(\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i))\ (B_b\ v) = (\sum i \leq m.\ ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}i))\ (r\text{-}dec_b\ ?d) + ((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}Suc\ m)\ v - v$ **.**
moreover have *L-pow v d m* = ν_b*-fin* (*mk-stationary-det d*) (*Suc m*) + $((l *_R \mathcal{P}_1\ ?d)^{\frown\frown}Suc\ m)\ v$
by (*simp only*: *L-pow-def L-iter lessThan-Suc-atMost*[*symmetric*])
ultimately show *?thesis*
by (*auto simp*: ν_b*-fin-eq lessThan-Suc-atMost*)
qed

lemma *L-pow-eq-W*:
assumes $d \in D_D$
shows *L-pow v* (*policy-improvement d v*) *m* = *W* (*mk-dec-det* (*policy-improvement d v*)) (*Suc m*) *v*
using *assms policy-improvement-improving*
by (*auto simp*: *W-eq-L-iter L-pow-def*)

lemma *L-pow-*$\mathcal{L}_b$*-mono-inv*:
assumes $d \in D_D$ $v \leq \mathcal{L}_b\ v$
shows *L-pow v* (*policy-improvement d v*) *m* $\leq \mathcal{L}_b$ (*L-pow v* (*policy-improvement d v*) *m*)
using *assms L-pow-eq-W* $\mathcal{L}_b$*-W-ge policy-improvement-improving*
by *auto*

3.5 The Modified Policy Iteration Algorithm

context
fixes *d0* :: $'s \Rightarrow 'a$
fixes *v0* :: $'s \Rightarrow_b real$
fixes *m* :: $nat \Rightarrow ('s \Rightarrow_b real) \Rightarrow nat$
assumes *d0*: $d0 \in D_D$
begin

We first define a function that executes the algorithm for n steps.

fun *mpi* :: $nat \Rightarrow (('s \Rightarrow 'a) \times ('s \Rightarrow_b real))$ **where**
mpi 0 = (*policy-improvement d0 v0*, *v0*) |
mpi (*Suc n*) =
(*let* (*d*, *v*) = *mpi n*; *v'* = *L-pow v d* (*m n v*) *in*
(*policy-improvement d v'*, *v'*))

definition *mpi-val* n = *snd* (*mpi* n)
definition *mpi-pol* n = *fst* (*mpi* n)

lemma *mpi-pol-zero*[*simp*]: *mpi-pol 0* = *policy-improvement d0 v0*
  unfolding *mpi-pol-def*
  by *auto*

lemma *mpi-pol-Suc*: *mpi-pol* (*Suc* n) = *policy-improvement* (*mpi-pol* n) (*mpi-val* (*Suc* n))
  by (*auto simp*: *case-prod-beta'* *Let-def mpi-pol-def mpi-val-def*)

lemma *mpi-pol-is-dec-det*: *mpi-pol* $n \in D_D$
  unfolding *mpi-pol-def*
  using *policy-improvement-is-dec-det d0*
  by (*induction* n) (*auto simp*: *Let-def split*: *prod.splits*)

lemma ν*-improving-mpi-pol*: ν*-improving* (*mpi-val* n) (*mk-dec-det* (*mpi-pol* n))
  using *d0 policy-improvement-improving mpi-pol-is-dec-det mpi-pol-Suc*
  by (*cases* n) (*auto simp*: *mpi-pol-def mpi-val-def*)

lemma *mpi-val-zero*[*simp*]: *mpi-val 0* = *v0*
  unfolding *mpi-val-def* **by** *auto*

lemma *mpi-val-Suc*: *mpi-val* (*Suc* n) = *L-pow* (*mpi-val* n) (*mpi-pol* n) (m n (*mpi-val* n))
  unfolding *mpi-val-def mpi-pol-def*
  by (*auto simp*: *case-prod-beta'* *Let-def*)

lemma *mpi-val-eq*: *mpi-val* (*Suc* n) =
  mpi-val n + ($\sum i \leq m\ n$ (*mpi-val* n). ($l *_R \mathcal{P}_1$ (*mk-dec-det* (*mpi-pol* n))) ^^ i) (B_b (*mpi-val* n))
  using *L-pow-eq*[*OF* ν*-improving-mpi-pol*] *mpi-val-Suc*
  by *auto*

Value Iteration is a special case of MPI where $\forall\, n\ v.\ m\ n\ v = 0$.

lemma *mpi-includes-value-it*:
  assumes $\forall\, n\ v.\ m\ n\ v = 0$
  shows *mpi-val* (*Suc* n) = $\mathcal{L}_b$ (*mpi-val* n)
  using *assms* B_b*-eq-*$\mathcal{L}_b$ *mpi-val-eq*
  by *auto*

3.6 Convergence Proof

We define the sequence w as an upper bound for the values of MPI.

fun w **where**
  w *0* = *v0* |

$w\ (Suc\ n) = U_b\ (Suc\ (m\ n\ (mpi\text{-}val\ n)))\ (w\ n)$

lemma *dist-ν_b-opt*: $dist\ (w\ (Suc\ n))\ \nu_b\text{-}opt \leq l * dist\ (w\ n)\ \nu_b\text{-}opt$
by (*fastforce simp*: *algebra-simps intro*: *order.trans*[*OF dist-U_b-opt*] *mult-left-mono power-le-one*
mult-left-le-one-le order.strict-implies-order)

lemma *dist-ν_b-opt-n*: $dist\ (w\ n)\ \nu_b\text{-}opt \leq l\hat{\ }n * dist\ v0\ \nu_b\text{-}opt$
by (*induction n*) (*fastforce simp*: *algebra-simps intro*: *order.trans*[*OF dist-ν_b-opt*] *mult-left-mono*)+

lemma *w-conv*: $w \longrightarrow \nu_b\text{-}opt$
proof −
have $(\lambda n.\ l\hat{\ }n * dist\ v0\ \nu_b\text{-}opt) \longrightarrow 0$
using *LIMSEQ-realpow-zero*
by (*cases* $v0 = \nu_b\text{-}opt$) *auto*
then show *?thesis*
by (*fastforce intro*: *metric-LIMSEQ-I order.strict-trans1*[*OF dist-ν_b-opt-n*] *simp*: *LIMSEQ-def*)
qed

MPI converges monotonically to the optimal value from below. The iterates are sandwiched between $\mathcal{L}_b$ from below and U_b from above.

theorem *mpi-conv*:
assumes $v0 \leq \mathcal{L}_b\ v0$
shows $mpi\text{-}val \longrightarrow \nu_b\text{-}opt$ **and** $\bigwedge n.\ mpi\text{-}val\ n \leq mpi\text{-}val\ (Suc\ n)$
proof −
define y **where** $y\ n = (\mathcal{L}_b\ \hat{\ }\hat{\ }\ n)\ v0$ **for** n
have *aux*: $mpi\text{-}val\ n \leq \mathcal{L}_b\ (mpi\text{-}val\ n) \wedge mpi\text{-}val\ n \leq mpi\text{-}val\ (Suc\ n) \wedge y\ n \leq mpi\text{-}val\ n \wedge mpi\text{-}val\ n \leq w\ n$ **for** n
proof (*induction n*)
case *0*
show *?case*
using *assms B_b-eq-$\mathcal{L}_b$*
unfolding *y-def*
by (*auto simp*: *mpi-val-eq blinfun.sum-left $\mathcal{P}_1$-n-disc-pos blincomp-scaleR-right sum-nonneg*)
next
case (*Suc n*)
have *val-eq-W*: $mpi\text{-}val\ (Suc\ n) = W\ (mk\text{-}dec\text{-}det\ (mpi\text{-}pol\ n))\ (Suc\ (m\ n\ (mpi\text{-}val\ n)))\ (mpi\text{-}val\ n)$
using *ν-improving-mpi-pol mpi-val-Suc W-eq-L-iter L-pow-def*
by *auto*
hence $*$: $mpi\text{-}val\ (Suc\ n) \leq \mathcal{L}_b\ (mpi\text{-}val\ (Suc\ n))$
using *Suc.IH $\mathcal{L}_b$-W-ge ν-improving-mpi-pol* **by** *presburger*
moreover have $mpi\text{-}val\ (Suc\ n) \leq mpi\text{-}val\ (Suc\ (Suc\ n))$
using $*$
by (*simp add*: *B_b-eq-$\mathcal{L}_b$ mpi-val-eq $\mathcal{P}_1$-n-disc-pos blincomp-scaleR-right*

blinfun.sum-left sum-nonneg)
  moreover have *mpi-val* (*Suc n*) $\leq$ *w* (*Suc n*)
    using *Suc.IH* ν*-improving-mpi-pol*
    by (*auto simp*: *val-eq-W intro*: *order.trans*[*OF - W-le-*U_b])
  moreover have *y* (*Suc n*) $\leq$ *mpi-val* (*Suc n*)
    using *Suc.IH* ν*-improving-mpi-pol W-ge-*$\mathcal{L}_b$
    by (*auto simp*: *y-def* B_b*-eq-*$\mathcal{L}_b$ *val-eq-W*)
  ultimately show *?case*
    by *auto*
 qed
 thus *mpi-val n* $\leq$ *mpi-val* (*Suc n*) **for** *n*
  by *auto*
 have *y* $\longrightarrow$ ν_b*-opt*
  using $\mathcal{L}_b$*-lim y-def* **by** *presburger*
 thus *mpi-val* $\longrightarrow$ ν_b*-opt*
  using *aux*
  by (*auto intro*: *tendsto-bfun-sandwich*[*OF - w-conv*])
qed

3.7 ϵ-Optimality

This gives an upper bound on the error of MPI.

lemma *mpi-pol-eps-opt*:
 assumes *2* $*$ *l* $*$ *dist* (*mpi-val n*) ($\mathcal{L}_b$ (*mpi-val n*)) < *eps* $*$ (*1* − *l*) *eps* > *0*
 shows *dist* (ν_b (*mk-stationary-det* (*mpi-pol n*))) ($\mathcal{L}_b$ (*mpi-val n*)) $\leq$ *eps* / *2*
proof −
 let *?p* = *mk-stationary-det* (*mpi-pol n*)
 let *?d* = *mk-dec-det* (*mpi-pol n*)
 let *?v* = *mpi-val n*
 have *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) = *dist* (*L ?d* (ν_b *?p*)) ($\mathcal{L}_b$ *?v*)
  using *L-*ν*-fix*
  by *force*
 also have ... = *dist* (*L ?d* (ν_b *?p*)) (*L ?d ?v*)
  by (*metis* ν*-improving-imp-*$\mathcal{L}_b$ ν*-improving-mpi-pol*)
 also have ... $\leq$ *dist* (*L ?d* (ν_b *?p*)) (*L ?d* ($\mathcal{L}_b$ *?v*)) + *dist* (*L ?d* ($\mathcal{L}_b$ *?v*)) (*L ?d ?v*)
  using *dist-triangle*
  by *blast*
 also have ... $\leq$ *l* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) + *dist* (*L ?d* ($\mathcal{L}_b$ *?v*)) (*L ?d ?v*)
  using *contraction-L* **by** *auto*
 also have ... $\leq$ *l* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) + *l* $*$ *dist* ($\mathcal{L}_b$ *?v*) *?v*
  using *contraction-L* **by** *auto*
 finally have *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) $\leq$ *l* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) + *l* $*$ *dist* ($\mathcal{L}_b$ *?v*) *?v*.
 hence $*$:(*1*−*l*) $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) $\leq$ *l* $*$ *dist* ($\mathcal{L}_b$ *?v*) *?v*
  by (*auto simp*: *left-diff-distrib*)

thus *?thesis*
proof (*cases* $l = 0$)
case *True*
thus *?thesis*
using *assms* $*$
by *auto*
next
case *False*
have $**$: *dist* ($\mathcal{L}_b$ *?v*) (*mpi-val n*) $<$ *eps* $*$ ($1 - l$) / ($2 * l$)
using *False le-neq-trans*[*OF zero-le-disc False*[*symmetric*]] *assms*
by (*auto simp*: *dist-commute pos-less-divide-eq Groups.mult-ac*(*2*))
have *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) $\leq$ (l/ ($1-l$)) $*$ *dist* ($\mathcal{L}_b$ *?v*) *?v*
using $*$
by (*auto simp*: *mult.commute pos-le-divide-eq*)
also have $\ldots \leq$ (l/ ($1-l$)) $*$ (*eps* $*$ ($1 - l$) / ($2 * l$))
using $**$
by (*fastforce intro*!: *mult-left-mono simp*: *divide-nonneg-pos*)
also have $\ldots =$ *eps* / *2*
using *False disc-lt-one*
by (*auto simp*: *order.strict-iff-order*)
finally show *dist* (ν_b *?p*) ($\mathcal{L}_b$ *?v*) $\leq$ *eps* / *2*.
qed
qed

lemma *mpi-pol-opt*:
assumes $2 * l *$ *dist* (*mpi-val n*) ($\mathcal{L}_b$ (*mpi-val n*)) $<$ *eps* $*$ ($1 - l$)
eps > 0
shows *dist* (ν_b (*mk-stationary-det* (*mpi-pol n*))) (ν_b-*opt*) $<$ *eps*
proof −
have *dist* (ν_b (*mk-stationary-det* (*mpi-pol n*))) (ν_b-*opt*) $\leq$ *eps*/*2* +
dist ($\mathcal{L}_b$ (*mpi-val n*)) ν_b-*opt*
by (*metis mpi-pol-eps-opt*[*OF assms*] *dist-commute dist-triangle-le*
add-right-mono)
thus *?thesis*
using *dist*-$\mathcal{L}_b$-*opt-eps assms*
by *fastforce*
qed

lemma *mpi-val-term-ex*:
assumes *v0* $\leq \mathcal{L}_b$ *v0 eps* > 0
shows $\exists$ *n*. $2 * l *$ *dist* (*mpi-val n*) ($\mathcal{L}_b$ (*mpi-val n*)) $<$ *eps* $*$ ($1 - l$)
proof −
note *dist*-$\mathcal{L}_b$-*lt-dist-opt*
have (λ*n*. *dist* (*mpi-val n*) ν_b-*opt*) $\longrightarrow$ *0*
using *mpi-conv*(*1*)[*OF assms*(*1*)] *tendsto-dist-iff*
by *blast*
hence (λ*n*. *dist* (*mpi-val n*) ($\mathcal{L}_b$ (*mpi-val n*))) $\longrightarrow$ *0*
using *dist*-$\mathcal{L}_b$-*lt-dist-opt*
by (*auto simp*: *metric-LIMSEQ-I intro*: *tendsto-sandwich*[*of* λ-. *0*

- - $\lambda n.\ 2 * dist\ (mpi\text{-}val\ n)\ \nu_b\text{-}opt])$
hence $\forall\ e > 0.\ \exists\ n.\ dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) < e$
by (*fastforce dest*!: *metric-LIMSEQ-D*)
hence $l \neq 0 \Longrightarrow \exists\ n.\ dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) < eps * (1 - l)\ /\ (2 * l)$
by (*simp add*: *assms order.not-eq-order-implies-strict*)
thus $\exists\ n.\ (2 * l) * dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) < eps * (1 - l)$
using *assms le-neq-trans*[*OF zero-le-disc*]
by (*cases* $l = 0$) (*auto simp*: *mult.commute pos-less-divide-eq*)
qed
end

3.8 Unbounded MPI

context
fixes *eps* δ :: *real* **and** *M* :: *nat*
begin

function (*domintros*) *mpi-algo* **where** $mpi\text{-}algo\ d\ v\ m = ($
$if\ 2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1 - l)$
$then\ (policy\text{-}improvement\ d\ v,\ v)$
$else\ mpi\text{-}algo\ (policy\text{-}improvement\ d\ v)\ (L\text{-}pow\ v\ (policy\text{-}improvement\ d\ v)\ (m\ 0\ v))\ (\lambda n.\ m\ (Suc\ n)))$
by *auto*

We define a tailrecursive version of *mpi* which more closely resembles *mpi-algo*.

fun *mpi'* **where**
$mpi'\ d\ v\ 0\ m = (policy\text{-}improvement\ d\ v,\ v)\ |$
$mpi'\ d\ v\ (Suc\ n)\ m = ($
$let\ d' = policy\text{-}improvement\ d\ v;\ v' = L\text{-}pow\ v\ d'\ (m\ 0\ v)\ in\ mpi'\ d'\ v'\ n\ (\lambda n.\ m\ (Suc\ n)))$

lemma *mpi-Suc'*:
assumes $d \in D_D$
shows $mpi\ d\ v\ m\ (Suc\ n) = mpi\ (policy\text{-}improvement\ d\ v)\ (L\text{-}pow\ v\ (policy\text{-}improvement\ d\ v)\ (m\ 0\ v))\ (\lambda a.\ m\ (Suc\ a))\ n$
using *assms policy-improvement-is-dec-det*
by (*induction n rule*: *nat.induct*) (*auto simp*: *Let-def*)

lemma
assumes $d \in D_D$
shows $mpi\ d\ v\ m\ n = mpi'\ d\ v\ n\ m$
using *assms*
proof (*induction n arbitrary*: *d v m rule*: *nat.induct*)
case (*Suc nat*)
thus *?case*
using *policy-improvement-is-dec-det*
by (*auto simp*: *Let-def mpi-Suc'*[*OF Suc*(*2*)] *Suc.IH*[*symmetric*])

qed *auto*

lemma *termination-mpi-algo*:
assumes $eps > 0$ $d \in D_D$ $v \le \mathcal{L}_b\ v$
shows *mpi-algo-dom* (*d*, *v*, *m*)
proof −
define *n* **where** $n = (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ d\ v\ m\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ d\ v\ m\ n)) < eps * (1 - l))$ (**is** $n = (LEAST\ n.\ ?P\ d\ v\ m\ n)$)
have *least0*: $\exists\, n.\ P\ n \Longrightarrow (LEAST\ n.\ P\ n) = (0 :: nat) \Longrightarrow P\ 0$ **for** *P*
by (*metis LeastI-ex*)
from *n-def assms* **show** *?thesis*
proof (*induction n arbitrary*: *v d m*)
case *0*
have $2 * l * dist\ (mpi\text{-}val\ d\ v\ m\ 0)\ (\mathcal{L}_b\ (mpi\text{-}val\ d\ v\ m\ 0)) < eps * (1 - l)$
using *least0 mpi-val-term-ex 0*
by (*metis* (*no-types*, *lifting*))
thus *?case*
using *0 mpi-algo.domintros mpi-val-zero*
by (*metis* (*no-types*, *opaque-lifting*))
next
case (*Suc n v d m*)
let *?d* = *policy-improvement d v*
have $Suc\ n = Suc\ (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ d\ v\ m\ (Suc\ n))\ (\mathcal{L}_b\ (mpi\text{-}val\ d\ v\ m\ (Suc\ n))) < eps * (1 - l))$
using *mpi-val-term-ex*[*OF Suc.prems*(*3*) ‹$v \le \mathcal{L}_b\ v$› ‹$0 < eps$›, *of m*] *Suc.prems*
by (*subst Nat.Least-Suc*[*symmetric*]) (*auto intro*: *LeastI-ex*)
hence $n = (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ d\ v\ m\ (Suc\ n))\ (\mathcal{L}_b\ (mpi\text{-}val\ d\ v\ m\ (Suc\ n))) < eps * (1 - l))$
by *auto*
hence *n-eq*: $n = (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ ?d\ (L\text{-}pow\ v\ ?d\ (m\ 0\ v))\ (\lambda a.\ m\ (Suc\ a))\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ ?d\ (L\text{-}pow\ v\ ?d\ (m\ 0\ v))\ (\lambda a.\ m\ (Suc\ a))\ n)) < eps * (1 - l))$
using *Suc.prems mpi-Suc′*
by (*auto simp*: *is-dec-det-pi mpi-val-def*)
have $\neg\ 2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1 - l)$
using *Suc mpi-val-zero* **by** *force*
moreover have *mpi-algo-dom* (*?d*, $L\text{-}pow\ v\ ?d\ (m\ 0\ v)$, $\lambda a.\ m\ (Suc\ a)$)
using *Suc.IH*[*OF n-eq* ‹$0 < eps$›] *Suc.prems is-dec-det-pi* $L\text{-}pow\text{-}\mathcal{L}_b\text{-}mono\text{-}inv$ **by** *auto*
ultimately show *?case*
using *mpi-algo.domintros*
by *blast*
qed
qed

abbreviation *mpi-alg-rec d v m* ≡
(*if* $2 * l *$ *dist v* ($\mathcal{L}_b$ *v*) < *eps* $* (1 - l)$ *then* (*policy-improvement d v*, *v*)
else mpi-algo (*policy-improvement d v*) (*L-pow v* (*policy-improvement d v*) (*m 0 v*))
(λ*n*. *m* (*Suc n*)))

lemma *mpi-algo-def′*:
assumes *d* ∈ D_D *v* ≤ $\mathcal{L}_b$ *v eps* > *0*
shows *mpi-algo d v m* = *mpi-alg-rec d v m*
using *mpi-algo.psimps termination-mpi-algo assms*
by *auto*

lemma *mpi-algo-eq-mpi*:
assumes *d* ∈ D_D *v* ≤ $\mathcal{L}_b$ *v eps* > *0*
shows *mpi-algo d v m* = *mpi d v m* (*LEAST n*. $2 * l *$ *dist* (*mpi-val d v m n*) ($\mathcal{L}_b$ (*mpi-val d v m n*)) < *eps* $* (1 - l)$)
proof −
define *n* **where** *n* = (*LEAST n*. $2 * l *$ *dist* (*mpi-val d v m n*) ($\mathcal{L}_b$ (*mpi-val d v m n*)) < *eps* $* (1 - l)$) (**is** *n* = (*LEAST n*. *?P d v m n*))
from *n-def assms* **show** *?thesis*
proof (*induction n arbitrary*: *d v m*)
case *0*
have *?P d v m 0*
by (*metis* (*no-types*, *lifting*) *assms*(*3*) *LeastI-ex 0 mpi-val-term-ex*)
thus *?case*
using *assms 0*
by (*auto simp*: *mpi-val-def mpi-algo-def′*)
next
case (*Suc n*)
hence *not0*: ¬ ($2 * l *$ *dist v* ($\mathcal{L}_b$ *v*) < *eps* $* (1 - l)$)
using *Suc*(*3*) *mpi-val-zero*
by *auto*
obtain *n′* **where** $2 * l *$ *dist* (*mpi-val d v m n′*) ($\mathcal{L}_b$ (*mpi-val d v m n′*)) < *eps* $* (1 - l)$
using *mpi-val-term-ex*[*OF Suc*(*3*) *Suc*(*4*), *of - m*] *assms* **by** *blast*
hence *n* = (*LEAST n*. *?P d v m* (*Suc n*))
using *Suc*(*2*) *Suc*
by (*subst* (*asm*) *Least-Suc*) *auto*
hence *n* = (*LEAST n*. *?P* (*policy-improvement d v*) (*L-pow v* (*policy-improvement d v*) (*m 0 v*)) (λ*n*. *m* (*Suc n*)) *n*)
using *Suc*(*3*) *policy-improvement-is-dec-det mpi-Suc′*
by (*auto simp*: *mpi-val-def*)
hence *mpi-algo d v m* = *mpi d v m* (*Suc n*)
unfolding *mpi-algo-def′*[*OF Suc.prems*(*2*−*4*)]
using *Suc*(*1*) *Suc.prems*(*2*−*4*) *is-dec-det-pi mpi-Suc′ not0 L-pow-*$\mathcal{L}_b$*-mono-inv* **by** *force*
thus *?case*

using *Suc.prems*(*1*) **by** *presburger*
qed
qed

lemma *mpi-algo-opt*:
assumes $v0 \leq \mathcal{L}_b$ *v0 eps* > 0 $d \in D_D$
shows *dist* (ν_b (*mk-stationary-det* (*fst* (*mpi-algo d v0 m*)))) ν_b*-opt* $<$ *eps*
proof −
let $?P = \lambda n.\ 2 * l *$ *dist* (*mpi-val d v0 m n*) ($\mathcal{L}_b$ (*mpi-val d v0 m n*)) $<$ *eps* $* (1 - l)$
let $?n =$ *Least* $?P$
have *mpi-algo d v0 m* = *mpi d v0 m ?n* **and** *?P ?n*
using *mpi-algo-eq-mpi LeastI-ex*[*OF mpi-val-term-ex*] *assms* **by** *auto*
thus *?thesis*
using *assms*
by (*auto simp*: *mpi-pol-opt mpi-pol-def*[*symmetric*])
qed

end

3.9 Initial Value Estimate *v0-mpi*

We define an initial estimate of the value function for which Modified Policy Iteration always terminates.

abbreviation *r-min* $\equiv (\prod s'\ a.\ r\ (s', a))$
definition *v0-mpi s* = *r-min* / $(1 - l)$

lift-definition $v0\text{-}mpi_b :: {}'s \Rightarrow_b real$ **is** *v0-mpi*
by *fastforce*

lemma $v0\text{-}mpi_b\text{-}le\text{-}\mathcal{L}_b$: $v0\text{-}mpi_b \leq \mathcal{L}_b\ v0\text{-}mpi_b$
proof (*rule less-eq-bfunI*)
fix *x*
have *r-min* $\leq r\ (s, a)$ **for** *s a*
by (*fastforce intro*: *cInf-lower2*)
hence *r-min* $\leq (1{-}l) * r\ (s, a) + l *$ *r-min* **for** *s a*
using *disc-lt-one zero-le-disc*
by (*meson order-less-imp-le order-refl segment-bound-lemma*)
hence *r-min* / $(1 - l) \leq ((1{-}l) * r\ (s, a) + l *$ *r-min*) / $(1 - l)$ **for** *s a*
using *order-less-imp-le*[*OF disc-lt-one*]
by (*auto intro*!: *divide-right-mono*)
hence *r-min* / $(1 - l) \leq r\ (s, a) + (l *$ *r-min*) / $(1 - l)$ **for** *s a*
using *disc-lt-one*
by (*auto simp*: *add-divide-distrib*)
thus $v0\text{-}mpi_b\ x \leq \mathcal{L}_b\ v0\text{-}mpi_b\ x$
unfolding $\mathcal{L}_b$*-eq-SUP-*L_a $v0\text{-}mpi_b$*.rep-eq v0-mpi-def*

by (*auto simp*: *A-ne intro*: *cSUP-upper2*[**where** x = *arb-act* (A x)])
qed

3.10 An Instance of Modified Policy Iteration with a Valid Conservative Initial Value Estimate

definition *mpi-user eps m* = (
if eps $\leq$ *0 then undefined else mpi-algo eps* (λx. *arb-act* (A x)) *v0-mpi*$_b$ *m*)

lemma *mpi-user-eq*:
assumes *eps* > *0*
shows *mpi-user eps* = *mpi-alg-rec eps* (λx. *arb-act* (A x)) *v0-mpi*$_b$
using *v0-mpi*$_b$*-le-*$\mathcal{L}_b$ *assms*
by (*auto simp*: *mpi-user-def mpi-algo-def′ A-ne is-dec-det-def*)

lemma *mpi-user-opt*:
assumes *eps* > *0*
shows *dist* (ν_b (*mk-stationary-det* (*fst* (*mpi-user eps n*)))) ν_b*-opt* < *eps*
unfolding *mpi-user-def* **using** *assms*
by (*auto intro*: *mpi-algo-opt simp*: *is-dec-det-def A-ne v0-mpi*$_b$*-le-*$\mathcal{L}_b$)

end

end
theory *Matrix-Util*
imports *HOL−Analysis.Analysis*
begin

4 Matrices

proposition *scalar-matrix-assoc′*:
fixes C :: (*′b*::*real-algebra-1*)^*′m*^*′n*
shows $k *_R (C ** D) = C ** (k *_R D)$
by (*simp add*: *matrix-matrix-mult-def sum-distrib-left mult-ac vec-eq-iff scaleR-sum-right*)

4.1 Nonnegative Matrices

lemma *nonneg-matrix-nonneg* [*dest*]: $0 \leq m \Longrightarrow 0 \leq m \$ i \$ j$
by (*simp add*: *Finite-Cartesian-Product.less-eq-vec-def*)

lemma *matrix-mult-mono*:
assumes $0 \leq E$ $0 \leq C$ (E :: *real*^*′c*^*′c*) $\leq B$ $C \leq D$
shows $E ** C \leq B ** D$
using *order.trans*[*OF assms*(*1*) *assms*(*3*)] *assms*
unfolding *Finite-Cartesian-Product.less-eq-vec-def*

by (*auto intro*!: *sum-mono mult-mono simp*: *matrix-matrix-mult-def*)

lemma *nonneg-matrix-mult*: $0 \le (C :: ('b::\{field,\ ordered\text{-}ring\})\hat{\ }\text{-}\hat{\ }\text{-})$ $\Longrightarrow 0 \le D \Longrightarrow 0 \le C ** D$
unfolding *Finite-Cartesian-Product.less-eq-vec-def*
by (*auto simp*: *matrix-matrix-mult-def intro*!: *sum-nonneg*)

lemma *zero-le-mat-iff* [*simp*]: $0 \le mat\ (x :: 'c :: \{zero,\ order\}) \longleftrightarrow$ $0 \le x$
by (*auto simp*: *Finite-Cartesian-Product.less-eq-vec-def mat-def*)

lemma *nonneg-mat-ge-zero*: $0 \le Q \Longrightarrow 0 \le v \Longrightarrow 0 \le Q *v\ (v ::$ $real\hat{\ }'c)$
unfolding *Finite-Cartesian-Product.less-eq-vec-def*
by (*auto intro*!: *sum-nonneg simp*: *matrix-vector-mult-def*)

lemma *nonneg-mat-mono*: $0 \le Q \Longrightarrow u \le v \Longrightarrow Q *v\ u \le Q *v\ (v$ $::\ real\hat{\ }'c)$
using *nonneg-mat-ge-zero*[*of* $Q\ v - u$]
by (*simp add*: *vec.diff*)

lemma *nonneg-mult-imp-nonneg-mat*:
assumes $\bigwedge v.\ v \ge 0 \Longrightarrow X *v\ v \ge 0$
shows $X \ge (0 :: real\ \hat{\ }\text{-}\ \hat{\ }\text{-})$
proof –
{ assume $\neg\ (0 \le X)$
then obtain $i\ j$ **where** *neg*: $X\ \$\ i\ \$\ j < 0$
by (*metis less-eq-vec-def not-le zero-index*)
let $?v = \chi\ k.\ if\ j = k\ then\ 1{::}real\ else\ 0$
have $(X *v\ ?v)\ \$\ i < 0$
using *neg*
by (*auto simp*: *matrix-vector-mult-def if-distrib cong*: *if-cong*)
hence $?v \ge 0 \wedge \neg\ ((X *v\ ?v) \ge 0)$
by (*auto simp*: *less-eq-vec-def not-le*)
hence $\exists\, v.\ v \ge 0 \wedge \neg\ X *v\ v \ge 0$
by *blast*
}
thus *?thesis*
using *assms* **by** *auto*
qed

lemma *nonneg-mat-iff*:
$(X \ge (0 :: real\ \hat{\ }\text{-}\ \hat{\ }\text{-})) \longleftrightarrow (\forall\, v.\ v \ge 0 \longrightarrow X *v\ v \ge 0)$
using *nonneg-mat-ge-zero nonneg-mult-imp-nonneg-mat* **by** *auto*

lemma *mat-le-iff*: $(X \le Y) \longleftrightarrow (\forall\, x{\ge}0.\ (X{::}real\hat{\ }\text{-}\hat{\ }\text{-}) *v\ x \le Y *v$ $x)$
by (*metis diff-ge-0-iff-ge matrix-vector-mult-diff-rdistrib nonneg-mat-iff*)

4.2 Matrix Powers

primrec *matpow* :: $'a$::*semiring-1*$^\frown$$'n$$^\frown$$'n \Rightarrow nat \Rightarrow$ $'a$$^\frown$$'n$$^\frown$$'n$ **where**
matpow-0: *matpow A 0* = *mat 1* |
matpow-Suc: *matpow A* (*Suc n*) = (*matpow A n*) ** *A*

lemma *nonneg-matpow*: $0 \leq X \Longrightarrow 0 \leq$ *matpow* (X :: *real* $^\frown$ - $^\frown$ -) i
by (*induction i*) (*auto simp*: *nonneg-matrix-mult*)

lemma *matpow-mono*: $0 \leq C \Longrightarrow C \leq D \Longrightarrow$ *matpow* (C :: *real*$^\frown$-$^\frown$-) $n \leq$ *matpow D n*
by (*induction n*) (*auto intro*!: *matrix-mult-mono nonneg-matpow*)

lemma *matpow-scaleR*: *matpow* ($c *_R$ (X :: $'b$ :: *real-algebra-1*$^\frown$-$^\frown$-)) n = ($c$$^\frown$$n$) $*_R$ (*matpow X*) n
proof (*induction n arbitrary*: *X c*)
case (*Suc n*)
have *matpow* ($c *_R X$) (*Suc n*) = ($c$$^\frown$$n$)$*_R$ (*matpow X*) n ** $c *_R X$
using *Suc* **by** *auto*
also have ... = $c *_R$ (($c$$^\frown$$n$) $*_R$ (*matpow X*) n ** X)
using *scalar-matrix-assoc'*
by (*auto simp*: *scalar-matrix-assoc'*)
finally show *?case*
by (*simp add*: *scalar-matrix-assoc*)
qed *auto*

lemma *matrix-vector-mult-code'*: (X *v x) \$ i = ($\sum j \in UNIV$. X \$ i \$ j * x \$ j)
by (*simp add*: *matrix-vector-mult-def*)

lemma *matrix-vector-mult-mono*: (*0*::*real*$^\frown$-$^\frown$-) $\leq X \Longrightarrow 0 \leq v \Longrightarrow X \leq Y \Longrightarrow X$ *v $v \leq Y$ *v v
by (*metis diff-ge-0-iff-ge matrix-vector-mult-diff-rdistrib nonneg-mat-iff*)

