Verified Algorithms for Solving Markov Decision Processes

Maximilian Schäffeler and Mohammad Abdulaziz

March 17, 2025

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 *MDP-fin*
  imports
  MDP−Rewards.MDP-reward
begin

locale *MDP-on* = *MDP-act-disc arb-act A K r l*
  for
    A **and**
    $K :: 's ::countable \times 'a ::countable \Rightarrow 's\ pmf$ **and** *r l arb-act* +
  fixes $S :: 's\ set$
  assumes
    fin-states: *finite S* **and**
    fin-actions: $\bigwedge s.$ *finite* (*A s*) **and**
    K-closed: *set-pmf* (*K* (*s*,*a*)) $\subseteq S$
begin

lemma $\mathcal{L}_b$*-indep*:
  assumes $\bigwedge s.\ s \in S \Longrightarrow$ *apply-bfun v s* = *apply-bfun v′ s*
    and $s \in S$
  shows $\mathcal{L}_b\ v\ s = \mathcal{L}_b\ v'\ s$
proof −
  have *measure-pmf.expectation* (*K* (*s*, *a*)) (*apply-bfun v*) = *measure-pmf.expectation* (*K* (*s*, *a*)) (*apply-bfun v′*) **for** *a*
    using *assms K-closed subsetD*
    by (*auto intro*!: *AE-pmfI Bochner-Integration.integral-cong-AE*)
  thus *?thesis*
    unfolding $\mathcal{L}_b$*-eq-SUP-*L_a L_a*-int* **by** *auto*
qed

end

locale *MDP-nat-type* = *MDP-act A K*
  for $A :: nat \Rightarrow nat\ set$ **and** *K* +
  assumes *A-fin* : $\bigwedge s.$ *finite* (*A s*)

locale *MDP-nat* = *MDP-nat-type* +
  fixes *states* :: *nat*
  assumes *K-closed*: $\forall s <$ *states*. *set-pmf* (*K* (*s*,*a*)) $\subseteq$ {*0..*<*states*}
  assumes *K-closed-compl*: $\forall s \geq$ *states*. *set-pmf* (*K* (*s*,*a*)) $\subseteq$ {*states..*}
  assumes *A-outside*: $\bigwedge s.\ s \geq$ *states* $\Longrightarrow A\ s = \{0\}$

locale *MDP-nat-disc* = *MDP-nat arb-act A K states* + *MDP-act-disc arb-act A K r l*
  for *A K r l arb-act states* +
  assumes *reward-zero-outside*: $\forall s \geq$ *states*. *r* (*s*,*a*) = *0*
begin
lemma $\mathcal{L}_b$*-eq-*L_a*-max′*: $\mathcal{L}_b\ v\ s = (MAX\ a \in A\ s.\ L_a\ a\ v\ s)$

unfolding $\mathcal{L}_b$*-eq-*L_a*-max*
using *finite-arg-max-eq-Max*[*of A s* λa. L_a *a v s*] *A-ne A-fin*
by *auto*

abbreviation *state-space* ≡ {*0..<states*}

lemma *set-pmf-Xn′*: *s* ∉ *state-space* ⟹ *set-pmf* (*Xn′ p s i*) ⊆ {*states..*}
using *K-closed-compl*
by (*induction i arbitrary*: *p s*) (*auto dest*!: *subsetD simp*: *Suc-Xn′ linorder-not-less*)

lemma *set-pmf-Pn′*: *s* ∉ *state-space* ⟹ (∀ *sa* ∈ *set-pmf* (*Pn′ p s i*). *fst sa*∉ *state-space*)
using *set-pmf-Xn′*[*unfolded Xn′-Pn′*] **by** *fastforce*

lemma *reward-Pn′-notin*: *s* ∉ *state-space* ⟹ (∀ *sa* ∈ *set-pmf* (*Pn′ p s i*). *r sa* = *0*)
using *set-pmf-Pn′ reward-zero-outside* **by** (*fastforce simp*: *linorder-not-less*)

lemma ν*-zero-notin*:
assumes *s* ∉ *state-space*
shows ν *p s* = *0*
proof −
have ν*-fin p n s* = *0* **for** *n*
using *assms reward-Pn′-notin*
by (*auto simp*: ν*-fin-eq-Pn intro*!: *sum.neutral integral-eq-zero-AE AE-pmfI*)
thus *?thesis*
unfolding ν*-def* **by** *auto*
qed

lemma ν*-opt-zero-notin*:
assumes *s* ∉ *state-space*
shows ν*-opt s* = *0*
unfolding ν*-opt-def* **using** *assms* ν*-zero-notin policies-ne* **by** *auto*

end

end

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.

abbreviation *term-measure* $\equiv (\lambda(eps,\ v).\ LEAST\ n.\ (2 * l * dist\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }(Suc\ n))\ v)\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }n)\ v) < eps * (1{-}l)))$

lemma *Least-Suc-less*:
  assumes $\exists\, n.\ P\ n\ \neg P\ 0$
  shows $Least\ (\lambda n.\ P\ (Suc\ n)) < Least\ P$
  using *assms* **by** (*auto simp*: *Least-Suc*)

function *value-iteration* :: $real \Rightarrow ('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow_b real)$ **where**
  $value\text{-}iteration\ eps\ v =$
  $(if\ 2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1{-}l) \vee eps \le 0\ then\ \mathcal{L}_b\ v\ else\ value\text{-}iteration\ eps\ (\mathcal{L}_b\ v))$
  by *auto*
termination
proof (*relation Wellfounded.measure term-measure*)
  fix *eps v*
  assume h: $\neg\ (2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1 - l) \vee eps \le 0)$
  show $((eps,\ \mathcal{L}_b\ v),\ eps,\ v) \in Wellfounded.measure\ term\text{-}measure$
  proof −
    have $(\lambda n.\ dist\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ Suc\ n)\ v)\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ n)\ v)) \longrightarrow 0$
      using *dist-*$\mathcal{L}_b$*-tendsto*
      by (*auto simp*: *dist-commute*)
    hence $*$: $\exists\, n.\ dist\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ Suc\ n)\ v)\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ n)\ v) < eps$ **if** $eps > 0$ **for** *eps*
      unfolding *LIMSEQ-def* **using** *that* **by** *auto*
    have $**$: $0 < l * 2$ **if** $0 \ne l$
      using *zero-le-disc that* **by** *linarith*
    hence $(LEAST\ n.\ (2 * l) * dist\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ (Suc\ (Suc\ n)))\ v)\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ (Suc\ n))\ v) < eps * (1 - l)) <$
      $(LEAST\ n.\ (2 * l) * dist\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ Suc\ n)\ v)\ ((\mathcal{L}_b \widehat{\ }\widehat{\ }\ n)\ v) < eps * (1 - l))$ **if** $0 \ne l$
        using $h\ *[of\ eps * (1{-}l)\ /\ (2 * l)]$ *that*

by (*fastforce simp*: ** *algebra-simps dist-commute pos-less-divide-eq intro*!: *Least-Suc-less*)
thus *?thesis*
using *h* **by** (*cases* $l = 0$) (*auto simp*: *funpow-swap1*)
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*: $(1 - l) * dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (\mathcal{L}_b\ v)$
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 $2 * dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt < eps$
proof –
have $2 * l * dist\ v\ \nu_b\text{-}opt * (1 - l) \leq 2 * l * dist\ v\ (\mathcal{L}_b\ v)$
using *contraction-$\mathcal{L}$-dist* **by** (*simp add*: *mult-left-mono mult.commute*)
hence $2 * l * dist\ v\ \nu_b\text{-}opt * (1 - l) < eps * (1-l)$
using *assms*(*2*) **by** *linarith*
hence $2 * l * dist\ v\ \nu_b\text{-}opt < eps$
by *force*
thus $2 * dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt < eps$
using *contraction-$\mathcal{L}$*[*of v ν_b-opt*] **by** *auto*
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

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 $2 * dist\ (value\text{-}iteration\ eps\ v)\ \nu_b\text{-}opt < eps$
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*)

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 $(\lambda n.$ *dist* (*vi v* (*Suc n*)) (*vi v n*)) $\longrightarrow$ *0*
  using *dist*-$\mathcal{L}_b$*-tendsto*[*of v*] **by** (*auto simp*: *dist-commute*)

end

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

lemma *is-arg-max-find-policy*: *is-arg-max* $(\lambda d.\ L_a\ d\ (apply\text{-}bfun\ v)\ s)$ $(\lambda d.\ d \in A\ s)$ $(find\text{-}policy\ v\ s)$
using *Sup-att*
by (*simp add*: *find-policy-def arg-max-on-def arg-max-def someI-ex max-L-ex-def has-arg-max-def*)

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-dist-* $\mathcal{L}_b$:
assumes $eps > 0$ $2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1-l)$
shows $2 * dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (find\text{-}policy\ (\mathcal{L}_b\ v))))\ (\mathcal{L}_b\ v) \leq eps$
proof –
let $?d = mk\text{-}dec\text{-}det\ (find\text{-}policy\ (\mathcal{L}_b\ v))$
let $?p = mk\text{-}stationary\ ?d$
have *L-eq-* $\mathcal{L}_b$: $L\ (mk\text{-}dec\text{-}det\ (find\text{-}policy\ v))\ v = \mathcal{L}_b\ v$ **for** v
by (*auto simp*: *L-eq-* L_a *-det* $\mathcal{L}_b$ *-eq-argmax-* L_a[*OF is-arg-max-find-policy*])
have $dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) = dist\ (L\ ?d\ (\nu_b\ ?p))\ (\mathcal{L}_b\ v)$
using *L-* ν *-fix* **by** *force*
also have $\ldots \leq dist\ (L\ ?d\ (\nu_b\ ?p))\ (\mathcal{L}_b\ (\mathcal{L}_b\ v)) + dist\ (\mathcal{L}_b\ (\mathcal{L}_b\ v))\ (\mathcal{L}_b\ v)$
using *dist-triangle* **by** *blast*
also have $\ldots = dist\ (L\ ?d\ (\nu_b\ ?p))\ (L\ ?d\ (\mathcal{L}_b\ v)) + dist\ (\mathcal{L}_b\ (\mathcal{L}_b\ v))\ (\mathcal{L}_b\ v)$
by (*auto simp*: *L-eq-* $\mathcal{L}_b$)
also have $\ldots \leq l * dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) + l * dist\ (\mathcal{L}_b\ v)\ v$
using *contraction-* $\mathcal{L}$ *contraction-L* **by** (*fastforce intro*!: *add-mono*)
finally have *aux*: $dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) \leq l * dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) + l * dist\ (\mathcal{L}_b\ v)\ v$ **.**
hence $dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) * (1 - l) \leq l * dist\ (\mathcal{L}_b\ v)\ v$
by (*auto simp*: *algebra-simps*)
hence $*$: $2 * dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) * (1 - l) \leq 2 * l * dist\ (\mathcal{L}_b\ v)\ v$
using *zero-le-disc mult-left-mono* **by** *auto*
hence $2 * dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) * (1 - l) \leq eps * (1 - l)$
using *assms* **by** (*fastforce simp*: *dist-commute intro*!: *order.trans*[*OF* $*$])
thus $2 * dist\ (\nu_b\ ?p)\ (\mathcal{L}_b\ v) \leq eps$
by *auto*
qed

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

```
    using dist-triangle by blast
  thus ?thesis
    using find-policy-dist-ℒ_b[OF assms] dist-ℒ_b-opt-eps[OF assms] by
simp
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 2 * l * dist (vi v (Suc n)) (vi v n) < eps * (1−l)
  shows 2 * dist (vi v (Suc n)) ν_b-opt < eps
  using dist-ℒ_b-opt-eps assms by (simp add: dist-commute)
end

context MDP-act-disc 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 is-arg-max-find-policy': is-arg-max (λd. L_a d (apply-bfun v) s)
(λd. d ∈ A s) (find-policy' v s)
  using is-opt-act-some by (auto simp: find-policy'-def is-opt-act-def)

lemma find-policy'-dist-ℒ_b:
  assumes eps > 0 2 * l * dist v (ℒ_b v) < eps * (1−l)
  shows 2 * dist (ν_b (mk-stationary-det (find-policy' (ℒ_b v)))) (ℒ_b v)
≤ eps
proof −
  let ?d = mk-dec-det (find-policy' (ℒ_b v))
  let ?p = mk-stationary ?d
  have L-eq-ℒ_b: L (mk-dec-det (find-policy' v)) v = ℒ_b v for v
   by (auto simp: L-eq-L_a-det ℒ_b-eq-argmax-L_a[OF is-arg-max-find-policy'])
  have dist (ν_b ?p) (ℒ_b v) = dist (L ?d (ν_b ?p)) (ℒ_b v)
    using L-ν-fix by force
  also have ... ≤ dist (L ?d (ν_b ?p)) (ℒ_b (ℒ_b v)) + dist (ℒ_b (ℒ_b v))
(ℒ_b v)
    using dist-triangle by blast
  also have ... = dist (L ?d (ν_b ?p)) (L ?d (ℒ_b v)) + dist (ℒ_b (ℒ_b
v)) (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*) $*$ (*1* − *l*) $\leq$ *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*
by (*auto simp*: *algebra-simps*)
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*
hence *2* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $*$ (*1* − *l*) $\leq$ *eps* $*$ (*1* − *l*)
using *assms* **by** (*fastforce simp*: *dist-commute intro*!: *order.trans*[*OF* $*$])
thus *2* $*$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) $\leq$ *eps*
by *auto*
qed

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 *?p* = *mk-stationary-det* (*find-policy′* ($\mathcal{L}_b$ *v*))
have *dist* (ν_b *?p*) ν_b*-opt* $\leq$ *dist* (ν_b *?p*) ($\mathcal{L}_b$ *v*) + *dist* ($\mathcal{L}_b$ *v*) ν_b*-opt*
using *dist-triangle* **by** *blast*
thus *?thesis*
using *find-policy′-dist-*$\mathcal{L}_b$[*OF assms*] *dist-*$\mathcal{L}_b$*-opt-eps*[*OF assms*] **by** *simp*
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 eps v*)
then show *?case*
using *find-policy′-error-bound* **by** (*auto simp*: *value-iteration.simps*[*of* - *v*])
qed

end
end

theory *DiffArray-Base*
imports
Main
HOL−*Library.Code-Target-Numeral*

begin

1.1 Additional List Operations

definition *tabulate n f* = *map f* [*0..<n*]

context
 notes [*simp*] = *tabulate-def*
begin

lemma *tabulate0*[*simp*]: *tabulate 0 f* = [] **by** *simp*

lemma *tabulate-Suc*: *tabulate* (*Suc n*) *f* = *tabulate n f* @ [*f n*] **by** *simp*

lemma *tabulate-Suc′*: *tabulate* (*Suc n*) *f* = *f 0* # *tabulate n* (*f o Suc*)
 by (*simp add*: *map-upt-Suc del*: *upt-Suc*)

lemma *tabulate-const*[*simp*]: *tabulate n* (λ-. *c*) = *replicate n c* **by** (*auto simp*: *map-replicate-trivial*)

lemma *length-tabulate*[*simp*]: *length* (*tabulate n f*) = *n* **by** *simp*
lemma *nth-tabulate*[*simp*]: *i*<*n* ⟹ *tabulate n f* ! *i* = *f i* **by** *simp*

lemma *upd-tabulate*: (*tabulate n f*)[*i*:=*x*] = *tabulate n* (*f*(*i*:=*x*))
 apply (*induction n*)
 by (*auto simp*: *list-update-append split*: *nat.split*)

lemma *take-tabulate*: *take n* (*tabulate m f*) = *tabulate* (*min n m*) *f*
 by (*simp add*: *min-def take-map*)

lemma *hd-tabulate*[*simp*]: *n*≠*0* ⟹ *hd* (*tabulate n f*) = *f 0*
 by (*cases n*) (*simp add*: *map-upt-Suc del*: *upt-Suc*)+

lemma *tl-tabulate*: *tl* (*tabulate n f*) = *tabulate* (*n*−*1*) (*f o Suc*)
 apply (*simp add*: *map-upt-Suc map-Suc-upt del*: *upt-Suc flip*: *map-tl map-map*)
 by (*cases n*; *simp*)

lemma *tabulate-cong*[*fundef-cong*]: *n*=*n′* ⟹ (⋀*i*. *i*<*n* ⟹ *f i* = *f′ i*) ⟹ *tabulate n f* = *tabulate n′ f′*
 by *simp*

lemma *tabulate-nth-take*: *n* ≤ *length xs* ⟹ *tabulate n* ((!) *xs*) = *take n xs*
 by (*rule nth-equalityI*, *auto*)

end

lemma *drop-tabulate*: *drop n* (*tabulate m f*) = *tabulate* (*m*−*n*) (*f o* (+)*n*)
apply (*induction n arbitrary*: *m f*)
apply (*auto simp*: *drop-Suc drop-tl tl-tabulate comp-def*)
done

1.2 Primitive Operations

typedef *'a array* = *UNIV* :: *'a list set*
morphisms *array-α Array*
by *blast*
setup-lifting *type-definition-array*

lift-definition *array-new* :: *nat* ⇒ *'a* ⇒ *'a array* **is** *λn a. replicate n a* **.**

lift-definition *array-tabulate* :: *nat* ⇒ (*nat* ⇒ *'a*) ⇒ *'a array* **is** *λn f. Array* (*tabulate n f*) **.**

lift-definition *array-length* :: *'a array* ⇒ *nat* **is** *length* **.**

lift-definition *array-get* :: *'a array* ⇒ *nat* ⇒ *'a* **is** *nth* **.**

lift-definition *array-set* :: *'a array* ⇒ *nat* ⇒ *'a* ⇒ *'a array* **is** *list-update* **.**

lift-definition *array-of-list* :: *'a list* ⇒ *'a array* **is** ‹*λx. x*› **.**

1.2.1 Refinement Lemmas

named-theorems *array-refine*
context
notes [*simp*] = *Array-inverse*
begin

lemma *array-α-inj*: *array-α a* = *array-α b* ⟹ *a*=*b* **by** *transfer auto*

lemma *array-eq-iff*: *a*=*b* ⟷ *array-α a* = *array-α b* **by** *transfer auto*

lemma *array-new-refine*[*simp*,*array-refine*]: *array-α* (*array-new n a*) = *replicate n a* **by** *transfer auto*

lemma *array-tabulate-refine*[*simp*,*array-refine*]: *array-α* (*array-tabulate n f*) = *tabulate n f* **by** *transfer auto*

lemma *array-length-refine*[*simp*,*array-refine*]: *array-length a* = *length* (*array-α a*) **by** *transfer auto*

lemma *array-get-refine*[*simp*,*array-refine*]: *array-get a i* = *array-α a* ! *i* **by** *transfer auto*

lemma *array-set-refine*[*simp*,*array-refine*]: *array-α* (*array-set a i x*) = (*array-α a*)[*i* := *x*] **by** *transfer auto*

lemma *array-of-list-refine*[*simp*,*array-refine*]: *array-α* (*array-of-list xs*) = *xs* **by** *transfer auto*

end

lifting-update *array.lifting*
lifting-forget *array.lifting*

1.3 Basic Derived Operations

These operations may have direct implementations in target language

definition *array-grow a n dflt* = (
let la = *array-length a in*
array-tabulate n (*λi. if i*<*la then array-get a i else dflt*)
)

lemma *tabulate-grow*: *tabulate n* (*λi. if i* < *length xs then xs*!*i else d*) = *take n xs* @ (*replicate* (*n*−*length xs*) *d*)
apply (*induction n*)
apply (*auto simp*: *tabulate-Suc take-Suc-conv-app-nth replicate-append-same Suc-diff-le*)
done

lemma *array-grow-refine*[*simp*,*array-refine*]:
array-α (*array-grow a n d*) = *take n* (*array-α a*) @ *replicate* (*n*−*length* (*array-α a*)) *d*
by (*auto simp*: *array-grow-def tabulate-grow cong*: *if-cong*)

definition *array-take a n* = (
let n = *min* (*array-length a*) *n in*
array-tabulate n (*array-get a*)
)

lemma *tabulate-take*: *tabulate* (*min* (*length xs*) *n*) ((!) *xs*) = *take n xs*
by (*auto simp*: *min-def tabulate-nth-take*)

lemma *array-take-refine*[*simp*,*array-refine*]: *array-α* (*array-take a n*) = *take n* (*array-α a*)
by (*auto simp*: *array-take-def tabulate-take cong*: *tabulate-cong*)

The following is a total version of *array-get*, which returns a default value in case the index is out of bounds. This can be efficiently implemented in the target language by catching exceptions.

definition *array-get-oo x a i* ≡
  if i<*array-length a then array-get a i else x*

lemma *array-get-oo-refine*[*simp*,*array-refine*]: *array-get-oo x a i* = (*if* i<*length* (*array-α a*) *then array-α a*!*i else x*)
  by (*simp add*: *array-get-oo-def*)

definition *array-set-oo f a i x* ≡
  if i<*array-length a then array-set a i x else f*()

lemma *array-set-oo-refine*[*simp*,*array-refine*]: *array-α* (*array-set-oo f a i x*)
  = (*if* i<*length* (*array-α a*) *then* (*array-α a*)[*i*:=*x*] *else array-α* (*f* ()))
  by (*simp add*: *array-set-oo-def*)

Map array. No old versions for intermediate results need to be tracked, which allows more efficient in-place implementation in case access to old versions is forbidden.

definition *array-map f a* ≡ *array-tabulate* (*array-length a*) (*f o array-get a*)

lemma *array-map-refine*[*simp*,*array-refine*]: *array-α* (*array-map f a*) = *map f* (*array-α a*)
  unfolding *array-map-def*
  apply (*auto simp*: *tabulate-def simp flip*: *map-map cong*: *map-cong*)
  by (*smt* (*z3*) *atLeastLessThan-iff length-map map-eq-conv map-nth nth-map set-upt*)

lemma *array-map-cong*[*fundef-cong*]: $a=a'$ ⟹ (⋀*x*. *x*∈*set* (*array-α a*) ⟹ $f\ x = f'\ x$) ⟹ *array-map f a* = *array-map* f' a'
  by (*simp add*: *array-eq-iff*)

context
  fixes f :: $'a \Rightarrow {'s} \Rightarrow {'s}$ **and** *xs* :: $'a$ *list*
begin
function *foldl-idx* **where**
  foldl-idx i s = (*if i*<*length xs then foldl-idx* (*i*+*1*) (*f* (*xs*!*i*) *s*) *else s*)

  by *pat-completeness auto*
termination
  apply (*relation measure* (λ(*i*,-). *length xs* − *i*))
  apply *auto*
  done

lemmas [*simp del*] = *foldl-idx.simps*

lemma *foldl-idx-eq*: *foldl-idx i s* = *fold f* (*drop i xs*) *s*
  apply (*induction i s rule*: *foldl-idx.induct*)
  apply (*subst foldl-idx.simps*)
  apply (*auto simp flip*: *Cons-nth-drop-Suc*)
  done

lemma *fold-by-idx*: *fold f xs s* = *foldl-idx 0 s* **using** *foldl-idx-eq* **by** *simp*

end

fun *foldr-idx* **where**
  foldr-idx f xs 0 s = *s*
| *foldr-idx f xs* (*Suc i*) *s* = *foldr-idx f xs i* (*f* (*xs*!*i*) *s*)

lemma *foldr-idx-eq*: *i*≤*length xs* ⟹ *foldr-idx f xs i s* = *foldr f* (*take i xs*) *s*
  apply (*induction i arbitrary*: *s*)
  apply (*auto simp*: *take-Suc-conv-app-nth*)
  done

lemma *foldr-by-idx*: *foldr f xs s* = *foldr-idx f xs* (*length xs*) *s* **apply** (*subst foldr-idx-eq*) **by** *auto*

context
  fixes *f* :: *'a* ⇒ *'s* ⇒ *'s* **and** *a* :: *'a array*
begin

function *array-foldl-idx* **where**
  array-foldl-idx i s = (*if i*<*array-length a then array-foldl-idx* (*i*+*1*) (*f* (*array-get a i*) *s*) *else s*)
  by *pat-completeness auto*
termination
  apply (*relation measure* (λ(*i*,-). *array-length a* − *i*))
  apply *auto*
  done

lemmas [*simp del*] = *array-foldl-idx.simps*

end

lemma *array-foldl-idx-refine*[*simp*, *array-refine*]: *array-foldl-idx f a i s* = *foldl-idx f* (*array-α a*) *i s*
  apply (*induction i s rule*: *foldl-idx.induct*)
  apply (*subst array-foldl-idx.simps*)
  apply (*subst foldl-idx.simps*)
  by *fastforce*

definition *array-fold f a s* ≡ *array-foldl-idx f a 0 s*
lemma *array-fold-refine*[*simp*, *array-refine*]: *array-fold f a s* = *fold f* (*array-α a*) *s*
  unfolding *array-fold-def*
  by (*simp add*: *fold-by-idx*)

fun *array-foldr-idx* **where**
  array-foldr-idx f xs 0 s = *s*
| *array-foldr-idx f xs* (*Suc i*) *s* = *array-foldr-idx f xs i* (*f* (*array-get xs i*) *s*)

lemma *array-foldr-idx-refine*[*simp*, *array-refine*]: *array-foldr-idx f xs i s* = *foldr-idx f* (*array-α xs*) *i s*
  apply (*induction i arbitrary*: *s*)
  by *auto*

definition *array-foldr f xs s* ≡ *array-foldr-idx f xs* (*array-length xs*) *s*

lemma *array-foldr-refine*[*simp*, *array-refine*]: *array-foldr f xs s* = *foldr f* (*array-α xs*) *s*
  by (*simp add*: *array-foldr-def foldr-by-idx*)

1.4 Code Generator Setup

1.4.1 Code-Numeral Preparation

definition [*code del*]: *array-new′* == *array-new o nat-of-integer*
definition [*code del*]: *array-tabulate′ n f* ≡ *array-tabulate* (*nat-of-integer n*) (*f o integer-of-nat*)

definition [*code del*]: *array-length′* == *integer-of-nat o array-length*
definition [*code del*]: *array-get′ a* == *array-get a o nat-of-integer*
definition [*code del*]: *array-set′ a* == *array-set a o nat-of-integer*
definition [*code del*]:
  array-get-oo′ x a == *array-get-oo x a o nat-of-integer*
definition [*code del*]:
  array-set-oo′ f a == *array-set-oo f a o nat-of-integer*

lemma [*code*]:
  array-new == *array-new′ o integer-of-nat*
  array-tabulate n f == *array-tabulate′* (*integer-of-nat n*) (*f o nat-of-integer*)
  array-length == *nat-of-integer o array-length′*
  array-get a == *array-get′ a o integer-of-nat*
  array-set a == *array-set′ a o integer-of-nat*
  array-get-oo x a == *array-get-oo′ x a o integer-of-nat*
  array-set-oo g a == *array-set-oo′ g a o integer-of-nat*
  by (*simp-all*

del: *array-refine*
add: *o-def*
add: *array-new'-def array-tabulate'-def array-length'-def array-get'-def array-set'-def*
array-get-oo'-def array-set-oo'-def)

Fallbacks

lemmas *array-get-oo'-fallback*[*code*] = *array-get-oo'-def*[*unfolded array-get-oo-def*[*abs-def*]]
lemmas *array-set-oo'-fallback*[*code*] = *array-set-oo'-def*[*unfolded array-set-oo-def*[*abs-def*]]

lemma *array-tabulate'-fallback*[*code*]:
array-tabulate' n f = *array-of-list* (*map* (*f o integer-of-nat*) [*0..<nat-of-integer n*])
unfolding *array-tabulate'-def*
by (*simp add*: *array-eq-iff tabulate-def*)

lemma *array-new'-fallback*[*code*]: *array-new' n x* = *array-of-list* (*replicate* (*nat-of-integer n*) *x*)
by (*simp add*: *array-new'-def array-eq-iff*)

1.4.2 Haskell

code-printing type-constructor *array* ⇀
(*Haskell*) *Array.ArrayType*/ -

code-reserved (*Haskell*) *array-of-list*

code-printing code-module *Array* ⇀
(*Haskell*) ‹*module Array where* {

−−import qualified Data.Array.Diff as Arr;
import qualified Data.Array as Arr;

type ArrayType = *Arr.Array Integer*;

array-of-size :: *Integer* −> [*e*] −> *ArrayType e*;
array-of-size n = *Arr.listArray* (*0*, *n*−*1*);

array-new :: *Integer* −> *e* −> *ArrayType e*;
array-new n a = *array-of-size n* (*repeat a*);

array-length :: *ArrayType e* −> *Integer*;
array-length a = *let* (*s*, *e*) = *Arr.bounds a in e*;

array-get :: *ArrayType e* −> *Integer* −> *e*;
array-get a i = *a Arr*.! *i*;

array-set :: *ArrayType e* −> *Integer* −> *e* −> *ArrayType e*;
array-set a i e = *a Arr*.// [(*i*, *e*)];

array-of-list :: [*e*] −> *ArrayType e*;
array-of-list xs = *array-of-size* (*toInteger* (*length xs*)) *xs*;

}›

code-printing constant *Array* ⇀ (*Haskell*) *Array.array′-of′-list*
code-printing constant *array-new′* ⇀ (*Haskell*) *Array.array′-new*
code-printing constant *array-length′* ⇀ (*Haskell*) *Array.array′-length*
code-printing constant *array-get′* ⇀ (*Haskell*) *Array.array′-get*
code-printing constant *array-set′* ⇀ (*Haskell*) *Array.array′-set*
code-printing constant *array-of-list* ⇀ (*Haskell*) *Array.array′-of′-list*

1.4.3 SML

We have the choice between single-threaded arrays, that raise an exception if an old version is accessed, and truly functional arrays, that update the array in place, but store undo-information to restore old versions.

code-printing code-module *FArray* ⇀
(*SML*)
‹
structure FArray = *struct*
datatype ′a Cell = *Value of ′a Array.array* | *Upd of* (*int*∗*′a*∗*′a Cell Unsynchronized.ref*);

type ′a array = *′a Cell Unsynchronized.ref*;

fun array (*size*,*v*) = *Unsynchronized.ref* (*Value* (*Array.array* (*size*,*v*)));
fun tabulate (*size*, *f*) = *Unsynchronized.ref* (*Value* (*Array.tabulate*(*size*, *f*)));
fun fromList l = *Unsynchronized.ref* (*Value* (*Array.fromList l*));

fun sub (*Unsynchronized.ref* (*Value a*), *idx*) = *Array.sub* (*a*,*idx*) |
sub (*Unsynchronized.ref* (*Upd* (*i*,*v*,*cr*)),*idx*) =
if i=*idx then v*
else sub (*cr*,*idx*);

fun length (*Unsynchronized.ref* (*Value a*)) = *Array.length a* |

```
      length (Unsynchronized.ref (Upd (i,v,cr))) = length cr;

  fun realize-aux (aref, v) =
    case aref of
      (Unsynchronized.ref (Value a)) => (
        let
          val len = Array.length a;
          val a' = Array.array (len,v);
        in
          Array.copy {src=a, dst=a', di=0};
          Unsynchronized.ref (Value a')
        end
      ) |
      (Unsynchronized.ref (Upd (i,v,cr))) => (
        let val res=realize-aux (cr,v) in
          case res of
             (Unsynchronized.ref (Value a)) => (Array.update (a,i,v);
res)
        end
      );

  fun realize aref =
    case aref of
      (Unsynchronized.ref (Value -)) => aref |
      (Unsynchronized.ref (Upd (i,v,cr))) => realize-aux(aref,v);

  fun update (aref,idx,v) =
    case aref of
      (Unsynchronized.ref (Value a)) => (
        let val nref=Unsynchronized.ref (Value a) in
          aref := Upd (idx,Array.sub(a,idx),nref);
          Array.update (a,idx,v);
          nref
        end
      ) |
      (Unsynchronized.ref (Upd -)) =>
        let val ra = realize-aux(aref,v) in
          case ra of
            (Unsynchronized.ref (Value a)) => Array.update (a,idx,v);
          ra
        end
      ;


structure IsabelleMapping = struct
type 'a ArrayType = 'a array;

fun array-new (n:IntInf.int) (a:'a) = array (IntInf.toInt n, a);
fun array-of-list (xs:'a list) = fromList xs;
```

fun array-tabulate (*n*:*IntInf.int*) (*f*:*IntInf.int* −> ′*a*) = *tabulate* (*IntInf.toInt n*, *f o IntInf.fromInt*)

fun array-length (*a*:′*a ArrayType*) = *IntInf.fromInt* (*length a*);

fun array-get (*a*:′*a ArrayType*) (*i*:*IntInf.int*) = *sub* (*a*, *IntInf.toInt i*);

fun array-set (*a*:′*a ArrayType*) (*i*:*IntInf.int*) (*e*:′*a*) = *update* (*a*, *IntInf.toInt i*, *e*);

fun array-get-oo (*d*:′*a*) (*a*:′*a ArrayType*) (*i*:*IntInf.int*) =
  sub (*a*,*IntInf.toInt i*) *handle Subscript* => *d*

fun array-set-oo (*d*:(*unit*−>′*a ArrayType*)) (*a*:′*a ArrayType*) (*i*:*IntInf.int*) (*e*:′*a*) =
  update (*a*, *IntInf.toInt i*, *e*) *handle Subscript* => *d* ()

end;
end;

›

code-printing
 type-constructor *array* ⇀ (*SML*) -/ *FArray.IsabelleMapping.ArrayType*
| **constant** *Array* ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-of*′*-list*
| **constant** *array-new*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-new*
| **constant** *array-tabulate*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-tabulate*
| **constant** *array-length*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-length*
| **constant** *array-get*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-get*
| **constant** *array-set*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-set*
| **constant** *array-of-list* ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-of*′*-list*
| **constant** *array-get-oo*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-get*′*-oo*
| **constant** *array-set-oo*′ ⇀ (*SML*) *FArray.IsabelleMapping.array*′*-set*′*-oo*

1.4.4 Scala

We use a DiffArray-Implementation in Scala.

code-printing code-module *DiffArray* ⇀
  (*Scala*) ‹
object DiffArray {

  import scala.collection.mutable.ArraySeq

  protected abstract sealed class DiffArray-D[*A*]
   final case class Current[*A*] (*a*:*ArraySeq*[*AnyRef*]) *extends DiffArray-D*[*A*]

```
 final case class Upd[A] (i:Int, v:A, n:DiffArray-D[A]) extends DiffAr-
ray-D[A]

  object DiffArray-Realizer {
    def realize[A](a:DiffArray-D[A]) : ArraySeq[AnyRef] = a match {
      case Current(a) => ArraySeq.empty ++ a
    case Upd(j,v,n) => {val a = realize(n); a.update(j, v.asInstanceOf[AnyRef]);
a}
    }
  }

  class T[A] (var d:DiffArray-D[A]) {

   def realize (): ArraySeq[AnyRef] = { val a=DiffArray-Realizer.realize(d);
d = Current(a); a }
    override def toString() = realize().toSeq.toString

    override def equals(obj:Any) =
      obj.isInstanceOf[T[A]] match {
        case true => obj.asInstanceOf[T[A]].realize().equals(realize())
        case false => false
      }
  }


 def array-of-list[A](l : List[A]) : T[A] = new T(Current(ArraySeq.empty
++ l.asInstanceOf[List[AnyRef]]))
 def array-new[A](sz : BigInt, v:A) = new T[A](Current[A](ArraySeq.fill[AnyRef](sz.intValue)(v.asInsta

  private def length[A](a:DiffArray-D[A]) : BigInt = a match {
    case Current(a) => a.length
    case Upd(-,-,n) => length(n)
  }

  def length[A](a : T[A]) : BigInt = length(a.d)

  private def sub[A](a:DiffArray-D[A], i:Int) : A = a match {
    case Current(a) => a(i).asInstanceOf[A]
    case Upd(j,v,n) => if (i==j) v else sub(n,i)
  }

  def get[A](a:T[A], i:BigInt) : A = sub(a.d,i.intValue)

   private def realize[A](a:DiffArray-D[A]): ArraySeq[AnyRef] = Dif-
fArray-Realizer.realize[A](a)

  def set[A](a:T[A], i:BigInt,v:A) : T[A] = a.d match {
    case Current(ad) => {
      val ii = i.intValue;
```

```
      a.d = Upd(ii,ad(ii).asInstanceOf[A],a.d);
      //ad.update(ii,v);
      ad(ii)=v.asInstanceOf[AnyRef]
      new T[A](Current(ad))
    }
   case Upd(-,-,-) => set(new T[A](Current(realize(a.d))), i.intValue,v)
  }


  def get-oo[A](d: => A, a:T[A], i:BigInt):A = try get(a,i) catch {
    case -:scala.IndexOutOfBoundsException => d
  }

  def set-oo[A](d: Unit => T[A], a:T[A], i:BigInt, v:A) : T[A] = try
set(a,i,v) catch {
    case -:scala.IndexOutOfBoundsException => d(())
  }

}

object Test {



 def assert (b : Boolean) : Unit = if (b) () else throw new java.lang.AssertionError(Assertion
Failed)

 def eql[A] (a:DiffArray.T[A], b:List[A]) = assert (a.realize.corresponds(b)((x,y)
=> x.equals(y)))


  def tests1(): Unit = {
    val a = DiffArray.array-of-list(1::2::3::4::Nil)
      eql(a,1::2::3::4::Nil)

    // Simple update
    val b = DiffArray.set(a,2,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)

    // Another update
    val c = DiffArray.set(b,3,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::9::9::Nil)

    // Update of old version (forces realize)
    val d = DiffArray.set(b,2,8)
      eql(a,1::2::3::4::Nil)
```

```
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::9::9::Nil)
      eql(d,1::2::8::4::Nil)

    }

  def tests2(): Unit = {
    val a = DiffArray.array-of-list(1::2::3::4::Nil)
      eql(a,1::2::3::4::Nil)

    // Simple update
    val b = DiffArray.set(a,2,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)

    // Grow of current version
/*     val c = DiffArray.grow(b,6,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::9::4::9::9::Nil)

    // Grow of old version
    val d = DiffArray.grow(a,6,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::9::4::9::9::Nil)
      eql(d,1::2::3::4::9::9::Nil)
*/
  }

  def tests3(): Unit = {
    val a = DiffArray.array-of-list(1::2::3::4::Nil)
      eql(a,1::2::3::4::Nil)

    // Simple update
    val b = DiffArray.set(a,2,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
/*
    // Shrink of current version
    val c = DiffArray.shrink(b,3)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::9::Nil)

    // Shrink of old version
    val d = DiffArray.shrink(a,3)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
```

```
      eql(c,1::2::9::Nil)
      eql(d,1::2::3::Nil)
*/
  }

  def tests4(): Unit = {
    val a = DiffArray.array-of-list(1::2::3::4::Nil)
      eql(a,1::2::3::4::Nil)

    // Update -oo (succeeds)
    val b = DiffArray.set-oo((-) => a,a,2,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)

    // Update -oo (current version,fails)
    val c = DiffArray.set-oo((-) => a,b,5,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::3::4::Nil)

    // Update -oo (old version,fails)
    val d = DiffArray.set-oo((-) => b,a,5,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)
      eql(c,1::2::3::4::Nil)
      eql(d,1::2::9::4::Nil)

  }

  def tests5(): Unit = {
    val a = DiffArray.array-of-list(1::2::3::4::Nil)
      eql(a,1::2::3::4::Nil)

    // Update
    val b = DiffArray.set(a,2,9)
      eql(a,1::2::3::4::Nil)
      eql(b,1::2::9::4::Nil)

    // Get-oo (current version, succeeds)
      assert (DiffArray.get-oo(0,b,2)==9)
    // Get-oo (current version, fails)
      assert (DiffArray.get-oo(0,b,5)==0)
    // Get-oo (old version, succeeds)
      assert (DiffArray.get-oo(0,a,2)==3)
    // Get-oo (old version, fails)
      assert (DiffArray.get-oo(0,a,5)==0)

  }
```

```
  def main(args: Array[String]): Unit = {
    tests1 ()
    tests2 ()
    tests3 ()
    tests4 ()
    tests5 ()


    Console.println(Tests passed)
  }

}

›
```

code-printing
  type-constructor *array* ⇀ (*Scala*) *DiffArray.T*[-]
| **constant** *Array* ⇀ (*Scala*) *DiffArray.array′-of′-list*
| **constant** *array-new′* ⇀ (*Scala*) *DiffArray.array′-new*((-).*toInt*,(-))
| **constant** *array-length′* ⇀ (*Scala*) *DiffArray.length*((-)).*toInt*
| **constant** *array-get′* ⇀ (*Scala*) *DiffArray.get*((-),(-).*toInt*)
| **constant** *array-set′* ⇀ (*Scala*) *DiffArray.set*((-),(-).*toInt*,(-))
| **constant** *array-of-list* ⇀ (*Scala*) *DiffArray.array′-of′-list*
| **constant** *array-get-oo′* ⇀ (*Scala*) *DiffArray.get′-oo*((-),(-),(-).*toInt*)
| **constant** *array-set-oo′* ⇀ (*Scala*) *DiffArray.set′-oo*((-),(-),(-).*toInt*,(-))

context begin
definition *test-diffarray-setup* ≡ (*Array*,*array-new′*,*array-length′*,*array-get′*, *array-set′*, *array-of-list*,*array-get-oo′*,*array-set-oo′*)
export-code *test-diffarray-setup* **checking** *SML OCaml? Haskell?*
end

1.5 Tests

definition *test1* ≡
  let a=*array-of-list* [*1*,*2*,*3*,*4*,*5*,*6*];
      b=*array-tabulate 6* (*Suc*);
      a′=*array-set a 3 42*;
      b′=*array-set b 3 42*;
      c=*array-new 6 0*
  in
    ∀ *i*∈{*0*..<*6*}.
      array-get a i = *i*+*1*
    ∧ *array-get b i* = *i*+*1*
    ∧ *array-get a′ i* = (*if i*=*3 then 42 else i*+*1*)
    ∧ *array-get b′ i* = (*if i*=*3 then 42 else i*+*1*)

$\wedge$ *array-get c i* = (*0*::*nat*)

lemma *enum-rangeE*:
assumes $i \in \{l..<h\}$
assumes *P l*
assumes $i \in \{Suc\ l..<h\} \Longrightarrow P\ i$
shows *P i*
using *assms*
by (*metis atLeastLessThan-iff less-eq-Suc-le nat-less-le*)

lemma *test1*
unfolding *test1-def Let-def*
apply (*intro ballI conjI*)
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
done

ML-val ‹
if not @{*code test1*} *then error ERROR else* ()
›

export-code *test1* **checking** *OCaml? Haskell? SML*

hide-const *test1*
hide-fact *test1-def*

experiment
begin

fun *allTrue* :: *bool list* ⇒ *nat* ⇒ *bool list* **where**
allTrue a 0 = *a* |
allTrue a (*Suc i*) = (*allTrue a i*)[*i* := *True*]

lemma *length-allTrue*: $n \leq$ *length a* $\Longrightarrow$ *length*(*allTrue a n*) = *length a*
by(*induction n*) (*auto*)

lemma $n \leq$ *length a* $\Longrightarrow \forall\, i < n.$ (*allTrue a n*) ! *i*
by(*induction n*) (*auto simp*: *nth-list-update length-allTrue*)

fun *allTrue'* :: *bool array* ⇒ *nat* ⇒ *bool array* **where**

allTrue′ a 0 = a |
allTrue′ a (*Suc i*) = *array-set* (*allTrue′ a i*) *i True*

lemma *array-α* (*allTrue′ xs i*) = *allTrue* (*array-α xs*) *i*
  apply (*induction xs i rule*: *allTrue′.induct*)
  apply *auto*
  done

end

end

2 Single Threaded Arrays

theory *DiffArray-ST*
imports *DiffArray-Base*
begin

2.1 Primitive Operations

typedef *′a starray* = *UNIV* :: *′a array set*
  morphisms *Rep-starray STArray*
  by *blast*
setup-lifting *type-definition-starray*

lift-definition *starray-new* :: *nat* ⇒ *′a* ⇒ *′a starray* **is** *array-new* **.**

lift-definition *starray-tabulate* :: *nat* ⇒ (*nat* ⇒ *′a*) ⇒ *′a starray* **is** *array-tabulate* **.**

lift-definition *starray-length* :: *′a starray* ⇒ *nat* **is** *array-length* **.**

lift-definition *starray-get* :: *′a starray* ⇒ *nat* ⇒ *′a* **is** *array-get* **.**

lift-definition *starray-set* :: *′a starray* ⇒ *nat* ⇒ *′a* ⇒ *′a starray* **is** *array-set* **.**

lift-definition *starray-of-list* :: *′a list* ⇒ *′a starray* **is** ‹*array-of-list*› **.**

lift-definition *starray-grow* :: *′a starray* ⇒ *nat* ⇒ *′a* ⇒ *′a starray* **is** *array-grow* **.**

lift-definition *starray-take* :: $'a$ *starray* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *starray* **is** *array-take* **.**

lift-definition *starray-get-oo* :: $'a \Rightarrow 'a$ *starray* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ **is** *array-get-oo* **.**

lift-definition *starray-set-oo* :: (*unit* $\Rightarrow$ $'a$ *starray*) $\Rightarrow$ $'a$ *starray* $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$ *starray* **is** *array-set-oo* **.**

lift-definition *starray-map* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *starray* $\Rightarrow$ $'b$ *starray* **is** *array-map* **.**

lift-definition *starray-fold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a$ *starray* $\Rightarrow 'b \Rightarrow 'b$ **is** *array-fold* **.**

lift-definition *starray-foldr* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a$ *starray* $\Rightarrow 'b \Rightarrow 'b$ **is** *array-foldr* **.**

definition *starray-*α = *array-*α *o Rep-starray*

2.1.1 Refinement Lemmas

context
notes [*simp*] = *STArray-inverse array-eq-iff starray-*α*-def*
begin

lemma *starray-*α*-inj*: *starray-*α *a* = *starray-*α *b* $\Longrightarrow$ *a*=*b* **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-eq-iff*: *a*=*b* $\longleftrightarrow$ *starray-*α *a* = *starray-*α *b* **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-new-refine*[*simp*,*array-refine*]: *starray-*α (*starray-new* *n a*) = *replicate n a* **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-tabulate-refine*[*simp*,*array-refine*]: *starray-*α (*starray-tabulate* *n f*) = *tabulate n f* **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-length-refine*[*simp*,*array-refine*]: *starray-length a* = *length* (*starray-*α *a*) **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-get-refine*[*simp*,*array-refine*]: *starray-get a i* = *starray-*α *a* ! *i* **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-set-refine*[*simp*,*array-refine*]: *starray-*α (*starray-set a* *i x*) = (*starray-*α *a*)[*i* := *x*] **unfolding** *starray-*α*-def* **by** *transfer auto*

lemma *starray-of-list-refine*[*simp*,*array-refine*]: *starray-*α (*starray-of-list*

xs) = xs **unfolding** *starray-α-def* **by** *transfer auto*

lemma *starray-grow-refine*[*simp*,*array-refine*]:
starray-α (*starray-grow a n d*) = *take n* (*starray-α a*) @ *replicate* (*n*−*length* (*starray-α a*)) *d*
unfolding *starray-α-def* **by** *transfer auto*

lemma *starray-take-refine*[*simp*,*array-refine*]: *starray-α* (*starray-take a n*) = *take n* (*starray-α a*)
unfolding *starray-α-def* **by** *transfer auto*

lemma *starray-get-oo-refine*[*simp*,*array-refine*]: *starray-get-oo x a i* = (*if i*<*length* (*starray-α a*) *then starray-α a*!*i else x*) **unfolding** *starray-α-def* **by** *transfer auto*

lemma *starray-set-oo-refine*[*simp*,*array-refine*]: *starray-α* (*starray-set-oo f a i x*)
= (*if i*<*length* (*starray-α a*) *then* (*starray-α a*)[*i*:=*x*] *else starray-α* (*f* ()))
unfolding *starray-α-def* **by** *transfer auto*

lemma *starray-map-refine*[*simp*,*array-refine*]: *starray-α* (*starray-map f a*) = *map f* (*starray-α a*)
unfolding *starray-α-def* **by** *transfer auto*

lemma *starray-fold-refine*[*simp*, *array-refine*]: *starray-fold f a s* = *fold f* (*starray-α a*) *s*
unfolding *starray-α-def* **by** *transfer auto*

lemma *starray-foldr-refine*[*simp*, *array-refine*]: *starray-foldr f a s* = *foldr f* (*starray-α a*) *s*
unfolding *starray-α-def* **by** *transfer auto*

end

lifting-update *starray.lifting*
lifting-forget *starray.lifting*

2.2 Code Generator Setup

2.2.1 Code-Numeral Preparation

definition [*code del*]: *starray-new′* == *starray-new o nat-of-integer*
definition [*code del*]: *starray-tabulate′ n f* ≡ *starray-tabulate* (*nat-of-integer n*) (*f o integer-of-nat*)

definition [*code del*]: *starray-length′* == *integer-of-nat o starray-length*
definition [*code del*]: *starray-get′ a* == *starray-get a o nat-of-integer*
definition [*code del*]: *starray-set′ a* == *starray-set a o nat-of-integer*

definition [*code del*]:
starray-get-oo′ x a == starray-get-oo x a o nat-of-integer
definition [*code del*]:
starray-set-oo′ f a == starray-set-oo f a o nat-of-integer

lemma [*code*]:
starray-new == starray-new′ o integer-of-nat
starray-tabulate n f == starray-tabulate′ (integer-of-nat n) (f o nat-of-integer)
starray-length == nat-of-integer o starray-length′
starray-get a == starray-get′ a o integer-of-nat
starray-set a == starray-set′ a o integer-of-nat
starray-get-oo x a == starray-get-oo′ x a o integer-of-nat
starray-set-oo g a == starray-set-oo′ g a o integer-of-nat
by (*simp-all*
del: *array-refine*
add: *o-def*
add: *starray-new′-def starray-tabulate′-def starray-length′-def starray-get′-def starray-set′-def*
starray-get-oo′-def starray-set-oo′-def)

Fallbacks

lemmas *starray-get-oo′-fallback*[*code*] = *starray-get-oo′-def*[*unfolded starray-get-oo-def*[*abs-def*]]
lemmas *starray-set-oo′-fallback*[*code*] = *starray-set-oo′-def*[*unfolded starray-set-oo-def*[*abs-def*]]

lemma *starray-tabulate′-fallback*[*code*]:
starray-tabulate′ n f = starray-of-list (map (f o integer-of-nat) [0..<nat-of-integer n])
unfolding *starray-tabulate′-def*
by (*simp add*: *starray-eq-iff tabulate-def*)

lemma *starray-new′-fallback*[*code*]: *starray-new′ n x = starray-of-list (replicate (nat-of-integer n) x)*
by (*simp add*: *starray-new′-def starray-eq-iff*)

code-printing code-module *STArray* ⇀
(*SML*)
‹
structure STArray = struct

```
datatype 'a Cell = Invalid | Value of 'a array;

exception AccessedOldVersion;

type 'a starray = 'a Cell Unsynchronized.ref;

fun fromList l = Unsynchronized.ref (Value (Array.fromList l));
fun starray (size, v) = Unsynchronized.ref (Value (Array.array (size,v)));
fun tabulate (size, f) = Unsynchronized.ref (Value (Array.tabulate(size,
f)));
fun sub (Unsynchronized.ref Invalid, idx) = raise AccessedOldVersion
|
    sub (Unsynchronized.ref (Value a), idx) = Array.sub (a,idx);
fun update (aref,idx,v) =
  case aref of
    (Unsynchronized.ref Invalid) => raise AccessedOldVersion |
    (Unsynchronized.ref (Value a)) => (
      aref := Invalid;
      Array.update (a,idx,v);
      Unsynchronized.ref (Value a)
    );

fun length (Unsynchronized.ref Invalid) = raise AccessedOldVersion |
    length (Unsynchronized.ref (Value a)) = Array.length a


structure IsabelleMapping = struct
type 'a ArrayType = 'a starray;

fun starray-new (n:IntInf.int) (a:'a) = starray (IntInf.toInt n, a);
fun starray-of-list (xs:'a list) = fromList xs;

fun starray-tabulate (n:IntInf.int) (f:IntInf.int -> 'a) = tabulate
(IntInf.toInt n, f o IntInf.fromInt)

fun starray-length (a:'a ArrayType) = IntInf.fromInt (length a);

fun starray-get (a:'a ArrayType) (i:IntInf.int) = sub (a, IntInf.toInt
i);

fun starray-set (a:'a ArrayType) (i:IntInf.int) (e:'a) = update (a,
IntInf.toInt i, e);

fun starray-get-oo (d:'a) (a:'a ArrayType) (i:IntInf.int) =
  sub (a,IntInf.toInt i) handle Subscript => d

fun starray-set-oo (d:(unit->'a ArrayType)) (a:'a ArrayType) (i:IntInf.int)
(e:'a) =
  update (a, IntInf.toInt i, e) handle Subscript => d ()
```

end;

end;
›

code-printing
type-constructor *starray* ⇀ (*SML*) -/ *STArray.IsabelleMapping.ArrayType*
| **constant** *STArray* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-of′-list*
| **constant** *starray-new′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-new*
| **constant** *starray-tabulate′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-tabulate*
| **constant** *starray-length′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-length*
| **constant** *starray-get′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-get*
| **constant** *starray-set′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-set*
| **constant** *starray-of-list* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-of′-list*
| **constant** *starray-get-oo′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-get′-oo*
| **constant** *starray-set-oo′* ⇀ (*SML*) *STArray.IsabelleMapping.starray′-set′-oo*

2.3 Tests

definition *test1* ≡
let a=starray-of-list [*1*,*2*,*3*,*4*,*5*,*6*];
b=starray-tabulate 6 (*Suc*);
a′=starray-set a 3 42;
b′=starray-set b 3 42;
c=starray-new 6 0
in
∀ *i*∈{*0*..<*6*}.
starray-get a′ i = (*if i=3 then 42 else i+1*)
∧ *starray-get b′ i* = (*if i=3 then 42 else i+1*)
∧ *starray-get c i* = (*0*::*nat*)

lemma *enum-rangeE*:
assumes *i*∈{*l*..<*h*}
assumes *P l*
assumes *i*∈{*Suc l*..<*h*} ⟹ *P i*
shows *P i*
using *assms*
by (*metis atLeastLessThan-iff less-eq-Suc-le nat-less-le*)

lemma *test1*
unfolding *test1-def Let-def*
apply (*intro ballI conjI*)
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*

apply (*erule enum-rangeE*, (*simp*; *fail*))+ **apply** *simp*
done

ML-val ‹
if not @{*code test1*} *then error ERROR else* ()
›

export-code *test1* **checking** *OCaml? Haskell? SML*

hide-const *test1*
hide-fact *test1-def*

experiment
begin

fun *allTrue* :: *bool list* ⇒ *nat* ⇒ *bool list* **where**
allTrue a 0 = *a* |
allTrue a (*Suc i*) = (*allTrue a i*)[*i* := *True*]

lemma *length-allTrue*: $n \leq$ *length a* $\Longrightarrow$ *length*(*allTrue a n*) = *length a*
by(*induction n*) (*auto*)

lemma $n \leq$ *length a* $\Longrightarrow \forall\, i < n.$ (*allTrue a n*) ! *i*
by(*induction n*) (*auto simp*: *nth-list-update length-allTrue*)

fun *allTrue'* :: *bool array* ⇒ *nat* ⇒ *bool array* **where**
allTrue' a 0 = *a* |
allTrue' a (*Suc i*) = *array-set* (*allTrue' a i*) *i True*

lemma *array-*α (*allTrue' xs i*) = *allTrue* (*array-*α *xs*) *i*
apply (*induction xs i rule*: *allTrue'.induct*)
apply *auto*
done

end

end
theory *Code-Setup*
imports
HOL−Library.IArray
HOL−Data-Structures.Array-Braun

```
    HOL−Data-Structures.RBT-Map

    ../MDP-fin
    ../Value-Iteration

    ./lib/DiffArray-ST

begin

context MDP-nat-disc begin
lemma L-zero:
  assumes ⋀s. s ≥ states ⟹ apply-bfun v s = 0 s ≥ states
  shows L d v s = 0
  using assms
  proof (induction s rule: less-induct)
    case (less x)
    moreover have r (x, a) = 0 if a ∈ A x for a
      by (simp add: less.prems reward-zero-outside)
    moreover have measure-pmf.expectation (K (x, a)) v = 0 for a
      using K-closed-compl assms less
      by (blast intro: integral-eq-zero-AE AE-pmfI)
    ultimately show ?case
      by (auto simp: A-ne A-fin L-eq-L_a reward-zero-outside)
    qed

lemma ℒ_b-zero:
  assumes ⋀s. s ≥ states ⟹ apply-bfun v s = 0 s ≥ states
  shows ℒ_b v s = 0
  using assms
  proof (induction s rule: less-induct)
    case (less x)
    have r (x, a) = 0 if a ∈ A x for a
      by (simp add: less.prems reward-zero-outside)
    moreover have measure-pmf.expectation (K (x, a)) v = 0 for a
      using K-closed-compl assms less
      by (blast intro: integral-eq-zero-AE AE-pmfI)
      ultimately show ?case
        by (auto simp: A-ne A-fin ℒ_b-eq-L_a-max')
    qed
end

lemma max-geI: finite A ⟹ A ≠ {} ⟹ (∃ a∈A. x ≤ a) ⟹ (x ≤
Max A) for x A
  by (simp add: Max-ge-iff)
```

3 Least argmax