4.3 Triangular Matrices

definition *lower-triangular-mat X* $\longleftrightarrow$ ($\forall$ *i j*. (*i* :: $'b$::{*finite*, *linorder*}) $< j \longrightarrow X$ \$ i \$ j = *0*)

definition *strict-lower-triangular-mat X* $\longleftrightarrow$ ($\forall$ *i j*. (*i* :: $'b$::{*finite*, *linorder*}) $\leq j \longrightarrow X$ \$ i \$ j = *0*)

definition *upper-triangular-mat X* $\longleftrightarrow$ ($\forall$ *i j*. $j < i \longrightarrow X$ \$ i \$ j = *0*)

lemma *stlI*: *strict-lower-triangular-mat X* $\Longrightarrow$ *lower-triangular-mat X*
unfolding *strict-lower-triangular-mat-def lower-triangular-mat-def*
by *auto*

lemma *lower-triangular-mat-mat*: *lower-triangular-mat* (*mat x*)
unfolding *lower-triangular-mat-def mat-def*
by *auto*

lemma *lower-triangular-mult*:
assumes *lower-triangular-mat X lower-triangular-mat Y*
shows *lower-triangular-mat* ($X ** Y$)
using *assms*
unfolding *matrix-matrix-mult-def lower-triangular-mat-def*
by (*auto intro*!: *sum.neutral*) (*metis mult-not-zero neqE less-trans*)

lemma *lower-triangular-pow*:
assumes *lower-triangular-mat X*
shows *lower-triangular-mat* (*matpow X i*)
using *assms lower-triangular-mult lower-triangular-mat-mat*
by (*induction i*) *auto*

lemma *lower-triangular-suminf*:
assumes $\bigwedge i$. *lower-triangular-mat* (f i) *summable* (f :: *nat* $\Rightarrow$ $'b$::*real-normed-vector*^'-^'-)
shows *lower-triangular-mat* ($\sum i.\ f\ i$)
using *assms*
unfolding *lower-triangular-mat-def*
by (*subst bounded-linear.suminf*) (*auto intro*: *bounded-linear-compose*)

lemma *lower-triangular-pow-eq*:
assumes *lower-triangular-mat X lower-triangular-mat Y* $\bigwedge s'.\ s' \le s \Longrightarrow row\ s'\ X = row\ s'\ Y$ $s' \le s$
shows *row* s' (*matpow X i*) = *row* s' (*matpow Y i*)
using *assms*
proof (*induction i*)
case (*Suc i*)
thus *?case*
proof −
have *ltX*: *lower-triangular-mat* (*matpow X i*)
by (*simp add*: *Suc*(*2*) *lower-triangular-pow*)
have *ltY*: *lower-triangular-mat* (*matpow Y i*)
by (*simp add*: *Suc*(*3*) *lower-triangular-pow*)
have ($\sum k{\in}UNIV.\ matpow\ X\ i\ \$\ s'\ \$\ k * X\ \$\ k\ \$\ j$) = ($\sum k{\in}UNIV.\ matpow\ Y\ i\ \$\ s'\ \$\ k * Y\ \$\ k\ \$\ j$) **for** j
proof −
have ($\sum k{\in}UNIV.\ matpow\ X\ i\ \$\ s'\ \$\ k * X\ \$\ k\ \$\ j$) = ($\sum k{\in}UNIV.$ *if* $s' < k$ *then 0 else* $matpow\ Y\ i\ \$\ s'\ \$\ k * X\ \$\ k\ \$\ j$)
using *Suc ltY*
by (*auto simp*: *row-def lower-triangular-mat-def intro*!: *sum.cong*)
also have ... = ($\sum k \in UNIV .\ matpow\ Y\ i\ \$\ s'\ \$\ k * Y\ \$\ k\ \$\ j$)
using *Suc ltY*
by (*auto simp*: *row-def lower-triangular-mat-def cong*: *if-cong*

intro!: *sum.cong*)
    finally show *?thesis*.
   qed
   thus *?thesis*
    by (*auto simp*: *row-def matrix-matrix-mult-def*)
  qed
qed *simp*

lemma *lower-triangular-mat-mult*:
  assumes *lower-triangular-mat M* $\bigwedge i.\ i \le j \Longrightarrow v\ \$\ i = v'\ \$\ i$
  shows $(M *v\ v)\ \$\ j = (M *v\ v')\ \$\ j$
proof −
  have $(M *v\ v)\ \$\ j = (\sum i{\in}UNIV.\ (\mathit{if}\ j < i\ \mathit{then}\ 0\ \mathit{else}\ \ M\ \$\ j\ \$\ i * v\ \$\ i))$
   using *assms* **unfolding** *lower-triangular-mat-def*
   by (*auto simp*: *matrix-vector-mult-def intro*!: *sum.cong*)
  also have $\ldots = (\sum i{\in}UNIV.\ (\mathit{if}\ j < i\ \mathit{then}\ 0\ \mathit{else}\ \ M\ \$\ j\ \$\ i * v'\ \$\ i))$
   using *assms*
   by (*auto intro*!: *sum.cong*)
  also have $\ldots = (M *v\ v')\ \$\ j$
   using *assms* **unfolding** *lower-triangular-mat-def*
   by (*auto simp*: *matrix-vector-mult-def intro*!: *sum.cong*)
  finally show *?thesis*.
qed

4.4 Inverses

lemma *matrix-inv*:
  assumes *invertible M*
  shows *matrix-inv-left*: *matrix-inv* $M ** M = \mathit{mat}\ 1$
   and *matrix-inv-right*: $M **$ *matrix-inv* $M = \mathit{mat}\ 1$
  using ‹*invertible M*› **and** *someI-ex* [*of* $\lambda\ N.\ M ** N = \mathit{mat}\ 1 \wedge N ** M = \mathit{mat}\ 1$]
  unfolding *invertible-def* **and** *matrix-inv-def*
  by *simp-all*

lemma *matrix-inv-unique*:
  fixes $A{::}'a{::}\{\mathit{semiring\text{-}1}\}\widehat{\ }'n\widehat{\ }'n$
  assumes *AB*: $A ** B = \mathit{mat}\ 1$ **and** *BA*: $B ** A = \mathit{mat}\ 1$
  shows *matrix-inv* $A = B$
  by (*metis AB BA invertible-def matrix-inv-right matrix-mul-assoc matrix-mul-lid*)

end
theory *Blinfun-Matrix*
  imports
   MDP−Rewards.Blinfun-Util

Matrix-Util
begin

5 Bounded Linear Functions and Matrices

definition *blinfun-to-matrix* (f :: ($'b$::*finite* $\Rightarrow_b$ *real*) $\Rightarrow_L$ ($'c$::*finite* $\Rightarrow_b$ -)) =
matrix ($\lambda v.$ (χ $j.$ f (*Bfun* (($) v)) j))

definition *matrix-to-blinfun* X = *Blinfun* ($\lambda v.$ *Bfun* ($\lambda i.$ (X $*v$ (χ $i.$ (*apply-bfun* v i))) \$ i))

lemma *plus-vec-eq*: (χ $i.$ f i + g i) = (χ $i.$ f i) + (χ $i.$ g i)
by (*simp add*: *Finite-Cartesian-Product.plus-vec-def*)

lemma *matrix-to-blinfun-mult*: *matrix-to-blinfun* m (v :: $'c$::*finite* $\Rightarrow_b$ *real*) i = (m $*v$ (χ $i.$ v i)) \$ i
proof −
have [*simp*]: (χ $i.$ $c * x$ i) = $c *_R$ (χ $i.$ x i) **for** c x
by (*simp add*: *vector-scalar-mult-def scalar-mult-eq-scaleR*[*symmetric*])

have *bounded-linear* ($\lambda v.$ *bfun.Bfun* (($) ($m$ $*v$ *vec-lambda* (*apply-bfun* v))))
proof (*rule bounded-linear-compose*[*of* $\lambda x.$ *bfun.Bfun* ($\lambda y.$ x \$ y)], *goal-cases*)
case *1*
then show *?case*
using *bounded-linear-bfun-nth*[*of id*, *simplified*] *bounded-linear-ident eq-id-iff*
by *metis*
next
case *2*
then show *?case*
using *norm-vec-le-norm-bfun*
by (*auto simp*: *matrix-vector-right-distrib plus-vec-eq*
intro!: *bounded-linear-intro bounded-linear-compose*[*OF matrix-vector-mul-bounded-linear*])
qed
thus *?thesis*
by (*auto simp*: *Blinfun-inverse matrix-to-blinfun-def Bfun-inverse*)
qed

lemma *blinfun-to-matrix-mult*: (*blinfun-to-matrix* f $*v$ (χ $i.$ *apply-bfun* v i)) \$ i = f v i
proof −
have (*blinfun-to-matrix* f $*v$ (χ $i.$ v i)) \$ i = ($\sum j{\in}UNIV.$ (f ((v j $*_R$ *bfun.Bfun* ($\lambda i.$ *if* $i = j$ *then* 1 *else* 0)))) i)
unfolding *blinfun-to-matrix-def matrix-def*
by (*auto simp*: *matrix-vector-mult-def mult.commute axis-def blin-*

fun.scaleR-right vec-lambda-inverse)
also have ... $= (\sum j{\in}UNIV.\ (f\ ((v\ j\ *_R\ bfun.Bfun\ (\lambda i.\ if\ i = j\ then\ 1\ else\ 0))))) \ i$
by (*auto intro*: *finite-induct*)
also have ... $= f\ (\sum j{\in}UNIV.\ (v\ j\ *_R\ bfun.Bfun\ (\lambda i.\ if\ i = j\ then\ 1\ else\ 0)))\ i$
by (*auto simp*: *blinfun.sum-right*)
also have ... $= f\ v\ i$
proof −
have $(\sum j{\in}UNIV.\ (v\ j\ *_R\ bfun.Bfun\ (\lambda i.\ if\ i = j\ then\ 1\ else\ 0)))\ x = v\ x$ **for** x
proof −
have $(\sum j{\in}UNIV.\ (v\ j\ *_R\ bfun.Bfun\ (\lambda i.\ if\ i = j\ then\ 1\ else\ 0)))\ x =$
$(\sum j{\in}UNIV.\ (v\ j\ *_R\ bfun.Bfun\ (\lambda i.\ if\ i = j\ then\ 1\ else\ 0)\ x))$
by (*auto intro*: *finite-induct*)
also have ... $= (\sum j{\in}UNIV.\ (v\ j\ *_R\ (\lambda i.\ if\ i = j\ then\ 1\ else\ 0)\ x))$
by (*subst Bfun-inverse*) (*metis vec-bfun vec-lambda-inverse*[*OF UNIV-I*, *symmetric*])+
also have ... $= (\sum j{\in}UNIV.\ ((if\ x = j\ then\ v\ j * 1\ else\ v\ j * 0)))$
by (*auto simp*: *if-distrib intro*!: *sum.cong*)
also have ... $= (\sum j{\in}UNIV.\ ((if\ x = j\ then\ v\ j\ else\ 0)))$
by (*meson more-arith-simps*(*6*) *mult-zero-right*)
also have ... $= v\ x$
by *auto*
finally show *?thesis*.
qed
thus *?thesis*
using *bfun-eqI*
by *fastforce*
qed
finally show *?thesis*.
qed

lemma *blinfun-to-matrix-mult′*: (*blinfun-to-matrix f* $*v$ *v*) \$ $i = f$ (*Bfun* ($\lambda i.\ v$ \$ i)) i
by (*metis bfun.Bfun-inverse blinfun-to-matrix-mult vec-bfun vec-nth-inverse*)

lemma *blinfun-to-matrix-mult′′*: (*blinfun-to-matrix f* $*v$ *v*) = (χ *i*. *f* (*Bfun* ($\lambda i.\ v$ \$ i)) *i*)
by (*metis blinfun-to-matrix-mult′ vec-lambda-unique*)

lemma *matrix-to-blinfun-inv*: *matrix-to-blinfun* (*blinfun-to-matrix f*) = *f*
by (*auto simp*: *matrix-to-blinfun-mult blinfun-to-matrix-mult intro*!: *blinfun-eqI*)

lemma *blinfun-to-matrix-add*: *blinfun-to-matrix* $(f + g) =$ *blinfun-to-matrix* $f +$ *blinfun-to-matrix* g
 by (*simp add*: *matrix-eq blinfun-to-matrix-mult*′′ *matrix-vector-mult-add-rdistrib blinfun.add-left plus-vec-eq*)

lemma *blinfun-to-matrix-diff*: *blinfun-to-matrix* $(f - g) =$ *blinfun-to-matrix* $f -$ *blinfun-to-matrix* g
  using *blinfun-to-matrix-add*
  by (*metis add-right-imp-eq diff-add-cancel*)

lemma *blinfun-to-matrix-scaleR*: *blinfun-to-matrix* $(c *_R f) = c *_R$ *blinfun-to-matrix* f
 by (*auto simp*: *matrix-eq blinfun-to-matrix-mult*′′ *scaleR-matrix-vector-assoc*[*symmetric*]

   blinfun.scaleR-left vector-scalar-mult-def[*of c*, *unfolded scalar-mult-eq-scaleR*])

lemma *matrix-to-blinfun-add*:
  matrix-to-blinfun ((*f* :: *real*^-^-) + *g*) = *matrix-to-blinfun f* + *matrix-to-blinfun g*
 by (*auto intro*!: *blinfun-eqI simp*: *matrix-to-blinfun-mult blinfun.add-left matrix-vector-mult-add-rdistrib*)

lemma *matrix-to-blinfun-diff*:
  matrix-to-blinfun ((*f* :: *real*^-^-) − *g*) = *matrix-to-blinfun f* − *matrix-to-blinfun g*
  using *matrix-to-blinfun-add diff-eq-eq*
  by *metis*

lemma *matrix-to-blinfun-scaleR*:
  matrix-to-blinfun ($c *_R$ (*f* :: *real*^-^-)) = $c *_R$ *matrix-to-blinfun f*
 by (*auto intro*!: *blinfun-eqI simp*: *matrix-to-blinfun-mult blinfun.scaleR-left*

   matrix-vector-mult-add-rdistrib scaleR-matrix-vector-assoc[*symmetric*])

lemma *matrix-to-blinfun-comp*: *matrix-to-blinfun* ((*m*:: *real*^-^-) ** *n*) = (*matrix-to-blinfun m*) o_L (*matrix-to-blinfun n*)
 by (*auto intro*!: *blinfun-eqI simp*: *matrix-vector-mul-assoc*[*symmetric*] *matrix-to-blinfun-mult*)

lemma *blinfun-to-matrix-comp*: *blinfun-to-matrix* (*f* o_L *g*) = (*blinfun-to-matrix f*) ** (*blinfun-to-matrix g*)
  by (*simp add*: *matrix-eq apply-bfun-inverse blinfun-to-matrix-mult*′′ *matrix-vector-mul-assoc*[*symmetric*])

lemma *blinfun-to-matrix-id*: *blinfun-to-matrix id-blinfun* = *mat 1*
  by (*simp add*: *Bfun-inverse blinfun-to-matrix-mult*′′ *matrix-eq*)

lemma *matrix-to-blinfun-id*: *matrix-to-blinfun* (*mat 1* :: (*real* ^-^-)) =

id-blinfun
  by (*auto intro*!: *blinfun-eqI simp*: *matrix-to-blinfun-mult*)

lemma *matrix-to-blinfun-inv*$_L$:
  assumes *invertible m*
  shows *matrix-to-blinfun* (*matrix-inv* (*m* :: *real*^-^-)) = *inv*$_L$ (*matrix-to-blinfun m*)
    invertible$_L$ (*matrix-to-blinfun m*)
proof −
  have *m* ∗∗ *matrix-inv m* = *mat 1 matrix-inv m* ∗∗ *m* = *mat 1*
    using *assms*
    by (*auto simp*: *matrix-inv-right matrix-inv-left*)
  hence *matrix-to-blinfun* (*matrix-inv m*) o_L *matrix-to-blinfun m* = *id-blinfun*
    matrix-to-blinfun m o_L *matrix-to-blinfun* (*matrix-inv m*) = *id-blinfun*
    by (*auto simp*: *matrix-to-blinfun-id matrix-to-blinfun-comp*[*symmetric*])
  thus *matrix-to-blinfun* (*matrix-inv m*) = *inv*$_L$ (*matrix-to-blinfun m*)
invertible$_L$ (*matrix-to-blinfun m*)
    by (*auto intro*: *inv*$_L$-*I*)
qed

lemma *blinfun-to-matrix-inverse*:
  assumes *invertible*$_L$ *X*
  shows *invertible* (*blinfun-to-matrix* (*X* :: ('*b*::*finite* ⇒$_b$ *real*) ⇒$_L$ '*c*::*finite* ⇒$_b$ *real*))
    blinfun-to-matrix (*inv*$_L$ *X*) = *matrix-inv* (*blinfun-to-matrix X*)
proof −
  have *X* o_L *inv*$_L$ *X* = *id-blinfun*
    by (*simp add*: *assms*)
  hence *1*: *blinfun-to-matrix X* ∗∗ *blinfun-to-matrix* (*inv*$_L$ *X*) = *mat 1*
    by (*metis blinfun-to-matrix-comp blinfun-to-matrix-id*)
  have *inv*$_L$ *X* o_L *X* = *id-blinfun*
    by (*simp add*: *assms*)
  hence *2*: *blinfun-to-matrix* (*inv*$_L$ *X*) ∗∗ *blinfun-to-matrix* (*X*) = *mat 1*
    by (*metis blinfun-to-matrix-comp blinfun-to-matrix-id*)
  thus *invertible* (*blinfun-to-matrix X*)
    using *1 invertible-def* **by** *blast*
  thus *blinfun-to-matrix* (*inv*$_L$ *X*) = *matrix-inv* (*blinfun-to-matrix X*)
    using *1 2 matrix-inv-right matrix-mul-assoc matrix-mul-lid*
    by *metis*
qed

lemma *blinfun-to-matrix-inv*[*simp*]: *blinfun-to-matrix* (*matrix-to-blinfun f*) = *f*
  by (*auto simp*: *matrix-eq blinfun-to-matrix-mult*′′ *matrix-to-blinfun-mult bfun.Bfun-inverse*)

lemma *invertible-invertible*$_L$*-I*: *invertible* (*blinfun-to-matrix f*) $\Longrightarrow$ *invertible*$_L$ *f*
invertible$_L$ (*matrix-to-blinfun X*) $\Longrightarrow$ *invertible* (*X* :: *real*^-^-)
using *matrix-to-blinfun-inv*$_L$(*2*) *blinfun-to-matrix-inverse*(*1*) *matrix-to-blinfun-inv blinfun-to-matrix-inv*
by *metis*+

lemma *bounded-linear-blinfun-to-matrix*: *bounded-linear* (*blinfun-to-matrix* :: (*'a* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*'b* $\Rightarrow_b$ *real*) $\Rightarrow$ *real*^*'a*^*'b*)
proof (*intro bounded-linear-intro*[*of - real CARD*(*'a*::*finite*) $*$ *real CARD*(*'b*::*finite*)])
show $\bigwedge$*x y*. *blinfun-to-matrix* (*x* + *y*) = *blinfun-to-matrix x* + *blinfun-to-matrix y*
by (*auto simp*: *blinfun-to-matrix-add blinfun-to-matrix-scaleR*)
next
show $\bigwedge$*r x*. *blinfun-to-matrix* (*r* $*_R$ *x*) = *r* $*_R$ *blinfun-to-matrix x*
by (*auto simp*: *blinfun-to-matrix-def matrix-def blinfun.scaleR-left vec-eq-iff*)
next
have $*$: $\bigwedge$*j*. (λ*i*. *if i* = *j then 1*::*real else 0*) $\in$ *bfun*
by *auto*
hence $**$: $\bigwedge$*j*. *norm* (*Bfun* (λ*i*. *if i* = *j then 1*::*real else 0*)) = *1*
by (*auto simp*: *Bfun-inverse*[*OF* $*$] *norm-bfun-def'* *intro*!: *cSup-eq-maximum*)
show *norm* (*blinfun-to-matrix x*) $\leq$ *norm x* $*$ (*real CARD*(*'a*) $*$ *real CARD*(*'b*)) **for** *x* :: (*'a* $\Rightarrow_b$ *real*) $\Rightarrow_L$ *'b* $\Rightarrow_b$ *real*
proof −
have *norm* (*blinfun-to-matrix x*) $\leq$ ($\sum$ *i*$\in$*UNIV*. $\sum$ *ia*$\in$*UNIV*. |(*x* (*bfun.Bfun* (λ*i*. *if i* = *ia then 1 else 0*))) *i*|)
unfolding *norm-vec-def blinfun-to-matrix-def matrix-def axis-def*
by(*auto simp*: *vec-lambda-inverse intro*!: *order.trans*[*OF L2-set-le-sum-abs*] *order.trans*[*OF sum-mono*[*OF L2-set-le-sum-abs*]])
also have ... $\leq$ ($\sum$ *i*$\in$(*UNIV*::*'b set*). $\sum$ *ia*$\in$(*UNIV* :: *'a set*). *norm x*)
using *norm-blinfun abs-le-norm-bfun*
by (*fastforce simp*: $**$ *intro*!: *sum-mono intro*: *order.trans*)
also have ... = *norm x* $*$ (*real CARD*(*'a*) $*$ *real CARD*(*'b*))
by *auto*
finally show *?thesis*.
qed
qed

lemma *summable-blinfun-to-matrix*:
assumes *summable* (*f* :: *nat* $\Rightarrow$ (*'c*::*finite* $\Rightarrow_b$ -) $\Rightarrow_L$ (*'c* $\Rightarrow_b$ -))
shows *summable* (λ*i*. *blinfun-to-matrix* (*f i*))
by (*simp add*: *assms bounded-linear.summable bounded-linear-blinfun-to-matrix*)

abbreviation *nonneg-blinfun Q* $\equiv$ *0* $\leq$ (*blinfun-to-matrix Q*)

lemma *nonneg-blinfun-mono*: *nonneg-blinfun* $Q \Longrightarrow u \leq v \Longrightarrow Q\ u \leq Q\ v$
using *nonneg-mat-mono*[*of blinfun-to-matrix Q vec-lambda u vec-lambda v*]
by (*fastforce simp*: *blinfun-to-matrix-mult''* *apply-bfun-inverse Finite-Cartesian-Product.less-eq-vec-def*)

lemma *nonneg-blinfun-nonneg*: *nonneg-blinfun* $Q \Longrightarrow 0 \leq v \Longrightarrow 0 \leq Q\ v$
using *nonneg-blinfun-mono blinfun.zero-right*
by *metis*

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

lemma *norm-nonneg-blinfun-one*:
assumes $0 \leq$ *blinfun-to-matrix X*
shows *norm X* = *norm* (*blinfun-apply X 1*)
by (*simp add*: *norm-blinfun-mono-eq-one assms nonneg-blinfun-nonneg*)

lemma *matrix-le-norm-mono*:
assumes $0 \leq$ (*blinfun-to-matrix C*)
and (*blinfun-to-matrix C*) $\leq$ (*blinfun-to-matrix D*)
shows *norm C* $\leq$ *norm D*
proof −
have $0 \leq C\ 1$ $0 \leq D\ 1$
using *assms zero-le-one*
by (*fastforce intro*!: *nonneg-blinfun-nonneg*)+
have $\bigwedge v.\ v \geq 0 \Longrightarrow$ *blinfun-to-matrix C* $*v$ $v \leq$ *blinfun-to-matrix D* $*v$ v
using *assms nonneg-mat-mono*[*of blinfun-to-matrix D* − *blinfun-to-matrix C*]
by (*fastforce simp*: *matrix-vector-mult-diff-rdistrib*)
hence $*$: $\bigwedge v.\ v \geq 0 \Longrightarrow C\ v \leq D\ v$
by (*auto simp*: *less-eq-vec-def less-eq-bfun-def blinfun-to-matrix-mult*[*symmetric*])
show *?thesis*
using *assms*(*1*) *assms*(*2*) ‹$0 \leq C\ 1$› ‹$0 \leq D\ 1$› *less-eq-bfunD*[*OF* $*$, *of 1*]
by (*fastforce intro*!: *cSUP-mono simp*: *norm-nonneg-blinfun-one norm-bfun-def' less-eq-bfun-def*)
qed

lemma *blinfun-to-matrix-matpow*: *blinfun-to-matrix* (*X* ^^ *i*) = *matpow* (*blinfun-to-matrix X*) *i*
by (*induction i*) (*auto simp*: *blinfun-to-matrix-id blinfun-to-matrix-comp blinfunpow-assoc simp del*: *blinfunpow.simps*(*2*))

lemma *nonneg-blinfun-iff*: *nonneg-blinfun X* $\longleftrightarrow$ ($\forall v \geq 0.\ X\ v \geq 0$)
using *nonneg-mat-iff*[*of blinfun-to-matrix X*] *nonneg-blinfun-nonneg*

by (*auto simp*: *blinfun-to-matrix-mult''* *bfun.Bfun-inverse less-eq-vec-def less-eq-bfun-def*)

lemma *blinfun-apply-mono*: $(0::real\hat{}-\hat{}-) \leq$ *blinfun-to-matrix* $X \Longrightarrow 0 \leq v \Longrightarrow$ *blinfun-to-matrix* $X \leq$ *blinfun-to-matrix* $Y \Longrightarrow X\ v \leq Y\ v$
by (*metis blinfun.diff-left blinfun-to-matrix-diff diff-ge-0-iff-ge nonneg-blinfun-nonneg*)

end

theory *Splitting-Methods*
imports
Blinfun-Matrix
Value-Iteration
Policy-Iteration
begin

6 Value Iteration using Splitting Methods

6.1 Regular Splittings for Matrices and Bounded Linear Functions

definition *is-splitting-mat* $X\ Q\ R \longleftrightarrow$
$X = Q - R \wedge$ *invertible* $Q \wedge 0 \leq$ *matrix-inv* $Q \wedge 0 \leq R$

definition *is-splitting-blin* $X\ Q\ R \longleftrightarrow$ *is-splitting-mat* (*blinfun-to-matrix* X) (*blinfun-to-matrix* Q) (*blinfun-to-matrix* R)

lemma *is-splitting-blin-def'*: *is-splitting-blin* $X\ Q\ R \longleftrightarrow$
$X = Q - R \wedge$ *invertible*$_L$ $Q \wedge$ *nonneg-blinfun* (*inv*$_L$ Q) $\wedge$ *nonneg-blinfun* R
proof −
have *blinfun-to-matrix* X = *blinfun-to-matrix* Q − *blinfun-to-matrix* $R \longleftrightarrow X = Q - R$
using *blinfun-to-matrix-diff matrix-to-blinfun-inv*
by *metis*
thus *?thesis*
unfolding *is-splitting-blin-def is-splitting-mat-def*
using *blinfun-to-matrix-inverse*[*of Q*] *matrix-to-blinfun-inv*
by (*fastforce simp*: *invertible-invertible*$_L$*-I*(*1*))
qed

lemma *is-splitting-blinD*[*dest*]:
assumes *is-splitting-blin* $X\ Q\ R$
shows $X = Q - R$ *invertible*$_L$ Q *nonneg-blinfun* (*inv*$_L$ Q) *nonneg-blinfun* R
using *is-splitting-blin-def' assms* **by** *auto*

6.2 Splitting Methods for MDPs

locale *MDP-QR* = *MDP-finite-type A K r l*
 for $A :: 's :: finite \Rightarrow ('a :: finite)\ set$
  and $K :: ('s \times 'a) \Rightarrow 's\ pmf$
  and $r\ l\ +$
 fixes $Q :: ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow_b real) \Rightarrow_L ('s \Rightarrow_b real)$
 fixes $R :: ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow_b real) \Rightarrow_L ('s \Rightarrow_b real)$
 assumes *is-splitting*: $\bigwedge d.\ d \in D_D \Longrightarrow$ *is-splitting-blin* (*id-blinfun* − $l *_R \mathcal{P}_1$ (*mk-dec-det d*)) ($Q\ d$) ($R\ d$)
 assumes *QR-contraction*: $(\bigsqcup d{\in}D_D.\ norm\ (inv_L\ (Q\ d)\ o_L\ R\ d)) < 1$
 assumes *arg-max-ex-split*: $\exists d.\ \forall s.\ is\text{-}arg\text{-}max\ (\lambda d.\ inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d) + R\ d\ v)\ s)\ (\lambda d.\ d \in D_D)\ d$
begin

lemma *inv-Q-mono*: $d \in D_D \Longrightarrow u \le v \Longrightarrow (inv_L\ (Q\ d))\ u \le (inv_L\ (Q\ d))\ v$
 using *is-splitting*
 by (*auto intro*!: *nonneg-blinfun-mono*)

lemma *splitting-eq*: $d \in D_D \Longrightarrow Q\ d - R\ d = (id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d))$
 using *is-splitting*
 by *fastforce*

lemma *Q-nonneg*: $d \in D_D \Longrightarrow 0 \le v \Longrightarrow 0 \le inv_L\ (Q\ d)\ v$
 using *is-splitting nonneg-blinfun-nonneg*
 by *auto*

lemma *Q-invertible*: $d \in D_D \Longrightarrow invertible_L\ (Q\ d)$
 using *is-splitting*
 by *auto*

lemma *R-nonneg*: $d \in D_D \Longrightarrow 0 \le v \Longrightarrow 0 \le R\ d\ v$
 using *is-splitting-blinD*[*OF is-splitting*]
 by (*fastforce simp*: *nonneg-blinfun-nonneg intro*: *nonneg-blinfun-mono*)

lemma *R-mono*: $d \in D_D \Longrightarrow u \le v \Longrightarrow (R\ d)\ u \le (R\ d)\ v$
 using *R-nonneg*[*of* $d\ v - u$]
 by (*auto simp*: *blinfun.bilinear-simps*)

lemma *QR-nonneg*: $d \in D_D \Longrightarrow 0 \le v \Longrightarrow 0 \le (inv_L\ (Q\ d)\ o_L\ R\ d)\ v$
 by (*simp add*: *Q-nonneg R-nonneg*)

lemma *QR-mono*: $d \in D_D \Longrightarrow u \le v \Longrightarrow (inv_L\ (Q\ d)\ o_L\ R\ d)\ u \le (inv_L\ (Q\ d)\ o_L\ R\ d)\ v$
 using *QR-nonneg*[*of* $d\ v - u$]
 by (*auto simp*: *blinfun.bilinear-simps*)

lemma *norm-QR-less-one*: $d \in D_D \Longrightarrow norm\ (inv_L\ (Q\ d)\ o_L\ R\ d) < 1$
using *QR-contraction*
by (*auto intro*!: *cSUP-lessD*[*of* $\lambda d.\ norm\ (inv_L\ (Q\ d)\ o_L\ R\ d)$])

lemma *splitting*: $d \in D_D \Longrightarrow id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d) = Q\ d - R\ d$
using *is-splitting*
by *auto*

6.3 Discount Factor *QR-disc*

abbreviation *QR-disc* $\equiv (\bigsqcup d \in D_D.\ norm\ (inv_L\ (Q\ d)\ o_L\ R\ d))$

lemma *QR-le-QR-disc*: $d \in D_D \Longrightarrow norm\ (inv_L\ (Q\ d)\ o_L\ (R\ d)) \le QR\text{-}disc$
by (*auto intro*: *cSUP-upper*)

lemma *a-nonneg*: $0 \le QR\text{-}disc$
using *QR-contraction norm-ge-zero ex-dec-det*
by (*fastforce intro*!: *cSUP-upper2*)