```
fun least-arg-max-max-ne where
  least-arg-max-max-ne f (x#xs) =
```

(*fold* (λ*y* (*am*, *m*). *let fy* = *f y in*
if m < *fy then* (*y*, *fy*) *else* (*am*, *m*)) *xs* (*x*, *f x*)) |
least-arg-max-max-ne a [] = *undefined*

fun *least-arg-max-ne* **where**
least-arg-max-ne f (*x*#*xs*) = *fst* (*least-arg-max-max-ne f* (*x*#*xs*)) |
least-arg-max-ne a [] = *undefined*

lemmas
least-arg-max-ne.simps[*simp del*]
least-arg-max-max-ne.simps[*simp del*]

lemma *least-arg-max-max-ne-Cons*: *least-arg-max-max-ne f* (*x*#*y*#*xs*)
=
(*if f x* < *f y then least-arg-max-max-ne f* (*y*#*xs*) *else least-arg-max-max-ne f* (*x*#*xs*))
by (*auto simp*: *least-arg-max-max-ne.simps*)

lemma *least-arg-max-max-ne-Cons1*: *f x* < *f y* ⟹ *least-arg-max-max-ne f* (*x*#*y*#*xs*) = *least-arg-max-max-ne f* (*y*#*xs*)
by (*auto simp*: *least-arg-max-max-ne.simps*)

lemma *least-arg-max-max-ne-Cons2*: ¬ *f x* < *f y* ⟹ *least-arg-max-max-ne f* (*x*#*y*#*xs*) = *least-arg-max-max-ne f* (*x*#*xs*)
by (*auto simp*: *least-arg-max-max-ne.simps*)

lemma *Max-insert-absorb*: *finite X* ⟹ (∃ *y* ∈ *X*. *x* ≤ *y*) ⟹ *Max* (*Set.insert x X*) = (*if X* = {} *then x else Max X*)
by (*simp add*: *Max-ge-iff*)

lemma *Max-insert-absorb′*: *finite X* ⟹ *y*∈*X* ⟹ *x* ≤ *y* ⟹ *Max* (*Set.insert x X*) = (*if X* = {} *then x else Max X*)
using *Max-insert-absorb*
by *blast*

lemma *fold-max-eq-arg-max*:
assumes *sorted* (*x*#*xs*)
shows *least-arg-max-max-ne f* (*x*#*xs*) = (*least-arg-max f* (*List.member* (*x*#*xs*)), *Max* (*f* ‘ *set* (*x*#*xs*)))
using *assms*
proof (*induction xs arbitrary*: *x*)
case *Nil*
then show *?case*
by (*auto simp*: *List.member-def least-arg-max-def least-arg-max-max-ne.simps is-arg-max-def intro*!: *Least-equality*[*symmetric*])
next
case (*Cons a xs*)
then show *?case*
proof (*cases is-arg-max f* (*List.member* (*x*#*a*#*xs*)) *x*)

case *True*
have *1*: *least-arg-max f* (*List.member* (*x#a#xs*)) = *x*
using *True Cons*
unfolding *least-arg-max-def*
by (*fastforce intro*!: *Least-equality simp*: *in-set-member*[*symmetric*])
have *2*: *Max* (*f* ' *set* (*x#a#xs*)) = *f x*
using *True* **unfolding** *is-arg-max-def*
by (*subst Max-eq-iff*) (*auto simp add*: *not-less in-set-member member-rec*(*1*))
show *?thesis*
unfolding *1 2*
using *True*
by (*induction xs*) (*auto simp*: *least-arg-max-max-ne.simps simp*: *is-arg-max-linorder member-rec*)+
next
case *False*
have *is-arg-max f* (*List.member* (*x#a#xs*)) = *is-arg-max f* (*List.member* (*a#xs*))
using *False* **by** (*fastforce simp*: *least-arg-max-max-ne.simps is-arg-max-linorder member-rec*)
hence *1*: *least-arg-max f* (*List.member* (*x#a#xs*)) = *least-arg-max f* (*List.member* (*a#xs*))
using *Cons False* **unfolding** *least-arg-max-def* **by** *auto*
have $f\ a \le f\ x \Longrightarrow$ *is-arg-max f* (*List.member* (*x#xs*)) = *is-arg-max f* (*List.member xs*)
using *False* **by** (*fastforce simp*: *is-arg-max-linorder member-rec*)

hence *4*: $f\ a \le f\ x \Longrightarrow$ *least-arg-max f* (*List.member* (*x#xs*)) = *least-arg-max f* (*List.member xs*)
using *Cons False* **unfolding** *least-arg-max-def* **by** *auto*
have $f\ a \le f\ x \Longrightarrow$ *is-arg-max f* (*List.member* (*a#xs*)) = *is-arg-max f* (*List.member xs*)
using *False* **by** (*fastforce simp*: *is-arg-max-linorder member-rec*(*1*))
hence *3*: $f\ a \le f\ x \Longrightarrow$ *least-arg-max f* (*List.member* (*a#xs*)) = *least-arg-max f* (*List.member xs*)
using *Cons False* **unfolding** *least-arg-max-def* **by** *auto*
have *2*: *Max* (*f*'*set* (*x#a#xs*)) = *Max* (*f*'*set* (*a#xs*))
using *False*
by (*fastforce simp*: *nle-le in-set-member is-arg-max-linorder Max-ge-iff simp*: *member-rec intro*!: *max-absorb2*)
have *5*: *Max* (*f*'*set* (*a#xs*)) = *Max* (*f*'*set* (*xs*)) ∧ *Max* (*f*'*set* (*x#xs*)) = *Max* (*f*'*set* (*xs*)) **if** $f\ a \le f\ x$
using *False that*
by (*cases xs* = []) (*auto simp*: *nle-le is-arg-max-linorder in-set-member*[*symmetric*] *intro*: *order.trans intro*!:*max-absorb2*)
show *?thesis*
unfolding *least-arg-max-max-ne-Cons 1 2* **using** *Cons 5 3 4* **by** *auto*

qed
qed

lemma *least-arg-max-ne-correct*:
assumes *sorted* $(x\#xs)$
shows *least-arg-max-ne* $(f :: - \Rightarrow 'b ::linorder)$ $(x\#xs)$ = *least-arg-max* *f* (*List.member* $(x\#xs)$)
using *assms*
by (*auto simp*: *fold-max-eq-arg-max least-arg-max-ne.simps*)

lemma *least-arg-max-ne-cong*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow g\ x = f\ x$
shows *least-arg-max-max-ne f xs* = *least-arg-max-max-ne g xs*
proof (*cases xs*)
case *Nil*
then show *?thesis*
by (*metis least-arg-max-max-ne.elims list.discI*)
next
case (*Cons a list*)
then show *?thesis*
using *assms*
by (*auto simp*: *least-arg-max-max-ne.simps intro*!: *List.fold-cong*)
qed

lemma *least-arg-max-max-ne-app*:
assumes $\bigwedge y.\ y \in set\ (x\#xs) \Longrightarrow f'\ (g\ y) = (f\ y)$
shows (*case* (*least-arg-max-max-ne f* $(x\#xs)$) *of* $(a, m) \Rightarrow (g\ a, m)$) = *least-arg-max-max-ne* f' (*map g* $(x\#xs)$))
using *assms*
proof (*induction xs arbitrary*: *x*)
case *Nil*
then show *?case*
by (*auto simp*: *least-arg-max-max-ne.simps*)
next
case (*Cons a xs*)
thus *?case*
by (*cases* $f\ x <\ f\ a$) (*auto simp*: *least-arg-max-max-ne-Cons1 least-arg-max-max-ne-Cons2*)
qed

lemma *least-arg-max-max-ne-app'*:
assumes $\bigwedge y.\ y \in set\ xs \Longrightarrow f'\ (g\ y) = (f\ y)$ $xs \neq []$
shows (*case* (*least-arg-max-max-ne f xs*) *of* $(a, m) \Rightarrow (g\ a, m)$) = *least-arg-max-max-ne* f' (*map g xs*)
using *assms*
by (*cases xs*) (*auto intro*!: *least-arg-max-max-ne-app*[*simplified*])

lemma *fold-max-eq-arg-max'*: $xs \neq [] \Longrightarrow$ *sorted xs* $\Longrightarrow$ *least-arg-max-max-ne f xs* = (*least-arg-max f* (*List.member xs*), *Max* (*f* ‘ *set xs*))

using *fold-max-eq-arg-max* **by** (*metis list.exhaust*)

lemma *least-arg-max-cong*: ($\bigwedge x.\ P\ x \Longrightarrow f\ x = g\ x$) $\Longrightarrow$ *least-arg-max* *f P* = *least-arg-max g P*
unfolding *least-arg-max-def* **using** *is-arg-max-cong′* **by** *metis*

lemma *least-arg-max-cong′*: $P = Q \Longrightarrow$ ($\bigwedge x.\ P\ x \Longrightarrow f\ x = g\ x$) $\Longrightarrow$ *least-arg-max f P* = *least-arg-max g Q*
unfolding *least-arg-max-def* **using** *is-arg-max-cong′* **by** *metis*

4 Congruence rule for fold

lemma *fold-cong′*:
assumes ($\bigwedge x\ acc.\ P\ acc \Longrightarrow x \in set\ xs \Longrightarrow f\ x\ acc = g\ x\ acc \wedge P\ (f\ x\ acc)$) *P a*
shows *fold f xs a* = *fold g xs a*
using *assms*
proof (*induction xs arbitrary*: *a*)
case (*Cons a xs y*)
show *?case*
using *Cons*(*2*)[*OF Cons*(*3*), *of a*]
by (*auto intro*!: *Cons*(*2*) *intro*: *Cons.IH*)
qed *auto*

5 MDP type

datatype *MDP* = *MDP* (*disc*: *real*) (*states*: *nat*)
(*transitions*: (((*nat* × (*real* × ((*nat* × *real*) *list*))) *RBT.rbt*)) *iarray*)

abbreviation *is-MDP-states mdp* ≡
IArray.length (*transitions mdp*) = *states mdp*

abbreviation *is-MDP-actions mdp* ≡ *IArray.all* (λ*t*.
rbt t ∧
sorted1 (*Tree2.inorder t*) ∧
t ≠ *empty* ∧
(∀ (-, -, *probs*) ∈ *set* (*inorder t*). *sum-list* (*map snd probs*) = *1*
∧ (*list-all* (λ(*s*, *p*). *p* ≥ *0* ∧ *s*<*states mdp*) *probs*))) (*transitions mdp*)

abbreviation *is-MDP-disc mdp* ≡ (*0* ≤ *disc mdp* ∧ *disc mdp* < *1*)

definition *is-MDP* :: *MDP* ⇒ *bool*
where *is-MDP mdp* ⟷ *is-MDP-states mdp* ∧ *is-MDP-disc mdp* ∧ *is-MDP-actions mdp*

definition *trivial-MDP* = *MDP 0 0* (*IArray* [])

lemma *trivial-MDP*: *is-MDP trivial-MDP*
  unfolding *trivial-MDP-def is-MDP-def* **by** *auto*

typedef *Valid-MDP* = {*mdp. is-MDP mdp*}
  using *trivial-MDP* **by** *auto*

setup-lifting *type-definition-Valid-MDP*

definition *error-mdp* = *trivial-MDP*

declare [[*code abort*: *error-mdp*]]

lift-definition *to-valid-MDP* :: *MDP* ⇒ *Valid-MDP* **is**
λ*mdp. if is-MDP mdp then mdp else Code.abort* (*STR ''not an MDP''*)
(λ-. *trivial-MDP*)
  by (*simp add*: *trivial-MDP-def is-MDP-def*)

context *Map-by-Ordered* **begin**
lemmas *map-specs*(*5*)[*intro*]

lemma *map-of-Some-in-set*: *AList-Upd-Del.map-of xs k* = *Some v* ⟹
(*k*, *v*) ∈ *set xs*
  by (*induction xs*) (*auto split*: *if-splits*)

lemma *map-of-None-notin-set*: *AList-Upd-Del.map-of xs k* = *None*
⟹ *k* ∉ *fst ‘ set xs*
  by (*induction xs*) (*fastforce split*: *if-splits*)+

definition *entries m* = *set* (*inorder m*)
definition *keys m* = *fst ‘ set* (*inorder m*)

lemma *lookup-some-set-a-inorder*:
  assumes *invar m lookup m x* = *Some y*
  shows (*x*, *y*) ∈ *entries m*
  using *inorder-lookup assms map-of-Some-in-set invar-def entries-def*
by *metis*

lemma *lookup-None-set-inorder*:
  assumes *invar m lookup m x* = *None*
  shows *x* ∉ *keys m*
  using *assms inorder-lookup map-of-None-notin-set keys-def invar-def*
by *metis*

lemma *entries-imp-keys*[*intro*]: (*x*,*y*) ∈ *entries m* ⟹ *x* ∈ *keys m*
  unfolding *keys-def entries-def* **by** *force*

lemma *lookup-some-set-key*: *invar m* ⟹ *lookup m x* = *Some y* ⟹
x ∈ *keys m*
  using *lookup-some-set-a-inorder* **by** *force*

lemma *lookup-in-keys*: *invar m* $\Longrightarrow$ *x* $\in$ *keys m* $\Longrightarrow$ $\exists$ *y*. *lookup m x* = *Some y*
  using *lookup-None-set-inorder* **by** *auto*

lemma *lookup-notin-keys*: *invar m* $\Longrightarrow$ *x* $\notin$ *keys m* $\Longrightarrow$ *lookup m x* = *None*
  by (*meson lookup-some-set-key not-Some-eq*)

lemma *inorder-delete*: *invar m* $\Longrightarrow$ *inorder m* = *kv#xs* $\Longrightarrow$ *inorder* ((*delete* (*fst kv*) *m*)) = *xs*
  unfolding *invar-def*
  using *AList-Upd-Del.del-list.simps*(*2*)[*of - fst kv snd kv*]
  by (*simp add*: *local.inorder-delete*)

lemma *inorder-lookup-Some*: *invar m* $\Longrightarrow$ (*k*, *v*) $\in$ *entries m* $\Longrightarrow$ *lookup m k* = *Some v*
  unfolding *entries-def*
proof (*induction inorder m arbitrary*: *m*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons a x*)
  show *?case*
  proof (*cases a* = (*k*,*v*))
    case *True*
    then show *?thesis*
      using *inorder-lookup Cons AList-Upd-Del.map-of.simps*(*2*) *invar-def* **by** *metis*
  next
    case *False*
    have *lookup* (*delete* (*fst a*) *m*) *k* = *Some v*
      using *False Cons*(*2*)[*symmetric*] *Cons*(*3*−*4*)
      by (*fastforce simp*: *inorder-delete map-specs intro*!: *Cons*(*1*))
    then show *?thesis*
       by (*metis map-delete fun-upd-other fun-upd-same Cons*(*3*) *option.distinct*(*1*))
  qed
qed

lemma *keys-eq-lookup-Some*: *invar m* $\Longrightarrow$ *keys m* = {*k*. $\exists$ *v*. *lookup m k* = *Some v*}
  using *lookup-some-set-key lookup-in-keys* **by** *auto*

lemma *keys-eq-fst-entries*: *invar m* $\Longrightarrow$ *keys m* = *fst* ‘ *entries m*
  unfolding *entries-def keys-def* **by** *auto*

lemma *keys-update*[*simp*]: *invar m* $\Longrightarrow$ *keys* (*update k v m*) = *Set.insert k* (*keys m*)
   by (*subst keys-eq-lookup-Some*) (*auto simp add*: *lookup-notin-keys*

lookup-in-keys map-specs split: *if-splits*)

definition *is-empty* $t \longleftrightarrow$ *inorder* $t = []$

lemma *is-empty-iff-entries-empty*: *is-empty* $t \longleftrightarrow$ *entries* $t = \{\}$
  by (*simp add*: *entries-def is-empty-def*)

lemma *is-empty-iff-keys-empty*: *is-empty* $t \longleftrightarrow$ *keys* $t = \{\}$
  by (*simp add*: *keys-def is-empty-def*)

lemma *finite-keys*: *finite* (*keys t*)
  by (*simp add*: *keys-def*)

lemma *finite-entries*: *finite* (*entries t*)
  by (*simp add*: *entries-def*)

lemma *keys-empty*[*simp*]: *keys empty* = {}
  by (*auto simp*: *keys-def inorder-empty*)

definition *lookup′ m k* = *the* (*lookup m k*)

6 Converting Lists to Maps

definition *from-list′ f xs* = *foldl* (λ*acc s. update s* (*f s*) *acc*) *empty xs*
definition *from-list xs* = *foldl* (λ*acc* (*k*,*v*). *update k v acc*) *empty xs*

lemmas *invar-empty*[*simp*, *intro*]

lemma *from-list-invar*[*simp*]: *invar* (*from-list′ f xs*)
proof −
  have *invar t* $\Longrightarrow$ *invar* (*foldl* (λ*acc s. update s* (*f s*) *acc*) *t xs*) **for** *t*
    by (*induction xs arbitrary*: *t*) *auto*
  thus *?thesis*
    unfolding *from-list′-def* **by** *auto*
qed

lemma *from-list-snoc*[*simp*]: (*from-list′ f* (*xs* @[*y*])) = *update y* (*f y*) (*from-list′ f xs*)
  by (*auto simp*: *from-list′-def*)

lemma *from-list-empty*[*simp*]: *from-list′ f* [] = *empty*
  unfolding *from-list′-def* **by** *simp*

lemma *from-list-keys*[*simp*]: *keys* (*from-list′ f xs*) = *set xs*
  by (*induction xs rule*: *List.rev-induct*) (*auto simp*: *map-update*)

lemma *from-list-lookup*[*simp*]: $x \in$ *set xs* $\Longrightarrow$ *lookup* (*from-list′ f xs*) x = *Some* (*f x*)

by (*induction xs rule*: *List.rev-induct*) (*auto simp*: *map-update*)

lemma *from-list-lookup′*[*simp*]: $x \in$ *set xs* $\Longrightarrow$ *lookup′* (*from-list′ f xs*) $x = f\ x$
unfolding *lookup′-def*
using *from-list-lookup*
by *auto*

lemma *from-list-snoc′*[*simp*]: (*from-list* (*xs* @[(*k*,*v*)])) = *update k v* (*from-list xs*)
by (*auto simp*: *from-list-def*)

lemma *from-list-invar′*[*simp*]: *invar* (*from-list xs*)
proof −
have *invar t* $\Longrightarrow$ *invar* (*foldl* (λ*acc* (*k*,*v*). *update k v acc*) *t xs*) **for** *t*
by (*induction xs arbitrary*: *t*) *auto*
thus *?thesis*
unfolding *from-list-def* **by** *auto*
qed

lemma *lookup-from-list-distinct*: (*x*,*y*) $\in$ *set xs* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$ *lookup* (*from-list xs*) *x* = *Some y*
by (*induction xs arbitrary*: *x y rule*: *List.rev-induct*) (*auto simp*: *rev-image-eqI map-update*)

lemma *lookup′-from-list-distinct*: (*x*,*y*) $\in$ *set xs* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$ *lookup′* (*from-list xs*) *x* = *y*
using *lookup-from-list-distinct* **unfolding** *lookup′-def*
by *auto*

lemma *distinct-inorder*: *invar m* $\Longrightarrow$ *distinct* (*map fst* (*inorder m*))
using *invar-def strict-sorted-iff* **by** *blast*

lemmas *map-empty*[*simp*]

lemma *from-list-lookup-notin*[*simp*]: $x \notin$ *set xs* $\Longrightarrow$ *lookup* (*from-list′ f xs*) *x* = *None*
by (*induction xs rule*: *List.rev-induct*) (*auto simp*: *map-update*)
end

locale *Map-by-Ordered-nat-zero* = *Map-by-Ordered empty update delete lookup inorder inv′* **for** *empty* **and** *update* :: *nat* ⇒ (′*a*::*zero*) ⇒ ′*t* ⇒ ′*t* **and** *delete lookup inorder inv′*
begin

definition *map-to-fun* :: ′*t* ⇒ *nat* ⇒ ′*a* **where**
map-to-fun m n = (*if invar m then case lookup m n of None* ⇒ *0* | *Some r* ⇒ *r else 0*)

lemma *map-to-fun-update*: *invar m* $\Longrightarrow$ (*map-to-fun* (*update k v m*)) = (*map-to-fun m*)(*k* := *v*)
  by (*fastforce simp*: *map-to-fun-def map-update*)
end

locale *Map-by-Ordered-nat-real* = *Map-by-Ordered empty update delete lookup inorder inv'* **for** *empty* **and** *update* :: *nat* $\Rightarrow$ *real* $\Rightarrow$ *'t* $\Rightarrow$ *'t* **and** *delete lookup inorder inv'*
begin

lift-definition *map-to-bfun* :: *'t* $\Rightarrow$ *nat* $\Rightarrow_b$ *real* **is**
  λ*m n. if invar m then case lookup m n of None* $\Rightarrow$ *0* | *Some r* $\Rightarrow$ *r else 0*
proof −
  fix *t*
  show (λ*n. if invar t then case lookup t n of None* $\Rightarrow$ *0* | *Some r* $\Rightarrow$ *r else 0*) $\in$ *bfun*
  proof (*cases is-empty t* $\vee$ $\neg$ *invar t*)
    case *True*
    then show *?thesis*
      by (*auto simp add*: *is-empty-iff-keys-empty lookup-notin-keys*)
  next
    case *False*
    have *norm* (*case lookup t x of None* $\Rightarrow$ *0* | *Some r* $\Rightarrow$ *r*) $\leq$ (*MAX a* $\in$ *entries t. abs* (*snd a*)) **for** *x*
      using *False is-empty-iff-entries-empty lookup-some-set-a-inorder*[*of t x*]
        by (*fastforce simp*: *Max-ge-iff finite-entries split*: *option.splits*)
      thus *?thesis*
        using *False* **by** (*intro bfun-normI*) *auto*
   qed
 qed

lemma *map-to-bfun-update*: *invar m* $\Longrightarrow$ *apply-bfun* (*map-to-bfun* (*update k v m*)) = (*map-to-bfun m*)(*k* := *v*)
  by (*fastforce simp*: *map-to-bfun.rep-eq map-update*)

end

locale *Array'* = *Array* +
  assumes *lookup-array*: *i* < *length xs* $\Longrightarrow$ *lookup* (*array xs*) *i* = *xs* ! *i*

locale *Array-real* = *Array' lookup update len array list invar* **for** *lookup* :: *'t* $\Rightarrow$ *nat* $\Rightarrow$ *real* **and** *update len array list invar*
begin

lift-definition *map-to-bfun* :: *'t* $\Rightarrow$ *nat* $\Rightarrow_b$ *real* **is**
  λ*m n. if invar m* $\wedge$ *n* < *len m then lookup m n else 0*
  using *bounded-const* **by** *fastforce*

lemma *map-to-bfun-update*:
assumes *invar m k* < *len m*
shows *apply-bfun* (*map-to-bfun* (*update k v m*)) = (*map-to-bfun m*)(*k* := *v*)
using *assms*
by (*auto simp*: *invar-update map-to-bfun.rep-eq len-array lookup update*)
end

locale *Array-zero* = *Array′ lookup update len array list invar* **for** *lookup* :: ′*t* ⇒ *nat* ⇒ ′*a*::*zero* **and** *update len array list invar*
begin

definition *map-to-fun* :: ′*t* ⇒ *nat* ⇒ ′*a* **where**
map-to-fun m n = (*if invar m* ∧ *n* < *len m then lookup m n else 0*)

lemma *map-to-fun-update*: *invar m* ⟹ *k* < *len m* ⟹ (*map-to-fun* (*update k v m*)) = (*map-to-fun m*)(*k* := *v*)
by (*auto simp*: *invar-update map-to-fun-def len-array lookup update*)

end

context *Array′* **begin**
lemma *lookup-in-list*: *invar m* ⟹ *x* < *len m* ⟹ *lookup m x* ∈ *set* (*list m*)
using *lookup len-array*
by *auto*

definition *arr-tabulate f n* = *array* (*map f* [*0*..<*n*])

lemma *invar-tabulate*[*simp*]: *invar* (*arr-tabulate f n*)
by (*auto simp*: *arr-tabulate-def invar-array*)

lemma *len-tabulate*[*simp*]: *len* (*arr-tabulate f n*) = *n*
using *arr-tabulate-def array invar-tabulate len-array* **by** *auto*

lemma *lookup-tabulate*[*simp*]: *i* < *n* ⟹ *lookup* (*arr-tabulate f n*) *i* = *f i*
by (*simp add*: *arr-tabulate-def lookup-array*)

lemmas *invar-update*[*intro*]
end

lemma *foldr-Cons*[*simp*]: *foldr* (#) *xs ys* = *xs*@*ys*
by (*induction xs*) *auto*

interpretation *starray-Array*:

Array′ starray-get λi x arr. starray-set arr i x starray-length starray-of-list
λarr. starray-foldr (λx xs. x # xs) arr [] λ-. True
by *standard auto*

definition *starray-to-list a* = *tabulate* (*starray-length a*) (*starray-get a*)

lemma *set-pmf-of-list*:
assumes *pmf-of-list-wf ps*
shows *set-pmf* (*pmf-of-list ps*) = {*a* | *a b*. (*a*,*b*) ∈ *set ps* ∧ *b* ≠ *0*}
proof *safe*
fix *x*
assume *x* ∈ *set-pmf* (*pmf-of-list ps*)
hence *sum-list* (*map snd* (*filter* (*λz*. *fst z* = *x*) *ps*)) ≠ *0*
using *assms*
by (*auto simp*: *set-pmf-eq pmf-pmf-of-list*)
hence ∃ *y* ∈ *set* (*map snd* (*filter* (*λz*. *fst z* = *x*) *ps*)). *y* ≠ *0*
by (*metis map-idI sum-list-0*)
then obtain *sp* **where** *snd sp* ≠ *0 fst sp* = *x sp* ∈ *set ps*
by *auto*
thus ∃ *a b*. *x* = *a* ∧ (*a*, *b*) ∈ *set ps* ∧ *b* ≠ *0*
by *force*
next
fix *x a b*
assume *h*: (*a*, *b*) ∈ *set ps b* ≠ *0*
have $\sum$ (*Set.insert a X*) = *a* + $\sum$ (*X*−{*a*}) **if** *finite X* **for** *X* **and** *a* :: *real*
using *that*
by (*meson sum.insert-remove*)
hence ∗: ∀ *b* ∈ *set ps*. *b* ≥ *0* ⟹ *b* ∈ *set ps* ⟹ *b* ≤ *sum-list ps* **for** *ps*
by (*induction ps*) (*auto intro*!: *sum-list-nonneg*)
have *pmf* (*pmf-of-list ps*) *a* ≥ *b*
using *assms* ‹(*a*, *b*) ∈ *set ps*›
by (*fastforce simp add*: *image-iff pmf-pmf-of-list pmf-of-list-wf-def intro*!: ∗)
thus *a* ∈ *set-pmf* (*pmf-of-list ps*)
unfolding *set-pmf-iff*
using *h assms pmf-of-list-wf-def* **by** *fastforce*
qed

lemma *set-pmf-of-list′*:
assumes *pmf-of-list-wf ps*
shows *set-pmf* (*pmf-of-list ps*) = {*a* | *a b*. (*a*,*b*) ∈ *set ps* ∧ *b* > *0*}
unfolding *set-pmf-of-list*[*OF assms*]
using *assms* **unfolding** *pmf-of-list-wf-def*
by *fastforce*

locale *MDP-Code-raw* =
S-Map : *Array′ s-lookup* :: *′ts* ⇒ *nat* ⇒ *′ta* *s-update s-len s-array s-list s-invar* +
A-Map : *Map-by-Ordered a-empty a-update* :: *nat* ⇒ (*real* × ((*nat* × *real*) *list*)) ⇒ *′ta* ⇒ *′ta a-delete a-lookup a-inorder a-inv*
for *s-lookup s-update s-len s-array s-list s-invar*
and *a-empty a-update a-delete a-lookup a-inorder a-inv* +
fixes
mdp :: *′ts* **and**
states :: *nat*
assumes
s-invar: *s-invar mdp* **and**
s-len: *s-len mdp* = *states* **and**
A-inv-locale: ∀ *am* ∈ *set* (*s-list mdp*). *A-Map.invar am* **and**
A-ne-locale: ∀ *am* ∈ *set* (*s-list mdp*). ¬ *A-Map.is-empty am* **and**
K-closed-locale:
∀ *am* ∈ *set* (*s-list mdp*). ∀ (-, -, *p*) ∈ *A-Map.entries am*.
list-all (λ(*s′*, *p*). *s′* <*states*) *p* **and**
lists-are-pmfs: ∀ *am* ∈ *set* (*s-list mdp*). ∀ (-, -, *p*) ∈ *A-Map.entries am*. *pmf-of-list-wf p*
begin

definition *a-lookup′ m x* = (
case (*a-lookup m x*) *of*
Some v ⇒ *v*
| *None* ⇒ *Code.abort* (*STR ′′MDP is missing action information′′*) (λ-. *undefined*))

definition *MDP-A s* = (*if s* < *states then A-Map.keys* (*s-lookup mdp s*) *else* {*0*})

definition *MDP-r sa* = (*if fst sa* ≥ *states then 0 else*
let a-map = *s-lookup mdp* (*fst sa*) *in*
(*case a-lookup a-map* (*snd sa*) *of Some* (*r*, -) ⇒ *r* | *None* ⇒ *0*)
)

definition *MDP-K sa* = (
if fst sa ≥ *states then*
return-pmf (*fst sa*)
else
let a-map = *s-lookup mdp* (*fst sa*) *in* (
case a-lookup a-map (*snd sa*) *of*
Some (-, *p*) ⇒ *pmf-of-list p*
| *None* ⇒ *return-pmf* (*fst sa*))
)

lemma *MDP-r-zero-notin-states*: *s* ≥ *states* ⟹ *MDP-r* (*s*, *a*) = *0*
for *s a*

unfolding *MDP-r-def*
by *auto*

lemma *a-lookup-some-in-A*: $s < states \Longrightarrow a\text{-}lookup\ (s\text{-}lookup\ mdp\ s)\ a = Some\ (aa,\ b) \Longrightarrow a \in MDP\text{-}A\ s$
using *A-Map.lookup-some-set-key A-inv-locale S-Map.lookup-in-list s-len s-invar*
by (*simp add*: *A-Map.keys-def MDP-A-def*)

lemma *a-lookup-None-notin-A*: $s < states \Longrightarrow a\text{-}lookup\ (s\text{-}lookup\ mdp\ s)\ a = None \Longrightarrow a \notin MDP\text{-}A\ s$
unfolding *MDP-A-def*
using *A-Map.lookup-None-set-inorder A-inv-locale S-Map.lookup-in-list s-invar s-len*
by *auto*

lemma *MDP-r-zero-notin-A*: $s < states \Longrightarrow a \notin MDP\text{-}A\ s \Longrightarrow MDP\text{-}r\ (s,\ a) = 0$ **for** *s a*
using *a-lookup-some-in-A*
by (*auto split*: *option.splits simp*: *MDP-r-def*)

lemma *MDP-r-in-A-eq*: $s < states \Longrightarrow a \in MDP\text{-}A\ s \Longrightarrow MDP\text{-}r\ (s,\ a) = fst\ ((a\text{-}lookup'\ (s\text{-}lookup\ mdp\ s)\ a))$
using *a-lookup-None-notin-A* **by** (*auto split*: *option.splits simp*: *a-lookup′-def MDP-r-def*)

lemma *range-MDP-r-subs*: $range\ (MDP\text{-}r) \subseteq \{0\} \cup \{fst\ ((a\text{-}lookup'\ (s\text{-}lookup\ mdp\ s)\ a)) \mid s\ a.\ s < states \wedge a \in MDP\text{-}A\ s\}$
using *MDP-r-in-A-eq MDP-r-zero-notin-A MDP-r-zero-notin-states*

by (*auto*) (*metis not-le*)

lemma *finite-MDP-A*[*simp*]: *finite* (*MDP-A s*)
unfolding *MDP-A-def*
by (*simp add*: *A-Map.finite-keys*)

lemma *finite-sa*: $finite\ \{(s,a).\ s < states \wedge a \in MDP\text{-}A\ s\}$
proof −
have $\{(s,a).\ s < states \wedge a \in MDP\text{-}A\ s\} \subseteq \{(s,a).\ s < states \wedge a \in (\bigcup s < states.\ MDP\text{-}A\ s)\}$
by *auto*
moreover have $finite\ \{(s,a).\ s < states \wedge a \in (\bigcup s < states.\ MDP\text{-}A\ s)\}$
by *auto*
ultimately show *?thesis*
using *finite-subset* **by** *blast*
qed

lemma *finite-r-lookup*: *finite* {*fst* ((*a-lookup′* (*s-lookup mdp s*) *a*)) | *s a*. *s* < *states* ∧ *a* ∈ *MDP-A s*}
proof –
have *aux*: {*fst* ((*a-lookup′* (*s-lookup mdp s*) *a*)) | *s a*. *s* < *states* ∧ *a* ∈ *MDP-A s*} = {*fst* ((*a-lookup′* (*s-lookup mdp* (*fst sa*)) (*snd sa*))) | *sa*. *fst sa* < *states* ∧ *snd sa* ∈ *MDP-A* (*fst sa*)}
by *auto*
show *?thesis*
unfolding *aux*
using *finite-sa*
by (*fastforce intro*!: *finite-image-set simp*: *case-prod-unfold*)
qed

lemma *bounded-MDP-r*: *bounded* (*range MDP-r*)
using *finite-r-lookup range-MDP-r-subs*
by (*simp add*: *finite-imp-bounded finite-subset*)

lemma *MDP-A-ne*[*simp*]: (*MDP-A s*) ≠ {}
using *A-ne-locale s-invar s-len*
by (*auto simp*: *MDP-A-def A-Map.is-empty-iff-keys-empty S-Map.lookup-in-list*)

lemma *K-closed-locale′*:
am ∈ *set* (*s-list mdp*) ⟹ (*x*, *y*, *p*) ∈ *A-Map.entries am* ⟹ (*s′*, *prob*) ∈ *set p* ⟹ *s′* <*states*
using *K-closed-locale*
by (*fastforce simp*: *list.pred-set case-prod-beta*)

lemma *MDP-K-closed*:
assumes *s*<*states*
shows *set-pmf* (*MDP-K* (*s*, *a*)) ⊆ {*0*..<*states*}
proof
fix *s′*
assume *h*: *s′* ∈ *set-pmf* (*MDP-K* (*s*, *a*))
show *s′* ∈ {*0*..<*states*}
proof (*cases a* ∈ *MDP-A s*)
case *False*
thus *?thesis*
using *assms h*
using *a-lookup-some-in-A*
by (*auto simp*: *MDP-K-def split*: *option.splits*)
next
case *True*
from *h* **obtain** *r ps* **where** *a-lookup* (*s-lookup mdp s*) *a* = *Some* (*r*, *ps*) **and** ∗∗:*s′* ∈ *set-pmf* (*pmf-of-list ps*)
unfolding *MDP-K-def* **using** *assms True a-lookup-None-notin-A*
by (*auto split*: *option.splits*)
have *pmf-of-list-wf ps*
using *lists-are-pmfs*
by (*metis A-Map.Map-by-Ordered-axioms A-inv-locale Map-by-Ordered.lookup-some-set-a-inorder*

S-Map.lookup-in-list ‹*a-lookup* (*s-lookup mdp s*) *a* = *Some* (*r*, *ps*)› *assms case-prod-conv s-invar s-len*)
  have $***$:(s'', *p*) ∈ *set ps* ⟹ *p* > *0* ⟹ s'' < *states* **for** s'' *p*
  by (*metis A-Map.Map-by-Ordered-axioms A-inv-locale K-closed-locale′ Map-by-Ordered.lookup-some-set-a-inorder S-Map.lookup-in-list* ‹*a-lookup* (*s-lookup mdp s*) *a* = *Some* (*r*, *ps*)› *assms s-invar s-len*)
  have s' < *states*
    using $***$ $**$ *set-pmf-of-list′*[*OF* ‹*pmf-of-list-wf ps*›]
    by *blast*
  then show *?thesis* **by** *auto*
 qed
qed

lemma *MDP-K-comp-closed*: *s* ≥ *states* ⟹ *set-pmf* (*MDP-K* (*s*, *a*)) ⊆ {*states..*}
 unfolding *MDP-K-def*
 by *auto*

lemma *MDP-A-outside*: *states* ≤ *s* ⟹ *MDP-A s* = {*0*}
 unfolding *MDP-A-def*
 by *auto*

lemma *invar-s-lookup*: *s* < *states* ⟹ *A-Map.invar* (*s-lookup mdp s*)
 by (*simp add*: *A-inv-locale S-Map.lookup-in-list s-invar s-len*)

lemma *ne-s-lookup*: *s* < *states* ⟹ ¬ *A-Map.is-empty* (*s-lookup mdp s*)
 using *A-ne-locale S-Map.lookup-in-list s-invar s-len* **by** *blast*

lemma *sa-lookup-eq*:
 assumes *s* < *states a* ∈ *MDP-A s* (*a-lookup* (*s-lookup mdp s*) *a*) = *Some* (*r*, *ps*)
 shows *r* = *MDP-r* (*s*,*a*) *pmf-of-list ps* = *MDP-K* (*s*, *a*)
 unfolding *MDP-K-def*
 using *assms a-lookup-None-notin-A*
 by (*auto split*: *option.splits simp*: *MDP-r-in-A-eq a-lookup′-def*)

lemma *fst-sa-lookup′-eq*:
 assumes *s* < *states a* ∈ *MDP-A s*
 shows *fst* (*a-lookup′* (*s-lookup mdp s*) *a*) = *MDP-r* (*s*, *a*)
 by (*simp add*: *MDP-r-in-A-eq assms*)

lemma *snd-sa-lookup′-eq*:
 assumes *s* < *states a* ∈ *MDP-A s*
 shows *pmf-of-list* (*snd* (*a-lookup′* (*s-lookup mdp s*) *a*)) = *MDP-K* (*s*, *a*)
 using *assms a-lookup′-def sa-lookup-eq a-lookup-None-notin-A*

by (*auto split*: *option.splits*)

lemma *entries-A-eq-r*: $s < states \Longrightarrow (a,\ r,\ succs) \in A\text{-}Map.entries$ $(s\text{-}lookup\ mdp\ s) \Longrightarrow r = MDP\text{-}r\ (s,\ a)$
using *sa-lookup-eq*[*OF - a-lookup-some-in-A*] *A-Map.inorder-lookup-Some*[*OF invar-s-lookup*]
by *simp*

lemma *entries-A-eq-K*: $s < states \Longrightarrow (a,\ r,\ succs) \in A\text{-}Map.entries$ $(s\text{-}lookup\ mdp\ s) \Longrightarrow pmf\text{-}of\text{-}list\ succs = MDP\text{-}K\ (s,\ a)$
using *sa-lookup-eq*[*OF - a-lookup-some-in-A*] *A-Map.inorder-lookup-Some*[*OF invar-s-lookup*]
by *simp*

lemma *a-inorderD*:
assumes $s < states$ $(a,\ r,\ succs) \in A\text{-}Map.entries\ (s\text{-}lookup\ mdp\ s)$
shows $a \in MDP\text{-}A\ s$ $r = MDP\text{-}r\ (s,\ a)$ $pmf\text{-}of\text{-}list\ succs = MDP\text{-}K\ (s,\ a)$
using *assms A-Map.inorder-lookup-Some a-lookup-some-in-A invar-s-lookup entries-A-eq-r entries-A-eq-K*
by *auto*

lemma *a-map-entries-lookup*: $s < states \Longrightarrow a \in MDP\text{-}A\ s \Longrightarrow (a,\ a\text{-}lookup'\ (s\text{-}lookup\ mdp\ s)\ a) \in A\text{-}Map.entries\ (s\text{-}lookup\ mdp\ s)$
by (*metis A-Map.lookup-in-keys A-Map.lookup-some-set-a-inorder MDP-A-def a-lookup'-def invar-s-lookup option.simps*(*5*))

lemma *lists-are-pmfs'*: $am \in set\ (s\text{-}list\ mdp) \Longrightarrow (a,r,p) \in A\text{-}Map.entries\ am \Longrightarrow pmf\text{-}of\text{-}list\text{-}wf\ p$
using *lists-are-pmfs* **by** *fastforce*

lemma *lists-are-pmfs''*: $am \in set\ (s\text{-}list\ mdp) \Longrightarrow (a,rp) \in A\text{-}Map.entries\ am \Longrightarrow pmf\text{-}of\text{-}list\text{-}wf\ (snd\ rp)$
using *lists-are-pmfs* **by** *fastforce*

lemma *lists-are-pmfs'''*: $s < states \Longrightarrow (a,rp) \in A\text{-}Map.entries\ (s\text{-}lookup\ mdp\ s) \Longrightarrow pmf\text{-}of\text{-}list\text{-}wf\ (snd\ rp)$
using *S-Map.lookup-in-list lists-are-pmfs'' s-invar s-len* **by** *blast*

lemma *pmf-of-list-wf-mdp*:
assumes $s < states$ $a \in MDP\text{-}A\ s$
shows $pmf\text{-}of\text{-}list\text{-}wf\ (snd\ (a\text{-}lookup'\ (s\text{-}lookup\ mdp\ s)\ a))$
using *assms a-map-entries-lookup*
by (*auto intro*: *lists-are-pmfs'''*[*of s a*])

lemma *set-list-pmf-in-states*:
assumes $s < states$ $a \in MDP\text{-}A\ s$ $(aa,\ b) \in set\ (snd\ (a\text{-}lookup'$

$(s\text{-}lookup\ mdp\ s)\ a))$
shows
  $aa < states$
proof −
  have$(s\text{-}lookup\ mdp\ s) \in set(\ s\text{-}list\ mdp)$
    using *S-Map.lookup-in-list assms*(*1*) *s-invar s-len* **by** *blast*
  moreover have $(a, (a\text{-}lookup'\ (s\text{-}lookup\ mdp\ s)\ a)) \in A\text{-}Map.entries$ $(s\text{-}lookup\ mdp\ s)$
    by (*metis A-Map.lookup-in-keys A-Map.lookup-some-set-a-inorder MDP-A-def a-lookup′-def assms*(*1*) *assms*(*2*) *invar-s-lookup option.case*(*2*))
  ultimately show *?thesis*
  using *K-closed-locale assms*
  by (*fastforce simp*: *case-prod-beta list-all-def*)
qed
end

lemma *sum-list-partition-fst*: $(\sum sp \leftarrow ps.\ f\ sp) = (\sum a \in fst\ `\ set\ ps.\ \sum sp \leftarrow filter\ (\lambda z.\ fst\ z = a)\ ps.\ f\ sp)$
proof (*induction ps*)
  case *Nil*
  then show *?case* **by** *auto*
next
    have $*{:}(if\ b\ then\ x\ else\ y) + z = (if\ b\ then\ x{+}z\ else\ y{+}z)$ **for** *b x y z*
      by *auto*
  case (*Cons a ps*)
  show *?case*
  proof (*cases* $fst\ a \in fst\ `\ set\ ps$)
    case *True*
     have $sum\text{-}list\ (map\ f\ (a\ \#\ ps)) = f\ a + (\sum a \in fst\ `\ set\ ps.\ sum\text{-}list\ (map\ f\ (filter\ (\lambda z.\ fst\ z = a)\ ps)))$
       by (*auto dest*: *simp*: *Cons if-distrib sum.insert-remove cong*: *sum.cong if-cong*)
    also have $\ldots = (\sum aa \in fst\ `\ set\ ps.\ (if\ fst\ a = aa\ then\ f\ a\ else\ 0)) + (\sum aa \in fst\ `\ set\ ps.\ sum\text{-}list\ (map\ f\ (filter\ (\lambda z.\ fst\ z = aa)\ ps)))$
      using *True* **by** *auto*
    also have $\ldots = (\sum aa \in fst\ `\ set\ ps.\ (if\ fst\ a = aa\ then\ f\ a\ else\ 0) + sum\text{-}list\ (map\ f\ (filter\ (\lambda z.\ fst\ z = aa)\ ps)))$
      by (*auto simp*: *sum.distrib*)
    also have $\ldots = (\sum aa \in fst\ `\ set\ (a\ \#\ ps).\ sum\text{-}list\ (map\ f\ (filter\ (\lambda z.\ fst\ z = aa)\ (a\ \#\ ps))))$
      by (*auto simp*: ∗ *True if-distrib*[*of map* -] *if-distrib*[*of sum-list*] *insert-absorb cong*: *if-cong*)
    finally show *?thesis***.**
  next
      case *False*
     have $sum\text{-}list\ (map\ f\ (a\ \#\ ps)) = f\ a + (\sum a \in fst\ `\ set\ ps.\ sum\text{-}list\ (map\ f\ (filter\ (\lambda z.\ fst\ z = a)\ ps)))$

by (*auto dest*: *simp*: *Cons if-distrib sum.insert-remove cong*: *sum.cong if-cong*)
also have ... = ($\sum$ *aa*∈*fst* ‘ *set* (*a*#*ps*). (*if fst a* = *aa then f a else 0*)) + ($\sum$ *aa*∈*fst* ‘ *set ps*. *sum-list* (*map f* (*filter* (λ*z*. *fst z* = *aa*) *ps*)))
using *False*
by (*auto simp*:)
also have ... = ($\sum$ *aa*∈*fst* ‘ *set* (*a*#*ps*). (*if fst a* = *aa then f a else 0*)) + ($\sum$ *aa*∈*fst* ‘ *set* (*a*#*ps*). *sum-list* (*map f* (*filter* (λ*z*. *fst z* = *aa*) *ps*)))
proof −
have ∗: (⋀*x*. *x* ∈ *set xs* ⟹ *x* = *0*) ⟹ *sum-list xs* = *0* **for** *xs*
by (*induction xs*) *auto*
have (*sum-list* (*map f* (*filter* (λ*z*. *fst z* = *fst a*) *ps*))) = *0*
using *False*
by (*intro* ∗) (*auto intro*: *fst-eqD*)
thus *?thesis*
by (*auto simp*: *False*)
qed
also have ... = ($\sum$ *aa*∈*fst* ‘ *set* (*a*#*ps*). (*if fst a* = *aa then f a else 0*) + *sum-list* (*map f* (*filter* (λ*z*. *fst z* = *aa*) *ps*)))
by (*auto simp*: *sum.distrib*)
also have ... = ($\sum$ *aa*∈*fst* ‘ *set* (*a* # *ps*). *sum-list* (*map f* (*filter* (λ*z*. *fst z* = *aa*) (*a* # *ps*))))
by (*auto simp*: ∗ *False if-distrib*[*of map* -] *if-distrib*[*of sum-list*] *insert-absorb cong*: *if-cong*)
finally show *?thesis*.
qed
qed

lemma *pmf-of-list-expectation*:
assumes *pmf-of-list-wf ps*
shows *measure-pmf.expectation* (*pmf-of-list ps*) *f* = ($\sum$ (*s*′, *p*)← *ps*. *p* ∗ *f s*′)
proof −
have *sumlist-cong*: *sum-list* (*map f xs*) = *sum-list* (*map g xs*) **if** ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x* **for** *f g xs*
using *that*
by (*induction xs*) *auto*
have ($\sum$ (*s*′, *p*)← *ps*. *p* ∗ *f s*′) = *sum-list* (*map* (λ*sp*. *snd sp* ∗ *f* (*fst sp*)) *ps*)
by (*metis case-prod-conv fst-def old.prod.exhaust snd-def*)
also have ... = ($\sum$ *a* ∈ *fst* ‘ (*set ps*). *sum-list* (*map* (λ*sp*. *snd sp* ∗ *f* (*fst sp*)) (*filter* (λ*z*. *fst z* = *a*) *ps*)))
using *sum-list-partition-fst*
by *auto*
also have ... = ($\sum$ *a* ∈ *fst* ‘ (*set ps*). *sum-list* (*map snd* (*filter* (λ*z*. *fst z* = *a*) *ps*)) ∗ *f a*)
by (*auto simp*: *add.commute set-filter map-eq-conv sum-list-mult-const*[*symmetric*]

intro!: *sumlist-cong sum.cong*)

also have $\ldots = (\sum a \in \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\} \cup \{u.\ \exists b.\ (u{,}b) \in \mathit{set\ ps} \wedge (\forall b.\ (u{,}b) \in \mathit{set\ ps} \longrightarrow b = 0)\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$

proof −

have $\mathit{fst}\ \text{`}\ (\mathit{set\ ps}) = \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps}\}$

by *force*

also have $\ldots = \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\} \cup \{u.\ \exists b.\ (u{,}b) \in \mathit{set\ ps} \wedge (\forall b.\ (u{,}b) \in \mathit{set\ ps} \longrightarrow b = 0)\}$

by *auto*

finally show *?thesis* **by** *auto*

qed

also have $\ldots = (\sum a \in \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\}\ .\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a) + (\sum a \in \{u.\ \exists b.\ (u{,}b) \in \mathit{set\ ps} \wedge (\forall b.\ (u{,}b) \in \mathit{set\ ps} \longrightarrow b = 0)\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$

proof −

have $\{u.\ (\exists b.\ (u,\ b) \in \mathit{set\ ps}) \wedge (\forall b.\ (u,\ b) \in \mathit{set\ ps} \longrightarrow b = 0)\} \subseteq \mathit{fst}\ \text{`}\ \mathit{set\ ps}$

by *force*

hence *finite* $\{u.\ (\exists b.\ (u,\ b) \in \mathit{set\ ps}) \wedge (\forall b.\ (u,\ b) \in \mathit{set\ ps} \longrightarrow b = 0)\}$

using *finite-surj* **by** *blast*

thus *?thesis*

using *assms finite-set-pmf-of-list set-pmf-of-list*

by (*subst sum.union-disjoint*) *fastforce*+

qed

also have $\ldots = (\sum a \in \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\}\ .\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$

by (*fastforce intro*!: *sum.neutral iffD2*[*OF sum-list-nonneg-eq-0-iff*])

also have $\ldots = (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$ **by** *blast*

finally have $\mathit{measure\text{-}pmf.expectation}\ (\mathit{pmf\text{-}of\text{-}list\ ps})\ f = (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$

using *finite-set-pmf-of-list*[*OF assms*]

by (*subst integral-measure-pmf*) (*fastforce simp add*: *pmf-pmf-of-list set-pmf-of-list assms*)+

thus *?thesis*

using ‹$(\sum (s',\ p){\leftarrow}\mathit{ps}.\ p * f\ s') = (\sum \mathit{sp}{\leftarrow}\mathit{ps}.\ \mathit{snd\ sp} * f\ (\mathit{fst\ sp}))$› ‹$(\sum a{\in}\mathit{fst}\ \text{`}\ \mathit{set\ ps}.\ \sum \mathit{sp}{\leftarrow}\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps}.\ \mathit{snd\ sp} * f\ (\mathit{fst\ sp})) = (\sum a{\in}\mathit{fst}\ \text{`}\ \mathit{set\ ps}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$› ‹$(\sum a{\in}\mathit{fst}\ \text{`}\ \mathit{set\ ps}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a) = (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\} \cup \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge (\forall b.\ (u,\ b) \in \mathit{set\ ps} \longrightarrow b = 0)\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$› ‹$(\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\} \cup \{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge (\forall b.\ (u,\ b) \in \mathit{set\ ps} \longrightarrow b = 0)\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a) = (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in \mathit{set\ ps} \wedge b \neq 0\}.\ \mathit{sum\text{-}list}\ (\mathit{map\ snd}\ (\mathit{filter}\ (\lambda z.\ \mathit{fst}\ z = a)\ \mathit{ps})) * f\ a)$

$+ (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in set\ ps \wedge (\forall b.\ (u,\ b) \in set\ ps \longrightarrow b = 0)\}.$ $sum\text{-}list\ (map\ snd\ (filter\ (\lambda z.\ fst\ z = a)\ ps)) * f\ a)$› ‹$(\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in set\ ps \wedge b \neq 0\}.\ sum\text{-}list\ (map\ snd\ (filter\ (\lambda z.\ fst\ z = a)\ ps)) * f\ a) + (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in set\ ps \wedge (\forall b.\ (u,\ b) \in set\ ps \longrightarrow b = 0)\}.\ sum\text{-}list\ (map\ snd\ (filter\ (\lambda z.\ fst\ z = a)\ ps)) * f\ a) = (\sum a{\in}\{u.\ \exists b.\ (u,\ b) \in set\ ps \wedge b \neq 0\}.\ sum\text{-}list\ (map\ snd\ (filter\ (\lambda z.\ fst\ z = a)\ ps)) * f\ a)$› ‹$(\sum sp{\leftarrow}ps.\ snd\ sp * f\ (fst\ sp)) = (\sum a{\in}fst\ `\ set\ ps.\ \sum sp{\leftarrow}filter\ (\lambda z.\ fst\ z = a)\ ps.\ snd\ sp * f\ (fst\ sp))$› **by** *presburger*
qed

locale *MDP-Code* = *MDP-Code-raw* +
  V-Map : *Array′ v-lookup* :: *′tv* ⇒ *nat* ⇒ *real v-update v-len v-array v-list v-invar* +
  D-Map : *Map-by-Ordered d-empty d-update* :: *nat* ⇒ *nat* ⇒ *′td* ⇒ *′td d-delete d-lookup d-inorder d-inv*