6.4 Bellman-Operator

abbreviation *L-split d v* $\equiv inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d) + R\ d\ v)$

definition $\mathcal{L}$*-split v s* $= (\bigsqcup d \in D_D.\ L\text{-}split\ d\ v\ s)$

lemma $\mathcal{L}$*-split-bfun-aux*:
assumes $d \in D_D$
shows $norm\ (L\text{-}split\ d\ v) \le (\bigsqcup d \in D_D.\ norm\ (inv_L\ (Q\ d))) * r_M + norm\ v$
proof −
have $norm\ (L\text{-}split\ d\ v) \le norm\ (inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d))) + norm\ (inv_L\ (Q\ d)\ (R\ d\ v))$
by (*simp add*: *blinfun.add-right norm-triangle-ineq*)
also have $\ldots \le norm\ (inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d))) + norm\ (inv_L\ (Q\ d)\ o_L\ R\ d) * norm\ v$
by (*auto simp*: *blinfun-apply-blinfun-compose*[*symmetric*] *norm-blinfun simp del*: *blinfun-apply-blinfun-compose*)
also have $\ldots \le norm\ (inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d))) + norm\ v$
using *norm-QR-less-one assms*
by (*fastforce intro*!: *mult-left-le-one-le*)
also have $\ldots \le norm\ (inv_L\ (Q\ d)) * r_M + norm\ v$
by (*auto intro*!: *order.trans*[*OF norm-blinfun*] *mult-left-mono simp*: *norm-r-dec-le*)
also have $\ldots \le (\bigsqcup d \in D_D.\ norm\ (inv_L\ (Q\ d))) * r_M + norm\ v$

by (*auto intro*!: *mult-right-mono cSUP-upper assms simp*: r_M*-nonneg*)
finally show *?thesis*.
qed

lift-definition $\mathcal{L}_b$*-split* :: ($'s \Rightarrow_b$ *real*) $\Rightarrow$ ($'s \Rightarrow_b$ *real*) **is** $\mathcal{L}$*-split*
by *fastforce*

lemma $\mathcal{L}_b$*-split-def'*: $\mathcal{L}_b$*-split* $v\ s = (\bigsqcup d{\in}D_D.\ L\text{-}split\ d\ v\ s)$
unfolding $\mathcal{L}_b$*-split.rep-eq* $\mathcal{L}$*-split-def*
by *auto*

lemma $\mathcal{L}_b$*-split-contraction*: *dist* ($\mathcal{L}_b$*-split* v) ($\mathcal{L}_b$*-split* u) $\leq$ *QR-disc* $*$ *dist* $v\ u$
proof −
have *aux*:
$\mathcal{L}_b$*-split* $v\ s$ − $\mathcal{L}_b$*-split* $u\ s \leq$ *QR-disc* $*$ *norm* $(v - u)$ **if** h: $\mathcal{L}_b$*-split* $u\ s \leq$ $\mathcal{L}_b$*-split* $v\ s$ **for** $u\ v\ s$
proof −
obtain d **where** d: *is-arg-max* ($\lambda d.$ inv_L ($Q\ d$) ($r\text{-}dec_b$ (*mk-dec-det* d) + $R\ d\ v$) s) ($\lambda d.\ d \in D_D$) d
using *finite-is-arg-max*[*of* D_D]
by *auto*
have $*$: inv_L ($Q\ d$) ($r\text{-}dec_b$ (*mk-dec-det* d) + $R\ d\ u$) $s \leq$ $\mathcal{L}_b$*-split* $u\ s$
using d
by (*auto simp*: $\mathcal{L}_b$*-split-def'* *is-arg-max-linorder intro*!: *cSUP-upper2*)
have inv_L ($Q\ d$) ($r\text{-}dec_b$ (*mk-dec-det* d) + $R\ d\ v$) s = $\mathcal{L}_b$*-split* $v\ s$
by (*auto simp*: $\mathcal{L}_b$*-split-def'* *arg-max-SUP*[*OF* d])
hence $\mathcal{L}_b$*-split* $v\ s$ − $\mathcal{L}_b$*-split* $u\ s$ = inv_L ($Q\ d$) ($r\text{-}dec_b$ (*mk-dec-det* d) + $R\ d\ v$) s − $\mathcal{L}_b$*-split* $u\ s$
by *auto*
also have ... $\leq$ (inv_L ($Q\ d$) o_L $R\ d$) $(v - u)$ s
using $*$
by (*auto simp*: *blinfun.bilinear-simps*)
also have ... $\leq$ *norm* ((inv_L ($Q\ d$) o_L $R\ d$)) $*$ *norm* $(v - u)$
by (*fastforce intro*: *order.trans*[*OF le-norm-bfun norm-blinfun*])
also have ... $\leq$ *QR-disc* $*$ *norm* $(v - u)$
using *QR-contraction* d
by (*auto simp*: *is-arg-max-linorder intro*!: *mult-right-mono cSUP-upper2*)
finally show *?thesis*.
qed
have $|$($\mathcal{L}_b$*-split* v − $\mathcal{L}_b$*-split* u) $s| \leq$ *QR-disc* $*$ *dist* $v\ u$ **for** s
using *aux*
by (*cases* $\mathcal{L}_b$*-split* $v\ s \leq$ $\mathcal{L}_b$*-split* $u\ s$) (*fastforce simp*: *dist-norm norm-minus-commute*)+
thus *?thesis*
by (*auto intro*!: *cSUP-least simp*: *dist-bfun.rep-eq dist-real-def*)
qed

lemma $\mathcal{L}_b$*-lim*:
$\exists!v.\ \mathcal{L}_b\text{-}split\ v = v$
$(\lambda n.\ (\mathcal{L}_b\text{-}split\ \widehat{\ }\widehat{\ }\ n)\ v) \longrightarrow (THE\ v.\ \mathcal{L}_b\text{-}split\ v = v)$
using *banach′*[*of* $\mathcal{L}_b$*-split*] *a-nonneg QR-contraction* $\mathcal{L}_b$*-split-contraction*
unfolding *is-contraction-def*
by *auto*

lemma $\mathcal{L}_b$*-split-tendsto-opt*: $(\lambda n.\ (\mathcal{L}_b\text{-}split\ \widehat{\ }\widehat{\ }\ n)\ v) \longrightarrow \nu_b\text{-}opt$
proof −
obtain L **where** *l-fix*: $\mathcal{L}_b\text{-}split\ L = L$
using $\mathcal{L}_b$*-lim*(*1*)
by *auto*
have $\nu_b\ (mk\text{-}stationary\text{-}det\ d) \le L$ **if** d: $d \in D_D$ **for** d
proof −
let $?QR = inv_L\ (Q\ d)\ o_L\ R\ d$
have $inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d) + R\ d\ L) \le \mathcal{L}_b\text{-}split\ L$
using *d l-fix*
by (*fastforce simp*: $\mathcal{L}_b$*-split-def′ intro*!: *cSUP-upper2*)
hence $inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d) + R\ d\ L) \le L$
using *l-fix* **by** *auto*
hence *aux*: $inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d)) \le (id\text{-}blinfun -$
$?QR)\ L$
using *that*
by (*auto simp*: *blinfun.bilinear-simps le-diff-eq*)
have *inv-eq*: $inv_L\ (id\text{-}blinfun - ?QR) = (\sum i.\ ?QR\ \widehat{\ }\widehat{\ }\ i)$
using *QR-contraction d norm-QR-less-one*
by (*auto intro*!: inv_L*-inf-sum*)
have *summable-QR*:*summable* $(\lambda i.\ norm\ (?QR\ \widehat{\ }\widehat{\ }\ i))$
using *QR-contraction d*
by (*fastforce simp*: *a-nonneg*
intro: *summable-comparison-test′*[**where** $g = \lambda i.\ QR\text{-}disc\widehat{\ }i$]
intro!: *cSUP-upper2 power-mono order.trans*[*OF norm-blinfunpow-le*])
have *summable* $(\lambda i.\ (?QR\ \widehat{\ }\widehat{\ }\ i)\ v\ s)$ **for** $v\ s$
by (*rule summable-comparison-test′*[**where** $g = \lambda i.\ norm\ (?QR$
$\widehat{\ }\widehat{\ }\ i) * norm\ v$])
(*auto intro*!: *summable-QR summable-norm-cancel order.trans*[*OF*
abs-le-norm-bfun] *order.trans*[*OF norm-blinfun*] *summable-mult2*)
moreover have $0 \le v \Longrightarrow 0 \le (\sum i{<}n.\ (?QR\ \widehat{\ }\widehat{\ }\ i)\ v\ s)$ **for** $n\ v\ s$
using *blinfunpow-nonneg*[*OF QR-nonneg*[*OF d*]]
by (*induction n*) (*auto simp add*: *less-eq-bfun-def*)
ultimately have $0 \le v \Longrightarrow 0 \le (\sum i.\ ((?QR\ \widehat{\ }\widehat{\ }\ i)\ v\ s))$ **for** $v\ s$
by (*auto intro*!: *summable-LIMSEQ LIMSEQ-le*)
hence $0 \le v \Longrightarrow 0 \le (\sum i.\ ((?QR\ \widehat{\ }\widehat{\ }\ i)))\ v\ s$ **for** $v\ s$
using *bounded-linear-apply-bfun summable-QR summable-comparison-test′*

by (*subst bounded-linear.suminf*[**where** $f = (\lambda i.\ apply\text{-}bfun$
$(blinfun\text{-}apply\ i\ v)\ s)$]])
(*fastforce intro*: *bounded-linear-compose*[*of* $\lambda s.\ apply\text{-}bfun\ s$ -])+
hence $0 \le v \Longrightarrow 0 \le inv_L\ (id\text{-}blinfun - ?QR)\ v$ **for** v

by (*simp add*: *inv-eq less-eq-bfun-def*)
hence (inv_L (*id-blinfun* − *?QR*)) ((inv_L (*Q d*)) ($r\text{-}dec_b$ (*mk-dec-det* *d*)))
≤ (inv_L (*id-blinfun* − *?QR*)) ((*id-blinfun* − *?QR*) *L*)
by (*metis aux blinfun.diff-right diff-ge-0-iff-ge*)
hence (inv_L (*id-blinfun* − *?QR*) o_L inv_L (*Q d*)) ($r\text{-}dec_b$ (*mk-dec-det* *d*)) ≤ *L*
using $invertible_L$*-inf-sum*[*OF norm-QR-less-one*[*OF that*]]
by *auto*
hence (inv_L (*Q d* o_L (*id-blinfun* − *?QR*))) ($r\text{-}dec_b$ (*mk-dec-det d*)) ≤ *L*
using *d norm-QR-less-one*
by (*auto simp*: inv_L*-compose*[*OF Q-invertible* $invertible_L$*-inf-sum*])
hence (inv_L (*Q d* − *R d*)) ($r\text{-}dec_b$ (*mk-dec-det d*)) ≤ *L*
using *Q-invertible d*
by (*auto simp*: *blinfun-compose-diff-right blinfun-compose-assoc*[*symmetric*])
thus ν_b (*mk-stationary-det d*) ≤ *L*
by (*auto simp*: ν*-stationary splitting*[*OF that*, *symmetric*] inv_L*-inf-sum* *blincomp-scaleR-right*)
qed
hence *opt-le*: ν_b*-opt* ≤ *L*
using *thm-6-2-10 finite* **by** *auto*

obtain *d* **where** *d*: *is-arg-max* (λ*d*. inv_L (*Q d*) ($r\text{-}dec_b$ (*mk-dec-det* *d*) + *R d L*) *s*) (λ*d*. *d* ∈ D_D) *d* **for** *s*
using *arg-max-ex-split* **by** *blast*
hence *d* ∈ D_D
unfolding *is-arg-max-linorder*
by *auto*
have *L* = inv_L (*Q d*) ($r\text{-}dec_b$ (*mk-dec-det d*) + *R d L*)
by (*subst l-fix*[*symmetric*]) (*fastforce simp*: $\mathcal{L}_b$*-split-def′ arg-max-SUP*[*OF* *d*])
hence *Q d L* = $r\text{-}dec_b$ (*mk-dec-det d*) + *R d L*
by (*metis Q-invertible* ‹*d* ∈ D_D› *inv-app2′*)
hence (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det d*)) *L* = $r\text{-}dec_b$ (*mk-dec-det* *d*)
using *splitting*[*OF* ‹*d* ∈ D_D›]
by (*simp add*: *blinfun.diff-left*)
hence *L* = inv_L ((*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det d*))) ($r\text{-}dec_b$ (*mk-dec-det d*))
using $invertible_L$*-inf-sum*[*OF norm-*$\mathcal{P}_1$*-l-less*] *inv-app1′*
by *metis*
hence *L* = ν_b (*mk-stationary-det d*)
by (*auto simp*: inv_L*-inf-sum* ν*-stationary blincomp-scaleR-right*)
hence ν_b*-opt* = *L*
using *opt-le* ‹*d* ∈ D_D› *is-markovian-def*
by (*auto intro*: *order.antisym*[*OF* - ν_b*-le-opt*])
thus *?thesis*
using $\mathcal{L}_b$*-lim l-fix the1-equality*[*OF* $\mathcal{L}_b$*-lim*(*1*)]

by *auto*
qed

lemma $\mathcal{L}_b$*-split-fix*[*simp*]: $\mathcal{L}_b$*-split* ν_b*-opt* = ν_b*-opt*
using $\mathcal{L}_b$*-lim* $\mathcal{L}_b$*-split-tendsto-opt the-equality limI*
by (*metis* (*mono-tags*, *lifting*))

lemma *dist-*$\mathcal{L}_b$*-split-opt-eps*:
assumes *eps* > *0 2* ∗ *QR-disc* ∗ *dist v* ($\mathcal{L}_b$*-split v*) < *eps* ∗ (*1*− *QR-disc*)
shows *dist* ($\mathcal{L}_b$*-split v*) ν_b*-opt* < *eps* / *2*
proof −
have (*1* − *QR-disc*) ∗ *dist v* ν_b*-opt* ≤ *dist v* ($\mathcal{L}_b$*-split v*)
using *dist-triangle* $\mathcal{L}_b$*-split-contraction*[*of v* ν_b*-opt*]
by (*fastforce simp*: *algebra-simps intro*: *order.trans*[*OF* - *add-left-mono*[*of dist* ($\mathcal{L}_b$*-split v*) ν_b*-opt*]])
hence *dist v* ν_b*-opt* ≤ *dist v* ($\mathcal{L}_b$*-split v*) / (*1* − *QR-disc*)
using *QR-contraction*
by (*simp add*: *mult.commute pos-le-divide-eq*)
hence *2* ∗ *QR-disc* ∗ *dist v* ν_b*-opt* ≤ *2* ∗ *QR-disc* ∗ (*dist v* ($\mathcal{L}_b$*-split v*) / (*1* − *QR-disc*))
using $\mathcal{L}_b$*-split-contraction assms mult-le-cancel-left-pos*[*of 2* ∗ *QR-disc*] *a-nonneg*
by (*fastforce intro*!: *mult-left-mono*[*of* - - *2* ∗ *QR-disc*])
hence *2* ∗ *QR-disc* ∗ *dist v* ν_b*-opt* < *eps*
using *a-nonneg QR-contraction*
by (*auto simp*: *assms*(*2*) *pos-divide-less-eq intro*: *order.strict-trans1*)
hence *dist v* ν_b*-opt* ∗ *QR-disc* < *eps* / *2*
by *argo*
thus *dist* ($\mathcal{L}_b$*-split v*) ν_b*-opt* < *eps* / *2*
using $\mathcal{L}_b$*-split-contraction*[*of v* ν_b*-opt*]
by (*auto simp*: *algebra-simps*)
qed

lemma *L-split-fix*:
assumes *d* ∈ D_D
shows *L-split d* (ν_b (*mk-stationary-det d*)) = ν_b (*mk-stationary-det d*)
proof −
let *?d* = *mk-dec-det d*
let *?p* = *mk-stationary-det d*
have (*Q d* − *R d*) (ν_b *?p*) = *r-dec*$_b$ *?d*
using *L-*ν*-fix*[*of mk-dec-det d*]
by (*simp add*: *L-def splitting*[*OF assms*, *symmetric*] *blinfun.bilinear-simps diff-eq-eq*)
thus *?thesis*
using *assms*
by (*auto simp*: *blinfun.bilinear-simps diff-eq-eq inv*$_L$*-cancel-iff*[*OF Q-invertible*])

qed

lemma *L-split-contraction*:
  assumes $d \in D_D$
  shows *dist* (*L-split d v*) (*L-split d u*) $\leq$ *QR-disc* $*$ *dist v u*
proof $-$
  have *aux*: *L-split d v s* $-$ *L-split d u s* $\leq$ *QR-disc* $*$ *dist v u* **if** *lea*: (*L-split d u s*) $\leq$ (*L-split d v s*) **for** *v s u*
  proof $-$
    have *L-split d v s* $-$ *L-split d u s* $=$ (inv_L (*Q d*) o_L (*R d*)) (*v* $-$ *u*) *s*
      by (*auto simp*: *blinfun.bilinear-simps*)
    also have ... $\leq$ *norm* ((inv_L (*Q d*) o_L (*R d*)) (*v* $-$ *u*))
      by (*simp add*: *le-norm-bfun*)
    also have ... $\leq$ *norm* ((inv_L (*Q d*) o_L (*R d*))) $*$ *dist v u*
      by (*auto simp only*: *dist-norm norm-blinfun*)
    also have ... $\leq$ *QR-disc* $*$ *dist v u*
      using *assms QR-le-QR-disc*
      by (*auto intro*!: *mult-right-mono*)
    finally show *?thesis*
      by *auto*
  qed
  have *dist* (*L-split d v s*) (*L-split d u s*) $\leq$ *QR-disc* $*$ *dist v u* **for** *v s u*
    using *aux aux*[*of v - u*]
    by (*cases L-split d v s* $\geq$ *L-split d u s*) (*auto simp*: *dist-real-def dist-commute*)
  thus *dist* (*L-split d v*) (*L-split d u*) $\leq$ *QR-disc* $*$ *dist v u*
    by (*simp add*: *dist-bound*)
qed

lemma *find-policy-QR-error-bound*:
  assumes *eps* $>$ *0 2* $*$ *QR-disc* $*$ *dist v* ($\mathcal{L}_b$*-split v*) $<$ *eps* $*$ (*1*$-$*QR-disc*)
  assumes *am*: $\bigwedge$*s. is-arg-max* (λ*d. L-split d* ($\mathcal{L}_b$*-split v*) *s*) (λ*d. d* $\in$ D_D) *d*
  shows *dist* (ν_b (*mk-stationary-det d*)) ν_b*-opt* $<$ *eps*
proof $-$
  let *?p* $=$ *mk-stationary-det d*
  have *L-eq-*$\mathcal{L}_b$: *L-split d* ($\mathcal{L}_b$*-split v*) $=$ $\mathcal{L}_b$*-split* ($\mathcal{L}_b$*-split v*)
    by (*auto simp*: $\mathcal{L}_b$*-split-def′ arg-max-SUP*[*OF am*])
  have *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $=$ *dist* (*L-split d* (ν_b *?p*)) ($\mathcal{L}_b$*-split v*)
    using *am*
    by (*auto simp*: *is-arg-max-linorder L-split-fix*)
  also have ... $\leq$ *dist* (*L-split d* (ν_b *?p*)) ($\mathcal{L}_b$*-split* ($\mathcal{L}_b$*-split v*)) $+$ *dist* ($\mathcal{L}_b$*-split* ($\mathcal{L}_b$*-split v*)) ($\mathcal{L}_b$*-split v*)
    by (*auto intro*: *dist-triangle*)
  also have ... $=$ *dist* (*L-split d* (ν_b *?p*)) (*L-split d* ($\mathcal{L}_b$*-split v*)) $+$ *dist* ($\mathcal{L}_b$*-split* ($\mathcal{L}_b$*-split v*)) ($\mathcal{L}_b$*-split v*)

by (*auto simp*: *L-eq-*$\mathcal{L}_b$)
also have ... $\leq$ *QR-disc* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $+$ *QR-disc* $*$ *dist* ($\mathcal{L}_b$*-split v*) *v*
using $\mathcal{L}_b$*-split-contraction L-split-contraction am* **unfolding** *is-arg-max-def*
by (*auto intro*!: *add-mono*)
finally have *aux*: *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $\leq$ *QR-disc* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $+$ *QR-disc* $*$ *dist* ($\mathcal{L}_b$*-split v*) *v* **.**
hence *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $-$ *QR-disc* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $\leq$ *QR-disc* $*$ *dist* ($\mathcal{L}_b$*-split v*) *v*
by *auto*
hence *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $*$ (*1* $-$ *QR-disc*) $\leq$ *QR-disc* $*$ *dist* ($\mathcal{L}_b$*-split v*) *v*
by *argo*
hence *2* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $*$ (*1*$-$*QR-disc*) $\leq$ *2* $*$ (*QR-disc* $*$ *dist* ($\mathcal{L}_b$*-split v*) *v*)
using *mult-left-mono*
by *auto*
hence *2* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $*$ (*1* $-$ *QR-disc*) $\leq$ *eps* $*$ (*1* $-$ *QR-disc*)
using *assms*
by (*auto intro*!: *mult-left-mono simp*: *dist-commute pos-divide-le-eq*)
hence *2* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) $\leq$ *eps*
using *QR-contraction mult-right-le-imp-le*
by *auto*
moreover have *2* $*$ *dist* ($\mathcal{L}_b$*-split v*) ν_b*-opt* $<$ *eps*
using *dist-*$\mathcal{L}_b$*-split-opt-eps assms*
by *fastforce*
ultimately show *?thesis*
using *dist-triangle*[*of* ν_b *?p* ν_b*-opt* $\mathcal{L}_b$*-split v*]
by *auto*
qed
end

context *MDP-ord* **begin**
lemma *inv-one-sub-Q'*:
fixes *Q* :: *'c* :: *banach* $\Rightarrow_L$ *'c*
assumes *onorm-le*: *norm* (*id-blinfun* $-$ *Q*) $<$ *1*
shows *inv*$_L$ *Q* $=$ ($\sum$ *i.* (*id-blinfun* $-$ *Q*)$\widehat{\ }\widehat{\ }$*i*)
by (*metis inv*$_L$*-I inv-one-sub-Q assms*)

An important theorem: allows to compare the rate of convergence for different splittings

lemma *norm-splitting-le*:
assumes *is-splitting-blin* (*id-blinfun* $-$ *l* $*_R$ $\mathcal{P}_1$ *d*) *Q1 R1*
and *is-splitting-blin* (*id-blinfun* $-$ *l* $*_R$ $\mathcal{P}_1$ *d*) *Q2 R2*
and (*blinfun-to-matrix R2*) $\leq$ (*blinfun-to-matrix R1*)
and (*blinfun-to-matrix R1*) $\leq$ (*blinfun-to-matrix* (*l* $*_R$ $\mathcal{P}_1$ *d*))
shows *norm* (*inv*$_L$ *Q2* o_L *R2*) $\leq$ *norm* (*inv*$_L$ *Q1* o_L *R1*)
proof $-$

let *?R1* = *blinfun-to-matrix R1*
let *?R2* = *blinfun-to-matrix R2*
let *?Q1* = *blinfun-to-matrix Q1*
let *?Q2* = *blinfun-to-matrix Q2*
have
inv-Q: inv_L *Q* = ($\sum$ *i*. (*id-blinfun* − *Q*)^^*i*) *norm* (*id-blinfun* − *Q*) < *1* **and**
splitting-eq: *id-blinfun* − *Q* = *l* $*_R$ $\mathcal{P}_1$ *d* − *R* **and**
nonneg-Q: *0* ≤ *blinfun-to-matrix* (*id-blinfun* − *Q*)
if (*blinfun-to-matrix R*) ≤ (*blinfun-to-matrix* (*l* $*_R$ $\mathcal{P}_1$ *d*))
and *is-splitting-blin* (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ *d*) *Q R* **for** *Q R*
proof −
let *?R* = *blinfun-to-matrix R*
show *splitting-eq*: *id-blinfun* − *Q* = *l* $*_R$ $\mathcal{P}_1$ *d* − *R*
using *that*
by (*auto simp*: *eq-diff-eq is-splitting-blin-def′*)
have *R-nonneg*: *0* ≤ *?R*
using *that*
by *blast*
show *nonneg-Q*: *0* ≤ *blinfun-to-matrix* (*id-blinfun* − *Q*)
using *that*
by (*auto simp*: *splitting-eq blinfun-to-matrix-diff*)
moreover have (*blinfun-to-matrix* (*id-blinfun* − *Q*)) ≤ (*blinfun-to-matrix* (*l* $*_R$ $\mathcal{P}_1$ *d*))
using *R-nonneg*
by (*auto simp*: *splitting-eq blinfun-to-matrix-diff*)
ultimately have *norm* (*id-blinfun* − *Q*) ≤ *norm* (*l* $*_R$ $\mathcal{P}_1$ *d*)
using *matrix-le-norm-mono* **by** *blast*
thus *norm* (*id-blinfun* − *Q*) < *1*
using *norm-*$\mathcal{P}_1$*-l-less*
by (*simp add*: *order.strict-trans1*)
thus inv_L *Q* = ($\sum$ *i*. (*id-blinfun* − *Q*) ^^ *i*)
using *inv-one-sub-Q′*
by *auto*
qed

have *i1*: inv_L *Q1* = ($\sum$ *i*. (*id-blinfun* − *Q1*) ^^ *i*) *norm* (*id-blinfun* − *Q1*) < *1*
and *i2*: inv_L *Q2* = ($\sum$ *i*. (*id-blinfun* − *Q2*) ^^ *i*) *norm* (*id-blinfun* − *Q2*) < *1*
using *assms*
by (*auto intro*: *inv-Q*[*of R2 Q2*] *inv-Q*[*of R1 Q1*])

have *Q1-le-Q2*: *blinfun-to-matrix* (*id-blinfun* − *Q1*) ≤ *blinfun-to-matrix* (*id-blinfun* − *Q2*)
using *assms* **unfolding** *is-splitting-blin-def′*
by (*auto simp*: *blinfun-to-matrix-diff eq-diff-eq blinfun-to-matrix-add*)

have *blinfun-to-matrix* (inv_L *Q1*) = *blinfun-to-matrix* (($\sum$ *i*. (*id-blinfun*

$-$ *Q1*) ^^ *i*))
  using *i1* **by** *auto*
 also have … = (($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*)))
  using *bounded-linear.suminf*[*OF bounded-linear-blinfun-to-matrix*] *summable-inv-Q i1*(*2*)
  by *auto*
 also have … $\leq$ ($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*))
 proof $-$
  have *le-n*: $\bigwedge$*n*. *0* $\leq$ *n* $\Longrightarrow$ ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*)) $\leq$ ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*))
   using *assms nonneg-Q*
  by (*auto intro*!: *sum-mono matpow-mono simp*: *blinfun-to-matrix-matpow Q1-le-Q2*)
 hence *le-n-elem*: $\bigwedge$*n*. *0* $\leq$ *n* $\Longrightarrow$ ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*)) $ *i* $ *j* $\leq$ ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*)) $ *i* $ *j* **for** *i j*
   by (*auto simp*: *less-eq-vec-def*)
  have (λ*n*. ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*))) $\longrightarrow$ ($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*))
    by (*auto intro*!: *bounded-linear.summable*[*of blinfun-to-matrix*] *summable-LIMSEQ simp add*: *bounded-linear-blinfun-to-matrix i1*(*2*) *summable-inv-Q*)
  hence *le1*: (λ*n*. ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*)) $ *j* $ *k*) $\longrightarrow$ ($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*)) $ *j* $ *k* **for** *j k*
   using *tendsto-vec-nth*
   by *metis*
  have (λ*n*. ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*))) $\longrightarrow$ ($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*))
    by (*auto intro*!: *bounded-linear.summable*[*of blinfun-to-matrix*] *summable-LIMSEQ simp add*: *bounded-linear-blinfun-to-matrix i2*(*2*) *summable-inv-Q*)
  hence *le2*: (λ*n*. ($\sum$ *i*<*n*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*)) $ *j* $ *k*) $\longrightarrow$ ($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*)) $ *j* $ *k* **for** *j k*
   using *tendsto-vec-nth*
   by *metis*
  have (($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q1*) ^^ *i*))$ *j* $ *k*) $\leq$ (($\sum$ *i*. *blinfun-to-matrix* ((*id-blinfun* $-$ *Q2*) ^^ *i*)) $ *j* $ *k*) **for** *j k*
   by (*fastforce intro*: *Topological-Spaces.lim-mono*[*OF le-n-elem le1 le2*])
  thus *?thesis*
   by (*simp add*: *less-eq-vec-def*)
 qed
 also have … = *blinfun-to-matrix* (*inv*$_L$ *Q2*)
  using *summable-inv-Q i2*(*2*) *i2*
 by (*auto intro*!: *bounded-linear.suminf*[*OF bounded-linear-blinfun-to-matrix*, *symmetric*])
finally have *Q1-le-Q2*: *blinfun-to-matrix* (*inv*$_L$ *Q1*) $\leq$ *blinfun-to-matrix*

$(inv_L\ Q2)$ **.**

have $*$: $0 \leq$ *blinfun-to-matrix* $((inv_L\ Q1)\ o_L\ R1)$ $0 \leq$ *blinfun-to-matrix* $((inv_L\ Q2)\ o_L\ R2)$
using *assms is-splitting-blin-def'*
by (*auto simp*: *blinfun-to-matrix-comp intro*: *nonneg-matrix-mult*)

have $0 \leq (id\text{-}blinfun - l *_R \mathcal{P}_1\ d)\ 1$
using *less-imp-le*[*OF disc-lt-one*]
by (*auto simp*: *blinfun.diff-left less-eq-bfun-def blinfun.scaleR-left*)
hence $(inv_L\ Q1)\ ((id\text{-}blinfun - l *_R \mathcal{P}_1\ d)\ 1) \leq (inv_L\ Q2)\ ((id\text{-}blinfun - l *_R \mathcal{P}_1\ d)\ 1)$
by (*metis Q1-le-Q2 blinfun.diff-left blinfun-to-matrix-diff diff-ge-0-iff-ge nonneg-blinfun-nonneg*)
hence $(inv_L\ Q1)\ ((Q1 - R1)\ 1) \leq (inv_L\ Q2)\ ((Q2 - R2)\ 1)$
by (*metis* (*no-types*, *opaque-lifting*) *assms*(*1*) *assms*(*2*) *is-splitting-blin-def'*)
hence $(inv_L\ Q1\ o_L\ Q1)\ 1 - (inv_L\ Q1\ o_L\ R1)\ 1 \leq (inv_L\ Q2\ o_L\ Q2)\ 1 - (inv_L\ Q2\ o_L\ R2)\ 1$
by (*auto simp*: *blinfun.add-left blinfun.diff-right blinfun.diff-left*)
hence $(inv_L\ Q2\ o_L\ R2)\ 1 \leq (inv_L\ Q1\ o_L\ R1)\ 1$
using *assms*
unfolding *is-splitting-blin-def'*
by *auto*
moreover have $0 \leq (inv_L\ Q2\ o_L\ R2)\ 1$
using $*$
by (*fastforce simp*: *less-eq-bfunI intro*!: *nonneg-blinfun-nonneg*)
ultimately have *norm* $((inv_L\ Q2\ o_L\ R2)\ 1) \leq$ *norm* $((inv_L\ Q1\ o_L\ R1)\ 1)$
by (*auto simp*: *less-eq-bfun-def norm-bfun-def' intro*!: *abs-ge-self cSUP-mono intro*: *order.trans*)
thus *norm* $((inv_L\ Q2\ o_L\ R2)) \leq$ *norm* $((inv_L\ Q1\ o_L\ R1))$
by (*auto simp*: *norm-nonneg-blinfun-one* $*$)
qed

6.5 Gauss Seidel Splitting

6.5.1 Definition of Upper and Lower Triangular Matrices

definition *P-dec d* $\equiv$ *blinfun-to-matrix* $(\mathcal{P}_1$ (*mk-dec-det d*))
definition *P-upper d* $\equiv (\chi\ i\ j.$ *if* $i \leq j$ *then P-dec d* \$ *i* \$ *j else 0*)
definition *P-lower d* $\equiv (\chi\ i\ j.$ *if* $j < i$ *then P-dec d* \$ *i* \$ *j else 0*)

definition $\mathcal{P}_U\ d$ = *matrix-to-blinfun* (*P-upper d*)
definition $\mathcal{P}_L\ d$ = *matrix-to-blinfun* (*P-lower d*)

lemma *P-dec-elem*: *P-dec d* \$ *i* \$ *j* = *pmf* (*K* (*i*, *d i*)) *j*
unfolding *blinfun-to-matrix-def matrix-def* $\mathcal{P}_1$*.rep-eq K-st-def P-dec-def push-exp.rep-eq vec-lambda-beta*
by (*subst pmf-expectation-bind*[*of* {*d i*}])

(*auto split*: *if-splits simp*: *mk-dec-det-def axis-def vec-lambda-inverse integral-measure-pmf*[*of* $\{j\}$])

lemma *nonneg-*$\mathcal{P}_U$: *nonneg-blinfun* ($\mathcal{P}_U$ *d*)
unfolding $\mathcal{P}_U$*-def Finite-Cartesian-Product.less-eq-vec-def blinfun-to-matrix-inv P-upper-def P-dec-elem*
by *auto*

lemma *nonneg-P-dec*: $0 \leq$ *P-dec d*
by (*simp add*: *Finite-Cartesian-Product.less-eq-vec-def P-dec-elem*)

lemma *nonneg-P-upper*: $0 \leq$ *P-upper d*
using *nonneg-P-dec*
by (*simp add*: *Finite-Cartesian-Product.less-eq-vec-def P-upper-def*)

lemma *nonneg-P-lower*: $0 \leq$ *P-lower d*
using *nonneg-P-dec*
by (*simp add*: *Finite-Cartesian-Product.less-eq-vec-def P-lower-def*)

lemma *nonneg-*$\mathcal{P}_L$: *nonneg-blinfun* ($\mathcal{P}_L$ *d*)
unfolding $\mathcal{P}_L$*-def Finite-Cartesian-Product.less-eq-vec-def blinfun-to-matrix-inv P-lower-def P-dec-elem*
by *auto*

lemma *nonneg-*$\mathcal{P}_1$: *nonneg-blinfun* ($\mathcal{P}_1$ *d*)
unfolding *blinfun-to-matrix-def matrix-def*
by (*auto simp*: *Finite-Cartesian-Product.less-eq-vec-def axis-def intro*!: $\mathcal{P}_1$*-pos less-eq-bfunD*[*of 0*, *simplified*])

lemma *norm-*$\mathcal{P}_L$*-le*: *norm* ($\mathcal{P}_L$ *d*) $\leq$ *norm* ($\mathcal{P}_1$ (*mk-dec-det d*))
using *nonneg-*$\mathcal{P}_1$
by (*fastforce intro*!: *matrix-le-norm-mono simp*: *Finite-Cartesian-Product.less-eq-vec-def P-dec-def P-lower-def* $\mathcal{P}_L$*-def*)

lemma *norm-*$\mathcal{P}_L$*-le-one*: *norm* ($\mathcal{P}_L$ *d*) ≤ 1
using *norm-*$\mathcal{P}_L$*-le norm-*$\mathcal{P}_1$ **by** *auto*

lemma *norm-*$\mathcal{P}_L$*-less-one*: *norm* ($l *_R \mathcal{P}_L$ *d*) < 1
using *order.strict-trans1*[*OF mult-left-le disc-lt-one*] *zero-le-disc norm-*$\mathcal{P}_L$*-le-one*
by *auto*

lemma $\mathcal{P}_L$*-le-*$\mathcal{P}_1$: $0 \leq v \Longrightarrow \mathcal{P}_L\ d\ v \leq \mathcal{P}_1$ (*mk-dec-det d*) *v*
proof −
assume $0 \leq v$
moreover have *P-lower d* $\leq$ *P-dec d*
using *nonneg-P-dec*
by (*auto simp*: *P-lower-def less-eq-vec-def*)
ultimately show *?thesis*

by (*metis P-dec-def* $\mathcal{P}_L$*-def blinfun-apply-mono blinfun-to-matrix-inv nonneg-*$\mathcal{P}_L$)
qed

lemma $\mathcal{P}_U$*-le-*$\mathcal{P}_1$: $0 \leq v \Longrightarrow \mathcal{P}_U\ d\ v \leq \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d)\ v$
proof −
assume $0 \leq v$
moreover have *P-upper d* ≤ *P-dec d*
using *nonneg-P-dec*
by (*auto simp*: *P-upper-def less-eq-vec-def*)
ultimately show *?thesis*
by (*metis P-dec-def* $\mathcal{P}_U$*-def blinfun-apply-mono blinfun-to-matrix-inv nonneg-*$\mathcal{P}_U$)
qed

lemma *row-P-upper-indep*: $d\ s = d'\ s \Longrightarrow row\ s\ (P\text{-}upper\ d) = row\ s\ (P\text{-}upper\ d')$
unfolding *row-def P-dec-elem P-upper-def*
by *auto*

lemma *row-P-lower-indep*: $d\ s = d'\ s \Longrightarrow row\ s\ (P\text{-}lower\ d) = row\ s\ (P\text{-}lower\ d')$
unfolding *row-def P-dec-elem P-lower-def*
by *auto*

lemma *triangular-mat-P-upper*: *upper-triangular-mat* (*P-upper d*)
unfolding *upper-triangular-mat-def P-upper-def*
by *auto*

lemma *slt-P-lower*: *strict-lower-triangular-mat* (*P-lower d*)
unfolding *strict-lower-triangular-mat-def P-lower-def*
by *auto*

lemma *lt-P-lower*: *lower-triangular-mat* (*P-lower d*)
unfolding *lower-triangular-mat-def P-lower-def*
by *auto*