  for *v-lookup v-update v-len v-array v-list v-invar*
  and *d-empty d-update d-delete d-lookup d-inorder d-inv* +
fixes
  l :: *real*
assumes
  zero-le-disc-locale: $0 \le l$ **and**
  disc-lt-one-locale: $l < 1$
begin

sublocale *V-Map*: *Array-real v-lookup v-update v-len v-array v-list v-invar*
  by *unfold-locales*

sublocale *V-Map*: *Array-zero v-lookup v-update v-len v-array v-list v-invar*
  by *unfold-locales*

sublocale *D-Map*: *Map-by-Ordered-nat-zero d-empty d-update d-delete d-lookup d-inorder d-inv*
  by *unfold-locales*

definition *d-lookup′ m x* = *the* (*d-lookup m x*)

lemma *map-to-fun-lookup*: *D-Map.invar f* ⟹ *s* ∈ *D-Map.keys f* ⟹ *D-Map.map-to-fun f s* = *d-lookup′ f s*
  unfolding *D-Map.map-to-fun-def d-lookup′-def*
 using *D-Map.lookup-None-set-inorder*
  by (*auto split*: *option.splits*)

sublocale *MDP*: *MDP-reward* (*MDP-A*) (*MDP-K*) (*MDP-r*) *l*
 using *MDP-A-ne bounded-MDP-r zero-le-disc-locale disc-lt-one-locale*
  by *unfold-locales auto*

sublocale *MDP*: *MDP-nat-disc* (*MDP-A*) (*MDP-K*) (*MDP-r*) *l* λX. *LEAST* y. $y \in X$ *states*
proof −
have [*simp*]: *MDP-reward-disc.max-L-ex MDP-A MDP-K MDP-r l s v* **for** *s v*
by (*simp add*: *MDP.MDP-reward-axioms MDP-reward-disc.intro MDP-reward-disc.max-L-ex-def MDP-reward-disc-axioms.intro disc-lt-one-locale finite-is-arg-max has-arg-max-def*)
have $X \neq \{\} \Longrightarrow$ (*LEAST* (y::*nat*). $y \in X$) $\in X$ **for** X
using *Inf-nat-def Inf-nat-def1* **by** *presburger*
thus *MDP-nat-disc MDP-A MDP-K MDP-r l* (λX. *LEAST* y. $y \in X$) *states*
using *MDP-K-closed MDP-K-comp-closed MDP-r-zero-notin-states MDP-A-outside disc-lt-one-locale*
by *unfold-locales auto*
qed

7 Code for *MDP*.L_a

definition L_a*-code rp v* = (
let (*r*, *ps*) = *rp in r* + *l* ∗ (*foldl* (λ *acc* (s', *p*). *p* ∗ *v-lookup v* s' + *acc*)) *0 ps*)

lemma L_a*-code-correct*:
assumes *s* < *states v-len v* = *states v-invar v pmf-of-list* (*snd rps*) = *MDP-K* (*s*, *a*)
pmf-of-list-wf (*snd rps*) *fst* ‘ *set* (*snd rps*) ⊆ {*0..<states*} *fst rps* = *MDP-r* (*s*, *a*)
shows L_a*-code rps v* = *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*
proof −
have *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*v-lookup v*) = *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*V-Map.map-to-bfun v*)
using *assms MDP.K-closed*
by (*force simp*: *V-Map.map-to-bfun.rep-eq split*: *option.splits intro*!: *Bochner-Integration.integral-cong-AE AE-pmfI*)
have *foldl* (λ*acc x*. *f x* + *acc*) *x xs* = ($\sum$ *x*←*xs*. *f x*) + *x* **for** *f xs* **and** *x* :: *real*
by (*induction xs arbitrary*: *x*) (*auto simp*: *algebra-simps*)
hence ∗: ($\sum$ *x*←*xs*. *f x*) = *foldl* (λ*acc x*. *f x* + *acc*) (*0*::*real*) *xs* **for** *f xs*
by (*metis add.right-neutral*)
have *foldl* (λ*acc* (s', *p*). *p* ∗ *v-lookup v* s' + *acc*) *0* (*snd rps*) = *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*apply-bfun* (*V-Map.map-to-bfun v*))
unfolding *assms*(*4*)[*symmetric*]
using *assms*(*5*,*6*,*7*)
by (*auto intro*!: *foldl-cong simp*: *pmf-of-list-expectation* ∗ *V-Map.map-to-bfun.rep-eq assms*(*2*,*3*))
thus *?thesis*

unfolding L_a*-code-def*
using *assms*
by (*simp add*: *case-prod-unfold*)
qed

lemma *L-GS-code-correct'*:
assumes *s* < *states v-len v* = *states v-invar v a* ∈ *MDP-A s*
shows L_a*-code* (*a-lookup'* (*s-lookup mdp s*) *a*) *v* = *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*
using *pmf-of-list-wf-mdp assms set-list-pmf-in-states*
by(*intro* L_a*-code-correct*)
(*auto simp*: *fst-sa-lookup'-eq*[*symmetric*] *snd-sa-lookup'-eq*)

lemma *v-lookup-map-to-bfun*: *v-invar m* ⟹ *k* < *v-len m* ⟹ *v-lookup m k* = *V-Map.map-to-bfun m k*
unfolding *V-Map.map-to-bfun.rep-eq*
by (*force split*: *option.splits*)

lemma *map-to-bfun-eq-fun*: *v-invar m* ⟹ *apply-bfun* (*V-Map.map-to-bfun v*) = *V-Map.map-to-fun v*
by (*auto simp*: *V-Map.map-to-bfun.rep-eq V-Map.map-to-fun-def*)

lemma *map-to-fun-notin*: *D-Map.invar d* ⟹ *k* ∉ *D-Map.keys d* ⟹ *D-Map.map-to-fun d k* = *0*
by (*auto simp*: *D-Map.map-to-fun-def D-Map.lookup-notin-keys split*: *option.splits*)

8 Folding lists to maps

lemma *v-lookup-update*: *v-invar m* ⟹ *k* < *v-len m* ⟹ *j* < *v-len m* ⟹ *v-lookup* (*v-update j x m*) *k* = (*if j* = *k then x else v-lookup m k*)
by (*auto simp add*: *V-Map.invar-update V-Map.len-array V-Map.lookup V-Map.update*)

lemma *V-invar-fold*: *v-invar m* ⟹ *set xs* ⊆ {*0..*<*v-len m*} ⟹ *v-invar* (*fold* (*λs v. v-update s* (*f s v*) *v*) *xs m*)
by (*induction xs arbitrary*: *m*) (*auto simp add*: *V-Map.invar-update V-Map.len-array V-Map.update*)

lemma *V-len-fold*: *v-invar m* ⟹ *set xs* ⊆ {*0..*<*v-len m*} ⟹ *v-len* (*fold* (*λs v. v-update s* (*f s v*) *v*) *xs m*) = *v-len m*
by (*induction xs arbitrary*: *m*) (*auto simp add*: *V-Map.invar-update V-Map.len-array V-Map.update*)

lemma *v-len-update*: *v-invar m* ⟹ *j* < *v-len m* ⟹ *v-len* (*v-update j x m*) = *v-len m*
by (*simp add*: *V-Map.invar-update V-Map.len-array V-Map.update*)

lemma *v-lookup-fold*: *v-invar m* $\Longrightarrow$ *n* $\leq$ *v-len m* $\Longrightarrow$ *k* $<$ *n* $\Longrightarrow$ *v-lookup* (*fold* (λ*s v. v-update s* (*f s*) *v*) [*0..<n*] *m*) *k* = (*f k*)
using *V-invar-fold*
by (*induction n arbitrary*: *m k*) (*auto intro*!: *V-invar-fold simp*: *v-lookup-update V-len-fold*)

lemma *keys-fold-map*: *D-Map.invar m* $\Longrightarrow$ *D-Map.keys* (*fold* (λ*s. d-update s* (*f s*)) *xs m*) = *D-Map.keys m* $\cup$ *set xs*
using *D-Map.map-specs*
by (*induction xs arbitrary*: *m*) *auto*

lemma *invar-fold-update*: *D-Map.invar m* $\Longrightarrow$ *D-Map.invar* (*fold* (λ*s. d-update s* (*f s*)) *xs m*)
using *D-Map.map-specs* **by** (*induction xs arbitrary*: *m*) *auto*

lemma *d-lookup-fold*: *k* $<$ *n* $\Longrightarrow$ *D-Map.invar m* $\Longrightarrow$ *d-lookup* (*fold* (λ*s v. d-update s* (*f s*) *v*) [*0..<n*] *m*) *k* = *Some* (*f k*)
using *D-Map.map-update invar-fold-update* **by** (*induction n*) *auto*

9 Code for $MDP.\mathcal{L}_b$

definition $\mathcal{L}$*-GS-code acts v* =
(*MAX* (*a*, *rs*) $\in$ *A-Map.entries acts.* L_a*-code rs v*)

lemma $\mathcal{L}$*-GS-code-correct*:
assumes *s* $<$ *states* *v-invar v v-len v* = *states*
shows $\mathcal{L}$*-GS-code* (*s-lookup mdp s*) *v* = ($\bigsqcup$ *a* $\in$ *MDP-A s. MDP.*L_a *a* (*V-Map.map-to-bfun v*) *s*)
unfolding $\mathcal{L}$*-GS-code-def*
proof (*subst cSup-eq-Max*[*symmetric*])
show *finite* ((λ(*a*, *rs*). L_a*-code rs v*) ` *A-Map.entries* (*s-lookup mdp s*))
using *A-Map.finite-entries* **by** *blast*
show (λ(*a*, *rs*). L_a*-code rs v*) ` *A-Map.entries* (*s-lookup mdp s*) $\neq$ {}
using *ne-s-lookup assms A-Map.is-empty-iff-entries-empty* **by** *blast*

have L_a*-code* (*r*,*s'*) *v* = *MDP.*L_a *a* (*V-Map.map-to-bfun v*) *s* **if** (*a*, *r*,*s'*) $\in$ *A-Map.entries* (*s-lookup mdp s*) **for** *a r s'*
proof −
have *r* = *MDP-r* (*s*, *a*)
by (*metis assms*(*1*) *entries-A-eq-r that*)
moreover have *fst* ` *set s'* $\subseteq$ *MDP.state-space*
using *K-closed-locale' S-Map.lookup-in-list assms*(*1*) *s-invar s-len that* **by** *fastforce*
moreover have *s'* = (*snd* (*a-lookup'* (*s-lookup mdp s*) *a*))
using *A-Map.inorder-lookup-Some a-lookup'-def assms*(*1*) *invar-s-lookup that* **by** *auto*

ultimately show *?thesis*
using *assms that a-inorderD pmf-of-list-wf-mdp*
by (*intro* L_a*-code-correct*) *auto*
qed
thus ($\bigsqcup$(*a*, *rs*)∈*A-Map.entries* (*s-lookup mdp s*). L_a*-code rs v*) = ($\bigsqcup$ *a*∈*MDP-A s*. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*)
using *invar-s-lookup*
by (*auto simp*: *MDP-A-def assms SUP-image A-Map.keys-eq-fst-entries intro*!: *SUP-cong*)
qed

definition $\mathcal{L}$*-code v* =
V-Map.arr-tabulate (λ*s*. $\mathcal{L}$*-GS-code* (*s-lookup mdp s*) *v*) *states*

lemma $\mathcal{L}$*-code-lookup*:
assumes *s* < *states v-len v* = *states v-invar v*
shows *v-lookup* ($\mathcal{L}$*-code v*) *s* = ($\mathcal{L}$*-GS-code* (*s-lookup mdp s*) *v*)
using *assms* **unfolding** $\mathcal{L}$*-code-def* **by** *auto*

lemma *keys-*$\mathcal{L}$*-code*[*simp*]: *v-invar v* ⟹ *v-len v* = *states* ⟹ *v-len* ($\mathcal{L}$*-code v*) = *v-len v*
unfolding $\mathcal{L}$*-code-def* **by** *auto*

lemma $\mathcal{L}$*-code-correct*:
assumes *s* < *states v-len v* = *states v-invar v*
shows *v-lookup* ($\mathcal{L}$*-code v*) *s* = *MDP*.$\mathcal{L}_b$ (*V-Map.map-to-bfun v*) *s*
unfolding $\mathcal{L}$*-code-lookup*[*OF assms*] *MDP*.$\mathcal{L}_b$*-eq-*L_a*-max*′
by (*auto intro*: *cSup-eq-Max simp*: *assms* $\mathcal{L}$*-GS-code-correct*)

lemma *invar-*$\mathcal{L}$*-code*: *v-invar v* ⟹ *v-invar* ($\mathcal{L}$*-code v*)
using *V-invar-fold* **unfolding** $\mathcal{L}$*-code-def*
using *V-Map.arr-tabulate-def V-Map.invar-array* **by** *presburger*

lemma $\mathcal{L}$*-code-correct*′:
assumes *v-len v* = *states v-invar v*
shows *V-Map.map-to-bfun* ($\mathcal{L}$*-code v*) = *MDP*.$\mathcal{L}_b$ (*V-Map.map-to-bfun v*)
using *MDP*.$\mathcal{L}_b$*-zero* *assms*
proof (*intro bfun-eqI*)
fix *x*
show *apply-bfun* (*V-Map.map-to-bfun* ($\mathcal{L}$*-code v*)) *x* = *apply-bfun* (*MDP*.$\mathcal{L}_b$ (*V-Map.map-to-bfun v*)) *x*
proof (*cases x* < *states*)
case *True*
then show *?thesis*

using *assms keys-$\mathcal{L}$-code $\mathcal{L}$-code-correct invar-$\mathcal{L}$-code v-lookup-map-to-bfun*
by *force*
next
case *False*
then show *?thesis*
using *assms keys-$\mathcal{L}$-code MDP.$\mathcal{L}_b$-zero*
by (*fastforce simp*: *V-Map.map-to-bfun.rep-eq dest*: *split*: *option.splits*)+
qed
qed

10 Code to check condition

definition *check-dist v v′ eps* = (
let m = *eps* ∗ (*1* − *l*) / (*2* ∗ *l*) *in*
(∀ *s* < *states*. *abs* (*v-lookup v s* − *v-lookup v′ s*) < *m*))

lemma *check-dist-correct*:
assumes *v-invar v v-invar v′ v-len v* = *states v-len v′* = *states eps* > *0 l* ≠ *0*
shows *check-dist v v′ eps* ⟷ *dist* (*V-Map.map-to-bfun v*) (*V-Map.map-to-bfun v′*) < *eps* ∗ (*1* − *l*) / (*2* ∗ *l*)
proof −
have *dist-zero-ge*: *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun v′*) *x*) = *0* **if** *x* ≥ *states* **for** *x*
using *assms that*
by (*auto simp*: *V-Map.map-to-bfun.rep-eq split*: *option.splits*)
have *univ*: *UNIV* = {*0*..<*states*} ∪ {*states*..} **by** *auto*
have *fin*: *finite* (*range* (λ*x*. *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun v′*) *x*)))
by (*auto simp*: *dist-zero-ge univ Set.image-Un Set.image-constant*[*of states*])
have *zero-less-eps*: *0* < *eps* ∗ (*1* − *l*) / (*2* ∗ *l*)
using *MDP.zero-le-disc assms MDP.disc-lt-one*
by (*auto intro*!: *mult-imp-less-div-pos simp*: *less-le*)
show *?thesis*
proof
assume *h*: *check-dist v v′ eps*
show *dist* (*V-Map.map-to-bfun v*) (*V-Map.map-to-bfun v′*) < *eps* ∗ (*1* − *l*) / (*2* ∗ *l*)
unfolding *dist-bfun.rep-eq*
proof (*rule finite-imp-Sup-less*[*OF fin*])
show *0* ∈ *range* (λ*x*. *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun v′*) *x*))
using *dist-zero-ge* **by** *fastforce*
have *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun v′*) *x*) < *eps* ∗ (*1* − *l*) / (*2* ∗ *l*) **if** *x* < *states* **for** *x*
using *assms h that*

unfolding *check-dist-def V-Map.map-to-bfun.rep-eq dist-real-def*
by (*auto split*: *option.splits*)
thus $x \in$ *range* (λx. *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun v'*) *x*)) $\Longrightarrow x <$ *eps* $*$ (*1* $-$ *l*) / (*2* $*$ *l*) **for** *x*
using *zero-less-eps dist-zero-ge imageE not-less*
by (*metis* (*no-types*, *lifting*))
qed
next
show *dist* (*V-Map.map-to-bfun v*) (*V-Map.map-to-bfun v'*) $<$ *eps* $*$ (*1* $-$ *l*) / (*2* $*$ *l*) $\Longrightarrow$ *check-dist v v' eps*
using *assms fin*
by (*auto simp*: *check-dist-def dist-bfun.rep-eq finite-Sup-less-iff dist-real-def v-lookup-map-to-bfun*)
qed
qed

11 Find policy

definition *find-policy-state-code-aux v s* =
(*least-arg-max-max-ne* (λ(-, *rsuccs*).
L_a-*code rsuccs v*) ((*a-inorder* (*s-lookup mdp s*))))

definition *find-policy-state-code-aux' v s* = (
case find-policy-state-code-aux v s of ((*a*, -, -), *v*) $\Rightarrow$ (*a*, *v*))

lemma *find-policy-state-code-aux-eq*:
assumes *s* $<$ *states*
shows *find-policy-state-code-aux' v s* = (*least-arg-max-max-ne* (λa.
L_a-*code* (*a-lookup'* (*s-lookup mdp s*) *a*) *v*) ((*map fst* (*a-inorder* (*s-lookup mdp s*)))))
unfolding *find-policy-state-code-aux'-def find-policy-state-code-aux-def*
using *assms ne-s-lookup invar-s-lookup A-Map.inorder-lookup-Some*
by (*subst least-arg-max-max-ne-app'*[*symmetric*])
(*auto simp*: *A-Map.entries-def a-lookup'-def case-prod-unfold A-Map.is-empty-def*)

lemma *find-policy-state-code-aux'-eq'*:
assumes *s* $<$ *states v-len v* = *states v-invar v*
shows *find-policy-state-code-aux' v s* =
(*least-arg-max* (λa. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) (λa. *a* $\in$ *MDP-A s*), *Max* ((λa. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) ‘ (*MDP-A s*)))
proof −
have *find-policy-state-code-aux' v s* = *least-arg-max-max-ne* (λa. L_a-*code* (*a-lookup'* (*s-lookup mdp s*) *a*) *v*) (*map fst* (*a-inorder* (*s-lookup mdp s*)))
using *find-policy-state-code-aux-eq assms* **by** *auto*
also have ‹. . . = (*least-arg-max* (λa. L_a-*code* (*a-lookup'* (*s-lookup*

mdp s) *a*) *v*) (*List.member* (*map fst* (*a-inorder* (*s-lookup mdp s*)))),
MAX a∈*set* (*map fst* (*a-inorder* (*s-lookup mdp s*))). L_a*-code* (*a-lookup′* (*s-lookup mdp s*) *a*) *v*)›
using *A-Map.is-empty-def assms*(*1*) *ne-s-lookup A-Map.invar-def A-inv-locale S-Map.lookup-in-list s-invar s-len*
by (*auto simp*: *fold-max-eq-arg-max′*)
also have ‹. . . = (*least-arg-max* (λ*a*. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) (*List.member* (*map fst* (*a-inorder* (*s-lookup mdp s*)))),
MAX a∈*set* (*map fst* (*a-inorder* (*s-lookup mdp s*))). *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*)›
using *assms a-inorderD*(*1*) *A-Map.keys-def MDP-A-def*
by (*auto intro*!: *least-arg-max-cong simp*: *L-GS-code-correct′ in-set-member*[*symmetric*])
also have ‹. . . = (*least-arg-max* (λ*a*. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) (λ*a*. *a* ∈ *MDP-A s*),
MAX a∈*MDP-A s*. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*)›
proof −
have ∗: *a* ∈ *fst* ‘ *set* (*a-inorder* (*s-lookup mdp s*)) ⟷ *List.member* (*map fst* ((*a-inorder* (*s-lookup mdp s*)))) *a* **for** *a*
by (*auto simp*: *List.member-def*)
show *?thesis*
using *assms* L_a*-code-correct A-Map.keys-def*
by (*auto intro*!: *least-arg-max-cong simp*: ∗ *MDP-A-def*)
qed
finally show *?thesis***.**
qed

definition *vi-find-policy-code* (*v*::′*tv*) = *D-Map.from-list′* (λ*s*. *fst* (*find-policy-state-code-aux′ v s*)) [*0*..<*states*]

lemma *d-invar-vi-find-policy-code*: *D-Map.invar* (*vi-find-policy-code v*)
using *D-Map.from-list-invar vi-find-policy-code-def* **by** *simp*

lemma *d-keys-vi-find-policy-code*: *D-Map.keys* (*vi-find-policy-code v*) = {*0*..<*states*}
using *D-Map.from-list-keys vi-find-policy-code-def* **by** *simp*

lemma *vi-find-policy-code-notin*:
assumes *s* ≥ *states* **shows** *d-lookup* (*vi-find-policy-code v*) *s* = *None*
using *D-Map.lookup-notin-keys assms d-invar-vi-find-policy-code d-keys-vi-find-policy-code* **by** *force*

lemma *vi-find-policy-code-in*:
assumes *s* < *states* **shows** ∃ *x*. *d-lookup* (*vi-find-policy-code v*) *s* = *Some x*
by (*simp add*: *D-Map.lookup-in-keys assms d-invar-vi-find-policy-code d-keys-vi-find-policy-code*)

lemma *vi-find-policy-code-ge*: *s* ≥ *states* ⟹ *D-Map.map-to-fun* (*vi-find-policy-code*

v) $s = 0$
using *vi-find-policy-code-notin vi-find-policy-code-def*
by (*auto simp*: *D-Map.map-to-fun-def*)

lemma *vi-find-policy-code-correct*:
assumes *v-len v* = *states v-invar v s* < *states*
shows *D-Map.map-to-fun* ((*vi-find-policy-code v*)) *s* = *least-arg-max* (λa. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) (λa. *a* ∈ *MDP-A s*)
using *assms*
by (*simp add*: *find-policy-state-code-aux′-eq′ vi-find-policy-code-def D-Map.map-to-fun-def*)

lemma *vi-find-policy-correct*:
assumes *v-len v* = *states v-invar v*
shows *D-Map.map-to-fun* (*vi-find-policy-code v*) = (*MDP.find-policy′* (*V-Map.map-to-bfun v*))
proof −
have *D-Map.map-to-fun* (*vi-find-policy-code v*) *s* = (*MDP.find-policy′* (*V-Map.map-to-bfun v*)) *s* **if** *s* ≥ *states* **for** *s*
using *vi-find-policy-code-ge that*
by (*auto simp*: *MDP.find-policy′-def MDP-A-def MDP.is-opt-act-def intro*!: *Least-equality*)
moreover have *D-Map.map-to-fun* (*vi-find-policy-code v*) *s* = (*MDP.find-policy′* (*V-Map.map-to-bfun v*)) *s* **if** *s* < *states* **for** *s*
using *that assms*
by (*auto simp*: *MDP.find-policy′-def vi-find-policy-code-correct least-arg-max-def MDP.is-opt-act-def*)
ultimately show *?thesis*
using *not-le* **by** *blast*
qed

definition *v0* = *V-Map.arr-tabulate* (λ-. *0*) *states*

lemma *v0-correct*: *v-invar v0 v-len v0* = *states*
unfolding *v0-def* **by** *auto*

definition *v-map-from-list xs* = *v-array xs*

end

hack: *pmf-of-list-wf* is polymorphic, so equality to *1* is checked for the sum of all probabilities. This fails for floats, so we reimplement the check monomorphically and change equality on floats to $(a = b) = (dist\ a\ b < 10\ /\ 10\ /\ 10^8)$.

lemmas *pmf-of-list-wf-code*[*code del*]

definition

pmf-of-list-wf′ xs ⟷ *list-all* (λ*z. snd z* ≥ *0*) *xs* ∧ *sum-list* (*map snd xs*) = (*1* :: *real*)

lemma *pmf-of-list-code* [*code abstract*]:
mapping-of-pmf (*pmf-of-list xs*) = (
if pmf-of-list-wf′ xs then
let xs′ = *filter* (λ*z. snd z*∗(*10^8*) ≠ *0*) *xs*
in Mapping.tabulate (*remdups* (*map fst xs′*))
(λ*x. sum-list* (*map snd* (*filter* (λ*z. fst z* = *x*) *xs′*)))
else
Code.abort (*STR ″Invalid list for pmf-of-list″*) (λ-. *mapping-of-pmf* (*pmf-of-list xs*)))
using *mapping-of-pmf-pmf-of-list′*[*of xs*] *pmf-of-list-wfI*
by (*auto simp add*: *pmf-of-list-wf′-def list-all-def*)

code-printing
constant *IArray.tabulate* ⇀ (*SML*) *case - of* (*n*, *f*) => *Vector.tabulate* (*IntInf.toInt n*, *fn i* => *f* ((*IntInf.fromInt i*)))
| **constant** *IArray.sub′* ⇀ (*SML*) *case - of* (*arr*, *i*) => *Vector.sub* (*arr*, *IntInf.toInt i*)
| **constant** *IArray.length′* ⇀ (*SML*) *IntInf.fromInt* (*Vector.length* -)

definition *nat-map-from-list* (*xs* :: (*nat* × -) *list*) = *foldr* (λ(*k*,*v*) *m*. *RBT-Map.update k v m*) *xs RBT-Set.empty*
definition *nat-pmf-of-list* (*xs* :: (*nat* × -) *list*) = *pmf-of-list xs*

definition *assoc-list-to-MDP d xs* =
to-valid-MDP (*MDP d* (*length xs*) (*IArray* (*map* (λ*as. foldr* (λ(*a*,(*r*,*p*)) *m. RBT-Map.update a* (*r*, *p*) *m*) *as RBT-Set.empty*) *xs*)))

lemma *starray-of-list-tabulate* [*code-unfold*]: *starray-of-list* (*map f* [*0*..<*n*]) = *starray-tabulate n f*
by (*simp add*: *starray-eq-iff tabulate-def*)

end
theory *VI-Code*
imports
Code-Setup
../Value-Iteration
HOL−Library.Code-Target-Numeral
begin

context *MDP-Code* **begin**

partial-function (*tailrec*) *VI-code-aux* **where**
VI-code-aux v eps = (
let v′ = *ℒ-code v in*
if check-dist v v′ eps

```
    then v′
    else VI-code-aux v′ eps)

lemmas VI-code-aux.simps[code]

definition VI-code v eps = (if l = 0 ∨ eps ≤ 0 then ℒ-code v else
VI-code-aux v eps)


lemma VI-code-aux-correct-aux:
  assumes eps > 0 v-invar v v-len v = states l ≠ 0
 shows V-Map.map-to-fun (VI-code-aux v eps) = MDP.value-iteration
eps (V-Map.map-to-bfun v)
  ∧ v-len (VI-code-aux v eps) = states
  ∧ v-invar (VI-code-aux v eps)
  using assms
proof (induction eps V-Map.map-to-bfun v arbitrary: v rule: MDP.value-iteration.induct)
  case (1 eps)
 have *: (check-dist v (ℒ-code v) eps) ⟷ 2 * l * dist (V-Map.map-to-bfun
v) (MDP.ℒ_b (V-Map.map-to-bfun v)) < eps * (1 − l)
  proof (subst check-dist-correct)
    have  0 < l using 1 MDP.zero-le-disc by linarith
   thus (dist (V-Map.map-to-bfun v) (V-Map.map-to-bfun (ℒ-code v))
< eps * (1 − l) / (2 * l)) =
   (2 * l * dist (V-Map.map-to-bfun v) (MDP.ℒ_b (V-Map.map-to-bfun
v)) < eps * (1 − l))
       by (subst pos-less-divide-eq) (fastforce simp: ℒ-code-correct′ 1
algebra-simps)+
  qed (auto simp: 1 intro: invar-ℒ-code)
 hence **: V-Map.map-to-fun (VI-code-aux v eps) = (MDP.value-iteration
eps (V-Map.map-to-bfun (ℒ-code v))) if ¬ (check-dist v (ℒ-code v) eps)
      using invar-ℒ-code 1 that by (auto simp: VI-code-aux.simps
ℒ-code-correct′)
 have V-Map.map-to-fun (VI-code-aux v eps) = (MDP.value-iteration
eps (V-Map.map-to-bfun v))
  proof (cases (check-dist v (ℒ-code v) eps))
    case True
    thus ?thesis
    using 1 invar-ℒ-code
    by (auto simp: MDP.value-iteration.simps VI-code-aux.simps[of v]
* map-to-bfun-eq-fun[symmetric] ℒ-code-correct′)
next
  case False
  thus ?thesis
    using 1 ℒ-code-correct′ ** * MDP.value-iteration.simps by auto
qed
  thus ?case
    using 1 VI-code-aux.simps ℒ-code-correct′ * invar-ℒ-code by auto
qed
```

lemma *VI-code-aux-correct*:
assumes *eps* > *0* *v-invar v v-len v* = *states l* ≠ *0*
shows *V-Map.map-to-fun* (*VI-code-aux v eps*) = *MDP.value-iteration eps* (*V-Map.map-to-bfun v*)
using *assms VI-code-aux-correct-aux* **by** *auto*

lemma *VI-code-aux-keys*:
assumes *eps* > *0* *v-invar v v-len v* = *states l* ≠ *0*
shows *v-len* (*VI-code-aux v eps*) = *states*
using *assms VI-code-aux-correct-aux* **by** *auto*

lemma *VI-code-aux-invar*:
assumes *eps* > *0* *v-invar v v-len v* = *states l* ≠ *0*
shows *v-invar* (*VI-code-aux v eps*)
using *assms VI-code-aux-correct-aux* **by** *auto*

lemma *VI-code-correct*:
assumes *eps* > *0* *v-invar v v-len v* = *states*
shows *V-Map.map-to-fun* (*VI-code v eps*) = *MDP.value-iteration eps* (*V-Map.map-to-bfun v*)
proof (*cases l* = *0*)
case *True*
then show *?thesis*
using *assms invar-ℒ-code ℒ-code-correct′*
unfolding *VI-code-def MDP.value-iteration.simps*[*of - V-Map.map-to-bfun v*]
by (*fastforce simp*: *map-to-bfun-eq-fun*)
next
case *False*
then show *?thesis*
using *assms*
by (*auto simp add*: *VI-code-def VI-code-aux-correct*)
qed

definition *VI-policy-code v eps* = *vi-find-policy-code* (*VI-code v eps*)

lemma *VI-policy-code-correct*:
assumes *eps* > *0* *v-invar v v-len v* = *states*
shows *D-Map.map-to-fun* (*VI-policy-code v eps*) = *MDP.vi-policy′ eps* (*V-Map.map-to-bfun v*)
proof −
have *V-Map.map-to-bfun* (*VI-code v eps*) = (*MDP.value-iteration eps* (*V-Map.map-to-bfun v*))
using *assms VI-code-correct*
by (*auto simp*: *VI-code-aux-invar map-to-bfun-eq-fun*)
moreover have *D-Map.map-to-fun* (*VI-policy-code v eps*) = *MDP.find-policy′* (*V-Map.map-to-bfun* (*VI-code v eps*))
unfolding *VI-code-def VI-policy-code-def*

using *assms invar-$\mathcal{L}$-code keys-$\mathcal{L}$-code vi-find-policy-correct vi-find-policy-correct VI-code-aux-correct-aux assms* **by** (*cases* $l = 0$) *auto*
ultimately show *?thesis*
unfolding *MDP.vi-policy'-def*
by *presburger*
qed

end

context *MDP-nat-disc*
begin

lemma *dist-opt-bound-$\mathcal{L}_b$*: *dist v ν_b-opt* $\leq$ *dist v* ($\mathcal{L}_b$ *v*) / ($1 - l$)
using *contraction-$\mathcal{L}$-dist*
by (*simp add*: *mult.commute mult-imp-le-div-pos*)

lemma *cert-$\mathcal{L}_b$*:
assumes $\varepsilon \geq 0$ *dist v* ($\mathcal{L}_b$ *v*) / ($1 - l$) $\leq \varepsilon$
shows *dist v ν_b-opt* $\leq \varepsilon$
using *assms dist-opt-bound-$\mathcal{L}_b$ order-trans* **by** *auto*

definition *check-value-$\mathcal{L}_b$ eps v* $\longleftrightarrow$ *dist v* ($\mathcal{L}_b$ *v*) / ($1 - l$) $\leq$ *eps*

definition *vi-policy-bound-error v* = (
let v' = ($\mathcal{L}_b$ *v*); *err* = ($2 * l$) $*$ *dist v v'* / ($1 - l$) *in*
(*err*, *find-policy' v'*))

lemma
assumes *vi-policy-bound-error v* = (*err*, *d*)
shows *dist* (ν_b (*mk-stationary-det d*)) *ν_b-opt* $\leq$ *err*
proof (*cases* $l = 0$)
case *True*
hence *vi-policy-bound-error v* = (0, *find-policy'* ($\mathcal{L}_b$ *v*))
unfolding *vi-policy-bound-error-def* **by** *auto*
have $\mathcal{L}_b$ *v* = $\mathcal{L}_b$ *ν_b-opt*
by (*auto simp*: *$\mathcal{L}_b$.rep-eq L-def simp del*: *$\mathcal{L}_b$-opt intro!*: *bfun-eqI simp*: *$\mathcal{L}$-def*) (*simp add*: *True*)
hence $\mathcal{L}_b$ *v* = *ν_b-opt*
by *auto*
hence ν_b (*mk-stationary-det* (*find-policy'* ($\mathcal{L}_b$ *v*))) = *ν_b-opt*
using *L-ν-fix ν-improving-opt-acts conserving-imp-opt*
unfolding *find-policy'-def ν-conserving-def*
by *auto*
then show *?thesis*
using *assms* **unfolding** *vi-policy-bound-error-def*
by (*auto simp*: *True*)
next
case *False*
then show *?thesis*

proof (*cases* $\mathcal{L}_b$ *v* = *v*)
case *True*
hence ν_b (*mk-stationary-det* (*find-policy'* ($\mathcal{L}_b$ *v*))) = ν_b*-opt*
using *L-*ν*-fix* ν*-improving-opt-acts conserving-imp-opt*
unfolding *find-policy'-def* ν*-conserving-def*
by *auto*
then show *?thesis*
using *assms* **unfolding** *vi-policy-bound-error-def*
by (*auto simp*: *True*)
next
case *False*
hence *1*: *dist v* ($\mathcal{L}_b$ *v*) > *0*
by *fastforce*
hence *2* ∗ *l* ∗ *dist v* ($\mathcal{L}_b$ *v*) > *0*
using ‹*l* ≠ *0*› *zero-le-disc* **by** (*simp add*: *less-le*)
hence *err* > *0*
using *assms* **unfolding** *vi-policy-bound-error-def* **by** *auto*
hence *dist* (ν_b (*mk-stationary-det* (*find-policy'* ($\mathcal{L}_b$ *v*)))) ν_b*-opt* <
err' **if** *err* < *err'* **for** *err'*
using *that assms*
unfolding *vi-policy-bound-error-def*
by (*auto simp*: *pos-divide-less-eq*[*symmetric*] *intro*: *find-policy'-error-bound*)
then show *?thesis*
using *assms* **unfolding** *vi-policy-bound-error-def Let-def*
by *force*
qed
qed

end

context *MDP-Code*
begin
definition *vi-policy-bound-error-code v* = (
let v' = ($\mathcal{L}$*-code v*);
d = *if states* = *0 then 0 else* (*MAX s* ∈ {..< *states*}. *dist* (*v-lookup*
v s) (*v-lookup v' s*));
err = (*2* ∗ *l*) ∗ *d* / (*1* − *l*) *in*
(*err*, *vi-find-policy-code v'*))

lemma
assumes *v-len v* = *states v-invar v*
shows *D-Map.map-to-fun* (*snd* (*vi-policy-bound-error-code v*)) = *snd*
(*MDP.vi-policy-bound-error* (*V-Map.map-to-bfun v*))
using *assms* $\mathcal{L}$*-code-correct' invar-*$\mathcal{L}$*-code vi-find-policy-correct*
by (*auto simp*: *vi-policy-bound-error-code-def MDP.vi-policy-bound-error-def*)

lemma *MAX-cong*:
assumes $\bigwedge x.\ x \in X \Longrightarrow f\ x = g\ x$
shows (*MAX x* ∈ *X*. *f x*) = (*MAX x* ∈ *X*. *g x*)

using *assms* **by** *auto*

lemma
assumes *v-len v = states v-invar v*
shows (*fst* (*vi-policy-bound-error-code v*)) = *fst* (*MDP.vi-policy-bound-error* (*V-Map.map-to-bfun v*))
proof–
have *dist-zero-ge*: *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun* ($\mathcal{L}$*-code v*)) *x*) = *0* **if** $x \geq$ *states* **for** *x*
using *assms that*
by (*auto simp*: *V-Map.map-to-bfun.rep-eq*)
have *univ*: *UNIV* = {*0..<states*} ∪ {*states..*} **by** *auto*
let *?d* = λ*x*. *dist* (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) (*apply-bfun* (*V-Map.map-to-bfun* ($\mathcal{L}$*-code v*)) *x*)

have *fin*: *finite* (*range* (λ*x*. *?d x*))
by (*auto simp*: *dist-zero-ge univ Set.image-Un Set.image-constant*[*of states*])

have *r*: *range* (λ*x*. *?d x*) = *?d* ‘ {*..<states*} ∪ *?d* ‘ {*states..*}
by *force*
hence *Sup* (*range ?d*) = *Max* (*range ?d*)
using *fin cSup-eq-Max* **by** *blast*
also have … = (*if states = 0 then* (*Max* (*?d* ‘ {*states..*})) *else max* (*Max* (*?d* ‘ {*..<states*})) (*Max* (*?d* ‘ {*states..*})))
using *r fin* **by** (*auto intro*: *Max-Un*)
also have … = (*if states = 0 then 0 else max* (*Max* (*?d* ‘ {*..<states*})) *0*)
using *dist-zero-ge*
by (*auto simp*: *Set.image-constant*[*of states*] *cSup-eq-Max*[*symmetric*, *of* (λ-. *0*) ‘ {*states..*}])
also have … = (*if states = 0 then 0 else* (*Max* (*?d* ‘ {*..<states*})))
by (*auto intro*!: *max-absorb1 max-geI*)
finally have *1*: *Sup* (*range ?d*) = (*if states = 0 then 0 else* (*Max* (*?d* ‘ {*..<states*}))).
thus *?thesis*
unfolding *MDP.vi-policy-bound-error-def vi-policy-bound-error-code-def dist-bfun-def*
using *assms v-lookup-map-to-bfun* $\mathcal{L}$*-code-correct′* $\mathcal{L}$*-code-correct*
by *fastforce*
qed

end

global-interpretation *VI-Code*:
MDP-Code

IArray.sub λ*n x arr*. *IArray* ((*IArray.list-of arr*)[*n*:= *x*]) *IArray.length IArray IArray.list-of* λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder rbt

MDP.transitions (*Rep-Valid-MDP mdp*) *MDP.states* (*Rep-Valid-MDP mdp*)

starray-get $\lambda i\ x\ arr.$ *starray-set arr i x starray-length starray-of-list* $\lambda arr.$ *starray-foldr* ($\lambda x\ xs.\ x\ \#\ xs$) *arr* [] λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder rbt

MDP.disc (*Rep-Valid-MDP mdp*)

for *mdp*
defines *VI-code* = *VI-Code.VI-code*
and *vi-policy-bound-error-code* = *VI-Code.vi-policy-bound-error-code*
and *VI-code-aux* = *VI-Code.VI-code-aux*
and L_a*-code* = *VI-Code.*L_a*-code*
and *a-lookup'* = *VI-Code.a-lookup'*
and *d-lookup'* = *VI-Code.d-lookup'*
and *find-policy-state-code-aux'* = *VI-Code.find-policy-state-code-aux'*
and *find-policy-state-code-aux* = *VI-Code.find-policy-state-code-aux*
and *check-dist* = *VI-Code.check-dist*
and $\mathcal{L}$*-code* = *VI-Code.*$\mathcal{L}$*-code*
and *VI-policy-code* = *VI-Code.VI-policy-code*
and $\mathcal{L}$*-GS-code* = *VI-Code.*$\mathcal{L}$*-GS-code*
and *v0* = *VI-Code.v0*
and *entries* = *M.entries*
and *from-list'* = *M.from-list'*
and *from-list* = *M.from-list*
and *vi-find-policy-code* = *VI-Code.vi-find-policy-code*
and *v-map-from-list* = *VI-Code.v-map-from-list*
and *arr-tabulate* = *starray-Array.arr-tabulate*
using *Rep-Valid-MDP*
by *unfold-locales*
(*fastforce simp*: *Ball-set-list-all*[*symmetric*] *case-prod-beta pmf-of-list-wf-def is-MDP-def RBT-Set.empty-def M.invar-def empty-def M.entries-def M.is-empty-def length-0-conv*[*symmetric*])+

lemmas *arr-tabulate-def*[*unfolded starray-Array.arr-tabulate-def*, *code*]
lemmas *entries-def*[*unfolded M.entries-def*, *code*]
lemmas *from-list'-def*[*unfolded M.from-list'-def*, *code*]
lemmas *from-list-def*[*unfolded M.from-list-def*, *code*]

function *tabulate* **where**
tabulate f acc upper n = (
  if n < *upper then tabulate f* (*update n* (*f n*) *acc*) *upper* (*Suc n*) *else acc*)
  by *auto*
termination
  by (*relation Wellfounded.measure* (λ(-, -, *i*,*N*). *i* − *N*)) *auto*

lemma *tabulate-Suc*: $j \le n' \Longrightarrow$ *update* n' (f n') (*tabulate* f m n' j) = *tabulate* f m (*Suc* n') j
proof (*induction* $n' - j$ *arbitrary*: m n' j)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc j*)
  then show *?case*
    by *auto*
qed

lemma *from-list′-upt* [*code-unfold*]: *from-list′* f [*0*..<n] = *tabulate f empty n 0*
proof −
  have $j \le n \Longrightarrow$ *foldl* (λ*acc s*. *update s* (*f s*) *acc*) *m* [*j*..<*n*] = *tabulate f m n j* **for** *m j*
  proof (*induction* $n - j$ *arbitrary*: *m n j*)
    case *0*
    then show *?case* **by** *auto*
  next
    case (*Suc x*)
    then obtain n' **where** n': n = *Suc* n'
      using *diff-le-self Suc-le-D* **by** *metis*
    then show *?case*
      using *Suc*
      by (*auto simp del*: *tabulate.simps simp*: n' *tabulate-Suc*)
  qed
  thus *?thesis*
  unfolding *from-list′-def M.from-list′-def*
  by *auto*
qed

end
theory *Code-Real-Approx-By-Float-Fix*
  imports
  HOL−*Library.Code-Real-Approx-By-Float*
begin

code-printing
    constant *Code-Real-Approx-By-Float.real-of-integer* ⇀
    (*SML*) *Real.fromLargeInt*

| **constant** *HOL.equal* :: *real* ⇒ *real* ⇒ *bool* ⇀
(*SML*) *Real.abs* (- − -) < *Math.pow* (*10.0*, *Real.~ 8.0*)
end
theory *VI-Code-Export-Float*
imports
VI-Code
Code-Real-Approx-By-Float-Fix
begin

export-code
to-valid-MDP MDP VI-policy-code v0 v-map-from-list vi-policy-bound-error-code RBT-Map.update nat-map-from-list assoc-list-to-MDP RBT-Set.empty nat-pmf-of-list pmf-of-list nat-of-integer Ratreal int-of-integer inverse-divide Tree2.inorder integer-of-nat
in *SML* **module-name** *VI-Code-Float* **file-prefix** *VI-Code-Float*

end
theory *VI-Code-Export-Rat*
imports
VI-Code
begin

export-code
ord-real-inst.less-eq-real quotient-of vi-policy-bound-error-code plus-real-inst.plus-real minus-real-inst.minus-real v0 to-valid-MDP MDP RBT-Map.update Rat.of-int divide divide-rat-inst.divide-rat divide-real-inst.divide-real nat-map-from-list assoc-list-to-MDP nat-pmf-of-list RBT-Set.empty VI-policy-code pmf-of-list nat-of-integer Ratreal int-of-integer inverse-divide Tree2.inorder integer-of-nat v-map-from-list
in *SML* **module-name** *VI-Code-Rat* **file-prefix** *VI-Code-Rat*

end

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

begin

12 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* $d = \nu_b$ (*mk-stationary-det* d)
end

context *MDP-act-disc*
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*: $\mathcal{L}_b$*-eq-argmax-*L_a *L-eq-*L_a*-det is-opt-act-def policy-improvement-def arg-max-SUP*)
thus *?thesis*
using *policy-improvement-is-dec-det assms* **by** (*auto simp*: ν*-improving-alt*)
qed

lemma *eval-policy-step-L*:
is-dec-det d $\Longrightarrow$ *L* (*mk-dec-det* (*policy-step d*)) (*policy-eval d*) = $\mathcal{L}_b$ (*policy-eval d*)
by (*auto simp*: *policy-step-def* 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-dec-det* (*policy-step d*)
let *?dp* = *mk-stationary-det d*
let *?P* = $\sum t.\ l \,\hat{}\, t *_R \mathcal{P}_1$ *?d'* $\hat{}\hat{}$ t

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

If policy iteration terminates, i.e. d = *policy-step* d, then it does so with optimal value.

lemma *policy-step-eq-imp-opt*:
assumes *is-dec-det d d* = *policy-step d*
shows ν_b (*mk-stationary-det d*) = ν_b*-opt*
using *L-*ν*-fix assms eval-policy-step-L*[*unfolded policy-eval-def*]
by (*fastforce intro*: $\mathcal{L}$*-fix-imp-opt*)

end

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

locale *MDP-PI-finite* = *MDP-act-disc arb-act A K r l*
for
A **and**
$K :: 's :: countable \times 'a :: countable \Rightarrow 's\ pmf$ **and** *r l arb-act* +
assumes *fin-states*: *finite* (*UNIV* :: *'s set*) **and** *fin-actions*: $\bigwedge s.$ *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 $?set = \{d.\ \forall x :: 's.\ (x \in UNIV \longrightarrow d\ x \in (\bigcup s.\ A\ s)) \wedge (x \notin UNIV \longrightarrow d\ x = undefined)\}$
have *finite* $(\bigcup s.\ A\ s)$
using *fin-actions fin-states* **by** *blast*
hence *finite ?set*
using *fin-states* **by** (*fastforce intro*: *finite-set-of-finite-funs*)
moreover have $D_D \subseteq ?set$
unfolding *is-dec-det-def* **by** *auto*
ultimately show *?thesis*
using *finite-subset* **by** *auto*
qed

lemma *finite-rel*: *finite* $\{(u, v).\ is\text{-}dec\text{-}det\ u \wedge is\text{-}dec\text{-}det\ v \wedge \nu_b\ (mk\text{-}stationary\text{-}det\ u) >$
$\nu_b\ (mk\text{-}stationary\text{-}det\ v)\}$
proof−
have *aux*: *finite* $\{(u, v).\ is\text{-}dec\text{-}det\ u \wedge is\text{-}dec\text{-}det\ v\}$
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 intro*!: *SUP-is-arg-max boundedI 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) $<$ ν_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) $\le \nu_b$ (*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-disc-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

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* λX. *Least* (λx. $x \in X$)
by *unfold-locales* (*auto intro*: *LeastI*)

end

end

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

13 Value Iteration using Splitting Methods

13.1 Regular Splittings for Matrices and Bounded Linear Functions

definition *is-splitting-blin X Q R* $\longleftrightarrow$
  $X = Q - R \wedge invertible_L\ Q \wedge nonneg\text{-}blinfun\ (inv_L\ Q) \wedge nonneg\text{-}blinfun\ R$

lemma *is-splitting-blinD*[*dest*]:
  assumes *is-splitting-blin X Q R*
  shows $X = Q - R$ $invertible_L\ Q$ $nonneg\text{-}blinfun\ (inv_L\ Q)$ $nonneg\text{-}blinfun\ R$
  using *is-splitting-blin-def assms* **by** *auto*

lemma *is-splitting-blinI*[*intro*]:
  assumes $X = Q - R$ $invertible_L\ Q$ $nonneg\text{-}blinfun\ (inv_L\ Q)$ $nonneg\text{-}blinfun\ R$
  shows *is-splitting-blin X Q R*
  using *is-splitting-blin-def assms* **by** *auto*

13.2 Splitting Methods for MDPs

locale $MDP\text{-}QR = MDP\text{-}att\text{-}\mathcal{L}\ A\ K\ r\ l$
  for $A :: 's::countable \Rightarrow 'a::countable\ set$
    and $K :: ('s \times 'a) \Rightarrow 's\ pmf$
    and $r\ l\ +$
  fixes $Q\ R :: ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow_b real) \Rightarrow_L ('s \Rightarrow_b real)$
  assumes *is-splitting*: $\bigwedge d.\ d \in D_D \Longrightarrow is\text{-}splitting\text{-}blin\ (id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d))\ (Q\ d)\ (R\ d)$
  and *QR-contraction*: $(\bigsqcup d{\in}D_D.\ norm\ (inv_L\ (Q\ d)\ o_L\ R\ d)) < 1$
  and *QR-bdd*: $bdd\text{-}above\ ((\lambda d.\ norm\ (inv_L\ (Q\ d)\ o_L\ R\ d))\ `\ D_D)$
  and *Q-bdd*: $bounded\ ((\lambda d.\ norm\ (inv_L\ (Q\ d)))\ `\ D_D)$
  and *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*: $d \in D_D \Longrightarrow u \leq v \Longrightarrow (inv_L\ (Q\ d))\ u \leq (inv_L\ (Q\ d))\ v$
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*: $d \in D_D \Longrightarrow 0 \leq v \Longrightarrow 0 \leq inv_L\ (Q\ d)\ v$
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*: $d \in D_D \Longrightarrow 0 \leq v \Longrightarrow 0 \leq R\ d\ v$
using *is-splitting-blinD*[*OF is-splitting*]
by (*fastforce simp*: *nonneg-blinfun-nonneg intro*: *nonneg-blinfun-mono*)

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

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

lemma *QR-mono*: $d \in D_D \Longrightarrow u \leq v \Longrightarrow (inv_L\ (Q\ d)\ o_L\ R\ d)\ u \leq (inv_L\ (Q\ d)\ o_L\ R\ d)\ v$
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-bdd*
by (*auto intro*: *cSUP-lessD*[*OF - QR-contraction*])

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*

13.3 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)) $\leq$ *QR-disc*
using *QR-bdd*
by (*auto intro*!: *cSUP-upper*)

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

13.4 Bellman-Operator

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

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

lemma $\mathcal{L}$-*split-bfun-aux*:
assumes $d \in D_D$
shows *norm* (*L-split d v*) $\leq$ ($\bigsqcup d \in D_D$. *norm* (inv_L (Q d))) $*$ r_M + *norm v*
proof −
have *norm* (*L-split d v*) $\leq$ *norm* (inv_L (Q d) ($r\text{-}dec_b$ (*mk-dec-det d*))) + *norm* (inv_L (Q d) (R d v))
by (*simp add*: *blinfun.add-right norm-triangle-ineq*)
also have ... $\leq$ *norm* (inv_L (Q d) ($r\text{-}dec_b$ (*mk-dec-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 ... $\leq$ *norm* (inv_L (Q d) ($r\text{-}dec_b$ (*mk-dec-det d*))) + *norm v*
using *norm-QR-less-one assms*
by (*fastforce intro*!: *mult-left-le-one-le*)
also have ... $\leq$ *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 ... $\leq$ ($\bigsqcup d \in D_D$. *norm* (inv_L (Q d))) $*$ r_M + *norm v*
using *Q-bdd bounded-imp-bdd-above*
by (*auto intro*!: *mult-right-mono cSUP-upper assms simp*: r_M-*nonneg*)
finally show *?thesis*.
qed

lemma *L-split-le*: $d \in D_D \Longrightarrow$ *L-split d v s* $\leq$ ($\bigsqcup d \in D_D$. *norm* (inv_L (Q d))) $*$ r_M + *norm v*
using *le-norm-bfun order.trans*[*OF le-norm-bfun* $\mathcal{L}$-*split-bfun-aux*]
by *auto*

lift-definition $\mathcal{L}_b$-*split* :: ($'s \Rightarrow_b$ *real*) $\Rightarrow$ ($'s \Rightarrow_b$ *real*) **is** $\mathcal{L}$-*split*
unfolding $\mathcal{L}$-*split-def bfun-def*
using *order.trans*[*OF abs-le-norm-bfun* $\mathcal{L}$-*split-bfun-aux*] *ex-dec-det*

by (*fastforce intro*!: *boundedI cSup-abs-le*)

lemma $\mathcal{L}_b$*-split-def'*: $\mathcal{L}_b$*-split v s* = ($\bigsqcup$ *d*$\in$$D_D$. *L-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
$\mathcal{L}_b$*-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* (λ*d*. inv_L (*Q d*) (*r-dec*$_b$ (*mk-dec-det d*) + *R d v*) *s*) (λ*d*. *d* $\in$ D_D) *d*
using *arg-max-ex-split* **by** *blast*
hence $*$: inv_L (*Q d*) (*r-dec*$_b$ (*mk-dec-det d*) + *R d u*) *s* $\leq$ $\mathcal{L}_b$*-split u s*
by (*fastforce simp*: $\mathcal{L}_b$*-split-def'* *is-arg-max-linorder intro*!: *cSUP-upper2 L-split-le*)
have inv_L (*Q d*) (*r-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-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 QR-bdd*
by (*auto simp*: *is-arg-max-linorder intro*!: *mult-right-mono cSUP-upper2*)
finally show *?thesis*.
qed
hence $|$($\mathcal{L}_b$*-split v* − $\mathcal{L}_b$*-split u*) *s*$|$ $\leq$ *QR-disc* $*$ *dist v u* **for** *s*
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*. $\mathcal{L}_b$*-split v* = *v*
(λ*n*. ($\mathcal{L}_b$*-split* ^^ *n*) *v*) $\longrightarrow$ (*THE v*. $\mathcal{L}_b$*-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*: (λ*n*. ($\mathcal{L}_b$*-split* ^^ *n*) *v*) $\longrightarrow$ ν_b*-opt*

proof −
obtain *L* **where** *l-fix*: $\mathcal{L}_b$*-split L = L*
using $\mathcal{L}_b$*-lim(1)* **by** *auto*
have ν_b *(mk-stationary-det d)* $\leq$ *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-dec*$_b$ *(mk-dec-det d) + R d L)* $\leq$ $\mathcal{L}_b$*-split L*
unfolding *less-eq-bfun-def* $\mathcal{L}_b$*-split-def'*
using *d L-split-le* **by** *(auto intro!: bdd-aboveI cSUP-upper2)*
hence *inv*$_L$ *(Q d) (r-dec*$_b$ *(mk-dec-det d) + R d L)* $\leq$ *L*
using *l-fix* **by** *auto*
hence *aux*: *inv*$_L$ *(Q d) (r-dec*$_b$ *(mk-dec-det d))* $\leq$ *(id-blinfun − ?QR) L*
using *that* **by** *(auto simp: blinfun.bilinear-simps le-diff-eq)*
have *inv-eq*: *inv*$_L$ *(id-blinfun − ?QR) = (*$\sum$ *i. ?QR ^^ i)*
using *QR-contraction d norm-QR-less-one*
by *(auto intro!: inv*$_L$*-inf-sum)*
have *summable-QR:summable (*λ*i. norm (?QR ^^ i))*
using *QR-contraction QR-bdd d*
by *(auto simp: a-nonneg*
intro!: summable-comparison-test'[of λ*i. QR-disc^i 0* λ*n. norm ((inv*$_L$ *(Q d) o*$_L$ *R d) ^^n)]*
cSUP-upper2 power-mono order.trans[OF norm-blinfunpow-le])