6.5.2 Gauss Seidel is a Regular Splitting

definition $Q\text{-}GS\ d = id\text{-}blinfun - l *_R \mathcal{P}_L\ d$
definition $R\text{-}GS\ d = l *_R \mathcal{P}_U\ d$

lemma *splitting-gauss*: *is-splitting-blin* ($id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d)$) (*Q-GS d*) (*R-GS d*)
unfolding *is-splitting-blin-def'*
proof *safe*
show *nonneg-blinfun* (*R-GS d*)
unfolding *R-GS-def* $\mathcal{P}_U$*-def blinfun-to-matrix-scaleR Finite-Cartesian-Product.less-eq-vec-def blinfun-to-matrix-inv*

using *nonneg-P-upper*
by (*auto intro*!: *mult-nonneg-nonneg*)
next
have $\mathcal{P}_L\ d + \mathcal{P}_U\ d = \mathcal{P}_1$ (*mk-dec-det d*) **for** *d*
proof −
have $\mathcal{P}_L\ d + \mathcal{P}_U\ d$ = *matrix-to-blinfun* (χ *i j*. ((*blinfun-to-matrix* ($\mathcal{P}_1$ (*mk-dec-det d*)))) $ *i* $ *j*)
unfolding $\mathcal{P}_L$-*def* $\mathcal{P}_U$-*def* *P-lower-def* *P-upper-def* *P-dec-def* *matrix-to-blinfun-add*[*symmetric*]
by (*auto simp*: *vec-eq-iff intro*!: *arg-cong*[*of - - matrix-to-blinfun*])
also have ... = ($\mathcal{P}_1$ (*mk-dec-det d*))
by (*simp add*: *matrix-to-blinfun-inv*)
finally show $\mathcal{P}_L\ d + \mathcal{P}_U\ d = \mathcal{P}_1$ (*mk-dec-det d*).
qed
thus *id-blinfun* − $l *_R \mathcal{P}_1$ (*mk-dec-det d*) = *Q-GS d* − *R-GS d*
unfolding *Q-GS-def R-GS-def*
by (*auto simp*: *algebra-simps scaleR-add-right*[*symmetric*] *simp del*: *scaleR-add-right*)
next
have *n-le*: *norm* ($l *_R \mathcal{P}_L\ d$) < *1*
using *mult-left-le*[*OF norm-*$\mathcal{P}_L$*-le-one*[*of d*] *zero-le-disc*] *order.strict-trans1*
by (*auto intro*: *disc-lt-one*)
thus *invertible*$_L$ (*Q-GS d*)
by (*simp add*: *Q-GS-def invertible*$_L$*-inf-sum*)
have *inv*$_L$ (*Q-GS d*) = ($\sum$ *i*. ($l *_R \mathcal{P}_L\ d$) ^^ *i*)
using *inv*$_L$*-inf-sum n-le* **unfolding** *Q-GS-def*
by *blast*
hence ∗: *blinfun-to-matrix* (*inv*$_L$ (*Q-GS d*)) $ *i* $ *j* = ($\sum$ *k*. *blinfun-to-matrix* (($l *_R \mathcal{P}_L\ d$) ^^ *k*) $ *i* $ *j*) **for** *i j*
using *summable-inv-Q*[*of Q-GS d*] *norm-*$\mathcal{P}_L$*-less-one*
unfolding *Q-GS-def*
by (*subst bounded-linear.suminf*[*symmetric*])
(*auto intro*!: *bounded-linear-compose*[*OF bounded-linear-vec-nth*] *bounded-linear-compose*[*OF bounded-linear-blinfun-to-matrix*])
have *0* ≤ *l^i* $*_R$ *matpow* (*P-lower d*) *i* **for** *i*
using *nonneg-matpow*[*OF nonneg-P-lower*]
by (*meson scaleR-nonneg-nonneg zero-le-disc zero-le-power*)
have *0* ≤ ($\sum$ *k*. *blinfun-to-matrix* (($l *_R \mathcal{P}_L\ d$) ^^ *k*) $ *i* $ *j*) **for** *i j*
proof (*intro suminf-nonneg*)
show *summable* (λ*k*. *blinfun-to-matrix* (($l *_R \mathcal{P}_L\ d$) ^^ *k*) $ *i* $ *j*)
using *summable-inv-Q*[*of Q-GS d*] *norm-*$\mathcal{P}_L$*-less-one*
unfolding *Q-GS-def*
by (*fastforce*
simp:
blinfun-to-matrix-matpow nonneg-matrix-nonneg blincomp-scaleR-right blinfun-to-matrix-scaleR
intro:
bounded-linear.summable[*of -* λ*i*. ($l *_R \mathcal{P}_L\ d$) ^^ *i*]
bounded-linear-compose[*OF bounded-linear-vec-nth*]

bounded-linear-compose[*OF bounded-linear-blinfun-to-matrix*])
show $\bigwedge n.\ 0 \le$ *blinfun-to-matrix* $((l *_R \mathcal{P}_L\ d)$ ^^ $n)$ \$ i \$ j
using *nonneg-matpow*[*OF nonneg-P-lower*]
by (*auto simp*: $\mathcal{P}_L$-*def nonneg-matrix-nonneg blinfun-to-matrix-scaleR matpow-scaleR blinfun-to-matrix-matpow*)
qed
thus *nonneg-blinfun* (inv_L (*Q-GS d*))
by (*simp add*: * *Finite-Cartesian-Product.less-eq-vec-def*)
qed

abbreviation $r\text{-}det_b\ d \equiv r\text{-}dec_b$ (*mk-dec-det d*)
abbreviation *r-vec d* $\equiv \chi\ i.\ r\text{-}dec_b$ (*mk-dec-det d*) *i*

abbreviation *Q-mat d* $\equiv$ *blinfun-to-matrix* (*Q-GS d*)
abbreviation *R-mat d* $\equiv$ *blinfun-to-matrix* (*R-GS d*)

lemma *Q-mat-def*: *Q-mat d* = *mat 1* $-$ *l* $*_R$ *P-lower d*
unfolding *Q-GS-def*
by (*simp add*: $\mathcal{P}_L$-*def blinfun-to-matrix-diff blinfun-to-matrix-id blinfun-to-matrix-scaleR*)

lemma *R-mat-def*: *R-mat d* = *l* $*_R$ *P-upper d*
unfolding *R-GS-def*
by (*simp add*: $\mathcal{P}_U$-*def blinfun-to-matrix-scaleR*)

lemma *triangular-mat-R*: *upper-triangular-mat* (*R-mat d*)
using *triangular-mat-P-upper*
unfolding *upper-triangular-mat-def R-mat-def*
by *auto*

definition *GS-inv d v* $\equiv$ *matrix-inv* (*Q-mat d*) $*v$ (*r-vec d* + *R-mat d* $*v$ *v*)

Q-mat can be expressed as an infinite sum of *P-lower*. It is therefore lower triangular.

lemma *inv-Q-mat-suminf*: *matrix-inv* (*Q-mat d*) = ($\sum k.$ (*matpow* (*l* $*_R$ (*P-lower d*)) *k*))
proof $-$
have *matrix-inv* (*Q-mat d*) = *blinfun-to-matrix* (inv_L (*Q-GS d*))
using *blinfun-to-matrix-inverse*(*2*) *is-splitting-blin-def′ splitting-gauss*
by *metis*
also have ... = *blinfun-to-matrix* ($\sum i.\ (l *_R \mathcal{P}_L\ d)$^^$i$)
using *norm*-$\mathcal{P}_L$-*less-one*
by (*auto simp*: *Q-GS-def* inv_L-*inf-sum*)
also have ... = ($\sum k.$ (*blinfun-to-matrix* $((l *_R \mathcal{P}_L\ d)$^^$k$)))
using *summable-inv-Q*[*of Q-GS d*] *norm*-$\mathcal{P}_L$-*less-one bounded-linear-blinfun-to-matrix*
unfolding *Q-GS-def row-def*
by (*subst bounded-linear.suminf*) *auto*
also have ... = ($\sum k.$ (*matpow* (*l* $*_R$ (*P-lower d*)) *k*))

by (*simp add*: *blinfun-to-matrix-scaleR blinfun-to-matrix-matpow* $\mathcal{P}_L$*-def blinfun-to-matrix-inv*)
finally show *?thesis***.**
qed

lemma *lt-Q-inv*: *lower-triangular-mat* (*matrix-inv* (*Q-mat d*))
unfolding *inv-Q-mat-suminf*
using *summable-inv-Q*[*of Q-GS d*] *norm-*$\mathcal{P}_L$*-less-one summable-blinfun-to-matrix*[*of* $\lambda i.$ $(l *_R \mathcal{P}_L\ d)$^^*i*]
by (*intro lower-triangular-suminf lower-triangular-pow*)
(*auto simp*: *lower-triangular-mat-def P-lower-def Q-GS-def blinfun-to-matrix-scaleR blinfun-to-matrix-matpow* $\mathcal{P}_L$*-def*)

Each row of the matrix *Q-mat d* only depends on *d*'s actions in lower states.

lemma *inv-Q-mat-indep*:
assumes $\bigwedge i.\ i \le s \Longrightarrow d\ i = d'\ i\ i \le s$
shows *row i* (*matrix-inv* (*Q-mat d*)) = *row i* (*matrix-inv* (*Q-mat d'*))
proof −
have *row i* (*matrix-inv* (*Q-mat d*)) = *row i* (*blinfun-to-matrix* (inv_L (*Q-GS d*)))
using *blinfun-to-matrix-inverse*(*2*) *is-splitting-blin-def' splitting-gauss*
by *metis*
also have ... = *row i* (*blinfun-to-matrix* ($\sum i.$ $(l *_R \mathcal{P}_L\ d)$^^*i*))
using *norm-*$\mathcal{P}_L$*-less-one*
by (*auto simp*: *Q-GS-def* inv_L*-inf-sum*)
also have ... = ($\sum k.$ *row i* (*blinfun-to-matrix* (($l *_R \mathcal{P}_L\ d$)^^*k*)))
using *summable-inv-Q*[*of Q-GS d*] *norm-*$\mathcal{P}_L$*-less-one*
unfolding *Q-GS-def row-def*
by (*subst bounded-linear.suminf*[*OF bounded-linear-compose*[*OF - bounded-linear-blinfun-to-matrix*]]) *auto*
also have ... = ($\sum k.$ *row i* (*matpow* ($l *_R$ (*P-lower d*)) *k*))
by (*simp add*: *blinfun-to-matrix-matpow blinfun-to-matrix-scaleR* $\mathcal{P}_L$*-def blinfun-to-matrix-inv*)
also have ... = ($\sum k.$ *l*^*k* $*_R$ *row i* (*matpow* ((*P-lower d*)) *k*))
by (*subst matpow-scaleR*) (*auto simp*: *row-def scaleR-vec-def*)
also have ... = ($\sum k.$ *l*^*k* $*_R$ *row i* (*matpow* ((*P-lower d'*)) *k*))
using *assms*
by (*subst lower-triangular-pow-eq*[*of P-lower d'*]) (*auto simp*: *P-dec-elem lt-P-lower row-P-lower-indep*[*of d' - d*])
also have ... = ($\sum k.$ *row i* (*matpow* ($l *_R$ (*P-lower d'*)) *k*))
by (*subst matpow-scaleR*) (*auto simp*: *row-def scaleR-vec-def*)
also have ... = ($\sum k.$ *row i* (*blinfun-to-matrix* (($l *_R \mathcal{P}_L\ d'$)^^*k*)))
by (*simp add*: $\mathcal{P}_L$*-def blinfun-to-matrix-inv blinfun-to-matrix-matpow blinfun-to-matrix-scaleR*)
also have ... = *row i* (*blinfun-to-matrix* ($\sum i.$ $(l *_R \mathcal{P}_L\ d')$^^*i*))
using *summable-inv-Q*[*of Q-GS d'*] *norm-*$\mathcal{P}_L$*-less-one*
unfolding *Q-GS-def row-def*

by (*auto intro*!: *bounded-linear.suminf*[*symmetric*]
bounded-linear-compose[*OF* - *bounded-linear-blinfun-to-matrix*])
also have ... = *row i* (*blinfun-to-matrix* (inv_L (*Q-GS d'*)))
by (*metis Q-GS-def* inv_L*-inf-sum norm-*$\mathcal{P}_L$*-less-one*)
also have ... = *row i* (*matrix-inv* (*Q-mat d'*))
by (*metis blinfun-to-matrix-inverse*(*2*) *is-splitting-blin-def' splitting-gauss*)
finally show *?thesis*.
qed

As a result, also *GS-inv* is independent of lower actions.

lemma *GS-indep-high-states*:
assumes $\bigwedge s'.\ s' \leq s \Longrightarrow d\ s' = d'\ s'$
shows *GS-inv d v* \$ *s* = *GS-inv d' v* \$ *s*
proof −
have *row i* (*P-upper d*) = *row i* (*P-upper d'*) **if** $i \leq s$ **for** *i*
using *assms that row-P-upper-indep* **by** *blast*
hence *R-eq-upto-s*: *row i* (*R-mat d*) = *row i* (*R-mat d'*) **if** $i \leq s$ **for** *i*
using *that*
by (*simp add*: *row-def R-mat-def*)

have *Qr-eq*: (*matrix-inv* (*Q-mat d*) *∗v r-vec d*) \$ *s* = (*matrix-inv* (*Q-mat d'*) *∗v r-vec d'*) \$ *s*
proof −
have (*matrix-inv* (*Q-mat d*) *∗v r-vec d*) \$ *s* = ($\sum$*j*∈*UNIV*. *matrix-inv* (*Q-mat d*) \$ *s* \$ *j* ∗ *r-vec d* \$ *j*)
unfolding *matrix-vector-mult-def*
by *simp*
also have ... = ($\sum$*j*∈*UNIV*. *if s* < *j then 0 else matrix-inv* (*Q-mat d*) \$ *s* \$ *j* ∗ *r-vec d* \$ *j*)
using *lt-Q-inv*
by (*auto intro*!: *sum.cong simp*: *lower-triangular-mat-def*)
also have ... = ($\sum$*j*∈*UNIV*. *if s* < *j then 0 else matrix-inv* (*Q-mat d'*) \$ *s* \$ *j* ∗ *r-vec d* \$ *j*)
using *inv-Q-mat-indep assms*
by (*fastforce intro*!: *sum.cong simp*: *row-def*)
also have ... = (*matrix-inv* (*Q-mat d'*) *∗v r-vec d'*) \$ *s*
using *lt-Q-inv*
by (*auto simp*: *matrix-vector-mult-def assms lower-triangular-mat-def intro*!: *sum.cong*)
finally show *?thesis*.
qed

have *QR-eq*: *row s* (*matrix-inv* (*Q-mat d*) ∗∗ *R-mat d*) = *row s* (*matrix-inv* (*Q-mat d'*) ∗∗ *R-mat d'*)
proof −
have *matrix-inv* (*Q-mat d*) \$ *s* \$ *k* ∗ *R-mat d* \$ *k* \$ *j* = *matrix-inv* (*Q-mat d'*) \$ *s* \$ *k* ∗ *R-mat d'* \$ *k* \$ *j* **for** *k j*

proof −
have *matrix-inv* (*Q-mat d*) $ *s* $ *k* ∗ *R-mat d* $ *k* $ *j* =
(*if s* < *k then 0 else matrix-inv* (*Q-mat d*) $ *s* $ *k* ∗ *R-mat d* $ *k* $ *j*)
using *lower-triangular-mat-def lt-Q-inv* **by** *auto*
also have ... = (*if s* < *k then 0 else matrix-inv* (*Q-mat d′*) $ *s* $ *k* ∗ *R-mat d* $ *k* $ *j*)
by (*metis* (*no-types*, *lifting*) *Finite-Cartesian-Product.row-def assms inv-Q-mat-indep order-refl vec-lambda-eta*)
also have ... = (*if s* < *k* ∨ *j* < *k then 0 else* (*matrix-inv* (*Q-mat d′*) $ *s* $ *k* ∗ *R-mat d* $ *k* $ *j*))
using *triangular-mat-R*
unfolding *upper-triangular-mat-def*
by (*auto split*: *if-splits*)
also have ... = (*if s* < *k* ∨ *j* < *k then 0 else* (*matrix-inv* (*Q-mat d′*) $ *s* $ *k* ∗ *R-mat d′* $ *k* $ *j*))
using *R-eq-upto-s*
by (*auto simp*: *row-def*)
also have ... = *matrix-inv* (*Q-mat d′*) $ *s* $ *k* ∗ *R-mat d′* $ *k* $ *j*
by (*metis lower-triangular-mat-def lt-Q-inv mult-not-zero triangular-mat-R upper-triangular-mat-def*)
finally show *?thesis*.
qed
thus *?thesis*
unfolding *row-def matrix-matrix-mult-def*
by *auto*
qed
show *?thesis*
using *QR-eq Qr-eq*
by (*auto simp add*: *GS-inv-def vec.add row-def matrix-vector-mul-assoc matrix-vector-mult-code′*)
qed

This recursive definition mimics the computation of the GS iteration.

lemma *GS-inv-rec*: *GS-inv d v* = *r-vec d* + *l* $*_R$ (*P-upper d* ∗*v v* + *P-lower d* ∗*v* (*GS-inv d v*))
proof −
have *Q-mat d* ∗*v* (*GS-inv d v*) = *r-vec d* + *R-mat d* ∗*v v*
using *splitting-gauss*[*of d*] *blinfun-to-matrix-inverse*(*1*)
unfolding *GS-inv-def matrix-vector-mul-assoc is-splitting-blin-def′*
by (*subst matrix-inv*(*2*)) *auto*
thus *?thesis*
unfolding *Q-mat-def R-mat-def*
by (*auto simp*: *algebra-simps scaleR-matrix-vector-assoc*)
qed

lemma *is-am-GS-inv-extend*:
assumes ⋀*s*. *s* < *k* ⟹ *is-arg-max* (*λd*. *GS-inv d v* $ *s*) (*λd*. *d* ∈

D_D) d
  and *is-arg-max* ($\lambda a.$ *GS-inv* (d (k := a)) v \$ k) ($\lambda a.$ $a \in A$ k) a
  and $s \leq k$
  and $d \in D_D$
 shows *is-arg-max* ($\lambda d.$ *GS-inv* d v \$ s) ($\lambda d.$ $d \in D_D$) (d (k := a))
proof −
 have *am-k*: *is-arg-max* ($\lambda d.$ *GS-inv* d v \$ k) ($\lambda d.$ $d \in D_D$) (d (k := a))
 proof (*rule is-arg-max-linorderI*)
  fix y
  assume $y \in D_D$
  have *GS-inv* y v \$ k = (*r-vec* y + l $*_R$ (*P-upper* y $*v$ v + *P-lower* y $*v$ (*GS-inv* y v))) \$ k
   using *GS-inv-rec* **by** *auto*
  also have ... = r (k, y k) + l $*$ ((*P-upper* y $*v$ v) \$ k + (*P-lower* y $*v$ *GS-inv* y v) \$ k)
   by *auto*
  also have ... $\leq$ r (k, (d(k := y k)) k) + l $*$ ((*P-upper* (d(k := y k)) $*v$ v) \$ k + (*P-lower* (d(k := y k)) $*v$ *GS-inv* (d(k := y k)) v) \$ k)
  proof (*rule add-mono*, *goal-cases*)
   case *2*
   thus *?case*
   proof (*intro mult-left-mono add-mono*, *goal-cases*)
    case *1*
    thus *?case*
    by (*auto simp*: *matrix-vector-mult-def P-dec-elem fun-upd-same P-upper-def cong*: *if-cong*)
   next
    case *2*
    thus *?case*
    proof −
     have (*P-lower* y $*v$ *GS-inv* y v) \$ k = (*P-lower* (d(k := y k)) $*v$ *GS-inv* y v) \$ k
      unfolding *matrix-vector-mult-def*
      by (*auto simp*: *P-dec-elem fun-upd-same P-lower-def cong*: *if-cong*)
     also have ... = ($\sum j \in UNIV$. (*if* $j < k$ *then pmf* (K (k, y k)) j $*$ *GS-inv* y v \$ j *else 0*))
      by (*auto simp*: *matrix-vector-mult-def P-dec-elem P-lower-def intro*!: *sum.cong*)
     also have ... $\leq$ ($\sum j \in UNIV$. (*if* $j < k$ *then pmf* (K (k, y k)) j $*$ *GS-inv* d v \$ j *else 0*))
      using *assms* ‹$y \in D_D$›
       by (*fastforce intro*!: *sum-mono mult-left-mono dest*: *is-arg-max-linorderD*)
     also have ... = ($\sum j \in UNIV$. (*if* $j < k$ *then pmf* (K (k, y k)) j $*$ *GS-inv* (d(k := y k)) v \$ j *else 0*))
      using *GS-indep-high-states*[*of* - d(k := y k) d, *symmetric*]

by (*fastforce intro*!: *sum.cong dest*: *leD*)
also have ... = (*P-lower* ($d(k := y\ k)$) $*v$ *GS-inv* ($d(k := y$ $k)$) *v*) \$ *k*
unfolding *matrix-vector-mult-def P-lower-def P-dec-elem*
by (*fastforce intro*!: *sum.cong*)
finally show *?thesis***.**
qed
qed *auto*
qed *auto*
also have ... = (*r-vec* ($d(k := y\ k)$) + $l *_R$ ((*P-upper* ($d(k := y$ $k)$) $*v$ *v*) + (*P-lower* ($d(k := y\ k)$) $*v$ *GS-inv* ($d(k := y\ k)$) *v*))) \$ *k*
by *auto*
also have ... = *GS-inv* ($d(k := y\ k)$) *v* \$ *k*
using *GS-inv-rec* **by** *presburger*
also have ... $\leq$ *GS-inv* ($d(k := a)$) *v* \$ *k*
using *is-arg-max-linorderD*(*2*)[*OF assms*(*2*)] ‹$y \in D_D$› *is-dec-det-def*
by *blast*
finally show *GS-inv y v* \$ $k \leq$ *GS-inv* ($d(k := a)$) *v* \$ *k***.**
next
show $d(k := a) \in D_D$
using *assms*
by (*auto simp*: *is-dec-det-def is-arg-max-linorder*)
qed
show *?thesis*
proof (*cases* $s < k$)
case *True*
thus *?thesis*
using *am-k assms*(*1*)[*OF True*] *GS-indep-high-states*[*of s d* ($k :=$ a) *d*]
by (*fastforce dest*: *is-arg-max-linorderD intro*!: *is-arg-max-linorderI*)
next
case *False*
thus *?thesis*
using *assms am-k*
by *auto*
qed
qed

lemma *is-arg-max-GS-le*:
$\exists\, d.\ \forall\, s{\leq}k.$ *is-arg-max* ($\lambda d.$ *GS-inv d v* \$ *s*) ($\lambda d.\ d \in D_D$) *d*
proof (*induction k rule*: *less-induct*)
case (*less x*)
show *?case*
proof (*cases* $\exists\, y.\ y < x$)
case *True*
define *y* **where** $y = Max\ \{y.\ y < x\}$
have $y < x$
using *Max-in*

by (*simp add*: *True y-def*)
obtain *d-opt* **where** *d-opt*: *is-arg-max* ($\lambda d.$ *GS-inv d v* \$ *s*) ($\lambda d.$ $d \in D_D$) *d-opt* **if** $s \leq y$ **for** *s*
using ‹$y < x$› *less* **by** *blast*

define *d-act* **where** *d-act*: *d-act a* = *d-opt*($x := a$) **for** *a*
have *le-y*: $a < x \Longrightarrow a \leq y$ **for** *a*
by (*simp add*: *y-def*)
have *1*: *GS-inv d v* = *r-vec d* + $l *_R$ (*P-upper d* $*v$ *v* + *P-lower d* $*v$ (*GS-inv d v*)) **for** *d*
proof −
have *Q-mat d* $*v$ (*GS-inv d v*) = (*R-mat d* $*v$ *v* + *r-vec d*)
unfolding *GS-inv-def*
using *splitting-gauss*[*unfolded is-splitting-blin-def'*]
by (*auto simp*: *matrix-vector-mul-assoc matrix-inv-right*[*OF blinfun-to-matrix-inverse*(*1*)]])
thus *?thesis*
unfolding *Q-mat-def R-mat-def*
by (*auto simp*: *scaleR-matrix-vector-assoc algebra-simps*)
qed
have ($\bigsqcup d \in D_D$. *GS-inv d v* \$ *x*) = ($\bigsqcup d \in D_D$. (*r-vec d* + $l *_R$ (*P-upper d* $*v$ *v* + *P-lower d* $*v$ (*GS-inv d v*))) \$ *x*)
using *1* **by** *auto*
also have ... = ($\bigsqcup a \in A\ x$. (*r-vec* (*d-act a*) + $l *_R$ (*P-upper* (*d-act a*) $*v$ *v* + *P-lower* (*d-act a*) $*v$ (*GS-inv* (*d-act a*) *v*))) \$ *x*)
proof (*rule antisym*, *rule cSUP-mono*, *goal-cases*)
case (*3 n*)
moreover have (*P-upper n* $*v$ *v*) \$ $x \leq$ (*P-upper* (*d-opt*($x := n\ x$)) $*v$ *v*) \$ *x*
unfolding *P-upper-def matrix-vector-mult-def*
by (*auto simp*: *P-dec-elem cong*: *if-cong*)
moreover
{
have $\bigwedge j.\ j < x \Longrightarrow$ *GS-inv n v* \$ $j \leq$ *GS-inv* (*d-opt*($x := n\ x$)) *v* \$ *j*
using *d-opt*[*OF le-y*] *3*
by (*subst GS-indep-high-states*[*of - d-opt*($x := n\ x$) *d-opt*]) (*auto simp*: *is-arg-max-linorder*)
hence (*P-lower n* $*v$ *GS-inv n v*) \$ $x \leq$ (*P-lower* (*d-opt*($x := n\ x$)) $*v$ *GS-inv* (*d-opt*($x := n\ x$)) *v*) \$ *x*
unfolding *matrix-vector-mult-def P-lower-def P-dec-elem*
by (*fastforce intro*!: *mult-left-mono sum-mono*)
}
ultimately show *?case*
unfolding *d-act*
by (*auto intro*!: *bexI*[*of - n x*] *mult-left-mono add-mono simp*: *is-dec-det-def*)
next
case *4*

then show *?case*
proof (*rule cSUP-mono*, *goal-cases*)
case (*3 n*)
then show *?case*
using *d-opt*
by (*fastforce simp*: *d-act is-dec-det-def is-arg-max-linorder intro*!: *bexI*[*of - d-act n*])
qed (*auto simp*: *A-ne*)
qed *auto*
also have ... $= (\bigsqcup a \in A\ x.\ \textit{GS-inv}\ (\textit{d-act}\ a)\ v\ \$\ x)$
using *1* **by** *auto*
finally have $*$: $(\bigsqcup d \in D_D.\ \textit{GS-inv}\ d\ v\ \$\ x) = (\bigsqcup a \in A\ x.\ \textit{GS-inv}\ (\textit{d-act}\ a)\ v\ \$\ x)$**.**
then obtain *a-opt* **where** *a-opt*: *is-arg-max* $(\lambda a.\ \textit{GS-inv}\ (\textit{d-act}\ a)\ v\ \$\ x)\ (\lambda a.\ a \in A\ x)$ *a-opt*
by (*metis A-ne finite finite-is-arg-max*)
hence $(\bigsqcup d \in D_D.\ \textit{GS-inv}\ d\ v\ \$\ x) = \textit{GS-inv}\ (\textit{d-act a-opt})\ v\ \$\ x$
by (*metis* $*$ *arg-max-SUP*)
hence *am-a-opt*: *is-arg-max* $(\lambda d.\ \textit{GS-inv}\ d\ v\ \$\ x)\ (\lambda d.\ d \in D_D)$ (*d-act a-opt*)
using *a-opt d-opt d-act* **unfolding** *is-dec-det-def*
by (*fastforce dest*: *is-arg-max-linorderD*(*1*) *intro*!: *SUP-is-arg-max*)
hence *is-arg-max* $(\lambda d.\ \textit{GS-inv}\ d\ v\ \$\ x')\ (\lambda d.\ d \in D_D)$ (*d-act a-opt*) **if** $x' < x$ **for** x'
proof −
have $s' \le x' \Longrightarrow \textit{d-act a-opt}\ s' = \textit{d-opt}\ s'$ **for** s'
using *d-act that is-arg-max-linorderD*[*OF d-opt*[*OF le-y*[*OF that*]]]
by *auto*
thus *?thesis*
using *am-a-opt is-arg-max-linorderD*[*OF d-opt*[*OF le-y*[*OF that*]]]
by (*auto simp*: *GS-indep-high-states*[*of - d-act a-opt d-opt*])
qed
thus *?thesis*
by (*metis am-a-opt antisym-conv1*)
next
case *False*
thus *?thesis*
using *finite-is-arg-max*[*OF finite-*D_D]
by (*fastforce simp*: *arg-max-def someI-ex dest*!: *le-neq-trans*)
qed
qed

lemma *ex-is-arg-max-GS*:
$\exists d.\ \forall s.\ \textit{is-arg-max}\ (\lambda d.\ \textit{GS-inv}\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
using *is-arg-max-GS-le*[*of Max UNIV*]
by *auto*

function *GS-rec-fun* **where**
GS-rec-fun $v\ s = (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l * ($
$(\sum s' < s.\ pmf\ (K\ (s,a))\ s' * (GS\text{-}rec\text{-}fun\ v\ s')) +$
$(\sum s' \in \{s'.\ s \le s'\}.\ pmf\ (K\ (s,a))\ s' * v\ s')))$
by *auto*
termination
proof (*relation* $\{(x,y).\ snd\ x < snd\ y\}$, *rule wfI-min*, *goal-cases*)
case (*1 x Q*)
assume $x \in Q$
hence $*$: $\{u.\ \exists a.\ (a,\ u) \in Q\} \neq \{\}$
by (*metis* (*mono-tags*, *lifting*) ‹$x \in Q$› *prod.collapse Collect-empty-eq*)
hence $\exists a\ x.\ (a,x) \in Q \wedge x = Min\ (snd\ `\ Q)$
by (*auto simp*: *image-iff*) (*metis* (*mono-tags*, *lifting*) *equals0D Min-in*[*OF finite*] *prod.collapse image-iff*)
then obtain x **where** $x \in Q\ snd\ x = Min\ \{snd\ x|\ x.\ x \in Q\}$
by (*metis Setcompr-eq-image snd-conv*)
thus *?case*
using $*$
by (*intro bexI*[*of - x*]) *auto*
qed *auto*

declare *GS-rec-fun.simps*[*simp del*]

definition *GS-rec-elem* $v\ s\ a = r\ (s,\ a) + l * ($
$(\sum s' < s.\ pmf\ (K\ (s,a))\ s' * (GS\text{-}rec\text{-}fun\ v\ s')) +$
$(\sum s' \in \{s'.\ s \le s'\}.\ pmf\ (K\ (s,a))\ s' * v\ s'))$

lemma *GS-rec-fun-elem*: *GS-rec-fun* $v\ s = (\bigsqcup a \in A\ s.\ GS\text{-}rec\text{-}elem\ v\ s\ a)$
unfolding *GS-rec-elem-def*
using *GS-rec-fun.simps*
by *blast*

definition *GS-rec* $v = (\chi\ s.\ GS\text{-}rec\text{-}fun\ (vec\text{-}nth\ v)\ s)$

lemma *GS-rec-def'*: *GS-rec* $v\ \$\ s = (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l * ($
$(\sum s' < s.\ pmf\ (K\ (s,a))\ s' * (GS\text{-}rec\ v\ \$\ s')) +$
$(\sum s' \in \{s'.\ s \le s'\}.\ pmf\ (K\ (s,a))\ s' * v\ \$\ s')))$
unfolding *GS-rec-def*
by (*auto simp*: *GS-rec-fun.simps*[*of - s*])

lemma *GS-rec-eq*: *GS-rec* $v\ \$\ s = (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l * ($
$(P\text{-}lower\ (d(s := a)) *v\ (GS\text{-}rec\ v))\ \$\ s + (P\text{-}upper\ (d(s := a)) *v\ v)\ \$\ s))$
unfolding *GS-rec-def'*[*of v s*] *P-lower-def P-upper-def P-dec-elem matrix-vector-mult-def*
by (*auto simp*: *if-distrib*[**where** $f = \lambda x.\ x * -\ \$\ -$] *sum.If-cases lessThan-def*)
definition *GS-rec-step* $d\ v \equiv r\text{-}vec\ d + l *_R (P\text{-}lower\ d *v\ GS\text{-}rec\ v$