have *summable (*λ*i. (?QR ^^ i) v s)* **for** *v s*
by *(rule summable-comparison-test'[***where** *g =* λ*i. norm (?QR ^^ 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* $\leq$ *v* $\Longrightarrow$ *0* $\leq$ *(*$\sum$ *i<n. (?QR ^^ 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* $\leq$ *v* $\Longrightarrow$ *0* $\leq$ *(*$\sum$ *i. ((?QR ^^ i) v s))* **for** *v s*
by *(auto intro!: summable-LIMSEQ LIMSEQ-le)*
hence *0* $\leq$ *v* $\Longrightarrow$ *0* $\leq$ *(*$\sum$ *i. ((?QR ^^ i))) v s* **for** *v s*
using *bounded-linear-apply-bfun summable-QR summable-comparison-test'*

by *(subst bounded-linear.suminf[***where** *f = (*λ*i. apply-bfun (blinfun-apply i v) s)])*
(fastforce intro: bounded-linear-compose[of λ*s. apply-bfun s -])+*
hence *0* $\leq$ *v* $\Longrightarrow$ *0* $\leq$ *inv*$_L$ *(id-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-dec*$_b$ *(mk-dec-det d)))*
$\leq$ *(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-dec*$_b$ *(mk-dec-det d))* $\leq$ *L*
using *invertible*$_L$*-inf-sum[OF norm-QR-less-one[OF that]]* **by** *auto*

hence $(inv_L\ (Q\ d\ o_L\ (id\text{-}blinfun - ?QR)))\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d))$
$\leq L$
using *d norm-QR-less-one*
by (*auto simp*: inv_L*-compose*[*OF Q-invertible invertible*$_L$*-inf-sum*])
hence $(inv_L\ (Q\ d - R\ d))\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d)) \leq L$
using *Q-invertible d*
by (*auto simp*: *blinfun-compose-diff-right blinfun-compose-assoc*[*symmetric*])
thus $\nu_b\ (mk\text{-}stationary\text{-}det\ d) \leq L$
by (*auto simp*: ν*-stationary splitting*[*OF that*, *symmetric*] inv_L*-inf-sum*
blincomp-scaleR-right)
qed
hence *opt-le*: $\nu_b\text{-}opt \leq L$
by (*metis* ν*-conserving-def conserving-imp-opt′ ex-improving-det*)
obtain *d* **where** *d*: *is-arg-max* $(\lambda d.\ inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det$
$d) + R\ d\ L)\ s)\ (\lambda d.\ d \in D_D)\ d$ **for** *s*
using *arg-max-ex-split* **by** *blast*
hence $d \in D_D$
unfolding *is-arg-max-linorder* **by** *auto*
have $L = inv_L\ (Q\ d)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d) + R\ d\ L)$
by (*subst l-fix*[*symmetric*]) (*fastforce simp*: $\mathcal{L}_b$*-split-def′ arg-max-SUP*[*OF*
d])
hence $Q\ d\ L = r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d) + R\ d\ L$
by (*metis Q-invertible* ‹$d \in D_D$› *inv-app2′*)
hence $(id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d))\ L = r\text{-}dec_b\ (mk\text{-}dec\text{-}det$
$d)$
using *splitting*[*OF* ‹$d \in D_D$›] **by** (*simp add*: *blinfun.diff-left*)
hence $L = inv_L\ ((id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d)))\ (r\text{-}dec_b$
$(mk\text{-}dec\text{-}det\ d))$
using *invertible*$_L$*-inf-sum*[*OF norm-*$\mathcal{P}_1$*-l-less*] *inv-app1′* **by** *metis*
hence $L = \nu_b\ (mk\text{-}stationary\text{-}det\ d)$
by (*auto simp*: inv_L*-inf-sum* ν*-stationary blincomp-scaleR-right*)
hence $\nu_b\text{-}opt = L$
using *opt-le* ‹$d \in D_D$› *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*]: $\mathcal{L}_b\text{-}split\ \nu_b\text{-}opt = \nu_b\text{-}opt$
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 $eps > 0\ 2 * QR\text{-}disc * dist\ v\ (\mathcal{L}_b\text{-}split\ v) < eps *$
$(1 - QR\text{-}disc)$
shows $dist\ (\mathcal{L}_b\text{-}split\ v)\ \nu_b\text{-}opt < eps\ /\ 2$
proof −
have $(1 - QR\text{-}disc) * dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (\mathcal{L}_b\text{-}split\ v)$
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* $\leq$ *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* $\leq$ *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 $d \in D_D$
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 *argmax-policy-error-bound*:
assumes *am*: $\bigwedge$*s. is-arg-max* (λ*d. L* (*mk-dec-det d*) ($\mathcal{L}_b$ *v*) *s*) (λ*d. d* $\in$ D_D) *d*
shows (*1* $-$ *l*) $*$ *dist* (ν_b (*mk-stationary-det d*)) ($\mathcal{L}_b$ *v*) $\leq$ *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*
proof $-$
have *dist* (ν_b (*mk-stationary-det d*)) ($\mathcal{L}_b$ *v*) = *dist* (*L* (*mk-dec-det d*) (ν_b (*mk-stationary-det d*))) ($\mathcal{L}_b$ *v*)
using *L-ν-fix* **by** *presburger*
also have ... $\leq$ *dist* (*L* (*mk-dec-det d*) (ν_b (*mk-stationary-det d*))) ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) + *dist* ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) ($\mathcal{L}_b$ *v*)
using *dist-triangle* **by** *blast*
also have ... = *dist* (*L* (*mk-dec-det d*) (ν_b (*mk-stationary-det d*))) (*L* (*mk-dec-det d*) ($\mathcal{L}_b$ *v*)) + *dist* ($\mathcal{L}_b$ ($\mathcal{L}_b$ *v*)) ($\mathcal{L}_b$ *v*)
using *$\mathcal{L}_b$-eq-SUP-det* **using** *arg-max-SUP*[*OF assms*, *symmetric*]
by (*auto simp*: *dist-bfun-def*)
also have ... $\leq$ *l* $*$ *dist* (ν_b (*mk-stationary-det d*)) ($\mathcal{L}_b$ *v*) + *l* $*$ *dist* ($\mathcal{L}_b$ *v*) *v*
by (*meson add-mono contraction-L contraction-$\mathcal{L}$*)
finally show *?thesis*
by (*auto simp*: *algebra-simps*)
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 ... ≤ *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 ... ≤ *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*) ≤ *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*) ≤ *QR-disc* ∗ *dist* ($\mathcal{L}_b$*-split v*) *v*
by *auto*
hence *dist* (ν_b *?p*) ($\mathcal{L}_b$*-split v*) ∗ (*1* − *QR-disc*) ≤ *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*) ≤ *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*) ≤ *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*) ≤ *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-att-*$\mathcal{L}$
begin

lemma *inv-one-sub-Q'*:

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

lemma *blinfun-le-trans*: *blinfun-le X Y* $\Longrightarrow$ *blinfun-le Y Z* $\Longrightarrow$ *blinfun-le X Z*
unfolding *blinfun-le-def nonneg-blinfun-def*
by (*fastforce simp*: *blinfun.diff-left*)

lemma *blinfun-leI*[*intro*]: ($\bigwedge v$. $v \geq 0 \Longrightarrow$ *blinfun-apply C v* $\leq$ *blinfun-apply D v*) $\Longrightarrow$ *blinfun-le C D*
unfolding *blinfun-le-def nonneg-blinfun-def*
by (*auto simp*: *algebra-simps blinfun.diff-left*)

lemma *blinfun-pow-mono*: *nonneg-blinfun* (*C* :: ($'c \Rightarrow_b$ *real*) $\Rightarrow_L$ ($'c \Rightarrow_b$ *real*)) $\Longrightarrow$ *blinfun-le C D* $\Longrightarrow$ *blinfun-le* (*C* $\widehat{\ }\widehat{\ }$ *n*) (*D* $\widehat{\ }\widehat{\ }$ *n*)
proof (*induction n*)
case *0*
then show *?case* **by** (*simp add*: *blinfun-le-def nonneg-blinfun-def*)
next
case (*Suc n*)
have $*$: $\bigwedge v$. *0* $\leq v \Longrightarrow$ *blinfun-apply* (*D* $\widehat{\ }\widehat{\ }$ *n*) (*blinfun-apply C v*) $\leq$ *blinfun-apply* (*D* $\widehat{\ }\widehat{\ }$ *n*) (*blinfun-apply D v*)
by (*metis* (*no-types*, *opaque-lifting*) *Suc.prems*(*1*) *Suc.prems*(*2*) *blinfun-apply-mono blinfunpow-nonneg le-left-mono nonneg-blinfun-def nonneg-blinfun-mono*)
thus *?case*
using *blinfun-apply-mono Suc*
by (*intro blinfun-leI*) (*auto simp*: *blinfunpow-assoc blinfunpow-nonneg nonneg-blinfun-def simp del*: *blinfunpow.simps intro*!: *blinfun-apply-mono order.trans*[*OF* - $*$])
qed

lemma *blinfun-le-iff*: *blinfun-le X Y* $\longleftrightarrow$ ($\forall v \geq$ *0*. *X v* $\leq$ *Y v*)
unfolding *blinfun-le-def nonneg-blinfun-def*
by (*auto simp*: *blinfun.diff-left*)

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-le R2 R1*
and *blinfun-le R1* (*l* $*_R$ $\mathcal{P}_1$ *d*)
shows *norm* (inv_L *Q2* o_L *R2*) $\leq$ *norm* (inv_L *Q1* o_L *R1*)
proof $-$
have
inv-Q: inv_L *Q* = ($\sum$ *i*. (*id-blinfun* $-$ *Q*)$\widehat{\ }\widehat{\ }$*i*) *norm* (*id-blinfun* $-$

Q) < *1* **and**
splitting-eq: *id-blinfun* − *Q* = *l* $*_R$ $\mathcal{P}_1$ *d* − *R* **and**
nonneg-Q: *nonneg-blinfun* (*id-blinfun* − *Q*)
if *blinfun-le R* (*l* $*_R$ $\mathcal{P}_1$ *d*)
and *is-splitting-blin* (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ *d*) *Q R* **for** *Q R*
proof −
show *splitting-eq*: *id-blinfun* − *Q* = *l* $*_R$ $\mathcal{P}_1$ *d* − *R*
using *that* **unfolding** *is-splitting-blin-def*
by (*auto simp*: *algebra-simps*)
have *R-nonneg*: *nonneg-blinfun R*
using *that* **by** *blast*
show *nonneg-Q*: *nonneg-blinfun* (*id-blinfun* − *Q*)
using *that* **by** (*simp add*: *blinfun-le-def splitting-eq*)
moreover have *blinfun-le* (*id-blinfun* − *Q*) (*l* $*_R$ $\mathcal{P}_1$ *d*)
using *R-nonneg* **by** (*simp add*: *splitting-eq blinfun-le-def*)
ultimately have *norm* (*id-blinfun* − *Q*) ≤ *norm* (*l* $*_R$ $\mathcal{P}_1$ *d*)
using *blinfun-le-def matrix-le-norm-mono* **by** *fast*
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_L*-inf-sum* **by** *fastforce*
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*: *blinfun-le-trans inv-Q*[*of R2 Q2*] *inv-Q*[*of R1 Q1*])

have *Q1-le-Q2*: *blinfun-le* (*id-blinfun* − *Q1*) (*id-blinfun* − *Q2*)
using *assms* **unfolding** *is-splitting-blin-def blinfun-le-def eq-diff-eq*
by *fastforce*

have (inv_L *Q1*) = (($\sum$ *i*. (*id-blinfun* − *Q1*) ^^ *i*))
using *i1* **by** *auto*
also have … = (($\sum$ *i*. ((*id-blinfun* − *Q1*) ^^ *i*)))
using *summable-inv-Q i1*(*2*)
by *auto*
have *blinfun-le* (($\sum$ *i*. ((*id-blinfun* − *Q1*) ^^ *i*))) ($\sum$ *i*. ((*id-blinfun* − *Q2*) ^^ *i*))
proof −
have *le-n*: $\bigwedge$*n v*. *0* ≤ *v* $\Longrightarrow$ ($\sum$ *i*<*n*. ((*id-blinfun* − *Q1*) ^^ *i*) *v*) ≤ ($\sum$ *i*<*n*. ((*id-blinfun* − *Q2*) ^^ *i*) *v*)
using *nonneg-Q blinfun-pow-mono*[*OF* - *Q1-le-Q2*] *assms*
by (*auto intro*!: *sum-mono simp*: *blinfun-le-iff*)
hence *le-n-elem*: $\bigwedge$*n v*. *0* ≤ *v* $\Longrightarrow$ ($\sum$ *i*<*n*. ((*id-blinfun* − *Q1*) ^^ *i*) *v s*) ≤ ($\sum$ *i*<*n*. ((*id-blinfun* − *Q2*) ^^ *i*) *v s*) **for** *s*
unfolding *less-eq-bfun-def*

by (*simp add*: *sum-apply-bfun*)
have *tt*: ($\lambda n.$ ($\sum i{<}n.$ ((*id-blinfun* $-$ *Q1*) ^^ *i*) *v s*)) $\longrightarrow$ ($\sum i.$ ((*id-blinfun* $-$ *Q1*) ^^ *i*)) *v s*
($\lambda n.$ ($\sum i{<}n.$ ((*id-blinfun* $-$ *Q2*) ^^ *i*) *v s*)) $\longrightarrow$ ($\sum i.$ ((*id-blinfun* $-$ *Q2*) ^^ *i*)) *v s* **for** *v s*
unfolding *blinfun.sum-left*[*symmetric*] *sum-apply-bfun*[*symmetric*]
using *summable-inv-Q*[*OF i1*(*2*)] *summable-inv-Q*[*OF i2*(*2*)]
by (*fastforce intro*: *bfun-tendsto-apply-bfun tendsto-blinfun-apply summable-LIMSEQ*)+
show *?thesis*
unfolding *blinfun-le-iff less-eq-bfun-def*
using *le-n-elem*
by (*auto simp add*: *less-eq-bfunI intro*: *Topological-Spaces.lim-mono*[*OF - tt*(*1*) *tt*(*2*)])
qed
also have ... = (inv_L *Q2*)
using *summable-inv-Q i2*(*2*) *i2* **by** *auto*
finally have *Q1-le-Q2*: *blinfun-le* (inv_L *Q1*) (inv_L *Q2*) **.**

have $*$: *nonneg-blinfun* ((inv_L *Q1*) o_L *R1*) *nonneg-blinfun* ((inv_L *Q2*) o_L *R2*)
using *assms is-splitting-blin-def*
by (*metis blinfun-apply-blinfun-compose nonneg-blinfun-def*)+
have $0 \le$ (*id-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-blinfun* $-$ $l *_R \mathcal{P}_1$ *d*) *1*) $\le$ (inv_L *Q2*) ((*id-blinfun* $-$ $l *_R \mathcal{P}_1$ *d*) *1*)
using *Q1-le-Q2*
by (*simp add*: *blinfun-le-iff*)
hence (inv_L *Q1*) ((*Q1* $-$ *R1*) *1*) $\le$ (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* $\le$ (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* $\le$ (inv_L *Q1* o_L *R1*) *1*
using *assms* **unfolding** *is-splitting-blin-def* **by** *auto*
moreover have $0 \le$ (inv_L *Q2* o_L *R2*) *1*
using $*$ *assms*(*2*) **by** (*fastforce simp*: *less-eq-bfunI intro*!: *nonneg-blinfun-nonneg*)
ultimately have *norm* ((inv_L *Q2* o_L *R2*) *1*) $\le$ *norm* ((inv_L *Q1* o_L *R1*) *1*)
by (*auto simp*: *less-eq-bfun-def norm-bfun-def'* *intro*!: *abs-le-norm-bfun abs-ge-self cSUP-mono bdd-above.I2 intro*: *order.trans*)
thus *norm* ((inv_L *Q2* o_L *R2*)) $\le$ *norm* ((inv_L *Q1* o_L *R1*))
by (*simp add*: $*$ *norm-nonneg-blinfun-one*)
qed

end

end
theory *Splitting-Methods-Fin*
imports
MDP−Rewards.Blinfun-Util
MDP-fin
Splitting-Methods
begin

13.5 Util

definition *upper-triangular-blin* :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real) \Rightarrow bool$ **where**
upper-triangular-blin $X \longleftrightarrow (\forall u\ v\ i.\ (\forall j \geq i.\ apply\text{-}bfun\ v\ j = apply\text{-}bfun\ u\ j) \longrightarrow X\ v\ i = X\ u\ i)$

definition *strict-upper-triangular-blin* :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real) \Rightarrow bool$ **where**
strict-upper-triangular-blin $X \longleftrightarrow (\forall u\ v\ i.\ (\forall j > i.\ apply\text{-}bfun\ v\ j = apply\text{-}bfun\ u\ j) \longrightarrow X\ v\ i = X\ u\ i)$

lemma *upper-triangularD*:
fixes X :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$
and $u\ v$ :: $'a \Rightarrow_b real$
assumes *upper-triangular-blin* X **and** $\bigwedge j.\ i \leq j \Longrightarrow v\ j = u\ j$
shows $X\ v\ i = X\ u\ i$
using *assms* **by** (*auto simp*: *upper-triangular-blin-def*)

lemma *upper-triangularI*[*intro*]:
fixes X :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$
assumes $\bigwedge i\ u\ v.\ (\bigwedge j.\ i \leq j \Longrightarrow apply\text{-}bfun\ v\ j = apply\text{-}bfun\ u\ j) \Longrightarrow X\ v\ i = X\ u\ i$
shows *upper-triangular-blin* X
using *assms* **by** (*fastforce simp*: *upper-triangular-blin-def*)

lemma *strict-upper-triangularD*:
fixes X :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$ **and** $u\ v$ :: $'a \Rightarrow_b real$
assumes *strict-upper-triangular-blin* X **and** $\bigwedge j.\ i < j \Longrightarrow v\ j = u\ j$
shows $X\ v\ i = X\ u\ i$
using *assms* **by** (*auto simp*: *strict-upper-triangular-blin-def*)

lemma *strict-imp-upper-triangular-blin*: *strict-upper-triangular-blin* $X \Longrightarrow$ *upper-triangular-blin* X
unfolding *strict-upper-triangular-blin-def upper-triangular-blin-def*
by *auto*

definition *lower-triangular-blin* :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b$

real) ⇒ *bool* **where**
lower-triangular-blin $X \longleftrightarrow (\forall u\ v\ i.\ (\forall j \leq i.\ \textit{apply-bfun}\ v\ j = \textit{apply-bfun}\ u\ j) \longrightarrow X\ v\ i = X\ u\ i)$

definition *strict-lower-triangular-blin* :: $('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real) \Rightarrow bool$ **where**
strict-lower-triangular-blin $X \longleftrightarrow (\forall u\ v\ i.\ (\forall j < i.\ \textit{apply-bfun}\ v\ j = \textit{apply-bfun}\ u\ j) \longrightarrow X\ v\ i = X\ u\ i)$

lemma *lower-triangularD*:
fixes $X :: ('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$
and $u\ v :: 'a \Rightarrow_b real$
assumes *lower-triangular-blin X* **and** $\bigwedge j.\ i \geq j \Longrightarrow v\ j = u\ j$
shows $X\ v\ i = X\ u\ i$
using *assms* **by** (*auto simp*: *lower-triangular-blin-def*)

lemma *lower-triangularI*[*intro*]:
fixes $X :: ('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$
assumes $\bigwedge i\ u\ v.\ (\bigwedge j.\ i \geq j \Longrightarrow \textit{apply-bfun}\ v\ j = \textit{apply-bfun}\ u\ j) \Longrightarrow X\ v\ i = X\ u\ i$
shows *lower-triangular-blin X*
using *assms* **by** (*fastforce simp*: *lower-triangular-blin-def*)

lemma *strict-lower-triangularI*[*intro*]:
fixes $X :: ('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$
assumes $\bigwedge i\ u\ v.\ (\bigwedge j.\ i > j \Longrightarrow \textit{apply-bfun}\ v\ j = \textit{apply-bfun}\ u\ j) \Longrightarrow X\ v\ i = X\ u\ i$
shows *strict-lower-triangular-blin X*
using *assms* **by** (*fastforce simp*: *strict-lower-triangular-blin-def*)

lemma *strict-lower-triangularD*:
fixes $X :: ('a::linorder \Rightarrow_b real) \Rightarrow_L ('a \Rightarrow_b real)$
and $u\ v :: 'a \Rightarrow_b real$
assumes *strict-lower-triangular-blin X* **and** $\bigwedge j.\ i > j \Longrightarrow v\ j = u\ j$
shows $X\ v\ i = X\ u\ i$
using *assms* **by** (*auto simp*: *strict-lower-triangular-blin-def*)

lemma *strict-imp-lower-triangular-blin*: *strict-lower-triangular-blin X* $\Longrightarrow$ *lower-triangular-blin X*
unfolding *strict-lower-triangular-blin-def lower-triangular-blin-def*
by *auto*

lemma *all-imp-Max*:
assumes *finite X* $X \neq \{\}$ $\forall x \in X.\ P\ (f\ x)$
shows $P\ (MAX\ x \in X.\ f\ x)$
proof −
have $(MAX\ x \in X.\ f\ x) \in f\ `\ X$
using *assms* **by** *auto*
thus *?thesis*

using *assms* **by** *force*
qed

lemma *bounded-mult*:
assumes *bounded* $((f :: 'c \Rightarrow real)$ ‘ $X)$ *bounded* $(g$ ‘ $X)$
shows *bounded* $((\lambda x.\ f\ x * g\ x)$ ‘ $X)$
using *assms mult-mono*
by (*fastforce simp*: *bounded-iff abs-mult intro*!: *mult-mono*)

context *MDP-nat-disc*
begin

13.6 Gauss Seidel Splitting

lemma $\mathcal{P}_1$*-det*: $\mathcal{P}_1$ (*mk-dec-det d*) $v\ s$ = *measure-pmf.expectation* (K $(s,\ d\ s))\ v$
by (*auto simp*: *mk-dec-det-def* $\mathcal{P}_1$*.rep-eq K-st-def bind-return-pmf*)

lift-definition $\mathcal{P}_U$ :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow_b real) \Rightarrow_L nat \Rightarrow_b real$
is $\lambda d\ (v :: nat \Rightarrow_b real).$
(*Bfun* ($\lambda s.$ ($\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* $(\lambda s'.\ s' < s)$ *0 v*) s)))
proof (*standard*, *goal-cases*)
let *?vl* = $\lambda v\ s.$ (*bfun-if* $(\lambda s'.\ s' < s)$ *0 v*)
have *norm-bfun-if-le*: *norm* (*?vl v s*) $\leq$ *norm v* **for** $v :: nat \Rightarrow_b real$
and s
by (*auto simp*: *norm-bfun-def′ bfun-if.rep-eq intro*!: *cSUP-mono bounded-imp-bdd-above*)
hence *is-bfun2*: ($\lambda s.$ $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl v s*) s) $\in$ *bfun* **for** $v ::$ $nat \Rightarrow_b real$ **and** d
by (*intro bfun-normI*) (*fastforce intro*: *order.trans*[*OF norm-blinfun*] *order.trans*[*OF norm-le-norm-bfun*])
case (*1 d u v*)
have $*$: $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl* $(u + v)$ x) x = $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl u x*) x + $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl v x*) x **for** x
by (*auto simp*: *bfun-if-zero-add blinfun.add-right*)
show *?case*
by (*simp add*: $*$ *eq-onp-same-args is-bfun2 plus-bfun.abs-eq*)
case (*2 d r v*)
have *?vl* $(r *_R v)$ x = $r *_R$ *?vl v x* **for** x
by (*auto simp*: *bfun-if.rep-eq*)
hence $*$: $r * \mathcal{P}_1$ (*mk-dec-det d*) (*?vl v x*) x = $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl* $(r *_R v)$ x) x **for** x
by (*auto simp*: *blinfun.scaleR-right*)
show *?case*
using *is-bfun2* **by** (*auto simp*: $*$)
case (*3 d*)
have [*simp*]: ($\lambda s.$ |*apply-bfun x s*|) $\in$ *bfun* **for** $x :: nat \Rightarrow_b real$
unfolding *bfun-def* **by** (*auto intro*!: *boundedI abs-le-norm-bfun*)
have $*$: |($\mathcal{P}_1$ (*mk-dec-det d*)) (*?vl v n*) n| $\leq$ $\mathcal{P}_1$ (*mk-dec-det d*)

(*bfun.Bfun* ($\lambda s.$ |*apply-bfun v s*|)) *n* **for** *v n*
  unfolding $\mathcal{P}_1$-*det*
  by (*subst Bfun-inverse*)
   (*auto simp*: *bfun-if.rep-eq abs-le-norm-bfun*
   intro!: *order.trans*[*OF integral-abs-bound*] *integral-mono AE-pmfI measure-pmf.integrable-const-bound*[*of - norm v*])
 have *norm* (*bfun.Bfun* ($\lambda s.$ (($\mathcal{P}_1$ (*mk-dec-det d*)) (*bfun-if* ($\lambda s'. s' < s$) *0 x*)) *s*)) $\le$ *norm x* **for** *x*
  by (*fastforce simp*: *norm-bfun-def′ Bfun-inverse*[*OF is-bfun2*]
   intro: *cSUP-least order.trans*[*OF* $*$[*of - x*]] *order.trans*[*OF le-norm-bfun*] *order.trans*[*OF norm-blinfun*])
 thus *?case*
  by (*auto intro*: *exI*[*of - 1*])
qed

lift-definition $\mathcal{P}_L$ :: (*nat* $\Rightarrow$ *nat*) $\Rightarrow$ (*nat* $\Rightarrow_b$ *real*) $\Rightarrow_L$ *nat* $\Rightarrow_b$ *real* **is** λd (v :: *nat* $\Rightarrow_b$ *real*).
 Bfun ($\lambda s.$ ($\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* ($\lambda s'. s' \ge s$) *0 v*) *s*))
proof (*standard*, *goal-cases*)
 let *?vl* = $\lambda v\ s.$ (*bfun-if* ($\lambda s'. s' \ge s$) *0 v*)
 have *norm* (*?vl v s*) $\le$ *norm v* **for** *v* :: *nat* $\Rightarrow_b$ *real* **and** *s*
  by (*auto simp*: *norm-bfun-def′ bfun-if.rep-eq intro*!: *cSUP-mono bounded-imp-bdd-above*)
 hence *is-bfun2*: ($\lambda s.$ $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl v s*) *s*) $\in$ *bfun* **for** *v* :: *nat* $\Rightarrow_b$ *real* **and** *d*
  by (*intro bfun-normI*) (*fastforce intro*: *order.trans*[*OF norm-blinfun*] *order.trans*[*OF norm-le-norm-bfun*])
 case (*1 d u v*)
 have $*$: $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl* (*u* + *v*) *x*) *x* = $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl u x*) *x* + $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl v x*) *x* **for** *x*
  by (*auto simp*: *bfun-if-zero-add blinfun.add-right*)
 show *?case*
  by (*simp add*: $*$ *eq-onp-same-args is-bfun2 plus-bfun.abs-eq*)
 case (*2 d r v*)
 have *?vl* ($r *_R v$) *x* = $r *_R$ *?vl v x* **for** *x*
  by (*auto simp*: *bfun-if.rep-eq*)
 hence $*$: $r *$ $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl v x*) *x* = $\mathcal{P}_1$ (*mk-dec-det d*) (*?vl* ($r *_R v$) *x*) *x* **for** *x*
  by (*auto simp*: *blinfun.scaleR-right*)
 show *?case*
  using *is-bfun2* **by** (*auto simp*: $*$)
 case (*3 d*)
 have [*simp*]: ($\lambda s.$ |*apply-bfun x s*|) $\in$ *bfun* **for** *x* :: *nat* $\Rightarrow_b$ *real*
  unfolding *bfun-def* **by** (*auto intro*!: *boundedI abs-le-norm-bfun*)
 have $*$: |($\mathcal{P}_1$ (*mk-dec-det d*)) (*?vl v n*) *n*| $\le$ $\mathcal{P}_1$ (*mk-dec-det d*) (*bfun.Bfun* ($\lambda s.$ |*apply-bfun v s*|)) *n* **for** *v n*
  unfolding $\mathcal{P}_1$-*det*
  by (*subst Bfun-inverse*) (*auto simp*: *bfun-if.rep-eq abs-le-norm-bfun*

intro!: *order.trans*[*OF integral-abs-bound*] *integral-mono AE-pmfI measure-pmf.integrable-const-bound*[*of - norm v*])
have *norm* (*bfun.Bfun* ($\lambda s.$ (($\mathcal{P}_1$ (*mk-dec-det d*)) (*bfun-if* ($\lambda s'.$ $s' \geq s$) *0 x*)) *s*)) $\leq$ *norm x* **for** *x*
by (*fastforce simp*: *norm-bfun-def′ Bfun-inverse*[*OF is-bfun2*]
intro!: *cSUP-least order.trans*[*OF* $*$[*of - x*]] *order.trans*[*OF le-norm-bfun*] *order.trans*[*OF norm-blinfun*])
thus *?case*
by (*auto intro*: *exI*[*of - 1*])
qed

lemma *is-bfun-*$\mathcal{P}$*-raw*[*simp*]:
fixes *v* :: *nat* $\Rightarrow_b$ *real* **and** *d*
shows ($\lambda s.$ $\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* ($\lambda s'.$ $s' \geq s$) *0 v*) *s*) $\in$ *bfun* (**is** *?t1*)
($\lambda s.$ $\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* ($\lambda s'.$ $s' < s$) *0 v*) *s*) $\in$ *bfun* (**is** *?t2*)
proof −
have $*$: *norm* ((*bfun-if* ($\lambda s'.$ $s' \geq s$) *0 v*)) $\leq$ *norm v norm* ((*bfun-if* ($\lambda s'.$ $s' < s$) *0 v*)) $\leq$ *norm v* **for** *v* :: *nat* $\Rightarrow_b$ *real* **and** *s*
by (*auto simp*: *norm-bfun-def′ bfun-if.rep-eq intro*!: *cSUP-mono bounded-imp-bdd-above*)