$+$ *P-upper d* $*v$ v)

lemma *GS-rec-eq'*: *GS-rec v* \$ $s = (\bigsqcup a \in A\ s.$ *GS-rec-step* $(d(s{:=}\ a))$ v \$ s)
using *GS-rec-eq GS-rec-step-def* **by** *auto*

lemma *GS-rec-eq-vec*:
GS-rec v \$ $s = (\bigsqcup d{\in}D_D.$ *GS-rec-step d v* \$ s)
proof –
obtain d **where** d: *is-arg-max* ($\lambda d.$ *GS-rec-step d v* \$ s) ($\lambda d.$ $d \in D_D$) d
using *finite-is-arg-max*[*OF finite*, *of* D_D] *ex-dec-det* **by** *blast*
have *GS-rec v* \$ s = *GS-rec-step d v* \$ s
unfolding *GS-rec-eq'*[*of - - d*]
proof (*intro antisym cSUP-least*)
show $\bigwedge x.\ x \in A\ s \Longrightarrow$ *GS-rec-step* $(d(s := x))$ v \$ $s \leq$ *GS-rec-step d v* \$ s
using *A-ne d*
by (*intro is-arg-max-linorderD*[*OF d*]) (*auto simp*: *is-dec-det-def is-arg-max-linorder*)
show *GS-rec-step d v* \$ $s \leq (\bigsqcup a{\in}A\ s.$ *GS-rec-step* $(d(s := a))$ v \$ s)
using d **unfolding** *is-arg-max-linorder is-dec-det-def fun-upd-triv*
by (*auto intro*!: *cSUP-upper2*[*of - - d s*])
qed (*auto simp*: *A-ne*)
thus *?thesis*
using d
by (*subst arg-max-SUP*[*symmetric*]) *auto*
qed

lift-definition $GS\text{-}rec\text{-}fun_b$ :: $('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow_b real)$ **is** *GS-rec-fun*
by *auto*

definition *GS-rec-fun-inner* $(v :: 's \Rightarrow_b real)\ s\ a \equiv r\ (s,\ a) + l * ($
$(\sum s' < s.$ *pmf* $(K\ (s,a))\ s' * (GS\text{-}rec\text{-}fun_b\ v\ s')) +$
$(\sum s' \in \{s'.\ s \leq s'\}.$ *pmf* $(K\ (s,a))\ s' * v\ s'))$

definition *GS-rec-iter* **where**
GS-rec-iter v s $= (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$
$(\sum s' \in UNIV.$ *pmf* $(K\ (s,a))\ s' * v\ s'))$

lemma *GS-rec-fun-eq-GS-iter*:
assumes $\forall s' < s.$ *v-next* s' = *GS-rec-fun v* s' $\forall s' \in \{s'.\ s \leq s'\}.$ *v-next* $s' = v\ s'$
shows *GS-rec-fun v s* = *GS-rec-iter v-next s*
proof –
have $\{s'.\ s' < s\} \cup \{s'.\ s \leq s'\} = UNIV$
by *auto*

hence $*$: $(\sum s'<s.\ f\ s') + (\sum s'\in Collect\ ((\leq)\ s).\ f\ s') = (\sum s' \in UNIV.\ f\ s')$ **for** f
by (*subst sum.union-disjoint*[*symmetric*]) (*auto simp add*: *lessThan-def*)
have *GS-rec-fun v s* = $(\bigsqcup a\in A\ s.\ r\ (s,\ a) + l * ((\sum s'<s.\ pmf\ (K\ (s,\ a))\ s' * v\text{-}next\ s') + (\sum s'\in Collect\ ((\leq)\ s).\ pmf\ (K\ (s,\ a))\ s' * v\ s')))$
using *assms*
by (*subst GS-rec-fun.simps*) *auto*
also have ... = $(\bigsqcup a\in A\ s.\ r\ (s,\ a) + l * ((\sum s'<s.\ pmf\ (K\ (s,\ a))\ s' * v\text{-}next\ s') + (\sum s'\in Collect\ ((\leq)\ s).\ pmf\ (K\ (s,\ a))\ s' * v\text{-}next\ s')))$
using *assms*
by *auto*
also have ... = *GS-rec-iter v-next s*
by (*auto simp*: $*$ *GS-rec-iter-def*)
finally show *?thesis* **.**
qed

lemma *foldl-upd-notin*: $x \notin set\ X \Longrightarrow foldl\ (\lambda f\ y.\ f(y := g\ f\ y))\ c\ X\ x = c\ x$
by (*induction X arbitrary*: *c*) *auto*

lemma *foldl-upd-notin'*: $x \notin set\ Y \Longrightarrow foldl\ (\lambda f\ y.\ f(y := g\ f\ y))\ c\ (X@Y)\ x = foldl\ (\lambda f\ y.\ f(y := g\ f\ y))\ c\ X\ x$
by (*induction X arbitrary*: *x c Y*) (*auto simp add*: *foldl-upd-notin*)

lemma *sorted-list-of-set-split*:
assumes *finite X*
shows *sorted-list-of-set X* = *sorted-list-of-set* $\{x \in X.\ x < y\}$ @ *sorted-list-of-set* $\{x \in X.\ y \leq x\}$
using *assms*
proof (*induction card X arbitrary*: *X*)
case (*Suc n X*)
have *sorted-list-of-set X* = *Min X* # *sorted-list-of-set* $(X - \{Min\ X\})$
using *Suc* **by** (*auto intro*: *sorted-list-of-set-nonempty*)
also have ... = *Min X* # *sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ x < y\}$ @ *sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ y \leq x\}$
using *Suc card.remove*[*OF Suc*(*3*) *Min-in*] *card.empty*
by (*fastforce simp*: *Suc*(*1*))+
also have ... = *sorted-list-of-set* $(\{x \in X.\ x < y\})$ @ *sorted-list-of-set* $\{x \in X.\ y \leq x\}$
proof (*cases Min X* $<$ *y*)
case *True*
hence *Min-eq*: *Min X* = *Min* $\{x \in X.\ x < y\}$
using *True Suc Min-in*
by (*subst eq-Min-iff*) *fastforce+*
have $\{x \in (X - \{Min\ X\}).\ x < y\} = \{x \in X.\ x < y\} - \{Min\ \{x \in X.\ x < y\}\}$
using *Min-eq* **by** *auto*

hence *Min X # sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ x < y\}$ =
Min $\{x \in X.\ x < y\}$ *# sorted-list-of-set* $(\{x \in X.\ x < y\} - \{Min\ \{x \in X.\ x < y\}\})$
using *Min-eq* **by** *auto*
also have ... = *sorted-list-of-set* $(\{x \in X.\ x < y\})$
using *Suc True Min-in Min-eq*
by (*subst sorted-list-of-set-nonempty*[*symmetric*]) *fastforce+*
finally have *Min X # sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ x < y\}$ = *sorted-list-of-set* $(\{x \in X.\ x < y\})$.
hence *Min X # sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ x < y\}$ @ *sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ y \le x\}$ =
sorted-list-of-set $(\{x \in X.\ x < y\})$ @ *sorted-list-of-set* $\{x \in (X - \{Min\ X\}).\ y \le x\}$
by *auto*
then show *?thesis*
using *True*
by (*auto simp*: *append-Cons*[*symmetric*] *simp del*: *append-Cons dest!*: *leD intro*: *arg-cong*)
next
case *False*
have *Min-eq*: *Min X* = *Min* $\{x \in X.\ y \le x\}$
using *False Suc Min-in*
by (*subst eq-Min-iff*) (*fastforce simp*: *linorder-class.not-less*)+
have *2*: $\{x \in (X - \{Min\ X\}).\ y \le x\} = \{x \in X.\ y \le x\} - \{Min\ \{x \in X.\ y \le x\}\}$
using *Min-eq* **by** *auto*
have $x \in X \Longrightarrow \neg\ x < y$ **for** *x*
using *False Min-less-iff Suc*(*3*) **by** *blast*
hence $\{x \in X.\ x < y\} = \{\}$
by *auto*
hence *Min X # sorted-list-of-set* $\{x \in X - \{Min\ X\}.\ x < y\}$ @ *sorted-list-of-set* $\{x \in X - \{Min\ X\}.\ y \le x\}$ =
Min X # sorted-list-of-set $\{x \in X - \{Min\ X\}.\ y \le x\}$
using *Suc* **by** *auto*
also have ... = *Min* $\{x \in X.\ y \le x\}$ *# sorted-list-of-set* $(\{x \in X.\ y \le x\} - \{Min\ \{x \in X.\ y \le x\}\})$
using *Min-eq 2*
by *auto*
also have ... = *sorted-list-of-set* $(\{x \in X.\ y \le x\})$
using *Suc False Min-in Min-eq*
by (*subst sorted-list-of-set-nonempty*[*symmetric*]) *fastforce+*
also have ... = *sorted-list-of-set* $(\{x \in X.\ x < y\})$@ *sorted-list-of-set* $(\{x \in X.\ y \le x\})$
by (*simp add*: ‹$\{x \in X.\ x < y\} = \{\}$›)
finally show *?thesis*.
qed
finally show *?case*.
qed *auto*

lemma *sorted-list-of-set-split'*:
assumes *finite X*
shows *sorted-list-of-set X* = *sorted-list-of-set* $\{x \in X.\ x \leq y\}$ @ *sorted-list-of-set* $\{x \in X.\ y < x\}$
using *sorted-list-of-set-split*[*of X*]
proof (*cases* $\exists\, x \in X.\ y < x$)
case *True*
hence $\{x \in X.\ x \leq y\} = \{x \in X.\ x < \mathit{Min}\ \{x \in X.\ y < x\}\}$
using *assms True* **by** (*subst Min-gr-iff*) *auto*
moreover have $\{x \in X.\ y < x\} = \{x \in X.\ \mathit{Min}\ \{x \in X.\ y < x\} \leq x\}$
using *assms True*
by (*subst Min-le-iff*) *auto*
ultimately show *?thesis*
using *sorted-list-of-set-split*[*OF assms*, *of Min* $\{x \in X.\ y < x\}$]
by *auto*
next
case *False*
hence $*$: $\{x \in X.\ y < x\} = \{\}$ $\{x \in X.\ x \leq y\} = X$
by (*auto simp add*:*linorder-class.not-less*)
thus *?thesis*
using *False*
by (*auto simp*: $*$)
qed

lemma *GS-rec-fun-code*: *GS-rec-fun v s* = *foldl* ($\lambda v\ s.\ v(s := $ *GS-rec-iter v s*$)$) *v* (*sorted-list-of-set* $\{..s\}$) *s*
proof (*induction s rule*: *less-induct*)
case (*less s*)
have *foldl* ($\lambda v\ s.\ v(s :=$ *GS-rec-iter v s*$)$) *v* (*sorted-list-of-set* $\{..s\}$) *s*
= *foldl* ($\lambda v\ s.\ v(s :=$ *GS-rec-iter v s*$)$) *v* (*sorted-list-of-set* $\{x \in \{..s\}.\ x < s\}$ @ *sorted-list-of-set* $\{x \in \{..s\}.\ s \leq x\}$) *s*
by (*subst sorted-list-of-set-split*[*of - s*]) *auto*
also have ... = *foldl* ($\lambda v\ s.\ v(s :=$ *GS-rec-iter v s*$)$) *v* (*sorted-list-of-set* $\{..{<}s\}$ @ *sorted-list-of-set* $\{s\}$) *s*
proof −
have $\{x \in \{..s\}.\ x < s\} = \{..{<}s\}$ $\{x \in \{..s\}.\ s \leq x\} = \{s\}$
by *auto*
thus *?thesis* **by** *auto*
qed
also have ... = *GS-rec-iter* (*foldl* ($\lambda v\ s.\ v(s :=$ *GS-rec-iter v s*$)$) *v* (*sorted-list-of-set* $\{..{<}s\}$)) *s*
by *auto*
also have ... = *GS-rec-fun v s*
proof (*intro GS-rec-fun-eq-GS-iter*[*symmetric*], *safe*, *goal-cases*)
case (*1* s')
assume $s' < s$
hence $*$: (*Collect* $((<)\ s')) \neq \{\}$
by *auto*

hence $\{x \in \{..<s\}.\ x < Min\ (Collect\ ((<)\ s'))\} = \{..s'\}$
using *leI 1*
by (*auto simp add*: *Min-gr-iff*[*OF finite*])
moreover have $\{x \in \{..<s\}.\ Min\ (Collect\ ((<)\ s')) \leq x\} = \{s'<..<s\}$
using $*$
by (*auto simp add*: *Min-le-iff*[*OF finite*])
ultimately have *foldl* ($\lambda v\ s.\ v(s := GS\text{-}rec\text{-}iter\ v\ s)$) *v* (*sorted-list-of-set* $\{..<s\}$) s'
$= foldl\ (\lambda v\ s.\ v(s := GS\text{-}rec\text{-}iter\ v\ s))\ v\ (sorted\text{-}list\text{-}of\text{-}set\ \{..s'\}\ @\ sorted\text{-}list\text{-}of\text{-}set\ \{s'<..<s\})\ s'$
by (*subst sorted-list-of-set-split*[*of - Min*$\{s.\ s' < s\}$]) *auto*
also have $\ldots = GS\text{-}rec\text{-}fun\ v\ s'$
using *1 less.IH* **by** (*subst foldl-upd-notin'*) *fastforce+*
finally show *?case*.
qed (*auto intro*: *foldl-upd-notin*)
finally show *?case*
by *metis*
qed

lemma *GS-rec-fun-code'*: $GS\text{-}rec\text{-}fun\ v\ s = foldl\ (\lambda v\ s.\ v(s := GS\text{-}rec\text{-}iter\ v\ s))\ v\ (sorted\text{-}list\text{-}of\text{-}set\ UNIV)\ s$
proof (*cases* $s = Max\ UNIV$)
case *True*
then show *?thesis*
by (*auto simp*: *GS-rec-fun-code atMost-def*)
next
case *False*
hence $*$: $(Collect\ ((<)\ s)) \neq \{\}$
by (*auto simp*: *not-le eq-Max-iff*[*OF finite*])
hence $\{x.\ x < Min\ (Collect\ ((<)\ s))\} = \{..s\}$
by (*auto simp*: *Min-less-iff*[*OF finite* $*$] *intro*: *leI*)
then show *?thesis*
unfolding *sorted-list-of-set-split*[*of UNIV Min*$\{s'.\ s < s'\}$, *OF finite*] *GS-rec-fun-code*
by (*subst foldl-upd-notin'*[*of s*]) *auto*
qed

lemma *GS-rec-fun-code''*: $GS\text{-}rec\text{-}fun\ v = foldl\ (\lambda v\ s.\ v(s := GS\text{-}rec\text{-}iter\ v\ s))\ v\ (sorted\text{-}list\text{-}of\text{-}set\ UNIV)$
using *GS-rec-fun-code'* **by** *auto*

lemma *GS-rec-eq-elem*: $GS\text{-}rec\ v\ \$\ s = GS\text{-}rec\text{-}fun\ (vec\text{-}nth\ v)\ s$
unfolding *GS-rec-def*
by *auto*

lemma *GS-rec-step-elem*: $GS\text{-}rec\text{-}step\ d\ v\ \$\ s = r\ (s,\ d\ s) + l * ((\sum s'$

$< s.$ *pmf* $(K\ (s,\ d\ s))\ s' *$ *GS-rec* v \$ $s') + (\sum s' \in \{s'.\ s \leq s'\}.$ *pmf* $(K\ (s,\ d\ s))\ s' * v$ \$ $s'))$
unfolding *GS-rec-step-def P-upper-def P-lower-def lessThan-def P-dec-elem matrix-vector-mult-def*
by (*auto simp*: *sum.If-cases algebra-simps if-distrib*[*of* $\lambda x.$ - \$ - $*$ x])

lemma *is-arg-max-GS-rec-step-act*:
assumes $d \in D_D$ *is-arg-max* ($\lambda a.$ *GS-rec-step* $(d'(s := a))$ v \$ s) ($\lambda a.$ $a \in A\ s$) a
shows *is-arg-max* ($\lambda d.$ *GS-rec-step* d v \$ s) ($\lambda d.$ $d \in D_D$) $(d(s := a))$
using *assms*
unfolding *GS-rec-step-elem is-arg-max-linorder is-dec-det-def*
by *auto*

lemma *is-arg-max-GS-rec-step-act'*:
assumes $d \in D_D$ *is-arg-max* ($\lambda a.$ *GS-rec-step* $(d'(s := a))$ v \$ s) ($\lambda a.$ $a \in A\ s$) $(d\ s)$
shows *is-arg-max* ($\lambda d.$ *GS-rec-step* d v \$ s) ($\lambda d.$ $d \in D_D$) d
using *is-arg-max-GS-rec-step-act*[*OF assms*]
by *fastforce*

lemma
is-arg-max-GS-rec:
assumes $\bigwedge s.$ *is-arg-max* ($\lambda d.$ *GS-rec-step* d v \$ s) ($\lambda d.$ $d \in D_D$) d
shows *GS-rec* v = *GS-rec-step* d v
using *arg-max-SUP*[*OF assms*]
by (*auto simp*: *vec-eq-iff GS-rec-eq-vec*)

lemma
is-arg-max-GS-rec':
assumes *is-arg-max* ($\lambda d.$ *GS-rec-step* d v \$ s) ($\lambda d.$ $d \in D_D$) d
shows *GS-rec* v \$ s = *GS-rec-step* d v \$ s
using *assms*
by (*auto simp*: *GS-rec-eq-vec arg-max-SUP*[*symmetric*])

lemma
GS-rec-eq-GS-inv:
assumes $\bigwedge s.$ *is-arg-max* ($\lambda d.$ *GS-rec-step* d v \$ s) ($\lambda d.$ $d \in D_D$) d
shows *GS-rec* v = *GS-inv* d v
proof −
have *GS-rec* v = *GS-rec-step* d v
using *is-arg-max-GS-rec*[*OF assms*]
by *auto*
hence *GS-rec* v = *r-vec* d + *R-mat* d $*v$ v + $(l *_R$ *P-lower* $d)$ $*v$ *GS-rec* v
unfolding *R-mat-def GS-rec-step-def*
by (*auto simp*: *scaleR-matrix-vector-assoc algebra-simps*)
hence *Q-mat* d $*v$ *GS-rec* v = *r-vec* d + *R-mat* d $*v$ v
unfolding *Q-mat-def*

by (*metis* (*no-types*, *lifting*) *add-diff-cancel matrix-vector-mult-diff-rdistrib matrix-vector-mul-lid*)
hence (*matrix-inv* (*Q-mat d*) $**$ *Q-mat d*) $*v$ *GS-rec v* = *matrix-inv* (*Q-mat d*) $*v$ (*r-vec d* + *R-mat d* $*v$ *v*)
by (*metis matrix-vector-mul-assoc*)
thus *GS-rec v* = *GS-inv d v*
using *splitting-gauss*
unfolding *GS-inv-def is-splitting-blin-def′*
by (*subst* (*asm*) *matrix-inv-left*) (*fastforce intro*: *blinfun-to-matrix-inverse*(*1*))+
qed

lemma
GS-rec-step-eq-GS-inv:
assumes $\bigwedge s$. *is-arg-max* (λd. *GS-rec-step d v* \$ *s*) (λd. $d \in D_D$) *d*
shows *GS-rec-step d v* = *GS-inv d v*
using *GS-rec-eq-GS-inv*[*OF assms*] *is-arg-max-GS-rec*[*OF assms*]
by *auto*

lemma *strict-lower-triangular-mat-mult*:
assumes *strict-lower-triangular-mat M* $\bigwedge i.\ i < j \Longrightarrow v$ \$ $i = v'$ \$ i
shows (M $*v$ v) \$ j = (M $*v$ v') \$ j
proof −
have (M $*v$ v) \$ j = ($\sum i \in UNIV$. (*if* $j \leq i$ *then* *0* *else* M \$ j \$ i $*$ v \$ i))
using *assms* **unfolding** *strict-lower-triangular-mat-def*
by (*auto simp*: *matrix-vector-mult-def intro*!: *sum.cong*)
also have ... = ($\sum i \in UNIV$. (*if* $j \leq i$ *then* *0* *else* M \$ j \$ i $*$ v' \$ i))
using *assms*
by (*auto intro*!: *sum.cong*)
also have ... = (M $*v$ v') \$ j
using *assms* **unfolding** *strict-lower-triangular-mat-def*
by (*auto simp*: *matrix-vector-mult-def intro*!: *sum.cong*)
finally show *?thesis***.**
qed

lemma *Q-mat-invertible*: *invertible* (*Q-mat d*)
by (*meson blinfun-to-matrix-inverse*(*1*) *is-splitting-blin-def′ splitting-gauss*)

lemma *GS-eq-GS-inv*:
assumes $\bigwedge s.\ s \leq k \Longrightarrow$ *is-arg-max* (λd. *GS-rec-step d v* \$ *s*) (λd. $d \in D_D$) *d*
assumes $s \leq k$
shows *GS-rec-step d v* \$ *s* = *GS-inv d v* \$ *s*
proof −
have $*$: *GS-rec v* \$ *s* = *GS-rec-step d v* \$ *s* **if** $s \leq k$ **for** *s*
using *assms is-arg-max-GS-rec′ that* **by** *presburger*

hence *GS-rec v* \$ *s* = (*r-vec d* + *R-mat d* ∗*v v* + (*l* $*_R$ *P-lower d*) ∗*v GS-rec v*) \$ *s* **if** *s* ≤ *k* **for** *s*
unfolding *R-mat-def GS-rec-step-def* **using** *that*
by (*simp add*: *scaleR-matrix-vector-assoc pth-6*)
hence (*Q-mat d* ∗*v GS-rec v*) \$ *s* = (*r-vec d* + *R-mat d* ∗*v v*) \$ *s* **if** *s* ≤ *k* **for** *s*
unfolding *Q-mat-def* **using** *that*
by (*simp add*: *matrix-vector-mult-diff-rdistrib*)
hence (*matrix-inv* (*Q-mat d*) ∗*v* (*Q-mat d* ∗*v GS-rec v*)) \$ *s* = (*matrix-inv* (*Q-mat d*) ∗*v* ((*r-vec d* + *R-mat d* ∗*v v*))) \$ *s*
using *assms lt-Q-inv* **by** (*auto intro*: *lower-triangular-mat-mult*)
thus *GS-rec-step d v* \$ *s* = *GS-inv d v* \$ *s*
unfolding *GS-inv-def*
using *matrix-inv-left*[*OF Q-mat-invertible*] *assms* ∗
by (*auto simp*: *matrix-vector-mul-assoc*)
qed

lemma *is-arg-max-GS-imp-splitting′*:
assumes ⋀*s*. *s* ≤ *k* ⟹ *is-arg-max* (λ*d*. *GS-rec-step d v* \$ *s*) (λ*d*. *d* ∈ D_D) *d*
assumes *s* ≤ *k*
shows *is-arg-max* (λ*d*. *GS-inv d v* \$ *s*) (λ*d*. *d* ∈ D_D) *d*
using *assms*
proof (*induction k arbitrary*: *s rule*: *less-induct*)
case (*less x*)
have *d*: *d* ∈ D_D
using *assms*(*1*) *is-arg-max-linorderD* **by** *fast*
have *is-arg-max* (λ*a*. *GS-inv* (*d*(*s* := *a*)) *v* \$ *s*) (λ*a*. *a* ∈ *A s*) (*d s*) **if** *s* ≤ *x* **for** *s*
proof −
have *is-arg-max* (λ*a*. *GS-rec-step* (*d*(*s* := *a*)) *v* \$ *s*) (λ*a*. *a* ∈ *A s*) (*d s*)
using *less*(*2*)[*OF that*]
unfolding *is-dec-det-def is-arg-max-linorder*
by *simp*
hence ∗: *is-arg-max* (λ*a*. *r* (*s*, *a*) + *l* ∗ ((*P-lower* (*d*(*s* := *a*)) ∗*v GS-rec v*) \$ *s* + (*P-upper* (*d*(*s* := *a*)) ∗*v v*) \$ *s*)) (λ*a*. *a* ∈ *A s*) (*d s*)
unfolding *GS-rec-step-def*
by *auto*
have *is-arg-max* (λ*a*. *r* (*s*, *a*) + *l* ∗ ((*P-lower* (*d*(*s* := *a*)) ∗*v GS-inv* (*d*(*s* := *a*)) *v*) \$ *s* + (*P-upper* (*d*(*s* := *a*)) ∗*v v*) \$ *s*)) (λ*a*. *a* ∈ *A s*) (*d s*)
proof −
have ((*P-lower* (*d*(*s* := *a*)) ∗*v GS-rec v*) \$ *s* = ((*P-lower* (*d*(*s* := *a*)) ∗*v GS-rec-step d v*) \$ *s*)) **for** *a*
using *is-arg-max-GS-rec′ less*(*2*) *that*
by (*auto intro*!: *lower-triangular-mat-mult*[*OF lt-P-lower*])
moreover have ((*P-lower* (*d*(*s* := *a*)) ∗*v GS-rec-step d v*) \$ *s*) = (*P-lower* (*d*(*s* := *a*)) ∗*v GS-inv d v*) \$ *s* **for** *a*

using *less*(*2*) *that GS-eq-GS-inv*
by (*fastforce intro*!: *lower-triangular-mat-mult*[*OF lt-P-lower*])
moreover have (*P-lower* (*d*(*s* := *a*)) ∗*v GS-inv d v*) $ *s* = (*P-lower* (*d*(*s* := *a*)) ∗*v GS-inv* (*d*(*s* := *a*)) *v*) $ *s* **for** *a*
using *GS-indep-high-states*[*of* - *d d*(*s* := *a*)]
by (*fastforce intro*!: *strict-lower-triangular-mat-mult*[*OF slt-P-lower*] *dest*!: *leD*)
ultimately show *?thesis*
using ∗
by *auto*
qed
hence *is-arg-max* (λ*a*. ((*r-vec* (*d*(*s* := *a*)) + *l* $*_R$ ((*P-lower* (*d*(*s* := *a*)) ∗*v GS-inv* (*d*(*s* := *a*)) *v*) + (*P-upper* (*d*(*s* := *a*)) ∗*v v*))) $ *s*)) (λ*a*. *a* ∈ *A s*) (*d s*)
by *auto*
hence ∗∗: *is-arg-max* (λ*a*. ((*r-vec* (*d*(*s* := *a*)) + *R-mat* (*d*(*s* := *a*)) ∗*v v*) + ((*l* $*_R$ *P-lower* (*d*(*s* := *a*))) ∗*v GS-inv* (*d*(*s* := *a*)) *v*)) $ *s*) (λ*a*. *a* ∈ *A s*) (*d s*)
unfolding *R-mat-def*
by (*auto simp*: *algebra-simps scaleR-matrix-vector-assoc*)
show *?thesis*
proof−
have (*r-vec d* + *R-mat d* ∗*v v*) = *Q-mat d* ∗*v* (*GS-inv d v*) **for** *d v*
unfolding *GS-inv-def matrix-vector-mul-assoc*
by (*metis* (*no-types*, *lifting*) *blinfun-to-matrix-inverse*(*1*) *is-splitting-blin-def′ matrix-inv*(*2*) *matrix-vector-mul-lid splitting-gauss*)
hence ((*r-vec d* + *R-mat d* ∗*v v*) + ((*l* $*_R$ *P-lower d*)) ∗*v GS-inv d v*) = *GS-inv d v* **for** *d*
unfolding *Q-mat-def*
by (*auto simp*: *matrix-vector-mult-diff-rdistrib*)
thus *?thesis*
using ∗∗
by *presburger*
qed
qed
thus *?case*
using *less d*
by (*fastforce intro*!: *is-am-GS-inv-extend*[*of x v d d x s*, *unfolded fun-upd-triv*])
qed

lemma *is-am-GS-rec-step-indep*:
assumes *d s* = *d′ s*
assumes *is-arg-max* (λ*d*. *GS-rec-step d v* $ *s*) (λ*d*. *d* ∈ D_D) *d*
shows *GS-rec v* $ *s* = *GS-rec-step d′ v* $ *s*
proof −
have *GS-rec v* $ *s* = *GS-rec-step d v* $ *s*
using *is-arg-max-GS-rec′ assms*(*2*) **by** *blast*

moreover have *GS-rec-step d v* \$ *s* = *GS-rec-step d′ v* \$ *s*
using *GS-rec-step-elem assms*(*1*) **by** *fastforce*
ultimately show *?thesis* **by** *auto*
qed

lemma *is-am-GS-rec-step-indep′*:
assumes *d s* = *d′ s*
assumes *is-arg-max* (λ*d*. *GS-rec-step d v* \$ *s*) (λ*d*. $d \in D_D$) *d*
shows *GS-rec v* \$ *s* = *GS-rec-step d′ v* \$ *s*
proof −
have *GS-rec v* \$ *s* = *GS-rec-step d v* \$ *s*
using *is-arg-max-GS-rec′ assms*(*2*) **by** *blast*
moreover have *GS-rec-step d v* \$ *s* = *GS-rec-step d′ v* \$ *s*
using *GS-rec-step-elem assms*(*1*) **by** *fastforce*
ultimately show *?thesis* **by** *auto*
qed

lemma *is-arg-max-GS-imp-splitting″*:
assumes ⋀*s*. $s \le k \Longrightarrow$ *is-arg-max* (λ*d*. *GS-inv d v* \$ *s*) (λ*d*. $d \in D_D$) *d*
assumes $s \le k$
shows *is-arg-max* (λ*d*. *GS-rec-step d v* \$ *s*) (λ*d*. $d \in D_D$) *d* ∧ *GS-inv d v* \$ *s* = *GS-rec v* \$ *s*
using *assms*
proof (*induction k arbitrary*: *s rule*: *less-induct*)
case (*less x*)
have *d*[*simp*]: $d \in D_D$ **using** *assms* **unfolding** *is-arg-max-linorder* **by** *blast*

have *is-arg-max* (λ*a*. *GS-rec-step* (*d*(*s* := *a*)) *v* \$ *s*) (λ*a*. *a* ∈ *A s*) (*d s*) **if** $s \le x$ **for** *s*
proof −
have *is-arg-max* (λ*a*. *GS-inv* (*d*(*s* := *a*)) *v* \$ *s*) (λ*a*. *a* ∈ *A s*) (*d s*)
using *less*(*2*)[*OF that*]
unfolding *is-dec-det-def is-arg-max-linorder*
by *auto*
hence ∗: *is-arg-max* (λ*a*. (*r-vec* (*d*(*s* := *a*)) + *l* $*_R$ (*P-lower* (*d*(*s* := *a*)) ∗*v* (*GS-inv* (*d*(*s* := *a*)) *v*) + *P-upper* (*d*(*s* := *a*)) ∗*v v*)) \$ *s*) (λ*a*. *a* ∈ *A s*) (*d s*)
by (*subst* (*asm*) *GS-inv-rec*) (*auto simp*: *add.commute*)

hence ∗: *is-arg-max* (λ*a*. (*r-vec* (*d*(*s* := *a*)) + *l* $*_R$ (*P-lower* (*d*(*s* := *a*)) ∗*v* (*GS-inv d v*) + *P-upper* (*d*(*s* := *a*)) ∗*v v*)) \$ *s*) (λ*a*. *a* ∈ *A s*) (*d s*)
proof −
have (*P-lower* (*d*(*s* := *a*)) ∗*v* (*GS-inv* (*d*(*s* := *a*)) *v*)) \$ *s* = (*P-lower* (*d*(*s* := *a*)) ∗*v* (*GS-inv d v*)) \$ *s* **for** *a*
using *GS-indep-high-states*[*of* - *d*(*s* := *a*) *d v*]
by (*rule strict-lower-triangular-mat-mult*[*OF slt-P-lower*]) (*metis*

array-rules(*4*) *leD*)
      thus *?thesis* **using** $*$ **by** *auto*
    qed
    thus *is-arg-max* (λa. *GS-rec-step* (*d*(*s* := *a*)) *v* $ *s*) (λa. $a \in A$ *s*) (*d s*)
    proof –
      have (*P-lower* (*d*(*s* := *a*)) $*v$ (*GS-inv d v*)) $ *s* = (*P-lower* (*d*(*s* := *a*)) $*v$ (*GS-rec v*)) $ *s* **for** *a*
        using *less*(*1*) *less*(*2*)*that*
        by (*intro strict-lower-triangular-mat-mult*[*OF slt-P-lower*]) *force*
      thus *?thesis*
        using $*$
        unfolding *GS-rec-step-def*
        by *auto*
    qed
  qed
  hence $*$: $\bigwedge s.\ s \le x \Longrightarrow$ *is-arg-max* (λd. *GS-rec-step d v* $ *s*) (λd. $d \in D_D$) *d*
    using *d*
    by (*intro is-arg-max-GS-rec-step-act*$'$[*of d d*]) *auto*
  moreover have *GS-inv d v* $ *s* = *GS-rec v* $ *s* **if** $s \le x$ **for** *s*
  proof –
    have *GS-rec v* $ *s* = *GS-rec-step d v* $ *s*
      using $*$[*OF that*]
      by (*auto simp*: *is-arg-max-GS-rec*$'$)
    thus *?thesis*
      using $*$ *GS-eq-GS-inv that* **by** *presburger*
  qed
  ultimately show *?case* **using** *less* **by** *blast*
qed

lemma *is-arg-max-GS-imp-splitting*$'''$:
  assumes $\bigwedge s.\ s \le k \Longrightarrow$ *is-arg-max* (λd. *GS-inv d v* $ *s*) (λd. $d \in D_D$) *d*
  assumes $s \le k$
  shows *is-arg-max* (λd. *GS-rec-step d v* $ *s*) (λd. $d \in D_D$) *d*
  using *assms* *is-arg-max-GS-imp-splitting*$''$ **by** *blast*

lemma *is-arg-max-GS-imp-splitting*:
  assumes $\bigwedge s.$ *is-arg-max* (λd. *GS-rec-step d v* $ *s*) (λd. $d \in D_D$) *d*
  shows *is-arg-max* (λd. *GS-inv d v* $ *k*) (λd. $d \in D_D$) *d*
  using *assms is-arg-max-GS-imp-splitting*$'$[*of Max UNIV*]
  by (*simp add*: *is-arg-max-linorder*)

lemma *is-arg-max-gs-iff*:
  assumes $d \in D_D$
  shows ($\forall s \le k$. *is-arg-max* (λd. *GS-inv d v* $ *s*) (λd. $d \in D_D$) *d*) $\longleftrightarrow$
    ($\forall s \le k$. *is-arg-max* (λd. *GS-rec-step d v* $ *s*) (λd. $d \in D_D$) *d*)

using *is-arg-max-GS-imp-splitting'* *is-arg-max-GS-imp-splitting''*
by *meson*

lemma *GS-opt-indep-high*:
assumes $(\bigwedge s'. \ s' < s \Longrightarrow$ *is-arg-max* $(\lambda d.$ *GS-rec-step* $d \ v \ \$ \ s')$ *is-dec-det* $d)$ $s' < s$ $a \in A \ s$
shows *is-arg-max* $(\lambda d.$ *GS-rec-step* $d \ v \ \$ \ s')$ *is-dec-det* $(d(s := a))$
proof (*rule is-arg-max-linorderI*)
fix y
assume *is-dec-det* y
hence *GS-rec-step* $y \ v \ \$ \ s' \leq r \ (s', d \ s') + l * ($*P-lower* $d * v$ *GS-rec* $v) \ \$ \ s' + l * ($*P-upper* $d * v \ v) \ \$ \ s'$
using *is-arg-max-linorderD*[*OF assms*(*1*)]
by (*auto simp*: *GS-rec-step-def algebra-simps assms*(*2*))
also have ... $= r \ (s', (d(s := a)) \ s') + l * ($*P-lower* $(d(s := a)) * v$ *GS-rec* $v) \ \$ \ s' + l * ($*P-upper* $(d(s := a)) * v \ v) \ \$ \ s'$
proof −
have $($*P-lower* $d * v$ *GS-rec* $v) \ \$ \ s' = ($*P-lower* $(d(s := a)) * v$ *GS-rec* $v) \ \$ \ s'$
using *assms*
by (*fastforce simp*: *matrix-vector-mult-def P-lower-def P-dec-elem intro*!: *sum.cong*)
moreover have $($*P-upper* $d * v \ v) \ \$ \ s' = ($*P-upper* $(d(s := a)) * v \ v) \ \$ \ s'$
using *assms*
by (*fastforce simp*: *matrix-vector-mult-def P-upper-def P-dec-elem intro*!: *sum.cong*)
ultimately show *?thesis*
using *assms*(*2*) **by** *force*
qed
also have ... $=$ *GS-rec-step* $(d(s := a)) \ v \ \$ \ s'$
by (*auto simp*: *GS-rec-step-def algebra-simps*)
finally show *GS-rec-step* $y \ v \ \$ \ s' \leq$ *GS-rec-step* $(d(s := a)) \ v \ \$ \ s'$**.**
next
show *is-dec-det* $(d(s := a))$
using *is-arg-max-linorderD*[*OF assms*(*1*)[*OF assms*(*2*)]] *assms*(*3*) *is-dec-det-def*
by *fastforce*
qed

lemma *mult-mat-vec-nth*: $(X * v \ x) \ \$ \ i =$ *scalar-product* $($*row* $i \ X) \ x$
by (*simp add*: *matrix-vector-mult-def row-def scalar-product-def*)

lemma *ext-GS-opt-le*:
assumes $(\bigwedge s'. \ s' < s \Longrightarrow$ *is-arg-max* $(\lambda d.$ *GS-rec-step* $d \ v \ \$ \ s')$ $(\lambda d. \ d \in D_D) \ d)$
and *is-arg-max* $(\lambda a.$ *GS-rec-step* $(d(s := a)) \ v \ \$ \ s)$ $(\lambda a. \ a \in A \ s)$

a $s' \leq s$
and $d \in D_D$
shows *is-arg-max* $(\lambda d.$ *GS-rec-step* d v \$ $s')$ $(\lambda d.$ $d \in D_D)$ $(d(s :=$ $a))$
using *assms is-arg-max-GS-rec-step-act is-arg-max-linorderD*(*1*)
by (*cases* $s = s'$) (*auto intro*!: *GS-opt-indep-high*)

lemma *ex-GS-opt-le*:
shows $\exists\, d.$ $(\forall\, s' \leq s.$ *is-arg-max* $(\lambda d.$ *GS-rec-step* d v \$ $s')$ $(\lambda d.$ $d \in$ $D_D)$ $d)$
proof (*induction s rule*: *less-induct*)
case (*less x*)
show *?case*
proof (*cases* $\exists\, y.$ $y < x$)
case *True*
hence $\{y.\ y < x\} \neq \{\}$
by *auto*
have *1*: $\bigwedge y.\ y \leq Max\ \{y.\ y < x\} \longleftrightarrow y < x$
using *True*
by (*auto simp*: *Max-ge-iff*[*OF finite*])
obtain d **where** d: *is-arg-max* $(\lambda d.$ *GS-rec-step* d v \$ $s')$ $(\lambda d.$ $d \in$ $D_D)$ d **if** $s'< x$ **for** s'
using *less*[*of* $Max\ \{y.\ y < x\}$] *1*
by *auto*
obtain a **where** a: *is-arg-max* $(\lambda a.$ *GS-rec-step* $(d(x := a))$ v \$ $x)$ $(\lambda a.$ $a \in A$ $x)$ a
using *finite-is-arg-max*[*OF finite A-ne*]
by *blast*
hence d': *is-arg-max* $(\lambda d.$ *GS-rec-step* d v \$ $s')$ $(\lambda d.$ $d \in D_D)$ $(d(x$ $:= a))$ **if** $s' < x$ **for** s'
using d *GS-opt-indep-high that is-arg-max-linorderD*(*1*)[*OF a*]
by *simp*
have d': *is-arg-max* $(\lambda d.$ *GS-rec-step* d v \$ $s')$ $(\lambda d.$ $d \in D_D)$ $(d(x$ $:= a))$ **if** $s' \leq x$ **for** s'
using *that a is-arg-max-linorderD*[*OF d*] *True*
by (*fastforce intro*!: *ext-GS-opt-le*[*OF d*])
thus *?thesis*
by *blast*
next
case *False*
define d **where** d $y = (SOME\ a.\ a \in A\ y)$ **for** y
obtain a **where** a: *is-arg-max* $(\lambda a.$ *GS-rec-step* $(d(x := a))$ v \$ $x)$ $(\lambda a.$ $a \in A$ $x)$ a
using *finite-is-arg-max*[*OF finite A-ne*]
by *blast*
have *1*: $y \leq x \Longrightarrow y = x$ **for** y
using *False*
by (*meson le-neq-trans*)
have *is-arg-max* $(\lambda d.$ *GS-rec-step* d v \$ $x)$ $(\lambda d.$ $d \in D_D)$ $(d(x :=$

a))
  using *False a SOME-is-dec-det* **unfolding** *d-def*
  by (*fastforce intro*!: *is-arg-max-GS-rec-step-act*)
 then show *?thesis*
  using *1*
  by *blast*
 qed
qed

lemma *ex-GS-opt*:
 shows $\exists\, d.\ \forall\, s.$ *is-arg-max* ($\lambda d.$ *GS-rec-step d v* \$ *s*) ($\lambda d.\ d \in D_D$) *d*
 using *ex-GS-opt-le*[*of Max UNIV*]
 by *auto*

lemma *GS-rec-eq-GS-inv'*: *GS-rec v* \$ *s* = ($\bigsqcup d{\in}D_D.$ *GS-inv d v* \$ *s*)
proof −
 obtain *d* **where** *d*: ($\bigwedge s.$ *is-arg-max* ($\lambda d.$ *GS-rec-step d v* \$ *s*) ($\lambda d.$
$d \in D_D$) *d*)
  using *ex-GS-opt* **by** *blast*
 have ($\bigsqcup d{\in}D_D.$ *GS-rec-step d v* \$ *s*) = *GS-rec-step d v* \$ *s*
  using *d is-arg-max-GS-rec GS-rec-eq-vec*
  by *metis*
 have ($\bigsqcup d{\in}D_D.$ *GS-inv d v* \$ *s*) = *GS-inv d v* \$ *s*
  using *is-arg-max-GS-imp-splitting*[*OF d*]
  by (*subst arg-max-SUP*[*symmetric*]) *auto*
 thus *?thesis*
  using *GS-rec-eq-GS-inv d*
  by *presburger*
qed

lemma *GS-rec-fun-eq-GS-inv*: *GS-rec-fun v s* = ($\bigsqcup d{\in}D_D.$ *GS-inv d*
(*vec-lambda v*) \$ *s*)
 using *GS-rec-eq-GS-inv'*[*of vec-lambda v*]
 unfolding *GS-rec-def*
 by (*auto simp*: *vec-lambda-inverse*)

lemma *invertible-Q-GS*: $invertible_L$ (*Q-GS d*) **for** *d*
 by (*simp add*: *Q-mat-invertible* $invertible\text{-}invertible_L$*-I*(*1*))

lemma *ex-opt-blinfun*: $\exists\, d.\ \forall\, s.$ *is-arg-max* ($\lambda d.$ ((inv_L (*Q-GS d*))
($r\text{-}det_b$ *d* + (*R-GS d*) *v*)) *s*) *is-dec-det d*
proof −
 have *GS-inv d* (*vec-lambda v*) \$ *s* = inv_L (*Q-GS d*) ($r\text{-}det_b$ *d* +
R-GS d v) *s* **for** *d s*
  unfolding *GS-inv-def plus-bfun-def*
  by (*simp add*: *invertible-Q-GS blinfun-to-matrix-mult' blinfun-to-matrix-inverse*(*2*)[*symmetric*]
apply-bfun-inverse)
 moreover obtain *d* **where** *is-arg-max* ($\lambda d.$ *GS-inv d* (*vec-lambda*

v) \$ *s*) *is-dec-det d* **for** *s*
  using *ex-GS-opt*[*of vec-lambda v*] *is-arg-max-GS-imp-splitting*
  by *auto*
 ultimately show *?thesis*
  by *auto*
qed

lemma *GS-inv-blinfun-to-matrix*: ((inv_L (*Q-GS d*)) (*r-det*$_b$ *d* + *R-GS d v*)) = *Bfun* (*vec-nth* (*GS-inv d* (*vec-lambda v*)))
 unfolding *GS-inv-def plus-bfun-def*
 by (*auto simp*: *invertible-Q-GS blinfun-to-matrix-inverse*(*2*)[*symmetric*] *blinfun-to-matrix-mult''* *apply-bfun-inverse*)

lemma *norm-GS-QR-le-disc*: *norm* (inv_L (*Q-GS d*) o_L *R-GS d*) ≤ *l*
proof −
 have *norm* (inv_L (*Q-GS d*) o_L *R-GS d*) ≤ *norm* (inv_L ((λ-. *id-blinfun*) *d*) o_L (*l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det d*)))
 proof (*rule norm-splitting-le*[*of mk-dec-det d*], *goal-cases*)
  case *1*
  then show *?case*
   unfolding *is-splitting-blin-def'*
    by (*auto simp*: *nonneg-id-blinfun blinfun-to-matrix-scaleR nonneg-*$\mathcal{P}_1$ *scaleR-nonneg-nonneg*)
 next
  case *3*
  then show *?case*
  unfolding *R-mat-def P-upper-def Finite-Cartesian-Product.less-eq-vec-def*
   using *nonneg-P-dec*
  by (*auto simp*: *P-dec-def nonneg-matrix-nonneg blinfun-to-matrix-scaleR*)
 qed (*auto simp*: *splitting-gauss*)
 also have ... = *norm* ((*l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det d*)))
  by *auto*
 also have ... ≤ *l*
  by *auto*
 finally show *?thesis*.
qed

sublocale *GS*: *MDP-QR A K r l Q-GS R-GS*
 rewrites *GS*.$\mathcal{L}_b$*-split* = *GS-rec-fun*$_b$
proof −
 have (⨆ *d*∈D_D. *norm* (inv_L (*Q-GS d*) o_L *R-GS d*)) < *1*
  using *norm-GS-QR-le-disc ex-dec-det*
  by (*fastforce intro*: *le-less-trans*[*of - l 1*] *intro*!: *cSUP-least*)
 thus *MDP-QR A K r l Q-GS R-GS*
  by *unfold-locales* (*auto simp*: *splitting-gauss ex-opt-blinfun*)
 thus *MDP-QR*.$\mathcal{L}_b$*-split A r Q-GS R-GS* = *GS-rec-fun*$_b$
  by (*fastforce simp*: *MDP-QR*.$\mathcal{L}_b$*-split.rep-eq MDP-QR*.$\mathcal{L}$*-split-def GS-rec-fun*$_b$*.rep-eq GS-rec-fun-eq-GS-inv GS-inv-blinfun-to-matrix*)
qed

abbreviation *gs-measure* $\equiv$ (λ(*eps*, *v*).
if $v = \nu_b$*-opt* $\vee$ $l = 0$
then 0
else nat (*ceiling* (*log* (*1*/*l*) (*dist v* ν_b*-opt*) $-$ *log* (*1*/*l*) (*eps* $*$ (*1*$-$*l*) / (*8* $*$ *l*)))))

lemma *dist-*$\mathcal{L}_b$*-split-lt-dist-opt*: *dist v* (*GS-rec-fun*$_b$ *v*) $\leq$ *2* $*$ *dist v* ν_b*-opt*
proof $-$
have *le1*: *dist v* (*GS-rec-fun*$_b$ *v*) $\leq$ *dist v* ν_b*-opt* + *dist* (*GS-rec-fun*$_b$ *v*) ν_b*-opt*
by (*simp add*: *dist-triangle dist-commute*)
have *le2*: *dist* (*GS-rec-fun*$_b$ *v*) ν_b*-opt* $\leq$ *GS.QR-disc* $*$ *dist v* ν_b*-opt*
using *GS.*$\mathcal{L}_b$*-split-contraction GS.*$\mathcal{L}_b$*-split-fix*
by (*metis* (*no-types*, *lifting*))
show *?thesis*
using *mult-right-mono*[*of GS.QR-disc 1*] *GS.QR-contraction*
by (*fastforce intro*!: *order.trans*[*OF le2*] *order.trans*[*OF le1*])
qed

lemma *GS-QR-disc-le-disc*: *GS.QR-disc* $\leq$ *l*
using *norm-GS-QR-le-disc ex-dec-det*
by (*fastforce intro*!: *cSUP-least*)

lemma *gs-rel-dec*:
assumes $l \neq 0$ *GS-rec-fun*$_b$ *v* $\neq$ ν_b*-opt*
shows $\lceil$*log* (*1* / *l*) (*dist* (*GS-rec-fun*$_b$ *v*) ν_b*-opt*) $-$ *c*$\rceil$ $<$ $\lceil$*log* (*1* / *l*) (*dist v* ν_b*-opt*) $-$ *c*$\rceil$
proof $-$
have *log* (*1* / *l*) (*dist* (*GS-rec-fun*$_b$ *v*) ν_b*-opt*) $-$ *c* $\leq$ *log* (*1* / *l*) (*l* $*$ *dist v* ν_b*-opt*) $-$ *c*
using *GS.*$\mathcal{L}_b$*-split-contraction*[*of - * ν_b*-opt*] *GS.QR-contraction norm-GS-QR-le-disc disc-lt-one GS-QR-disc-le-disc*
by (*fastforce simp*: *assms less-le intro*!: *log-le order.trans*[*OF GS.*$\mathcal{L}_b$*-split-contraction*[*of v* ν_b*-opt*, *simplified*]] *mult-right-mono*)
also have ... = *log* (*1* / *l*) *l* + *log* (*1*/*l*) (*dist v* ν_b*-opt*) $-$ *c*
using *assms disc-lt-one*
by (*auto simp*: *less-le intro*!: *log-mult*)
also have ... = $-$(*log* (*1* / *l*) (*1*/*l*)) + (*log* (*1*/*l*) (*dist v* ν_b*-opt*)) $-$ *c*
using *assms disc-lt-one*
by (*subst log-inverse*[*symmetric*]) (*auto simp*: *less-le right-inverse-eq*)
also have ... = (*log* (*1*/*l*) (*dist v* ν_b*-opt*)) $-$ *1* $-$ *c*
using *assms order.strict-implies-not-eq*[*OF disc-lt-one*]
by (*auto intro*!: *log-eq-one neq-le-trans*)
finally have *log* (*1* / *l*) (*dist* (*GS-rec-fun*$_b$ *v*) ν_b*-opt*) $-$ *c* $\leq$ *log* (*1* / *l*) (*dist v* ν_b*-opt*) $-$ *1* $-$ *c* **.**
thus *?thesis*

by *linarith*
qed

function *gs-iteration* :: *real* ⇒ ('*s* ⇒$_b$ *real*) ⇒ ('*s* ⇒$_b$ *real*) **where**
gs-iteration eps v =
(*if 2 ∗ l ∗ dist v* (*GS-rec-fun*$_b$ *v*) < *eps* ∗ (*1*−*l*) ∨ *eps* ≤ *0 then GS-rec-fun*$_b$ *v else gs-iteration eps* (*GS-rec-fun*$_b$ *v*))
by *auto*
termination
proof (*relation Wellfounded.measure gs-measure*, (*simp*; *fail*), *cases l* = *0*)
case *False*
fix *eps v*
assume *h*: ¬ (*2* ∗ *l* ∗ *dist v* (*GS-rec-fun*$_b$ *v*) < *eps* ∗ (*1* − *l*) ∨ *eps* ≤ *0*)
show ((*eps*, *GS-rec-fun*$_b$ *v*), *eps*, *v*) ∈ *Wellfounded.measure gs-measure*
proof −
have *gt-zero*[*simp*]: *l* ≠ *0 eps* > *0* **and** *dist-ge*: *eps* ∗ (*1* − *l*) ≤ *dist v* (*GS-rec-fun*$_b$ *v*) ∗ (*2* ∗ *l*)
using *h*
by (*auto simp*: *algebra-simps*)
have *v-not-opt*: *v* ≠ ν_b*-opt*
using *h*
by *auto*
have *log* (*1* / *l*) (*eps* ∗ (*1* − *l*) / (*8* ∗ *l*)) < *log* (*1* / *l*) (*dist v* ν_b*-opt*)
proof (*intro log-less*)
show *1* < *1* / *l*
by (*auto intro*!: *mult-imp-less-div-pos intro*: *neq-le-trans*)
show *0* < *eps* ∗ (*1* − *l*) / (*8* ∗ *l*)
by (*auto intro*!: *mult-imp-less-div-pos intro*: *neq-le-trans*)
show *eps* ∗ (*1* − *l*) / (*8* ∗ *l*) < *dist v* ν_b*-opt*
using *dist-pos-lt*[*OF v-not-opt*] *dist-*$\mathcal{L}_b$*-split-lt-dist-opt*[*of v*] *gt-zero zero-le-disc*
mult-strict-left-mono[*of dist v* (*GS-rec-fun*$_b$ *v*) (*4* ∗ *dist v* ν_b*-opt*) *l*]
by (*intro mult-imp-div-pos-less le-less-trans*[*OF dist-ge*], *argo*+)
qed
thus *?thesis*
using *gs-rel-dec h*
by *auto*
qed
qed *auto*

lemma *THE-fix-GS-rec-fun*$_b$: (*THE v. GS-rec-fun*$_b$ *v* = *v*) = ν_b*-opt*
using *GS.*$\mathcal{L}_b$*-lim*(*1*) *GS.*$\mathcal{L}_b$*-split-fix*
by *blast*+

The distance between an estimate for the value and the optimal value can be bounded with respect to the distance between the estimate and the result of applying it to $\mathcal{L}_b$

lemma *contraction-$\mathcal{L}$-split-dist*: $(1 - l) * dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (GS\text{-}rec\text{-}fun_b\ v)$
  using *GS-QR-disc-le-disc*
  by (*fastforce*
      simp: *THE-fix-GS-rec-fun$_b$*
       intro: *order.trans*[*OF - contraction-dist*, *of - l*] *order.trans*[*OF GS.$\mathcal{L}_b$-split-contraction*] *mult-right-mono*)+

lemma *dist-$\mathcal{L}_b$-split-opt-eps*:
  assumes $eps > 0$ $2 * l * dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) < eps * (1-l)$
  shows $dist\ (GS\text{-}rec\text{-}fun_b\ v)\ \nu_b\text{-}opt < eps / 2$
proof −
  have $dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) / (1 - l)$
    using *contraction-$\mathcal{L}$-split-dist*
    by (*simp add*: *mult.commute pos-le-divide-eq*)
  hence $2 * l * dist\ v\ \nu_b\text{-}opt \leq 2 * l * (dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) / (1 - l))$
    using *contraction-$\mathcal{L}$-dist assms mult-le-cancel-left-pos*[*of* $2 * l$]
    by (*fastforce intro*!: *mult-left-mono*[*of - -* $2 * l$])
  hence $2 * l * dist\ v\ \nu_b\text{-}opt < eps$
   by (*auto simp*: *assms*(*2*) *pos-divide-less-eq intro*: *order.strict-trans1*)
  hence $dist\ v\ \nu_b\text{-}opt * l < eps / 2$
    by *argo*
  hence $*$: $l * dist\ v\ \nu_b\text{-}opt < eps / 2$
    by (*auto simp*: *algebra-simps*)
  show $dist\ (GS\text{-}rec\text{-}fun_b\ v)\ \nu_b\text{-}opt < eps / 2$
   using *GS.$\mathcal{L}_b$-split-contraction*[*of v ν_b-opt*] *order.trans mult-right-mono*[*OF GS-QR-disc-le-disc zero-le-dist*]
    by (*fastforce intro*!: *le-less-trans*[*OF -* $*$])
qed
end

context *MDP-ord*
begin

lemma *is-am-GS-inv-extend'*:
  assumes $(\bigwedge s.\ s < x \Longrightarrow is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}inv\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d)$
   assumes $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ x)\ (\lambda d.\ d \in D_D)\ (d(x := a))$
  assumes $s \leq x$ $d \in D_D$
  shows $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}inv\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (d(x := a))$
proof −
   have *a*: $a \in A\ x$ **using** *assms*(*2*) **unfolding** *is-arg-max-linorder is-dec-det-def* **by** (*auto split*: *if-splits*)
  have $*$: $\exists y.\ y < x \Longrightarrow s \leq Max\ \{y.\ y < x\} \longleftrightarrow s < x$ **for** $x\ s :: 's$
    by (*auto simp*: *linorder-class.Max-ge-iff*[*OF finite*])

have $(\bigwedge s.\ s < x \Longrightarrow$ *is-arg-max* $(\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d)$
using *is-arg-max-gs-iff*[*OF assms*(*4*), *of Max* $\{y.\ y < x\}$] *assms*(*1*)
by (*cases* $\exists\, y.\ y < x$) (*auto simp*: ∗)
hence $(\bigwedge s.\ s < x \Longrightarrow$ *is-arg-max* $(\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (d(x := a)))$
using *GS-opt-indep-high a* **by** *auto*
hence $(\bigwedge s.\ s \le x \Longrightarrow$ *is-arg-max* $(\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (d(x := a)))$
using *assms*(*2*) *antisym-conv1* **by** *blast*
thus *?thesis*
using *is-arg-max-gs-iff*[*of d*$(x := a)$ *s*] *assms*(*4*) *assms a*
by (*intro is-arg-max-GS-imp-splitting′*) *auto*
qed

definition *opt-policy-gs′ d v s* = (*LEAST a. is-arg-max* $(\lambda a.\ GS\text{-}rec\text{-}step\ (d(s := a))\ v\ \$\ s)\ (\lambda a.\ a \in A\ s)\ a)$

definition *GS-iter a v s* = $r\ (s,\ a) + l * (\sum s' \in UNIV.\ pmf\ (K(s,a))\ s' * v\ \$\ s')$

definition *GS-iter-max v s* = $(\bigsqcup a \in A\ s.\ GS\text{-}iter\ a\ v\ s)$

lemma *GS-rec-eq-iter*:
assumes $\bigwedge s.\ s < k \Longrightarrow v'\ \$\ s = GS\text{-}rec\ v\ \$\ s$ $\bigwedge s.\ k \le s \Longrightarrow v'\ \$\ s = v\ \$\ s$
shows $GS\text{-}rec\text{-}step\ (d(k := a))\ v\ \$\ k = GS\text{-}iter\ a\ v'\ k$
proof −
have $(P\text{-}lower\ d *v\ GS\text{-}rec\ v)\ \$\ k = (P\text{-}lower\ d *v\ v')\ \$\ k$ **for** *d*
using *slt-P-lower assms*
by (*auto intro*!: *strict-lower-triangular-mat-mult*)
moreover have $(P\text{-}upper\ d *v\ v)\ \$\ k = (P\text{-}upper\ d *v\ v')\ \$\ k$ **for** *d*
unfolding *P-upper-def* **using** *assms*
by (*auto simp*: *matrix-vector-mult-def if-distrib*[*of* $\lambda x.\ x *$ - \$ -] *cong*: *if-cong*)
moreover have $P\text{-}lower\ d + P\text{-}upper\ d = P\text{-}dec\ d$ **for** *d*
by (*auto simp*: *P-lower-def P-upper-def Finite-Cartesian-Product.vec-eq-iff*)
ultimately show *?thesis*
unfolding *vector-add-component*[*symmetric*] *matrix-vector-mult-diff-rdistrib*[*symmetric*] *GS-rec-step-def*
matrix-vector-mult-def P-dec-elem P-lower-def P-upper-def GS-iter-def
by (*fastforce simp*: *sum.distrib*[*symmetric*] *intro*!: *sum.cong*)
qed

lemma *GS-rec-eq-iter-max*:
assumes $\bigwedge s.\ s < k \Longrightarrow v'\ \$\ s = GS\text{-}rec\ v\ \$\ s$ $\bigwedge s.\ k \le s \Longrightarrow v'\ \$\ s = v\ \$\ s$
shows $GS\text{-}rec\ v\ \$\ k = GS\text{-}iter\text{-}max\ v'\ k$
using *GS-rec-eq-iter*[*OF assms*] *GS-rec-eq′*[*of* - - *undefined*] *GS-iter-max-def*

by *auto*

definition *GS-iter-arg-max v s* = (*LEAST a. is-arg-max* (λ*a. GS-iter a v s*) (λ*a. a* $\in$ *A s*) *a*)

definition *GS-rec-am-code v d s* = *foldl* (λ*vd s. vd*(*s* := (*GS-iter-max* (χ *s. fst* (*vd s*)) *s*, *GS-iter-arg-max* (χ *s. fst* (*vd s*)) *s*))) (λ*s.* (*v* \$ *s*, *d s*)) (*sorted-list-of-set* {..*s*}) *s*
definition *GS-rec-am-code*$'$ *v d s* = *foldl* (λ*vd s. vd*(*s* := (*GS-iter-max* (χ *s. fst* (*vd s*)) *s*, *GS-iter-arg-max* (χ *s. fst* (*vd s*)) *s*))) (λ*s.* (*v* \$ *s*, *d s*)) (*sorted-list-of-set UNIV*) *s*

lemma *GS-rec-am-code*$'$: *GS-rec-am-code* = *GS-rec-am-code*$'$
proof −
 have ∗: *sorted-list-of-set UNIV* = *sorted-list-of-set*{..*s*} @ *sorted-list-of-set*{*s*<..}
for *s* :: $'s$
  using *sorted-list-of-set-split*$'$[*OF finite*, *of UNIV s*]
  by (*auto simp*: *atMost-def greaterThan-def*)
 have *GS-rec-am-code v d s* = *GS-rec-am-code*$'$ *v d s* **for** *v d s*
  unfolding *GS-rec-am-code-def GS-rec-am-code*$'$*-def* ∗[*of s*]
  by (*fastforce intro*!: *foldl-upd-notin*$'$[*symmetric*])
 thus *?thesis*
  by *blast*
qed

lemma *opt-policy-gs*$'$*-eq-GS-iter*:
 assumes $\bigwedge$*s. s* < *k* $\Longrightarrow$ *v*$'$ \$ *s* = *GS-rec v* \$ *s* $\bigwedge$*s. k* $\leq$ *s* $\Longrightarrow$ *v*$'$ \$ *s* = *v* \$ *s*
 shows *opt-policy-gs*$'$ *d v k* = *GS-iter-arg-max v*$'$ *k*
 unfolding *opt-policy-gs*$'$*-def GS-iter-arg-max-def*
 by (*subst GS-rec-eq-iter*[*OF assms*, *of k d*]) *auto*

lemma *opt-policy-gs*$'$*-eq-GS-iter*$'$:
 opt-policy-gs$'$ *d v k* = *GS-iter-arg-max* (χ *s. if s* < *k then GS-rec v* \$ *s else v* \$ *s*) *k*
 using *opt-policy-gs*$'$*-eq-GS-iter*
 by (*simp add*: *leD*)

lemma *opt-policy-gs*$'$*-is-dec-det*: *opt-policy-gs*$'$ *d v* $\in D_D$
 unfolding *opt-policy-gs*$'$*-def is-dec-det-def*
 using *finite-is-arg-max*[*OF finite A-ne*]
 by (*auto intro*: *LeastI2-ex*)

lemma *opt-policy-gs*$'$*-is-arg-max*: *is-arg-max* (λ*d. GS-inv d v* \$ *s*) (λ*d. d* $\in D_D$) (*opt-policy-gs*$'$ *d v*)
proof (*induction arbitrary*: *d rule*: *less-induct*)
 case (*less x*)
  have *s* < *x* $\Longrightarrow$ *is-arg-max* (λ*d. GS-inv d v* \$ *s*) (λ*d. d* $\in D_D$)

$(opt\text{-}policy\text{-}gs'\ d\ v)$ **for** $d\ s$
  using *less*
  by *auto*
 hence $*{:}s < x \Longrightarrow is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (opt\text{-}policy\text{-}gs'\ d\ v)$ **for** $d\ s$
  by $(intro\ is\text{-}arg\text{-}max\text{-}GS\text{-}imp\text{-}splitting''')\ auto$
 have $is\text{-}arg\text{-}max\ (\lambda a.\ GS\text{-}rec\text{-}step\ (d(x := a))\ v\ \$\ x)\ (\lambda a.\ a \in A\ x)\ (opt\text{-}policy\text{-}gs'\ d\ v\ x)$ **for** d
  unfolding *opt-policy-gs′-def*
  using *finite-is-arg-max*[*OF - A-ne*]
  by (*auto intro*: *LeastI-ex*)
 hence $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ x)\ (\lambda d.\ d \in D_D)\ (opt\text{-}policy\text{-}gs'\ d\ v)$ **for** d
  using *opt-policy-gs′-is-dec-det*
  by $(intro\ is\text{-}arg\text{-}max\text{-}GS\text{-}rec\text{-}step\text{-}act')\ auto$
 hence $s \le x \Longrightarrow is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (opt\text{-}policy\text{-}gs'\ d\ v)$ **for** $d\ s$
  using $*$
  by (*auto simp*: *order.order-iff-strict*)
 hence $s \le x \Longrightarrow is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}inv\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (opt\text{-}policy\text{-}gs'\ d\ v)$ **for** $d\ s$
  using *is-arg-max-GS-imp-splitting′*
  by *blast*
 thus *?case*
  by *blast*
qed

lemma $GS\text{-}rec\text{-}am\text{-}code\ v\ d\ s = (GS\text{-}rec\ v\ \$\ s,\ opt\text{-}policy\text{-}gs'\ d\ v\ s)$
proof (*induction s arbitrary*: *d rule*: *less-induct*)
 case (*less x*)
 show *?case*
 proof $(cases\ \exists\, x'.\ x' < x)$
  case *True*
   let $?f = (\lambda vd\ s.\ vd(s := (GS\text{-}iter\text{-}max\ (\chi\ s.\ fst\ (vd\ s))\ s,\ GS\text{-}iter\text{-}arg\text{-}max\ (\chi\ s.\ fst\ (vd\ s))\ s)))$
  define x' **where** $x' = Max\ \{x'.\ x' < x\}$
  let $?old = (foldl\ ?f\ (\lambda s.\ (v\ \$\ s,\ d\ s))\ (sorted\text{-}list\text{-}of\text{-}set\ \{..x'\}))$
  have $1{:}\ s < x \Longrightarrow (s \notin set\ (sorted\text{-}list\text{-}of\text{-}set\ \{s' \in \{..x'\}.\ s < s'\}))$ **for** $s :: 's$
   by *auto*
  have $s < x \Longrightarrow foldl\ ?f\ (\lambda s.\ (v\ \$\ s,\ d\ s))\ (sorted\text{-}list\text{-}of\text{-}set\ \{..x'\})\ s = foldl\ ?f\ (\lambda s.\ (v\ \$\ s,\ d\ s))\ (sorted\text{-}list\text{-}of\text{-}set\ \{s' \in \{..x'\}.\ s' \le s\}\ @\ sorted\text{-}list\text{-}of\text{-}set\ \{s' \in \{..x'\}.\ s < s'\})\ s$ **for** s
   by (*subst sorted-list-of-set-split′*[*symmetric*, *OF finite*]) *blast*
  hence $s < x \Longrightarrow foldl\ ?f\ (\lambda s.\ (v\ \$\ s,\ d\ s))\ (sorted\text{-}list\text{-}of\text{-}set\ \{..x'\})\ s = foldl\ ?f\ (\lambda s.\ (v\ \$\ s,\ d\ s))\ (sorted\text{-}list\text{-}of\text{-}set\ \{s' \in \{..x'\}.\ s' \le s\})\ s$ **for** s
   using *foldl-upd-notin′*[*OF 1*]
   by *fastforce*

hence *1*: $s < x \Longrightarrow$ *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* $\{..x'\}$) s = *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* $\{..s\}$) s **for** s
unfolding *x′-def*
using *True*
by (*auto simp*: *atMost-def Max-ge-iff*[*OF finite*]) *meson*
have *fst-IH*: *fst* (*?old s*) = *GS-rec v* \$ *s* **if** $s < x$ **for** s
using *1*[*OF that*] *less*[*unfolded GS-rec-am-code-def*] *that*
by *auto*
have *fst-IH′*: *fst* (*?old s*) = *v* \$ *s* **if** $x \le s$ **for** s
using *True that*
by (*subst foldl-upd-notin*) (*auto simp*: *x′-def Max-ge-iff*)
have *fst-IH″*: *fst* (*?old s*) = (*if* $s < x$ *then GS-rec v* \$ *s else v* \$ *s*) **for** s
using *fst-IH fst-IH′* **by** *auto*
have *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* $\{..x\}$) = *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* $\{..x'\}$ @ *sorted-list-of-set* $\{x\}$)
proof −
have $*$: $\{x'.\ x' < x\} \neq \{\}$ **using** *True* **by** *auto*
hence $**$: $\{..x'\} = \{y \in \{..x\}.\ y < x\}$ $\{x\} = \{y \in \{..x\}.\ x \le y\}$
by (*auto simp*: *x′-def Max-ge-iff*[*OF finite* $*$])
show *?thesis*
unfolding $**$ *sorted-list-of-set-split*[*symmetric*, *OF finite*] **by** *auto*
qed
also have ... = *?f* (*foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* $\{..x'\}$)) x
by *auto*
also have ... = (*?old* (x := (*GS-rec v* \$ *x*, *GS-iter-arg-max* (χ *s. fst* (*?old s*)) *x*)))
proof (*subst GS-rec-eq-iter-max*[*of* - (χ *s. fst* (*?old s*))], *goal-cases*)
case (*1 s*)
then show *?case*
using *fst-IH* **by** *auto*
next
case (*2 s*)
then show *?case*
unfolding *vec-lambda-inverse*[*OF UNIV-I*]
using *True*
by (*subst foldl-upd-notin*) (*auto simp*: *x′-def Max-ge-iff*[*OF finite*])
qed *auto*
also have ... = (*?old* (x := (*GS-rec v* \$ *x*, *opt-policy-gs′ d v x*)))
by (*auto simp*: *fst-IH″ opt-policy-gs′-eq-GS-iter′*)
finally have *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* $\{..x\}$) = (*?old* (x := (*GS-rec v* \$ *x*, *opt-policy-gs′ d v x*))).
thus *?thesis*
unfolding *GS-rec-am-code-def*
by *auto*
next

case *False*
hence $\{..x\} = \{x\}$
by (*auto simp*: *not-less antisym*)
thus *?thesis*
unfolding *GS-rec-am-code-def*
using *opt-policy-gs'-eq-GS-iter*[*of x v*] *GS-rec-eq-iter-max*[*of x v*] *False*
by *fastforce*
qed
qed

lemma *GS-rec-am-code-eq*: *GS-rec-am-code v d s* = (*GS-rec v* \$ *s*, *opt-policy-gs' d v s*)
proof (*induction s arbitrary*: *d rule*: *less-induct*)
case (*less x*)
show *?case*
proof (*cases* $\exists x'. x' < x$)
case *True*
let *?f* = (λ*vd s*. *vd*(*s* := (*GS-iter-max* (χ *s*. *fst* (*vd s*)) *s*, *GS-iter-arg-max* (χ *s*. *fst* (*vd s*)) *s*)))
define x' **where** $x' =$ *Max* $\{x'. x' < x\}$
let *?old* = (*foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{..x'\}$))
have *1*: $s < x \Longrightarrow$ ($s \notin$ *set* (*sorted-list-of-set* $\{s' \in \{..x'\}. s < s'\}$)) **for** *s* :: *'s*
by *auto*
have $s < x \Longrightarrow$ *foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{..x'\}$) *s* = *foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{s' \in \{..x'\}. s' \le s\}$ @ *sorted-list-of-set* $\{s' \in \{..x'\}. s < s'\}$) *s* **for** *s*
by (*subst sorted-list-of-set-split'*[*symmetric*, *OF finite*]) *blast*
hence $s < x \Longrightarrow$ *foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{..x'\}$) *s* = *foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{s' \in \{..x'\}. s' \le s\}$) *s* **for** *s*
using *foldl-upd-notin'*[*OF 1*]
by *fastforce*
hence *1*: $s < x \Longrightarrow$ *foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{..x'\}$) *s* = *foldl ?f* (λ*s*. (*v* \$ *s*, *d s*)) (*sorted-list-of-set* $\{..s\}$) *s* **for** *s*
unfolding *x'-def*
using *True*
by (*auto simp*: *atMost-def Max-ge-iff*[*OF finite*]) *meson*
have *fst-IH*: *fst* (*?old s*) = *GS-rec v* \$ *s* **if** $s < x$ **for** *s*
unfolding *1*[*OF that*] *less*[*unfolded GS-rec-am-code-def*, *OF that*]
by *auto*
have *fst-IH'*: *fst* (*?old s*) = *v* \$ *s* **if** $x \le s$ **for** *s*
using *True that*
by (*subst foldl-upd-notin*) (*auto simp*: *x'-def atMost-def Max-ge-iff*[*OF finite*])
have *fst-IH''*: *fst* (*?old s*) = (*if* $s < x$ *then GS-rec v* \$ *s else v* \$ *s*) **for** *s*
using *fst-IH fst-IH'* **by** *auto*

have *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* {..x}) = *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* {..x'} @ *sorted-list-of-set* {x})
proof –
have ∗: {x'. $x' < x$} ≠ {} **using** *True* **by** *auto*
hence *1*: {..x'} = {$y \in$ {..x}. $y < x$}
by (*auto simp*: x'*-def Max-ge-iff*[*OF finite* ∗])
have *2*: {x} = {$y \in$ {..x}. $x \le y$}
by *auto*
thus *?thesis*
unfolding *1 2 sorted-list-of-set-split*[*symmetric*, *OF finite*] **by** *auto*
qed
also have ... = *?f* (*foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* {..x'})) x
by *auto*
also have ... = (*?old* (x := (*GS-rec* v \$ x, *GS-iter-arg-max* (χ s. *fst* (*?old* s)) x)))
proof (*subst GS-rec-eq-iter-max*[*of* - (χ s. *fst* (*?old* s))], *goal-cases*)
case (*2 s*)
then show *?case*
unfolding *vec-lambda-inverse*[*OF UNIV-I*]
using *True*
by (*subst foldl-upd-notin*) (*auto simp*: x'*-def Max-ge-iff*[*OF finite*])
qed (*auto simp*: *fst-IH*)
also have ... = (*?old* (x := (*GS-rec* v \$ x, *opt-policy-gs'* d v x)))
by (*auto simp*: *fst-IH''* *opt-policy-gs'-eq-GS-iter'*)
finally have *foldl ?f* ($\lambda s.$ (v \$ s, d s)) (*sorted-list-of-set* {..x}) = (*?old* (x := (*GS-rec* v \$ x, *opt-policy-gs'* d v x))).
thus *?thesis*
unfolding *GS-rec-am-code-def*
by *auto*
next
case (*False*)
hence {..x} = {x}
by (*auto simp*: *not-less antisym*)
hence ∗: (*sorted-list-of-set* {..x}) = [x]
by *auto*
show *?thesis*
unfolding *GS-rec-am-code-def*
using *opt-policy-gs'-eq-GS-iter*[*of x v*] *GS-rec-eq-iter-max*[*of x v*] *False*
by (*fastforce simp*: ∗)
qed
qed

definition *GS-rec-iter-arg-max* **where**
GS-rec-iter-arg-max v s = (*LEAST a. is-arg-max* ($\lambda a.$ r (s, a) + l ∗ ($\sum s' \in UNIV$. *pmf* (K (s,a)) s' ∗ v s')) ($\lambda a.$ $a \in A$ s) a)

definition *opt-policy-gs v s* = (*LEAST a. is-arg-max* (λa. *GS-rec-fun-inner v s a*) (λa. $a \in A\ s$) *a*)

lemma *opt-policy-gs-eq′*: *opt-policy-gs v* = *opt-policy-gs′ d* (*vec-lambda v*)
  unfolding *opt-policy-gs-def opt-policy-gs′-def GS-rec-fun-inner-def GS-rec-step-elem*
  by (*auto simp*: *GS-rec-fun$_b$.rep-eq GS-rec-def vec-lambda-inverse*)

declare *gs-iteration.simps*[*simp del*]

lemma *gs-iteration-error*:
  assumes *eps* > *0*
  shows *dist* (*gs-iteration eps v*) ν_b-*opt* < *eps* / *2*
  using *assms dist-$\mathcal{L}_b$-split-opt-eps gs-iteration.simps*
  by (*induction eps v rule*: *gs-iteration.induct*) *auto*

lemma *GS-rec-fun-inner-vec*: *GS-rec-fun-inner v s a* = *GS-rec-step* (*d*(*s* := *a*)) (*vec-lambda v*) \$ *s*
  unfolding *GS-rec-step-elem*
  by (*auto simp*: *GS-rec-fun-inner-def GS-rec-def GS-rec-fun$_b$.rep-eq vec-lambda-inverse*)

lemma *find-policy-error-bound-gs*:
  assumes *eps* > *0 2* ∗ *l* ∗ *dist v* (*GS-rec-fun$_b$ v*) < *eps* ∗ (*1*−*l*)
  shows *dist* (ν_b (*mk-stationary-det* (*opt-policy-gs* (*GS-rec-fun$_b$ v*)))) ν_b-*opt* < *eps*
proof (*rule GS.find-policy-QR-error-bound*[*OF assms*(*1*)])
  have *2* ∗ *GS.QR-disc* ∗ *dist v* (*GS-rec-fun$_b$ v*) ≤ *2* ∗ *l* ∗ *dist v* (*GS-rec-fun$_b$ v*)
    using *GS-QR-disc-le-disc*
    by (*auto intro*!: *mult-right-mono*)
  also have ... < *eps* ∗ (*1*−*l*) **using** *assms* **by** *auto*
  also have ... ≤ *eps* ∗ (*1* − *GS.QR-disc*)
    using *assms GS-QR-disc-le-disc*
    by (*auto intro*!: *mult-left-mono*)
  finally show *2* ∗ *GS.QR-disc* ∗ *dist v* (*GS-rec-fun$_b$ v*) < *eps* ∗ (*1* − *GS.QR-disc*)**.**
next
  obtain *d* **where** *d*: *is-dec-det d*
    using *ex-dec-det* **by** *blast*
  show *is-arg-max* (λd. *apply-bfun* (*GS.L-split d* (*GS-rec-fun$_b$ v*)) *s*) (λd. $d \in D_D$) (*opt-policy-gs* (*GS-rec-fun$_b$ v*)) **for** *s*
    unfolding *opt-policy-gs-eq′*[*of - d*] *GS-inv-blinfun-to-matrix*
    using *opt-policy-gs′-is-arg-max*
    by *simp*
qed

definition *vi-gs-policy eps v = opt-policy-gs (gs-iteration eps v)*

lemma *vi-gs-policy-opt*:
  assumes *0 < eps*
  shows *dist* (ν_b (*mk-stationary-det* (*vi-gs-policy eps v*))) ν_b*-opt* < *eps*
  unfolding *vi-gs-policy-def*
  using *assms*
proof (*induction eps v rule*: *gs-iteration.induct*)
  case (*1 v*)
  then show *?case*
    using *find-policy-error-bound-gs*
    by (*subst gs-iteration.simps*) *auto*
qed

lemma *GS-rec-iter-eq-iter-max*: *GS-rec-iter v = GS-iter-max* (*vec-lambda v*)
  unfolding *GS-rec-iter-def GS-iter-max-def GS-iter-def*
  by *auto*
end

end

theory *Algorithms*
  imports
    Value-Iteration
    Policy-Iteration
    Modified-Policy-Iteration
    Splitting-Methods
begin
end

theory *Code-DP*
  imports
    Value-Iteration
    Policy-Iteration
    Modified-Policy-Iteration
    Splitting-Methods

HOL−Library.Code-Target-Numeral
Gauss-Jordan.Code-Generation-IArrays
begin

7 Code Generation for MDP Algorithms

7.1 Least Argmax

lemma *least-list*:

assumes *sorted xs* $\exists\, x \in$ *set xs. P x*
shows (*LEAST x* $\in$ *set xs. P x*) = *the* (*find P xs*)
using *assms*
proof (*induction xs*)
case (*Cons a xs*)
thus *?case*
proof (*cases P a*)
case *False*
have (*LEAST x* $\in$ *set* (*a* # *xs*). *P x*) = (*LEAST x* $\in$ *set xs. P x*)
using *False Cons*(*2*)
by *simp metis*
thus *?thesis*
using *False Cons*
by *simp*
qed (*auto intro*: *Least-equality*)
qed *auto*

definition *least-enum P* = *the* (*find P* (*sorted-list-of-set* (*UNIV* :: ('*b*:: {*finite*, *linorder*}) *set*)))

lemma *least-enum-eq*: $\exists\, x$. *P x* $\Longrightarrow$ *least-enum P* = (*LEAST x. P x*)
by (*auto simp*: *least-list*[*symmetric*] *least-enum-def*)

definition *least-max-arg-max-list f init xs* =
foldl (λ(*am*, *m*) *x*. *if f x* > *m then* (*x*, *f x*) *else* (*am*, *m*)) *init xs*

lemma *snd-least-max-arg-max-list*:
snd (*least-max-arg-max-list f* (*n*, *f n*) *xs*) = (*MAX x* $\in$ *insert n* (*set xs*). *f x*)
unfolding *least-max-arg-max-list-def*
proof (*induction xs arbitrary*: *n*)
case (*Cons a xs*)
then show *?case*
by (*cases xs* = []) (*fastforce simp*: *max.assoc*[*symmetric*])+
qed *auto*

lemma *least-max-arg-max-list-snd-fst*: *snd* (*least-max-arg-max-list f* (*x*, *f x*) *xs*) = *f* (*fst* (*least-max-arg-max-list f* (*x*, *f x*) *xs*))
by (*induction xs arbitrary*: *x*) (*auto simp*: *least-max-arg-max-list-def*)

lemma *fst-least-max-arg-max-list*:
fixes *f* :: - $\Rightarrow$ - :: *linorder*
assumes *sorted* (*n*#*xs*)
shows *fst* (*least-max-arg-max-list f* (*n*, *f n*) *xs*)
= (*LEAST x. is-arg-max f* (λ*x*. *x* $\in$ *insert n* (*set xs*)) *x*)
unfolding *least-max-arg-max-list-def*
using *assms* **proof** (*induction xs arbitrary*: *n*)
case *Nil*
then show *?case*

by (*auto simp*: *is-arg-max-def intro*!: *Least-equality*[*symmetric*])
next
case (*Cons a xs*)
hence *sorted* (*a#xs*)
by *auto*
then show *?case*
proof (*cases f a* > *f n*)
case *True*
then show *?thesis*
by (*fastforce simp*: *is-arg-max-def Cons.IH*[*OF* ‹*sorted* (*a#xs*)›] *intro*!: *cong*[*of Least*, *OF refl*])
next
case *False*
have (*LEAST b. is-arg-max f* (λx. $x \in$ *insert n* (*set* (*a* # *xs*))) *b*)
= (*LEAST b. is-arg-max f* (λx. $x \in$ (*set* (*n* # *xs*))) *b*)
proof (*cases is-arg-max f* (λx. $x \in$ *set* (*n* #*a*# *xs*)) *a*)
case *True*
hence (*LEAST b. is-arg-max f* (λx. $x \in$ (*set* (*n#a* # *xs*))) *b*) = *n*
using *Cons False*
by (*fastforce simp*: *is-arg-max-linorder intro*!: *Least-equality*)
thus *?thesis*
using *False True Cons*
by (*fastforce simp*: *is-arg-max-linorder intro*!: *Least-equality*[*symmetric*])
next
case *False*
thus *?thesis*
by (*fastforce simp*: *not-less is-arg-max-linorder intro*!: *cong*[*of Least*, *OF refl*])
qed
thus *?thesis*
using *False Cons*
by (*auto simp add*: *Cons.IH*[*OF* ‹*sorted* (*a#xs*)›])
qed
qed

definition *least-arg-max-enum f X* = (
let xs = *sorted-list-of-set* (*X* :: (- :: {*finite*, *linorder*}) *set*) *in*
fst (*least-max-arg-max-list f* (*hd xs*, *f* (*hd xs*)) (*tl xs*)))

definition *least-max-arg-max-enum f X* = (
let xs = *sorted-list-of-set* (*X* :: (- :: {*finite*, *linorder*}) *set*) *in*
(*least-max-arg-max-list f* (*hd xs*, *f* (*hd xs*)) (*tl xs*)))

lemma *least-arg-max-enum-correct*:
assumes $X \neq \{\}$
shows
(*least-arg-max-enum* (*f* :: - $\Rightarrow$ (- :: *linorder*)) *X*) = (*LEAST x. is-arg-max f* (λx. $x \in X$) *x*)

proof −
have ∗: *xs* ≠ [] ⟹ (*x* = *hd xs* ∨ *x* ∈ *set* (*tl xs*)) ⟷ *x* ∈ *set xs* **for** *x xs*
using *list.set-sel list.exhaust-sel set-ConsD* **by** *metis*
thus *?thesis*
unfolding *least-arg-max-enum-def*
using *assms*
by (*auto simp*: *Let-def fst-least-max-arg-max-list* ∗)
qed

lemma *least-max-arg-max-enum-correct1*:
assumes *X* ≠ {}
shows *fst* (*least-max-arg-max-enum* (*f* :: - ⇒ (- :: *linorder*)) *X*) = (*LEAST x*. *is-arg-max f* (λ*x*. *x* ∈ *X*) *x*)
proof −
have ∗: *xs* ≠ [] ⟹ (*x* = *hd xs* ∨ *x* ∈ *set* (*tl xs*)) ⟷ *x* ∈ *set xs* **for** *x xs*
using *list.set-sel list.exhaust-sel set-ConsD* **by** *metis*
thus *?thesis*
using *assms*
by (*auto simp*: *least-max-arg-max-enum-def Let-def fst-least-max-arg-max-list* ∗)
qed

lemma *least-max-arg-max-enum-correct2*:
assumes *X* ≠ {}
shows *snd* (*least-max-arg-max-enum* (*f* :: - ⇒ (- :: *linorder*)) *X*) = (*MAX x* ∈ *X*. *f x*)
proof −
have ∗: *xs* ≠ [] ⟹ *insert* (*hd xs*) (*set* (*tl xs*)) = *set xs* **for** *xs*
using *list.exhaust-sel list.simps*(*15*)
by *metis*
show *?thesis*
using *assms*
by (*auto simp*: *least-max-arg-max-enum-def Let-def snd-least-max-arg-max-list* ∗)
qed

7.2 Functions as Vectors

typedef (′*a*, ′*b*) *Fun* = *UNIV* :: (′*a* ⇒ ′*b*) *set*
by *blast*

setup-lifting *type-definition-Fun*

lift-definition *to-Fun* :: (′*a* ⇒ ′*b*) ⇒ (′*a*, ′*b*) *Fun* **is** *id***.**

definition *fun-to-vec* (*v* :: (′*a*::*finite*, ′*b*) *Fun*) = *vec-lambda* (*Rep-Fun* *v*)

lift-definition *vec-to-fun* :: $'b^{\frown}'a \Rightarrow ('a, 'b)$ *Fun* **is** *vec-nth*.

lemma *Fun-inverse*[*simp*]: *Rep-Fun* (*Abs-Fun f*) = *f*
 using *Abs-Fun-inverse* **by** *auto*

lift-definition *zero-Fun* :: ('*a*, '*b*::*zero*) *Fun* **is** *0*.

code-datatype *vec-to-fun*

lemmas *vec-to-fun.rep-eq*[*code*]

instantiation *Fun* :: (*enum*, *equal*) *equal*
begin
definition *equal-Fun* (*f* :: ('*a*::*enum*, '*b*::*equal*) *Fun*) *g* = (*Rep-Fun f* = *Rep-Fun g*)
instance
 by *standard* (*auto simp*: *equal-Fun-def Rep-Fun-inject*)
end

7.3 Bounded Functions as Vectors

lemma *Bfun-inverse-fin*[*simp*]: *apply-bfun* (*Bfun* (*f* :: '*c* :: *finite* ⇒ -)) = *f*
 using *finite* **by** (*fastforce intro*!: *Bfun-inverse simp*: *bfun-def*)

definition *bfun-to-vec* (*v* :: ('*a*::*finite*) $\Rightarrow_b$ ('*b*::*metric-space*)) = *vec-lambda v*
definition *vec-to-bfun v* = *Bfun* (*vec-nth v*)

code-datatype *vec-to-bfun*

lemma *apply-bfun-vec-to-bfun*[*code*]: *apply-bfun* (*vec-to-bfun f*) *x* = *f* $ *x*
 by (*auto simp*: *vec-to-bfun-def*)

lemma [*code*]: *0* = *vec-to-bfun 0*
 by (*auto simp*: *vec-to-bfun-def*)

7.4 IArrays with Lengths in the Type

typedef ('*s* :: *mod-type*, '*a*) *iarray-type* = {*arr* :: '*a iarray*. *IArray.length arr* = *CARD*('*s*)}
 using *someI-ex*[*OF Ex-list-of-length*]
 by (*auto intro*!: *exI*[*of - IArray* (*SOME xs. length xs* = *CARD*('*s*))])

setup-lifting *type-definition-iarray-type*

lift-definition *fun-to-iarray-t* :: ('*s*::{*mod-type*} ⇒ '*a*) ⇒ ('*s*, '*a*) *iarray-type* **is** λ*f*. *IArray.of-fun* (λ*s*. *f* (*from-nat s*)) (*CARD*('*s*)) by *auto*

lift-definition *iarray-t-sub* :: (*'s*::*mod-type*, *'a*) *iarray-type* ⇒ *'s* ⇒ *'a*
is $\lambda v\ x.$ *IArray.sub v* (*to-nat x*).

lift-definition *iarray-to-vec* :: (*'s*, *'a*) *iarray-type* ⇒ *'a*^*'s*::{*mod-type*, *finite*}
is $\lambda v.$ (χ *s. IArray.sub v* (*to-nat s*)).

lift-definition *vec-to-iarray* :: *'a*^*'s*::{*mod-type*, *finite*} ⇒ (*'s*, *'a*) *iarray-type*
is $\lambda v.$ *IArray.of-fun* ($\lambda s.$ *v* $ ((*from-nat s*) :: *'s*)) (*CARD*(*'s*))
by *auto*

lemma *length-iarray-type* [*simp*]: *length* (*IArray.list-of* (*Rep-iarray-type* (*v*:: (*'s*::{*mod-type*}, *'a*) *iarray-type*))) = *CARD*(*'s*)
using *Rep-iarray-type* **by** *auto*

lemma *iarray-t-eq-iff*: (*v* = *w*) = (∀ *x. iarray-t-sub v x* = *iarray-t-sub w x*)
unfolding *iarray-t-sub.rep-eq IArray.sub-def*
by (*metis Rep-iarray-type-inject iarray-exhaust2 length-iarray-type list-eq-iff-nth-eq to-nat-from-nat-id*)

lemma *iarray-to-vec-inv*: *iarray-to-vec* (*vec-to-iarray v*) = *v*
by (*auto simp*: *to-nat-less-card iarray-to-vec.rep-eq vec-to-iarray.rep-eq vec-eq-iff*)

lemma *vec-to-iarray-inv*: *vec-to-iarray* (*iarray-to-vec v*) = *v*
by (*auto simp*: *to-nat-less-card iarray-to-vec.rep-eq vec-to-iarray.rep-eq iarray-t-eq-iff iarray-t-sub.rep-eq*)

code-datatype *iarray-to-vec*

lemma *vec-nth-iarray-to-vec*[*code*]: *vec-nth* (*iarray-to-vec v*) *x* = *iarray-t-sub v x*
by (*auto simp*: *iarray-to-vec.rep-eq iarray-t-sub.rep-eq*)

lemma *vec-lambda-iarray-t*[*code*]: *vec-lambda v* = *iarray-to-vec* (*fun-to-iarray-t v*)
by (*auto simp*: *iarray-to-vec.rep-eq fun-to-iarray-t.rep-eq to-nat-less-card*)

lemma *zero-iarray*[*code*]: *0* = *iarray-to-vec* (*fun-to-iarray-t 0*)
by (*auto simp*: *iarray-to-vec.rep-eq fun-to-iarray-t.rep-eq to-nat-less-card vec-eq-iff*)

7.5 Value Iteration

locale *vi-code* =

MDP-ord A K r l **for** *A* :: *'s::mod-type* ⇒ (*'a::{finite, wellorder}*) *set*
and *K* :: (*'s::{finite, mod-type}* × *'a::{finite, wellorder}*) ⇒ *'s pmf*
and *r l*
begin
definition *vi-test* (*v::'s*$\Rightarrow_b$ *real*) *v' eps* = *2* ∗ *l* ∗ *dist v v'*

partial-function (*tailrec*) *value-iteration-partial* **where** [*code*]: *value-iteration-partial eps v* =
(*let v'* = $\mathcal{L}_b$ *v in*
(*if 2* ∗ *l* ∗ *dist v v'* < *eps* ∗ (*1* − *l*) *then v' else* (*value-iteration-partial eps v'*)))

lemma *vi-eq-partial*: *eps* > *0* ⟹ *value-iteration-partial eps v* = *value-iteration eps v*
proof (*induction eps v rule*: *value-iteration.induct*)
case (*1 eps v*)
then show *?case*
by (*auto simp*: *Let-def value-iteration.simps value-iteration-partial.simps*)
qed

definition *L-det d* = *L* (*mk-dec-det d*)

lemma *code-L-det* [*code*]: *L-det d* (*vec-to-bfun v*) = *vec-to-bfun* (*χ s.* L_a (*d s*) (*vec-nth v*) *s*)
by (*auto simp*: *L-det-def vec-to-bfun-def L-eq-*L_a*-det*)

lemma *code-*$\mathcal{L}_b$ [*code*]: $\mathcal{L}_b$ (*vec-to-bfun v*) = *vec-to-bfun* (*χ s.* (*MAX a* ∈ *A s. r* (*s*, *a*) + *l* ∗ *measure-pmf.expectation* (*K* (*s*, *a*)) (*vec-nth v*)))
by (*auto simp*: *vec-to-bfun-def* $\mathcal{L}_b$*-fin-eq-det A-ne cSup-eq-Max*)

lemma *code-value-iteration*[*code*]: *value-iteration eps* (*vec-to-bfun v*) =
(*if eps* ≤ *0 then* $\mathcal{L}_b$ (*vec-to-bfun v*) *else value-iteration-partial eps* (*vec-to-bfun v*))
by (*simp add*: *value-iteration.simps vi-eq-partial*)

lift-definition *find-policy-impl* :: (*'s* $\Rightarrow_b$ *real*) ⇒ (*'s*, *'a*) *Fun* **is** *λv. find-policy' v*.
lemma *code-find-policy-impl*: *find-policy-impl v* = *vec-to-fun* (*χ s.* (*LEAST x. x* ∈ *opt-acts v s*))
by (*auto simp*: *vec-to-fun-def find-policy-impl-def find-policy'-def Abs-Fun-inject*)

lemma *code-find-policy-impl-opt*[*code*]: *find-policy-impl v* = *vec-to-fun* (*χ s. least-arg-max-enum* (*λa.* L_a *a v s*) (*A s*))
by (*auto simp*: *is-opt-act-def code-find-policy-impl least-arg-max-enum-correct*[*OF A-ne*])

lemma *code-vi-policy'*[*code*]: *vi-policy' eps v* = *Rep-Fun* (*find-policy-impl* (*value-iteration eps v*))
unfolding *vi-policy'-def find-policy-impl-def* **by** *auto*

7.6 Policy Iteration

partial-function (*tailrec*) *policy-iteration-partial* **where** [*code*]: *policy-iteration-partial d* =
(*let d'* = *policy-step d in if d* = *d' then d else policy-iteration-partial d'*)

lemma *pi-eq-partial*: $d \in D_D \Longrightarrow$ *policy-iteration-partial d* = *policy-iteration d*
proof (*induction d rule*: *policy-iteration.induct*)
case (*1 d*)
then show *?case*
by (*auto simp*: *Let-def is-dec-det-pi policy-step-def policy-iteration-partial.simps*)
qed

definition *P-mat d* = (χ *i j. pmf* (*K* (*i*, *Rep-Fun d i*)) *j*)

definition *r-vec' d* = (χ *i. r*(*i*, *Rep-Fun d i*))

lift-definition *policy-eval'* :: (*'s*::{*mod-type*, *finite*}, *'a*) *Fun* ⇒ (*'s* $\Rightarrow_b$ *real*) **is** *policy-eval*.

lemma *mat-eq-blinfun*: *mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*)) = *blinfun-to-matrix* (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det* (*vec-nth d*)))
unfolding *blinfun-to-matrix-diff blinfun-to-matrix-id blinfun-to-matrix-scaleR*
unfolding *blinfun-to-matrix-def P-mat-def* $\mathcal{P}_1$*.rep-eq K-st-def push-exp-def matrix-def axis-def vec-to-fun-def*
by (*auto simp*: *if-distrib mk-dec-det-def integral-measure-pmf*[*of UNIV*] *pmf-expectation-bind*[*of UNIV*] *pmf-bind cong*: *if-cong*)

lemma ν_b*-vec*: *policy-eval'* (*vec-to-fun d*) = *vec-to-bfun* (*matrix-inv* (*mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*))) *∗v* (*r-vec'* (*vec-to-fun d*)))
proof −
let *?d* = *Rep-Fun* (*vec-to-fun d*)
have *vec-to-bfun* (*matrix-inv* (*mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*))) *∗v* (*r-vec'* (*vec-to-fun d*))) = *matrix-to-blinfun* (*matrix-inv* (*mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*)))) (*vec-to-bfun* (*r-vec'* (*vec-to-fun d*)))
by (*auto simp*: *matrix-to-blinfun-mult vec-to-bfun-def r-vec'-def*)
also have ... = *matrix-to-blinfun* (*matrix-inv* (*blinfun-to-matrix* (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det ?d*)))) (*r-dec*$_b$ (*mk-dec-det ?d*))
unfolding *mat-eq-blinfun*
by (*auto simp*: *r-vec'-def vec-to-bfun-def vec-lambda-inverse r-dec*$_b$*-def vec-to-fun-def*)
also have ... = *inv*$_L$ (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det ?d*)) (*r-dec*$_b$ (*mk-dec-det ?d*))

by (*auto simp*: *blinfun-to-matrix-inverse*(*2*)[*symmetric*] *invertible*$_L$*-inf-sum matrix-to-blinfun-inv*)
finally have *vec-to-bfun* (*matrix-inv* (*mat 1* $-$ *l* $*_R$ (*P-mat* (*vec-to-fun d*))) $*v$ (*r-vec*$'$ (*vec-to-fun d*))) $=$ *inv*$_L$ (*id-blinfun* $-$ *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det ?d*)) (*r-dec*$_b$ (*mk-dec-det ?d*))**.**
thus *?thesis*
by (*auto simp*: ν*-stationary policy-eval*$'$*.rep-eq policy-eval-def inv*$_L$*-inf-sum blincomp-scaleR-right*)
qed

lemma ν_b*-vec-opt*[*code*]: *policy-eval*$'$ (*vec-to-fun d*) $=$ *vec-to-bfun* (*Matrix-To-IArray.iarray-to-vec* (*Matrix-To-IArray.vec-to-iarray* ((*fst* (*Gauss-Jordan-PA* ((*mat 1* $-$ *l* $*_R$ (*P-mat* (*vec-to-fun d*)))))) $*v$ (*r-vec*$'$ (*vec-to-fun d*)))))
using ν_b*-vec*
by (*auto simp*: *mat-eq-blinfun matrix-inv-Gauss-Jordan-PA blinfun-to-matrix-inverse*(*1*) *invertible*$_L$*-inf-sum iarray-to-vec-vec-to-iarray*)

lift-definition *policy-improvement*$'$ :: ($'s$, $'a$) *Fun* $\Rightarrow$ ($'s \Rightarrow_b$ *real*) $\Rightarrow$ ($'s$, $'a$) *Fun*
is *policy-improvement***.**

lemma [*code*]: *policy-improvement*$'$ (*vec-to-fun d*) *v* $=$ *vec-to-fun* (χ *s.* (*if is-arg-max* ($\lambda a.$ L_a *a v s*) ($\lambda a.$ $a \in A$ *s*) (*d* \$ *s*) *then d* \$ *s else LEAST x. is-arg-max* ($\lambda a.$ L_a *a v s*) ($\lambda a.$ $a \in A$ *s*) *x*))
by (*auto simp*: *is-opt-act-def policy-improvement*$'$*-def vec-to-fun-def vec-lambda-inverse policy-improvement-def Abs-Fun-inject*)

lift-definition *policy-step*$'$ :: ($'s$, $'a$) *Fun* $\Rightarrow$ ($'s$, $'a$) *Fun*
is *policy-step***.**

lemma [*code*]: *policy-step*$'$ *d* $=$ *policy-improvement*$'$ *d* (*policy-eval*$'$ *d*)
by (*auto simp*: *policy-step*$'$*-def policy-step-def policy-improvement*$'$*-def policy-eval*$'$*-def apply-bfun-inverse*)

lift-definition *policy-iteration-partial*$'$ :: ($'s$, $'a$) *Fun* $\Rightarrow$ ($'s$, $'a$) *Fun*
is *policy-iteration-partial***.**

lemma [*code*]: *policy-iteration-partial*$'$ *d* $=$ (*let d*$'$ $=$ *policy-step*$'$ *d in if d* $=$ *d*$'$ *then d else policy-iteration-partial*$'$ *d*$'$)
by (*auto simp*: *policy-iteration-partial*$'$*.rep-eq policy-step*$'$*.rep-eq Let-def policy-iteration-partial.simps Rep-Fun-inject*[*symmetric*])

lift-definition *policy-iteration*$'$ :: ($'s$, $'a$) *Fun* $\Rightarrow$ ($'s$, $'a$) *Fun* **is** *policy-iteration***.**

lemma *code-policy-iteration*$'$[*code*]: *policy-iteration*$'$ *d* $=$
(*if Rep-Fun d* $\notin D_D$ *then d else* (*policy-iteration-partial*$'$ *d*))
by *transfer* (*auto simp*: *pi-eq-partial*)

lemma *code-policy-iteration*[*code*]: *policy-iteration d* = *Rep-Fun* (*policy-iteration′* (*vec-to-fun* (*vec-lambda d*)))
by (*auto simp add*: *vec-lambda-inverse policy-iteration′.rep-eq vec-to-fun-def*)

7.7 Gauss-Seidel Iteration

partial-function (*tailrec*) *gs-iteration-partial* **where**
[*code*]: *gs-iteration-partial eps v* = (
let v′ = (*GS-rec-fun*$_b$ *v*) *in*
(*if* $2 * l * dist\ v\ v' < eps * (1 - l)$ *then v′ else gs-iteration-partial eps v′*))

lemma *gs-iteration-partial-eq*: *eps* > *0* ⟹ *gs-iteration-partial eps v* = *gs-iteration eps v*
by (*induction eps v rule*: *gs-iteration.induct*) (*auto simp*: *gs-iteration-partial.simps Let-def gs-iteration.simps*)

lemma *gs-iteration-code-opt*[*code*]: *gs-iteration eps v* = (*if eps* ≤ *0 then GS-rec-fun*$_b$ *v else gs-iteration-partial eps v*)
by (*auto simp*: *gs-iteration-partial-eq gs-iteration.simps*)

definition *vec-upd v i x* = (χ *j*. *if i* = *j then x else v* $ *j*)

lemma *GS-rec-eq-fold*: *GS-rec v* = *foldl* (λ*v s*. (*vec-upd v s* (*GS-iter-max v s*))) *v* (*sorted-list-of-set UNIV*)
proof −
have *vec-lambda* (*foldl* (λ*v s*. *v*(*s* := *GS-rec-iter v s*)) (($) *v*) *xs*) = *foldl* (λ*v s*. *vec-upd v s* (*GS-iter-max v s*)) *v xs* **for** *xs*
proof (*induction xs arbitrary*: *v*)
case (*Cons a xs*)
show *?case*
by (*auto simp*: *vec-upd-def*[*of v*] *Cons*[*symmetric*] *eq-commute GS-rec-iter-eq-iter-max cong*: *if-cong*)
qed *auto*
thus *?thesis*
unfolding *GS-rec-def GS-rec-fun-code′*
by *auto*
qed

lemma *GS-rec-fun-code′′′′*[*code*]: *GS-rec-fun*$_b$ (*vec-to-bfun v*) = *vec-to-bfun* (*foldl* (λ*v s*. (*vec-upd v s* (*GS-iter-max v s*))) *v* (*sorted-list-of-set UNIV*))
by (*auto simp add*: *GS-rec-eq-fold*[*symmetric*] *GS-rec-eq-elem GS-rec-fun*$_b$*.rep-eq vec-to-bfun-def*)

lemma *GS-iter-max-code* [*code*]: *GS-iter-max v s* = (*MAX a* ∈ *A s*. *GS-iter a v s*)
using *A-ne*
by (*auto simp*: *GS-iter-max-def cSup-eq-Max*)

lift-definition *opt-policy-gs''* :: $('s \Rightarrow_b real) \Rightarrow ('s, 'a)$ *Fun* **is** *opt-policy-gs*.

declare *opt-policy-gs''.rep-eq*[*symmetric*, *code*]

lemma *GS-rec-am-code'-prod*: *GS-rec-am-code' v d* =
 ($\lambda s'$. (
 let (v', d') = *foldl* ($\lambda(v,d)$ *s*. ($v(s$:= (*GS-iter-max* (*vec-lambda* *v*) *s*)), *d*(*s* := *GS-iter-arg-max* (*vec-lambda* *v*) *s*))) (*vec-nth v*, *d*) (*sorted-list-of-set UNIV*)
 in (v' s', d' s')))
proof −
 have *1*: (λx. (f x, g x))(y := (z, w)) = (λx. ((f(y := z)) x, (g(y := w)) x)) **for** *f g y z w*
 by *auto*
 have *2*: (($) *f*)(*a* := *y*) = ($) (*vec-lambda* ((*vec-nth f*)(*a* := *y*))) **for** *f a y*
 by *auto*
 have *foldl* (λvd *s*. *vd*(*s* := (*GS-iter-max* (χ *s*. *fst* (*vd s*)) *s*, *GS-iter-arg-max* (χ *s*. *fst* (*vd s*)) *s*))) (λs. (*v* $ *s*, *d s*)) *xs* =
 ($\lambda s'$. *let* (v', d') = *foldl* ($\lambda(v, d)$ *s*. (*v*(*s* := *GS-iter-max* (*vec-lambda* *v*) *s*), *d*(*s* := *GS-iter-arg-max* (*vec-lambda v*) *s*))) (($) *v*, *d*) *xs in* ($v'$ s', d' s')) **for** *xs*
 proof (*induction xs arbitrary*: *v d*)
 case *Nil*
 then show *?case* **by** *auto*
 next
 case (*Cons a xs*)
 show *?case*
 by (*simp add*: *1 Cons.IH*[*of* (*vec-lambda* ((($) *v*)(*a* := *GS-iter-max* *v a*))), *unfolded 2*[*symmetric*]] *del*: *fun-upd-apply*)
 qed
 thus *?thesis*
 unfolding *GS-rec-am-code'-def* **by** *blast*
qed

lemma *code-GS-rec-am-arr-opt*[*code*]: *opt-policy-gs''* (*vec-to-bfun v*) = *vec-to-fun* ((*snd* (*foldl* ($\lambda(v, d)$ *s*.
 let (*am*, *m*) = *least-max-arg-max-enum* (λa. *r* (*s*, *a*) + *l* ∗ ($\sum s' \in$ *UNIV*. *pmf* (*K* (*s*,*a*)) s' ∗ *v* $ s')) (*A s*) *in*
 (*vec-upd v s m*, *vec-upd d s am*))
 (*v*, (χ *s*. (*least-enum* (λa. $a \in A$ *s*)))) (*sorted-list-of-set UNIV*))))
proof −
 have *1*: *opt-policy-gs''* v' = *Abs-Fun* (*opt-policy-gs* v') **for** v'
 by (*simp add*: *opt-policy-gs''.abs-eq*)
 have *2*: *opt-policy-gs* (*vec-to-bfun v*) = *opt-policy-gs' d v* **for** *v d*
 by (*metis Bfun-inverse-fin opt-policy-gs-eq' vec-lambda-eta vec-to-bfun-def*)
 have *3*: *opt-policy-gs' d v* = (λs. *snd* (*GS-rec-am-code v d s*)) **for** *d*

by (*simp add*: *GS-rec-am-code-eq*)
have *4*: *GS-rec-am-code v d* = ($\lambda s'$. *let* (v', d') = *foldl* ($\lambda(v, d)$ *s*. ($v(s$:= *GS-iter-max* (*vec-lambda v*) *s*), $d(s$:= *GS-iter-arg-max* (*vec-lambda v*) *s*))) (($) *v*, *d*) (*sorted-list-of-set UNIV*) *in* (v' s', d' s')) **for** *d s v*
using *GS-rec-am-code'* *GS-rec-am-code'-prod* **by** *presburger*
have *5*: *foldl* ($\lambda(v, d)$ *s*. ($v(s$:= *GS-iter-max* (*vec-lambda v*) *s*), $d(s$:= *GS-iter-arg-max* (*vec-lambda v*) *s*))) (($) *v*, ($) *d*) *xs* =
(*let* (v', d') = *foldl* ($\lambda(v, d)$ *s*. (*vec-upd v s* (*GS-iter-max v s*), *vec-upd d s* (*GS-iter-arg-max v s*))) (*v*, *d*) *xs in* (*vec-nth* v', *vec-nth* d')) **for** *d v xs*
proof (*induction xs arbitrary*: *d v*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
show *?case*
unfolding *vec-lambda-inverse Let-def*
using *Cons*[*symmetric*, *unfolded Let-def*]
by *simp* (*auto simp*: *vec-lambda-inverse vec-upd-def Let-def eq-commute cong*: *if-cong*)
qed
have *6*: *opt-policy-gs''* (*vec-to-bfun v*) = *vec-to-fun* (*snd* (*foldl* ($\lambda(v, d)$ *s*. (*vec-upd v s* (*GS-iter-max v s*), *vec-upd d s* (*GS-iter-arg-max v s*))) (*v*, *d*) (*sorted-list-of-set UNIV*)))) **for** *d*
unfolding *1*
unfolding *2*[*of - Rep-Fun* (*vec-to-fun d*)]
unfolding *3*
unfolding *4*
using *5*
by (*auto simp*: *Let-def case-prod-beta vec-to-fun-def*)
show *?thesis*
unfolding *Let-def case-prod-beta*
unfolding *least-max-arg-max-enum-correct1*[*OF A-ne*]
using *least-max-arg-max-enum-correct2*[*OF A-ne*]
unfolding *least-max-arg-max-enum-correct2*[*OF A-ne*]
using *6 fin-actions A-ne*
unfolding *GS-iter-max-def GS-iter-def GS-iter-arg-max-def*
by (*auto simp*: *cSup-eq-Max split-beta'*)
qed

7.8 Modified Policy Iteration

sublocale *MDP-MPI A K r l* λX. *Least* (λx. $x \in X$)
using *MDP-act-axioms MDP-reward-axioms*
by *unfold-locales auto*

definition *d0 s* = (*LEAST a*. $a \in A\ s$)
lift-definition *d0'* :: ('*s*, '*a*) *Fun* **is** *d0*.

lemma *d0-dec-det*: *is-dec-det d0*
  using *A-ne* **unfolding** *d0-def is-dec-det-def*
  by *simp*

lemma *v0-code*[*code*]: $v0\text{-}mpi_b$ = *vec-to-bfun* (χ *s. r-min* / (*1* − *l*))
  by (*auto simp*: $v0\text{-}mpi_b$*-def v0-mpi-def vec-to-bfun-def*)

lemma *d0′-code*[*code*]: *d0′* = *vec-to-fun* (χ *s.* (*LEAST a. a* ∈ *A s*))
 by (*auto simp*: *d0′.rep-eq d0-def Rep-Fun-inject*[*symmetric*] *vec-to-fun-def*)

lemma *step-value-code*[*code*]: *L-pow v d m* = (*L-det d* ^^ *Suc m*) *v*
  unfolding *L-pow-def L-det-def*
  by *auto*

partial-function (*tailrec*) *mpi-partial* **where** [*code*]: *mpi-partial eps d v m* =
  (*let d′* = *policy-improvement d v in* (
    if 2 ∗ *l* ∗ *dist v* ($\mathcal{L}_b$ *v*) < *eps* ∗ (*1* − *l*)
    then (*d′*, *v*)
    else mpi-partial eps d′ (*L-pow v d′* (*m 0 v*)) (λ*n. m* (*Suc n*))))

lemma *mpi-partial-eq-algo*:
  assumes *eps* > *0 d* ∈ D_D *v* ≤ $\mathcal{L}_b$ *v*
  shows *mpi-partial eps d v m* = *mpi-algo eps d v m*
proof −
  have *mpi-algo-dom eps* (*d*,*v*,*m*)
    using *assms termination-mpi-algo* **by** *blast*
  thus *?thesis*
   by (*induction rule*: *mpi-algo.pinduct*) (*auto simp*: *Let-def mpi-algo.psimps mpi-partial.simps*)
qed

lift-definition *mpi-partial′* :: *real* ⇒ (*′s*, *′a*) *Fun* ⇒ (*′s* $\Rightarrow_b$ *real*) ⇒ (*nat* ⇒ (*′s* $\Rightarrow_b$ *real*) ⇒ *nat*)
    ⇒ (*′s*, *′a*) *Fun* × (*′s* $\Rightarrow_b$ *real*) **is** *mpi-partial*.

lemma *mpi-partial′-code*[*code*]: *mpi-partial′ eps d v m* =
  (*let d′* = *policy-improvement′ d v in* (
    if 2 ∗ *l* ∗ *dist v* ($\mathcal{L}_b$ *v*) < *eps* ∗ (*1* − *l*)
    then (*d′*, *v*)
     else mpi-partial′ eps d′ (*L-pow v* (*Rep-Fun d′*) (*m 0 v*)) (λ*n. m* (*Suc n*))))
 by (*auto simp*: *mpi-partial′-def mpi-partial.simps Let-def policy-improvement′-def*)

lemma *r-min-code*[*code-unfold*]: *r-min* = (*MIN s. MIN a. r*(*s*,*a*))
  by (*auto simp*: *cInf-eq-Min*)

lemma *mpi-user-code*[*code*]: *mpi-user eps m* =

(*if eps* ≤ *0 then undefined else*
  let (*d*, *v*) = *mpi-partial′ eps d0′ v0-mpi*$_b$ *m in* (*Rep-Fun d*, *v*))
unfolding *mpi-user-def case-prod-beta mpi-partial′-def*
using *mpi-partial-eq-algo A-ne v0-mpi*$_b$*-le-*$\mathcal{L}_b$ *d0-dec-det*
by (*auto simp*: *d0′.rep-eq d0-def*[*symmetric*])
end

7.9 Auxiliary Equations

lemma [*code-unfold*]: *dist* (*f*::′*a*::*finite* ⇒$_b$ ′*b*::*metric-space*) *g* = (*MAX a. dist* (*apply-bfun f a*) (*g a*))
  by (*auto simp*: *dist-bfun-def cSup-eq-Max*)

lemma *member-code*[*code del*]: *x* ∈ *List.coset xs* ⟷ ¬ *List.member xs x*
  by (*auto simp*: *in-set-member*)

lemma [*code*]: *iarray-to-vec v* + *iarray-to-vec u* = (*Matrix-To-IArray.iarray-to-vec* (*Rep-iarray-type v* + *Rep-iarray-type u*))
  by (*metis* (*no-types*, *lifting*) *Matrix-To-IArray.vec-to-iarray-def iarray-to-vec-vec-to-iarray vec-to-iarray.rep-eq vec-to-iarray-inv vec-to-iarray-plus*)

lemma [*code*]: *iarray-to-vec v* − *iarray-to-vec u* = (*Matrix-To-IArray.iarray-to-vec* (*Rep-iarray-type v* − *Rep-iarray-type u*))
  unfolding *minus-iarray-def Matrix-To-IArray.iarray-to-vec-def iarray-to-vec-def*
  by (*auto simp*: *vec-eq-iff to-nat-less-card*)

lemma *matrix-to-iarray-minus*[*code-unfold*]: *matrix-to-iarray* (*A* − *B*) = *matrix-to-iarray A* − *matrix-to-iarray B*
  unfolding *matrix-to-iarray-def o-def minus-iarray-def Matrix-To-IArray.vec-to-iarray-def*
  by *simp*

declare *matrix-to-iarray-fst-Gauss-Jordan-PA*[*code-unfold*]

end
theory *Code-Mod*
  imports *Code-DP*
begin

8 Code Generation for Concrete Finite MDPs

locale *mod-MDP* =
  fixes *transition* :: ′*s*::{*enum*, *mod-type*} × ′*a*::{*enum*, *mod-type*} ⇒ ′*s pmf*
    and *A* :: ′*s* ⇒ ′*a set*
    and *reward* :: ′*s* × ′*a* ⇒ *real*
    and *discount* :: *real*
begin

sublocale *mdp*: *vi-code*
$\lambda s.$ (*if Set.is-empty* (*A s*) *then UNIV else A s*)
transition
reward
(*if* $1 \leq$ *discount* $\vee$ *discount* $<$ *0 then 0 else discount*)
defines $\mathcal{L}_b$ = *mdp*.$\mathcal{L}_b$
and *L-det* = *mdp.L-det*
and *value-iteration* = *mdp.value-iteration*
and *vi-policy'* = *mdp.vi-policy'*
and *find-policy'* = *mdp.find-policy'*
and *find-policy-impl* = *mdp.find-policy-impl*
and *is-opt-act* = *mdp.is-opt-act*
and *value-iteration-partial* = *mdp.value-iteration-partial*
and *policy-iteration* = *mdp.policy-iteration*
and *is-dec-det* = *mdp.is-dec-det*
and *policy-step* = *mdp.policy-step*
and *policy-improvement* = *mdp.policy-improvement*
and *policy-eval* = *mdp.policy-eval*
and *mk-markovian* = *mdp.mk-markovian*
and *policy-eval'* = *mdp.policy-eval'*
and *policy-iteration-partial'* = *mdp.policy-iteration-partial'*
and *policy-iteration'* = *mdp.policy-iteration'*
and *policy-iteration-policy-step'* = *mdp.policy-step'*
and *policy-iteration-policy-eval'* = *mdp.policy-eval'*
and *policy-iteration-policy-improvement'* = *mdp.policy-improvement'*
and *gs-iteration* = *mdp.gs-iteration*
and *gs-iteration-partial* = *mdp.gs-iteration-partial*
and *vi-gs-policy* = *mdp.vi-gs-policy*
and *opt-policy-gs* = *mdp.opt-policy-gs*
and *opt-policy-gs''* = *mdp.opt-policy-gs''*
and *P-mat* = *mdp.P-mat*
and *r-vec'* = *mdp.r-vec'*
and *GS-rec-fun*$_b$ = *mdp.GS-rec-fun*$_b$
and *GS-iter-max* = *mdp.GS-iter-max*
and *GS-iter* = *mdp.GS-iter*
and *mpi-user* = *mdp.mpi-user*
and *v0-mpi*$_b$ = *mdp.v0-mpi*$_b$
and *mpi-partial'* = *mdp.mpi-partial'*
and *L-pow* = *mdp.L-pow*
and *v0-mpi* = *mdp.v0-mpi*
and *r-min* = *mdp.r-min*
and *d0* = *mdp.d0*
and *d0'* = *mdp.d0'*
and ν_b = *mdp*.ν_b
and *vi-test* = *mdp.vi-test*
by *unfold-locales* (*auto simp add*: *Set.is-empty-def*)
end

global-interpretation *mod-MDP transition A reward discount*
for *transition A reward discount*
defines *mod-MDP-*$\mathcal{L}_b$ = *mdp.*$\mathcal{L}_b$
and *mod-MDP-*$\mathcal{L}_b$*-L-det* = *mdp.L-det*
and *mod-MDP-value-iteration* = *mdp.value-iteration*
and *mod-MDP-vi-policy′* = *mdp.vi-policy′*
and *mod-MDP-find-policy′* = *mdp.find-policy′*
and *mod-MDP-find-policy-impl* = *mdp.find-policy-impl*
and *mod-MDP-is-opt-act* = *mdp.is-opt-act*
and *mod-MDP-value-iteration-partial* = *mdp.value-iteration-partial*
and *mod-MDP-policy-iteration* = *mdp.policy-iteration*
and *mod-MDP-is-dec-det* = *mdp.is-dec-det*
and *mod-MDP-policy-step* = *mdp.policy-step*
and *mod-MDP-policy-improvement* = *mdp.policy-improvement*
and *mod-MDP-policy-eval* = *mdp.policy-eval*
and *mod-MDP-mk-markovian* = *mdp.mk-markovian*
and *mod-MDP-policy-eval′* = *mdp.policy-eval′*
and *mod-MDP-policy-iteration-partial′* = *mdp.policy-iteration-partial′*
and *mod-MDP-policy-iteration′* = *mdp.policy-iteration′*
and *mod-MDP-policy-iteration-policy-step′* = *mdp.policy-step′*
and *mod-MDP-policy-iteration-policy-eval′* = *mdp.policy-eval′*
and *mod-MDP-policy-iteration-policy-improvement′* = *mdp.policy-improvement′*
and *mod-MDP-gs-iteration* = *mdp.gs-iteration*
and *mod-MDP-gs-iteration-partial* = *mdp.gs-iteration-partial*
and *mod-MDP-vi-gs-policy* = *mdp.vi-gs-policy*
and *mod-MDP-opt-policy-gs* = *mdp.opt-policy-gs*
and *mod-MDP-opt-policy-gs′′* = *mdp.opt-policy-gs′′*
and *mod-MDP-P-mat* = *mdp.P-mat*
and *mod-MDP-r-vec′* = *mdp.r-vec′*
and *mod-MDP-GS-rec-fun*$_b$ = *mdp.GS-rec-fun*$_b$
and *mod-MDP-GS-iter-max* = *mdp.GS-iter-max*
and *mod-MDP-GS-iter* = *mdp.GS-iter*
and *mod-MDP-mpi-user* = *mdp.mpi-user*
and *mod-MDP-v0-mpi*$_b$ = *mdp.v0-mpi*$_b$
and *mod-MDP-mpi-partial′* = *mdp.mpi-partial′*
and *mod-MDP-L-pow* = *mdp.L-pow*
and *mod-MDP-v0-mpi* = *mdp.v0-mpi*
and *mod-MDP-r-min* = *mdp.r-min*
and *mod-MDP-d0* = *mdp.d0*
and *mod-MDP-d0′* = *mdp.d0′*
and *mod-MDP-*ν_b = *mdp.*ν_b
and *mod-MDP-vi-test* = *mdp.vi-test*
.

end
theory *Code-Real-Approx-By-Float-Fix*
imports

HOL−Library.Code-Real-Approx-By-Float
Gauss-Jordan.Code-Real-Approx-By-Float-Haskell
beginend

theory *Code-Inventory*
imports
Code-Mod

Code-Real-Approx-By-Float-Fix
begin

9 Inventory Management Example

lemma [*code abstype*]: *embed-pmf* (*pmf P*) = *P*
by (*metis* (*no-types*, *lifting*) *td-pmf-embed-pmf type-definition-def*)

lemmas [*code-abbrev del*] = *pmf-integral-code-unfold*

lemma [*code-unfold*]:
measure-pmf.expectation P (f :: $'a$ :: *enum* ⇒ *real*) = ($\sum x \in$ *UNIV*. *pmf P x* ∗ *f x*)
by (*metis* (*no-types*, *lifting*) *UNIV-I finite-class.finite-UNIV integral-measure-pmf*
real-scaleR-def sum.cong)

lemma [*code*]: *pmf* (*return-pmf x*) = (λy. *indicat-real* {*y*} *x*)
by *auto*

lemma [*code*]:
pmf (*bind-pmf N f*) = (λi :: $'a$. *measure-pmf.expectation N* (λ(x :: $'b$ ::*enum*). *pmf* (*f x*) *i*))
using *Probability-Mass-Function.pmf-bind*
by *fast*

lemma *pmf-finite-le*: *finite* (*X* :: ($'a$::*finite*) *set*) ⟹ *sum* (*pmf p*) *X* ≤ *1*
by (*subst sum-pmf-eq-1*[*symmetric*, *of UNIV p*]) (*auto intro*: *sum-mono2*)

lemma *mod-less-diff*:
assumes *0* < (*x*::$'s$::{*mod-type*}) *x* ≤ *y*
shows *y* − *x* < *y*
proof −
have *0* ≤ *Rep y* − *Rep x*
using *assms mono-Rep* **unfolding** *mono-def*
by *auto*
have *1*: *Rep y* − *Rep x* = *Rep* (*y* − *x*)
unfolding *mod-type-class.diff-def Rep-Abs-mod*
using *Rep-ge-0*

by (*auto intro*!: *mod-pos-pos-trivial*[*OF* ‹*0* $\leq$ *Rep y* $-$ *Rep x*› *order.strict-trans1*[*OF* - *Rep-less-n*, *of* - *y*], *symmetric*])
have *Rep y* $-$ *Rep x* $<$ *Rep y*
using *assms*(*1*) *strict-mono-Rep Rep-ge-0 le-less not-less*
by (*fastforce simp*: *strict-mono-def*)
hence *Rep* (*y* $-$ *x*) $<$ *Rep y*
unfolding *1* **by** *blast*
thus *?thesis*
by (*metis not-less-iff-gr-or-eq strict-mono-Rep strict-mono-def*)
qed

locale *inventory* =
fixes *fixed-cost* :: *real*
and *var-cost* :: *'s*::{*mod-type*, *finite*} $\Rightarrow$ *real*
and *inv-cost* :: *'s* $\Rightarrow$ *real*
and *demand-prob* :: *'s pmf*
and *revenue* :: *'s* $\Rightarrow$ *real*
and *discount* :: *real*
begin
definition *order-cost u* = (*if u* = *0 then 0 else fixed-cost* + *var-cost u*)
definition *prob-ge-inv u* = *1* $-$ ($\sum$ *j*<*u*. *pmf demand-prob j*)
definition *exp-rev u* = ($\sum$ *j*<*u*. *revenue j* $*$ *pmf demand-prob j*) + *revenue u* $*$ *prob-ge-inv u*
definition *reward sa* = (*case sa of* (*s*,*a*) $\Rightarrow$ *exp-rev* (*s* + *a*) $-$ *order-cost a* $-$ *inv-cost* (*s* + *a*))
lift-definition *transition* :: (*'s* $\times$ *'s*) $\Rightarrow$ *'s pmf* **is** λ(*s*, *a*) *s'*.
(*if CARD*(*'s*) $\leq$ *Rep s* + *Rep a*
then (*if s'* = *0 then 1 else 0*)
else (*if s* + *a* $<$ *s' then 0 else*
if s' = *0 then prob-ge-inv* (*s*+*a*)
else pmf demand-prob (*s* + *a* $-$ *s'*)))

proof (*safe*, *goal-cases*)
case (*1 a b x*)
then show *?case* **unfolding** *prob-ge-inv-def* **using** *pmf-finite-le* **by** *auto*
next
case (*2 a b*)
then show *?case*
proof (*cases int CARD*(*'s*) $\leq$ *Rep a* + *Rep b*) **next**
case *False*
hence ($\int^+$ *x*. *ennreal* (*if int CARD*(*'s*) $\leq$ *Rep a* + *Rep b then if x* = *0 then 1 else 0 else if a* + *b* $<$ *x then 0 else if x* = *0 then prob-ge-inv* (*a* + *b*) *else pmf demand-prob* (*a* + *b* $-$ *x*)) ∂*count-space UNIV*) =
($\sum$ *x* $\in$ *UNIV*. (*if a* + *b* $<$ *x then 0 else if x* = *0 then prob-ge-inv* (*a* + *b*) *else pmf demand-prob* (*a* + *b* $-$ *x*)))
using *pmf-nonneg prob-ge-inv-def pmf-finite-le*
by (*auto simp*: *nn-integral-count-space-finite intro*!: *sum-ennreal*)

also have ... $=(\sum x \in UNIV.\ (if\ x = 0\ then\ prob\text{-}ge\text{-}inv\ (a + b)$ $else\ if\ a + b < x\ then\ 0\ else\ pmf\ demand\text{-}prob\ (a + b - x)))$
by (*auto intro*!: *sum.cong ennreal-cong simp add*: *leD least-mod-type*)
also have ... $= prob\text{-}ge\text{-}inv\ (a + b) + (\sum x \in UNIV - \{0\}.\ (if\ a + b < x\ then\ 0\ else\ pmf\ demand\text{-}prob\ (a + b - x)))$
by (*auto simp*: *sum.remove*[*of UNIV 0*])
also have ... $= prob\text{-}ge\text{-}inv\ (a + b) + (\sum x\in\{0<..\}.\ (if\ a + b < x\ then\ 0\ else\ pmf\ demand\text{-}prob\ (a + b - x)))$
by (*auto simp add*: *greaterThan-def le-neq-trans least-mod-type intro*!: *cong*[*of sum -*] *ennreal-cong*)
also have ... $= prob\text{-}ge\text{-}inv\ (a + b) + (\sum x\in\{0<..a+b\}.\ (pmf\ demand\text{-}prob\ (a + b - x)))$
unfolding *atMost-def greaterThan-def greaterThanAtMost-def*
by (*auto simp*: *Collect-neg-eq*[*symmetric*] *not-less sum.If-cases*)
also have ... $= 1 - (\sum j<(a + b).\ pmf\ demand\text{-}prob\ j) + (\sum x\in\{0<..a+b\}.\ pmf\ demand\text{-}prob\ (a + b - x))$
unfolding *prob-ge-inv-def*
by *auto*
also have ... $= 1 - (\sum j<(a + b).\ pmf\ demand\text{-}prob\ j) + (\sum x\in\{..<a+b\}.\ (pmf\ demand\text{-}prob\ x))$
proof −
have $(\sum x\in\{0<..a+b\}.\ pmf\ demand\text{-}prob\ (a + b - x)) = (\sum x\in\{..<a+b\}.\ (pmf\ demand\text{-}prob\ x))$
proof (*intro sum.reindex-bij-betw bij-betw-imageI*)
show $inj\text{-}on\ ((-)\ (a + b))\ \{0<..a + b\}$
unfolding *inj-on-def*
by (*metis add.left-cancel diff-add-cancel*)
have $x + (a + b) = a + (b + x)$ **for** x
by (*metis add.assoc add.commute add-to-nat-def*)
moreover have $x < a + b \Longrightarrow 0 < a + b - x$ **for** x
by (*metis add.left-neutral diff-add-cancel least-mod-type less-le*)
moreover have $x < a + b \Longrightarrow a + b - x \le a + b$ **for** x
by (*metis diff-0-right least-mod-type less-le mod-less-diff not-less*)
ultimately have $x < a + b \Longrightarrow \exists xa.\ x = a + b - xa \wedge 0 < xa \wedge xa \le a + b$ **for** x
by (*auto simp*: *algebra-simps intro*: *exI*[*of - a + b − x*])
thus $(-)\ (a + b)$ ‘ $\{0<..a + b\} = \{..<a + b\}$
by (*fastforce intro*!: *mod-less-diff*)
qed
thus *?thesis*
by *auto*
qed
also have ... $= 1$
by *auto*
finally show *?thesis*.
qed (*simp add*: *nn-integral-count-space-finite*)
qed

definition *A-inv* $(s::'s) = \{a::'s.\ Rep\ s + Rep\ a < CARD('s)\}$

end

definition *var-cost-lin* $(c::real)\ n = c * Rep\ n$
definition *inv-cost-lin* $(c::real)\ n = c * Rep\ n$
definition *revenue-lin* $(c::real)\ n = c * Rep\ n$

lift-definition *demand-unif* :: $('a::finite)\ pmf$ **is** λ-. *1 / card* $(UNIV::'a\ set)$
  by (*auto simp*: *ennreal-divide-times divide-ennreal*[*symmetric*] *ennreal-of-nat-eq-real-of-nat*)

lift-definition *demand-three* :: *3 pmf* **is** $\lambda i.$ *if* $i = 1$ *then 1/4 else if* $i = 2$ *then 1/2 else 1/4*
proof −
  have ∗: $(UNIV :: (3\ set)) = \{0,1,2\}$
    using *exhaust-3*
    by *fastforce*
  show *?thesis*
    apply (*auto simp*: *nn-integral-count-space-finite*)
    unfolding ∗
    by *auto*
qed

abbreviation *fixed-cost* ≡ *4*
abbreviation *var-cost* ≡ *var-cost-lin 2*
abbreviation *inv-cost* ≡ *inv-cost-lin 1*
abbreviation *revenue* ≡ *revenue-lin 8*
abbreviation *discount* ≡ *0.99*
type-synonym *capacity* = *30*

lemma *card-ge-2-imp-ne*: $CARD('a) \geq 2 \Longrightarrow \exists (x::'a::finite)\ y::'a.\ x \neq y$
  by (*meson card-2-iff′ ex-card*)

global-interpretation *inventory-ex*: *inventory fixed-cost var-cost*:: *capacity* ⇒ *real inv-cost demand-unif revenue discount*
  defines *A-inv* = *inventory-ex.A-inv*
    and *transition* = *inventory-ex.transition*
    and *reward* = *inventory-ex.reward*
    and *prob-ge-inv* = *inventory-ex.prob-ge-inv*
    and *order-cost* = *inventory-ex.order-cost*
    and *exp-rev* = *inventory-ex.exp-rev*.

abbreviation *K* ≡ *inventory-ex.transition*
abbreviation *A* ≡ *inventory-ex.A-inv*
abbreviation *r* ≡ *inventory-ex.reward*
abbreviation *l* ≡ *0.95*

definition *eps = 0.1*

definition *fun-to-list f = map f (sorted-list-of-set UNIV)*
definition *benchmark-gs (- :: unit) = map Rep (fun-to-list (vi-policy′ K A r l eps 0))*
definition *benchmark-vi (- :: unit) = map Rep (fun-to-list (vi-gs-policy K A r l eps 0))*
definition *benchmark-mpi (- :: unit) = map Rep (fun-to-list (fst (mpi-user K A r l eps (λ- -. 3))))*
definition *benchmark-pi (- :: unit) = map Rep (fun-to-list (policy-iteration K A r l 0))*

fun *vs-n* **where**
vs-n 0 v = v
| *vs-n (Suc n) v = vs-n n (mod-MDP-*$\mathcal{L}_b$ *K A r l v)*

definition *vs-n′ n = vs-n n 0*

definition *benchmark-vi-n n = (fun-to-list (vs-n n 0))*
definition *benchmark-vi-nopol = (fun-to-list (mod-MDP-value-iteration K A r l (1/10) 0))*

export-code *dist vs-n′ benchmark-vi-nopol benchmark-vi-n nat-of-integer integer-of-int benchmark-gs benchmark-vi benchmark-mpi benchmark-pi*
in *Haskell* **module-name** *DP*

export-code *integer-of-int benchmark-gs benchmark-vi benchmark-mpi benchmark-pi* **in** *SML* **module-name** *DP*

end

theory *Code-Gridworld*
imports
Code-Mod
begin

10 Gridworld Example

lemma [*code abstype*]: *embed-pmf (pmf P) = P*
by (*metis (no-types, lifting) td-pmf-embed-pmf type-definition-def*)

lemmas [*code-abbrev del*] = *pmf-integral-code-unfold*

lemma [*code-unfold*]:
measure-pmf.expectation P (f :: 'a :: enum ⇒ real) = ($\sum$ *x ∈ UNIV.*

pmf P x ∗ f x)
  by (*metis* (*no-types*, *lifting*) *UNIV-I finite-class.finite-UNIV integral-measure-pmf*
  real-scaleR-def sum.cong)

lemma [*code*]: *pmf* (*return-pmf x*) = (λ*y. indicat-real* {*y*} *x*)
  by *auto*

lemma [*code*]:
  pmf (*bind-pmf N f*) = (λ*i* :: *'a. measure-pmf.expectation N* (λ(*x* :: *'b* ::*enum*). *pmf* (*f x*) *i*))
  using *Probability-Mass-Function.pmf-bind*
  by *fast*

type-synonym *state-robot* = *13*

definition *from-state x* = (*Rep x div 4*, *Rep x mod 4*)
definition *to-state x* = (*Abs* (*fst x ∗ 4* + *snd x*) :: *state-robot*)

type-synonym *action-robot* = *4*

fun *A-robot* :: *state-robot* ⇒ *action-robot set* **where**
  A-robot pos = *UNIV*

abbreviation *noise* ≡ (*0.2* :: *real*)

lift-definition *add-noise* :: *action-robot* ⇒ *action-robot pmf* **is** λ*det rnd.* (
  if det = *rnd then 1* − *noise else if det* = *rnd* − *1* ∨ *det* = *rnd* + *1 then noise* / *2 else 0*)
  subgoal for *n*
    unfolding *nn-integral-count-space-finite*[*OF finite*] *UNIV-4*
    using *exhaust-4*[*of n*]
    by *fastforce*
  done

fun *r-robot* :: (*state-robot* × *action-robot*) ⇒ *real* **where**
  r-robot (*s*,*a*) = (
  if from-state s = (*2*,*3*) *then 1 else*
  if from-state s = (*1*,*3*) *then* −*1 else*
  if from-state s = (*3*,*0*) *then 0 else*
  0)

fun *K-robot* :: (*state-robot* × *action-robot*) ⇒ *state-robot pmf* **where**
  K-robot (*loc*, *a*) =
  do {
  a ← *add-noise a*;

$let\ (y,\ x) = from\text{-}state\ loc;$
$let\ (y',\ x') =$
$(if\ a = 0\ then\ (y + 1\ ,\ x)$
$else\ if\ a = 1\ then\ (y,\ x{+}1)$
$else\ if\ a = 2\ then\ (y{-}1,\ x)$
$else\ if\ a = 3\ then\ (y,\ x{-}1)$
$else\ undefined);$
$return\text{-}pmf\ ($
$if\ (y,x) = (2,3) \vee (y,x) = (1,3) \vee (y,x) = (3,0)$
$then\ to\text{-}state\ (3,0)$
$else\ if\ y' < 0 \vee y' > 2 \vee x' < 0 \vee x' > 3 \vee (y',x') = (1,1)$
$then\ to\text{-}state\ (y,\ x)$
$else\ to\text{-}state\ (y',\ x'))$
$\}$

definition *l-robot* = *0.9*

lemma *vi-code A-robot r-robot l-robot*
by *standard* (*auto simp*: *l-robot-def*)

abbreviation *to-gridworld f* ≡ *f K-robot r-robot l-robot*
abbreviation *to-gridworld′ f* ≡ *f K-robot A-robot r-robot l-robot*

abbreviation *gridworld-policy-eval′* ≡ *to-gridworld mod-MDP-policy-eval′*
abbreviation *gridworld-policy-step′* ≡ *to-gridworld′ mod-MDP-policy-iteration-policy-step′*
abbreviation *gridworld-mpi-user* ≡ *to-gridworld′ mod-MDP-mpi-user*
abbreviation *gridworld-opt-policy-gs* ≡ *to-gridworld′ mod-MDP-opt-policy-gs*
abbreviation *gridworld-*$\mathcal{L}_b$ ≡ *to-gridworld′ mod-MDP-*$\mathcal{L}_b$
abbreviation *gridworld-find-policy′* ≡ *to-gridworld′ mod-MDP-find-policy′*
abbreviation *gridworld-GS-rec-fun*$_b$ ≡ *to-gridworld′ mod-MDP-GS-rec-fun*$_b$
abbreviation *gridworld-vi-policy′* ≡ *to-gridworld′ mod-MDP-vi-policy′*
abbreviation *gridworld-vi-gs-policy* ≡ *to-gridworld′ mod-MDP-vi-gs-policy*
abbreviation *gridworld-policy-iteration* ≡ *to-gridworld′ mod-MDP-policy-iteration*

fun *pi-robot-n* **where**
pi-robot-n 0 d = (*d*, *gridworld-policy-eval′ d*) |
pi-robot-n (*Suc n*) *d* = *pi-robot-n n* (*gridworld-policy-step′ d*)

definition *mpi-robot eps* = *gridworld-mpi-user eps* (λ-. *3*)

fun *gs-robot-n* **where**
gs-robot-n (*0* :: *nat*) *v* = (*gridworld-opt-policy-gs v*, *v*) |
gs-robot-n (*Suc n* :: *nat*) *v* = *gs-robot-n n* (*gridworld-GS-rec-fun*$_b$ *v*)

fun *vi-robot-n* **where**
vi-robot-n (*0* :: *nat*) *v* = (*gridworld-find-policy′ v*, *v*) |
vi-robot-n (*Suc n* :: *nat*) *v* = *vi-robot-n n* (*gridworld-*$\mathcal{L}_b$ *v*)

definition *mpi-result eps* =
  (*let* (*d*, *v*) = *mpi-robot eps in* (*d*,*v*))

definition *gs-result n* =
  (*let* (*d*,*v*) = *gs-robot-n n 0 in* (*d*,*v*))

definition *vi-result-n n* =
  (*let* (*d*, *v*) = *vi-robot-n n 0 in* (*d*,*v*))

definition *pi-result-n n* =
  (*let* (*d*, *v*) = *pi-robot-n n* (*vec-to-fun 0*) *in* (*d*,*v*))

definition *fun-to-list f* = *map f* (*sorted-list-of-set UNIV*)

definition *benchmark-gs* = *fun-to-list* (*gridworld-vi-policy′ 0.1 0*)
definition *benchmark-vi* = *fun-to-list* (*gridworld-vi-gs-policy 0.1 0*)
definition *benchmark-mpi* = *fun-to-list* (*fst* (*gridworld-mpi-user 0.1* (λ- -. *3*)))
definition *benchmark-pi* = *fun-to-list* (*gridworld-policy-iteration 0*)

export-code *benchmark-gs benchmark-vi benchmark-mpi benchmark-pi*
in *Haskell* **module-name** *DP*
export-code *benchmark-gs benchmark-vi benchmark-mpi benchmark-pi*
in *SML* **module-name** *DP*
end

theory *Examples*
  imports
    Code-Inventory
    Code-Gridworld
begin
end

References

[1] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Statistics. Wiley, 1994.