thus *?t1 ?t2*
by (*fastforce intro*!: *bfun-normI order.trans*[*OF norm-blinfun*] *order.trans*[*OF norm-le-norm-bfun*])+
qed

lemma $\mathcal{P}_U$*-rep-eq′*: $\mathcal{P}_U$ *d v s* = $\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* ((>) *s*) *0 v*) *s*
by (*auto simp*: $\mathcal{P}_U$*.rep-eq*)

lemma $\mathcal{P}_L$*-rep-eq′*: $\mathcal{P}_L$ *d v s* = $\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* (($\leq$) *s*) *0 v*) *s*
by (*auto simp*: $\mathcal{P}_L$*.rep-eq*)

lemma *apply-bfun-plus*: *apply-bfun f a* + *apply-bfun g a* = *apply-bfun* (*f* + *g*) *a*
by *auto*

lemma $\mathcal{P}_1$*-sum-lower-upper*: $\mathcal{P}_1$ (*mk-dec-det d*) = $\mathcal{P}_L$ *d* + $\mathcal{P}_U$ *d*
proof −
have *bfun-if-sum*: *bfun-if* (($\leq$) *s*) *0 v* + *bfun-if* ($\lambda s'.$ $s' < s$) *0 v* = *v* **for** *s* **and** *v* :: *nat* $\Rightarrow_b$ *real*
by (*auto simp*: *bfun-if.rep-eq*)
show *?thesis*
by (*fastforce intro*: *blinfun-eqI simp*: *blinfun.add-left* $\mathcal{P}_L$*-rep-eq′* $\mathcal{P}_U$*-rep-eq′ apply-bfun-plus blinfun.add-right*[*symmetric*] *bfun-if-sum*)
qed

lemma *nonneg-*$\mathcal{P}_U$: *nonneg-blinfun* ($\mathcal{P}_U$ *d*)

using $\mathcal{P}_1$-*nonneg is-bfun-*$\mathcal{P}$*-raw*
by (*auto simp*: *nonneg-blinfun-def* $\mathcal{P}_U$*.rep-eq bfun-if.rep-eq less-eq-bfun-def*)

lemma *nonneg-*$\mathcal{P}_L$: *nonneg-blinfun* ($\mathcal{P}_L$ *d*)
using $\mathcal{P}_1$-*nonneg is-bfun-*$\mathcal{P}$*-raw*
by (*auto simp*: *nonneg-blinfun-def* $\mathcal{P}_L$*.rep-eq bfun-if.rep-eq less-eq-bfun-def*)

lemma *norm-*$\mathcal{P}_L$*-le*: *norm* ($\mathcal{P}_L$ *d*) $\leq$ *norm* ($\mathcal{P}_1$ (*mk-dec-det d*))
using *nonneg-*$\mathcal{P}_L$ $\mathcal{P}_1$*-mono*
by (*fastforce intro*!: *matrix-le-norm-mono simp*: *bfun-if.rep-eq non-neg-blinfun-def blinfun.diff-left* $\mathcal{P}_L$*.rep-eq less-eq-bfun-def*)

lemma *norm-*$\mathcal{P}_U$*-le*: *norm* ($\mathcal{P}_U$ *d*) $\leq$ *norm* ($\mathcal{P}_1$ (*mk-dec-det d*))
using *nonneg-*$\mathcal{P}_U$ $\mathcal{P}_1$*-mono*
by (*fastforce intro*!: *matrix-le-norm-mono simp*: *bfun-if.rep-eq non-neg-blinfun-def blinfun.diff-left* $\mathcal{P}_U$*.rep-eq less-eq-bfun-def*)

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

lemma *norm-*$\mathcal{P}_U$*-le-one*: *norm* ($\mathcal{P}_U$ *d*) $\leq$ *1*
using *norm-*$\mathcal{P}_U$*-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 *norm-*$\mathcal{P}_U$*-less-one*: *norm* ($l *_R \mathcal{P}_U$ *d*) < *1*
using *order.strict-trans1*[*OF mult-left-le disc-lt-one*] *zero-le-disc norm-*$\mathcal{P}_U$*-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*
using $\mathcal{P}_1$*-mono*
by (*auto simp*: *bfun-if.rep-eq* $\mathcal{P}_L$*-rep-eq*$'$ *less-eq-bfun-def intro*!:)

lemma $\mathcal{P}_U$*-le-*$\mathcal{P}_1$: $0 \leq v \Longrightarrow \mathcal{P}_U\ d\ v \leq \mathcal{P}_1$ (*mk-dec-det d*) *v*
using $\mathcal{P}_1$*-mono*
by (*auto simp*: *bfun-if.rep-eq* $\mathcal{P}_U$*-rep-eq*$'$ *less-eq-bfun-def intro*!:)

lemma $\mathcal{P}_U$*-indep*: $d\ s = d'\ s \Longrightarrow \mathcal{P}_U\ d\ v\ s = \mathcal{P}_U\ d'\ v\ s$
unfolding $\mathcal{P}_U$*-rep-eq*$'$ $\mathcal{P}_1$*-det* **by** *simp*
lemma $\mathcal{P}_L$*-indep*: $d\ s = d'\ s \Longrightarrow \mathcal{P}_L\ d\ v\ s = \mathcal{P}_L\ d'\ v\ s$
unfolding $\mathcal{P}_L$*-rep-eq*$'$ $\mathcal{P}_1$*-det* **by** *simp*

lemma $\mathcal{P}_U$*-indep2*:
assumes $d\ s = d'\ s$ ($\bigwedge s'.\ s' \geq s \Longrightarrow$ *apply-bfun v s*$'$ = *apply-bfun v*$'$ *s*$'$)
shows $\mathcal{P}_U\ d\ v\ s = \mathcal{P}_U\ d'\ v'\ s$
using *assms* **by** (*auto simp*: $\mathcal{P}_U$*-rep-eq*$'$ $\mathcal{P}_1$*-det bfun-if.rep-eq cong*:

if-cong)

lemma $\mathcal{P}_L$*-indep2*: $d\ s = d'\ s \Longrightarrow (\bigwedge s'.\ s' < s \Longrightarrow$ *apply-bfun* $v\ s' =$ *apply-bfun* $v'\ s') \Longrightarrow \mathcal{P}_L\ \ d\ v\ s = \mathcal{P}_L\ \ d'\ v'\ s$
  by (*auto simp*: $\mathcal{P}_L$*-rep-eq'* $\mathcal{P}_1$*-det bfun-if.rep-eq cong*: *if-cong*)

lemma $\mathcal{P}_1$*-indep*: $d\ s = d'\ s \Longrightarrow \mathcal{P}_1\ \ d\ v\ s = \mathcal{P}_1\ \ d'\ v\ s$
  by (*simp add*: *K-st-def* $\mathcal{P}_1$*.rep-eq*)

lemma $\mathcal{P}_U$*-upper*: *upper-triangular-blin* ($\mathcal{P}_U\ \ d$)
  using $\mathcal{P}_U$*-indep2* **by** *fastforce*

lemma $\mathcal{P}_L$*-strict-lower*: *strict-lower-triangular-blin* ($\mathcal{P}_L\ \ d$)
  using $\mathcal{P}_L$*-indep2* **by** *fastforce*

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

lemma *nonneg-R-GS*: *nonneg-blinfun* (*R-GS* d)
  by (*simp add*: *R-GS-def nonneg-*$\mathcal{P}_U$ *nonneg-blinfun-scaleR*)

lemma *splitting-gauss*: *is-splitting-blin* (*id-blinfun* $- \ l *_R \mathcal{P}_1$ (*mk-dec-det* d)) (*Q-GS* d) (*R-GS* d)
  unfolding *is-splitting-blin-def*
proof *safe*
  show *nonneg-blinfun* (*R-GS* d)
    using *nonneg-R-GS*.
next
  show *id-blinfun* $- \ l *_R \mathcal{P}_1$ (*mk-dec-det* d) = *Q-GS* d $-$ *R-GS* d
    using $\mathcal{P}_1$*-sum-lower-upper*
    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) = $(\sum i.\ (l *_R \mathcal{P}_L\ \ d)\ \widehat{\ }\widehat{\ }\ i)$
    using *inv*$_L$*-inf-sum n-le* **unfolding** *Q-GS-def* **by** *blast*
  have *nonneg-blinfun* (*R-GS* $d\ \widehat{\ }\widehat{\ }\ i$) **for** i
    using *nonneg-R-GS* **by** (*auto simp*: *nonneg-blinfun-def intro*: *blinfunpow-nonneg*)
  have s: *summable* $(\lambda k.\ ((l *_R \mathcal{P}_L\ \ d)\widehat{\ }\widehat{\ }k))$
    using *summable-inv-Q*[*of Q-GS d*] *norm-*$\mathcal{P}_L$*-less-one*
      by (*simp add*: *Q-GS-def algebra-simps blinfun.scaleR-left blincomp-scaleR-right*)
  hence s': *summable* $(\lambda k.\ ((l *_R \mathcal{P}_L\ \ d)\widehat{\ }\widehat{\ }k)\ v)$ **for** v

using *tendsto-blinfun-apply*
by (*auto simp*: *summable-def sums-def blinfun.sum-left*[*symmetric*])
hence s'': *summable* (λk. (($l *_R \mathcal{P}_L\ d$)$\widehat{\ }\widehat{\ }k$) $v\ s$) **for** $v\ s$
by (*fastforce simp*: *summable-def sums-def sum-apply-bfun*[*symmetric*] *intro*: *bfun-tendsto-apply-bfun*)
have $0 \leq (\sum k.\ ((l *_R \mathcal{P}_L\ d)\widehat{\ }\widehat{\ }k)\ v\ s)$ **if** $v \geq 0$ **for** $v\ s$
by (*rule suminf-nonneg*[*OF* s''])
(*metis blinfunpow-nonneg that less-eq-bfun-def nonneg-*$\mathcal{P}_L$ *nonneg-blinfun-nonneg nonneg-blinfun-scaleR zero-bfun.rep-eq zero-le-disc*)
hence $0 \leq (\sum k.\ ((l *_R \mathcal{P}_L\ d)\widehat{\ }\widehat{\ }k)\ v)$ **if** $v \geq 0$ **for** v
using *that* **unfolding** *less-eq-bfun-def suminf-apply-bfun*[*OF* s'] **by** *auto*
hence *nonneg-blinfun* ($\sum k.\ ((l *_R \mathcal{P}_L\ d)\widehat{\ }\widehat{\ }k)$)
unfolding *nonneg-blinfun-def* **by** (*simp add*: *blinfun-apply-suminf s*)
thus *nonneg-blinfun* (inv_L (*Q-GS d*))
by (*simp add*: ‹inv_L (*Q-GS d*) = ($\sum i.\ (l *_R \mathcal{P}_L\ d)\ \widehat{\ }\widehat{\ }\ i$)›)
qed

abbreviation $r\text{-}det_b\ d \equiv r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d)$

definition *GS-inv d v* = inv_L (*Q-GS d*) ($r\text{-}dec_b$ (*mk-dec-det d*) + *R-GS d v*)

Q-GS can be expressed as an infinite sum of $\mathcal{P}_L$.

lemma *inv-Q-suminf*: inv_L (*Q-GS d*) = ($\sum k.\ (l *_R (\mathcal{P}_L\ d))\ \widehat{\ }\widehat{\ }\ k$)
unfolding *Q-GS-def* **using** inv_L*-inf-sum norm-*$\mathcal{P}_L$*-less-one* **by** *blast*

This recursive definition mimics the computation of the GS iteration.

lemma *GS-inv-rec*: *GS-inv d v* = $r\text{-}det_b\ d + l *_R (\mathcal{P}_U\ d\ v + \mathcal{P}_L\ d$ (*GS-inv d v*))
proof −
have *Q-GS d* (*GS-inv d v*) = $r\text{-}det_b\ d$ + *R-GS d v*
using *splitting-gauss*[*of d*] **unfolding** *GS-inv-def is-splitting-blin-def* **by** *simp*
thus *?thesis*
unfolding *R-GS-def Q-GS-def* **by** (*auto simp*: *algebra-simps blinfun.diff-left blinfun.scaleR-left*)
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*
using *assms*
proof (*induction s arbitrary*: $d\ d'\ v$ *rule*: *less-induct*)
case (*less x*)
have $r\text{-}det_b\ d\ x = r\text{-}det_b\ d'\ x$

by (*simp add*: *less.prems*)
moreover have $\mathcal{P}_U$ *d v x* = $\mathcal{P}_U$ *d′ v x*
by (*meson* $\mathcal{P}_U$*-indep le-refl less.prems*)
moreover have $\mathcal{P}_L$ *d* (*GS-inv d v*) *x* = $\mathcal{P}_L$ *d′* (*GS-inv d′ v*) *x*
using $\mathcal{P}_L$*-indep2 less.IH less.prems* **by** *fastforce*
ultimately show *?case*
by (*subst GS-inv-rec*[*of d*], *subst GS-inv-rec*[*of d′*]) *auto*
qed

lemma *is-am-GS-inv-extend*:
assumes $\bigwedge s.\ s < k \Longrightarrow$ *is-arg-max* ($\lambda d.$ *GS-inv d v s*) ($\lambda d.\ d \in D_D$) *d*
and *is-arg-max* ($\lambda a.$ *GS-inv* (*d* ($k := a$)) *v k*) ($\lambda a.\ a \in A\ k$) *a*
and $s \le k$
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 *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 $y \in D_D$
let *?d* = *d*($k :=$ *y k*)
have *GS-inv y v k* $\le$ *GS-inv ?d v k*
proof −
have $\mathcal{P}_L$ *y* (*GS-inv y v*) *k* = ($\mathcal{P}_L$ *?d* (*GS-inv y v*)) *k*
by (*auto intro*!: $\mathcal{P}_L$*-indep2 GS-indep-high-states*)
also have … $\le$ ($\mathcal{P}_L$ *?d* (*bfun-if* ($\lambda s.\ s < k$) (*GS-inv d v*) (*GS-inv y v*))) *k*
using *assms*(*1*) ‹$y \in D_D$›
by (*fastforce intro*!: *nonneg-blinfun-mono*[*THEN less-eq-bfunD*] *simp*: *bfun-if.rep-eq less-eq-bfun-def nonneg-*$\mathcal{P}_L$)
also have … = ($\mathcal{P}_L$ *?d* (*GS-inv d v*)) *k*
by (*metis* (*no-types*, *lifting*) $\mathcal{P}_L$*-strict-lower bfun-if.rep-eq strict-lower-triangularD*)
also have … = $\mathcal{P}_L$ *?d* (*GS-inv ?d v*) *k*
using *GS-indep-high-states* $\mathcal{P}_L$*-strict-lower*
by (*fastforce intro*: *strict-lower-triangularD*[*OF* $\mathcal{P}_L$*-strict-lower*])
finally have $\mathcal{P}_L$ *y* (*GS-inv y v*) *k* $\le$ $\mathcal{P}_L$ *?d* (*GS-inv ?d v*) *k*.
thus *?thesis*
by (*subst GS-inv-rec*[*of y*], *subst GS-inv-rec*[*of ?d*])
(*auto simp*: $\mathcal{P}_U$*-indep*[*of y - ?d*] *intro*!: *mult-left-mono*)
qed
thus *GS-inv y v k* $\le$ *GS-inv* (*d*($k := a$)) *v k*
using *is-arg-max-linorderD*[*OF assms*(*2*)] ‹$y \in D_D$› *is-dec-det-def* **by** *fastforce*
next
show *d*($k := a$) $\in D_D$
using *assms* **by** (*auto simp*: *is-dec-det-def is-arg-max-linorder*)
qed

thus *?thesis*
using *assms GS-indep-high-states*[*of s d* (*k* := *a*) *d*] **by** (*cases s* < *k*) *fastforce*+
qed

lemma *is-am-GS-inv-extend′*:
assumes $\bigwedge s.\ s < k \Longrightarrow$ *is-arg-max* ($\lambda d.$ *GS-inv d v s*) ($\lambda d.\ d \in D_D$) *d*
and *is-arg-max* ($\lambda a.$ *GS-inv* (*d* (*k* := *a*)) *v k*) ($\lambda a.\ a \in A\ k$) (*d k*)
and $s \le k$
and $d \in D_D$
shows *is-arg-max* ($\lambda d.$ *GS-inv d v s*) ($\lambda d.\ d \in D_D$) *d*
using *assms is-am-GS-inv-extend*[*of k - d d k*] **by** *auto*

lemma *norm-$\mathcal{P}_L$-pow*: *norm* (($\sum k.$ ($l *_R \mathcal{P}_L\ d$) ^^ *k*)) $\le$ *1* / (*1*−*l*)
by (*fastforce simp*: *norm-$\mathcal{P}_L$-le-one mult-left-le power-mono suminf-geometric*
intro: *order.trans*[*OF summable-norm*] *summable-comparison-test′*[*of* λn :: *nat*. *l* ^ *n 0*]
order.trans[*OF suminf-le*[*of* - $\lambda n.$ *l*^*n*]] *order.trans*[*OF norm-blinfunpow-le*])

lemma *summable-disc-$\mathcal{P}_L$*: *summable* ($\lambda i.$ (($l *_R \mathcal{P}_L\ d$)^^*i*))
by (*metis add-diff-cancel-left′ diff-add-cancel norm-$\mathcal{P}_L$-less-one summable-inv-Q*)

lemma *norm-$\mathcal{P}_L$-pow-elem*: *norm* (($\sum k.$ ($l *_R \mathcal{P}_L\ d$) ^^ *k*) *v*) $\le$ *norm v* / (*1*−*l*)
using *norm-$\mathcal{P}_L$-le-one*
by (*subst blinfun-apply-suminf*[*symmetric*, *OF summable-disc-$\mathcal{P}_L$*])
(*auto simp*: *blincomp-scaleR-right blinfun.scaleR-left intro*!: *power-le-one sum-disc-bound′*
order.trans[*OF norm-blinfunpow-le*] *order.trans*[*OF norm-blinfun*] *mult-left-le-one-le*)

lemma *norm-Q-GS*: *norm* (inv_L (*Q-GS d*) *v*) $\le$ *norm v* / (*1*−*l*)
using *inv-Q-suminf norm-$\mathcal{P}_L$-pow-elem* **by** *auto*

lemma *norm-GS-inv-le*: *norm* (*GS-inv d v*) $\le$ (r_M + *l* ∗ *norm v*) / (*1* − *l*)
proof −
have *0* < (*1* − *l*)
using *disc-lt-one* **by** *auto*
thus *?thesis*
unfolding *GS-inv-def inv-Q-suminf R-GS-def*
using *norm-r-dec-le norm-$\mathcal{P}_U$-le-one order.strict-implies-order*[*OF disc-lt-one*]
by (*intro order.trans*[*OF norm-$\mathcal{P}_L$-pow-elem*])
(*auto simp*: *blinfun.scaleR-left intro*!: *mult-left-le-one-le order.trans*[*OF norm-blinfun*] *mult-left-mono divide-right-mono order.trans*[*OF norm-triangle-ineq*] *add-mono*)

qed

lemma *GS-inv-elem-eq*: *GS-inv d v s* = (*r-det*$_b$ *d* + *l* $*_R$ ($\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* ($\lambda s'$. $s \le s'$) *v* (*GS-inv d v*)))) *s*
proof −
have *bfun-if* ($\lambda s'$. $s' < s$) *0 v* + *bfun-if* (($\le$) *s*) *0* (*GS-inv d v*) = *bfun-if* (($\le$) *s*) *v* (*GS-inv d v*)
by (*auto simp*: *bfun-if.rep-eq*)
thus *?thesis*
by (*subst GS-inv-rec*) (*auto simp*: $\mathcal{P}_U$*-rep-eq'* $\mathcal{P}_L$*-rep-eq' apply-bfun-plus blinfun.add-right*[*symmetric*])
qed

13.7 Maximizing Decision Rule for GS

lemma *ex-GS-inv-arg-max*: $\exists$ *a*. *is-arg-max* (λa. *GS-inv* (*d*(*s*:= *a*)) *v s*) (λa. $a \in A$ *s*) *a*
proof −
have $\exists$ *a*. *is-arg-max* (λa. (*r-det*$_b$ (*d*(*s* := *a*)) + *l* $*_R$ ($\mathcal{P}_1$ (*mk-dec-det* (*d*(*s* := *a*))) (*bfun-if* ($\lambda s'$. $s \le s'$) *v* (*GS-inv d v*)))) *s*) (λa. $a \in A$ *s*) *a*
using *Sup-att* **by** (*auto simp*: $\mathcal{P}_1$*-det max-L-ex-def has-arg-max-def*)
moreover have(*bfun-if* ($\lambda s'$. $s \le s'$) *v* (*GS-inv* (*d*(*s* := *a*)) *v*)) = (*bfun-if* ($\lambda s'$. $s \le s'$) *v* (*GS-inv d v*)) **for** *a*
using *GS-indep-high-states* **by** (*fastforce simp*: *bfun-if.rep-eq*)
ultimately show *?thesis*
by (*auto simp*: *GS-inv-elem-eq*)
qed

This shows that there always exists a decision rule that maximized *GS-inv* for all states simultaneously.

abbreviation *some-dec* $\equiv$ (*SOME d*. $d \in D_D$)

fun *d-GS-least* :: (*nat* $\Rightarrow_b$ *real*) $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
d-GS-least v (*0*::*nat*) = (*LEAST a*. *is-arg-max* (λa. *GS-inv* (*some-dec*(*0* := *a*)) *v 0*) (λa. $a \in A$ *0*) *a*) |
d-GS-least v (*Suc n*) = (*LEAST a*. *is-arg-max* (λa. *GS-inv* ((λs. *if* $s <$ *Suc n then d-GS-least v s else SOME a*. $a \in A$ *s*)(*Suc n*:= *a*)) *v* (*Suc n*)) (λa. $a \in A$ (*Suc n*)) *a*)

lemma *d-GS-least-is-dec*: *d-GS-least v* $\in D_D$
unfolding *is-dec-det-def*
proof *safe*
fix *s*
show *d-GS-least v s* $\in A$ *s*
using *LeastI-ex*[*OF ex-GS-inv-arg-max*] **by** (*cases s*) *auto*
qed

lemma *d-GS-least-eq*: *d-GS-least v n* = (*LEAST a*. *is-arg-max* (λa.

GS-inv ((*d-GS-least v*)(*n* := *a*)) *v n*) (λ*a*. *a* $\in$ *A n*) *a*)
proof (*induction n*)
  case *0*
  have *aux*: *apply-bfun* (*GS-inv* ((*d-GS-least v*)(*0* := *a*)) *v*) *0* = *GS-inv* (*some-dec*(*0* := *a*)) *v 0* **for** *a*
    by (*auto intro*: *GS-indep-high-states*)
  show *?case*
    unfolding *aux* **by** *auto*
next
  case (*Suc n*)
  have *aux*: *GS-inv* ((λ*s*. *if s* < *Suc n then d-GS-least v s else SOME a*. *a* $\in$ *A s*)(*Suc n* := *a*)) *v* (*Suc n*) =
      (*GS-inv* ((*d-GS-least v*)(*Suc n* := *a*)) *v*) (*Suc n*) **for** *a*
    using *GS-indep-high-states* **by** *fastforce*
  show *?case*
    unfolding *aux*[*symmetric*] **by** *simp*
qed

lemma *d-GS-least-is-arg-max*: *is-arg-max* (λ*d*. *GS-inv d v s*) (λ*d*. *d* $\in$ D_D) (*d-GS-least v*)
proof (*induction s rule*: *nat-less-induct*)
  case (*1 n*)
  assume $\forall$ *m*<*n*. *is-arg-max* (λ*d*. *apply-bfun* (*GS-inv d v*) *m*) (λ*d*. *d* $\in$ D_D) (*d-GS-least v*)
  show *?case*
   using *is-am-GS-inv-extend'*[*of n* - (*d-GS-least v*)] *1 d-GS-least-is-dec*

   by (*fastforce simp*: *ex-GS-inv-arg-max d-GS-least-eq*[*of v n*] *LeastI-ex*)
qed

13.8 Gauss-Seidel is a Valid Regular Splitting

lemma *norm-GS-QR-le-disc*: *norm* (inv_L (*Q-GS d*) o_L *R-GS d*) $\leq$ *l*
proof –
 have *norm* (inv_L (*Q-GS d*) o_L *R-GS d*) $\leq$ *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 nonneg-blinfun-def*
      by (*auto simp*: $\mathcal{P}_1$*-pos blinfun.scaleR-left scaleR-nonneg-nonneg*)
  next
    case *3*
    then show *?case*
    by (*simp add*: *R-GS-def* $\mathcal{P}_U$*-le-*$\mathcal{P}_1$ *blinfun-le-iff scaleR-blinfun.rep-eq scaleR-left-mono*)
  qed (*auto simp*: *splitting-gauss blinfun-le-iff*)
  also have ... $\leq$ *l*
    by *auto*

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

lemma *ex-GS-arg-max-all*: $\exists$ *d. is-arg-max* (λ*d. GS-inv d v s*) (λ*d. d* $\in D_D$) *d*
using *d-GS-least-is-arg-max* **by** *blast*

sublocale *GS*: *MDP-QR A K r l Q-GS R-GS*
proof –
have ($\bigsqcup d \in D_D$. *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*
using *norm-GS-QR-le-disc norm-*$\mathcal{P}_L$*-pow d-GS-least-is-arg-max*
by *unfold-locales* (*fastforce intro*!: *bdd-above.I2 simp*: *splitting-gauss bounded-iff inv-Q-suminf GS-inv-def*)+
qed

13.9 Termination

lemma *dist-*$\mathcal{L}_b$*-split-lt-dist-opt*: *dist v* (*GS.*$\mathcal{L}_b$*-split v*) $\leq$ *2* $*$ *dist v* ν_b*-opt*
proof –
have *le1*: *dist v* (*GS.*$\mathcal{L}_b$*-split v*) $\leq$ *dist v* ν_b*-opt* + *dist* (*GS.*$\mathcal{L}_b$*-split v*) ν_b*-opt*
by (*simp add*: *dist-triangle dist-commute*)
have *le2*: *dist* (*GS.*$\mathcal{L}_b$*-split 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*)

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 *gs-rel-dec*:
assumes *l* $\neq$ *0 GS.*$\mathcal{L}_b$*-split v* $\neq \nu_b$*-opt*
shows $\lceil$*log* (*1* / *l*) (*dist* (*GS.*$\mathcal{L}_b$*-split v*) ν_b*-opt*) − *c*$\rceil$ < $\lceil$*log* (*1* / *l*) (*dist v* ν_b*-opt*) − *c*$\rceil$
proof –
have *log* (*1* / *l*) (*dist* (*GS.*$\mathcal{L}_b$*-split v*) ν_b*-opt*) − *c* $\leq$ *log* (*1* / *l*) (*l* $*$ *dist v* ν_b*-opt*) − *c*
proof (*intro Transcendental.log-mono diff-mono*)
show *dist* (*GS.*$\mathcal{L}_b$*-split v*) ν_b*-opt* $\leq$ *l* $*$ *dist v* ν_b*-opt*

using *GS.*$\mathcal{L}_b$*-split-contraction*[*of* - ν_b*-opt*]
by (*smt* (*verit*, *ccfv-SIG*) *GS.*$\mathcal{L}_b$*-split-fix GS-QR-disc-le-disc mult-right-mono zero-le-dist*)
show $1 < 1/l$
by (*metis* ‹$l \neq 0$› *disc-lt-one less-divide-eq-1-pos less-le zero-le-disc*)
qed (*use assms* **in** *auto*)
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 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 $log\ (1\ /\ l)\ (dist\ (GS.\mathcal{L}_b\text{-}split\ v)\ \nu_b\text{-}opt) - c \leq log\ (1\ /\ l)\ (dist\ v\ \nu_b\text{-}opt) - 1 - c$ **.**
thus *?thesis*
by *linarith*
qed

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

function *gs-iteration* :: $real \Rightarrow (nat \Rightarrow_b real) \Rightarrow (nat \Rightarrow_b real)$ **where**
gs-iteration eps v =
(*if* $2 * l * dist\ v\ (GS.\mathcal{L}_b\text{-}split\ v) < eps * (1 - l) \vee eps \leq 0$ *then* $GS.\mathcal{L}_b\text{-}split\ v$ *else gs-iteration eps* $(GS.\mathcal{L}_b\text{-}split\ v)$)
by *auto*
termination
proof (*relation Wellfounded.measure gs-measure*; *cases* $l = 0$)
case *False*
fix *eps v*
assume *h*: $\neg\ (2 * l * dist\ v\ (GS.\mathcal{L}_b\text{-}split\ v) < eps * (1 - l) \vee eps \leq 0)$
show $((eps,\ GS.\mathcal{L}_b\text{-}split\ v),\ eps,\ v) \in$ *Wellfounded.measure gs-measure*
proof −
have *gt-zero*[*simp*]: $l \neq 0$ $eps > 0$ **and** *dist-ge*: $eps * (1 - l) \leq dist\ v\ (GS.\mathcal{L}_b\text{-}split\ v) * (2 * l)$
using *h* **by** (*auto simp*: *algebra-simps*)
have *v-not-opt*: $v \neq \nu_b\text{-}opt$
using *h* **by** *auto*
have $log\ (1\ /\ l)\ (eps * (1 - l)\ /\ (8 * l)) < log\ (1\ /\ l)\ (dist\ v\ \nu_b\text{-}opt)$
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.*$\mathcal{L}_b$*-split 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*

13.10 Optimality

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

lemma *contraction-*$\mathcal{L}$*-split-dist*: (*1 − l*) * *dist v* ν_b*-opt* ≤ *dist v* (*GS.*$\mathcal{L}_b$*-split v*)
using *GS-QR-disc-le-disc*
by (*fastforce simp*: *THE-fix-GS*
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.*$\mathcal{L}_b$*-split v*) < *eps * (1−l)*
shows *dist* (*GS.*$\mathcal{L}_b$*-split v*) ν_b*-opt < eps / 2*
proof −
have *dist v* ν_b*-opt* ≤ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) / (*1 − l*)
using *contraction-*$\mathcal{L}$*-split-dist*
by (*simp add*: *mult.commute pos-le-divide-eq*)
hence *2 * l * dist v* ν_b*-opt* ≤ *2 * l ** (*dist v* (*GS.*$\mathcal{L}_b$*-split 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* ν_b*-opt < eps*
by (*auto simp*: *assms*(*2*) *pos-divide-less-eq intro*: *order.strict-trans1*)
hence *dist v* ν_b*-opt * l < eps / 2*
by *argo*
hence *: *l * dist v* ν_b*-opt < eps / 2*
by (*auto simp*: *algebra-simps*)
show *dist* (*GS.*$\mathcal{L}_b$*-split v*) ν_b*-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

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 *find-policy-error-bound-gs*:
assumes *eps* > *0 2* ∗ *l* ∗ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) < *eps* ∗ (*1*−*l*)
shows *dist* (ν_b (*mk-stationary-det* (*d-GS-least* (*GS.*$\mathcal{L}_b$*-split v*)))) ν_b*-opt* < *eps*
proof (*rule GS.find-policy-QR-error-bound*[*OF assms*(*1*)])
have *2* ∗ *GS.QR-disc* ∗ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) ≤ *2* ∗ *l* ∗ *dist v* (*GS.*$\mathcal{L}_b$*-split 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.*$\mathcal{L}_b$*-split 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* (λ*d.* (*GS.L-split d* (*GS.*$\mathcal{L}_b$*-split v*)) *s*) (λ*d. d* ∈ D_D) (*d-GS-least* (*GS.*$\mathcal{L}_b$*-split v*)) **for** *s*
unfolding *GS-inv-def*[*symmetric*] **using** *d-GS-least-is-arg-max* **by** *auto*
qed

definition *vi-gs-policy eps v* = *d-GS-least* (*gs-iteration eps v*)

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

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

14 Preparation for Codegen

lemma $\mathcal{L}_b$*-split-eq-GS-inv*: *GS.*$\mathcal{L}_b$*-split v* = *GS-inv* (*d-GS-least v*) *v*
using *arg-max-SUP*[*OF d-GS-least-is-arg-max*]
by (*auto simp*: *GS.*$\mathcal{L}_b$*-split.rep-eq GS.*$\mathcal{L}$*-split-def GS-inv-def*[*symmetric*])

lemma $\mathcal{L}_b$-*split-GS*: *GS*.$\mathcal{L}_b$-*split* v s = ($\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$ *measure-pmf.expectation* (K (s, a)) (*bfun-if* ($\lambda s'.\ s' < s$) (*GS*.$\mathcal{L}_b$-*split* v) v))
proof –
let *?d* = *d-GS-least* v
have *GS*.$\mathcal{L}_b$-*split* v s = *GS-inv* *?d* v s
using $\mathcal{L}_b$-*split-eq-GS-inv* **by** *auto*
also have ... = ($\bigsqcup a \in A\ s.$ *GS-inv* (*?d*($s := a$)) v s)
proof (*subst arg-max-SUP*[*symmetric*, *of* - - *?d s*])
show *is-arg-max* ($\lambda a.$ (*GS-inv* (*?d*($s := a$)) v) s) ($\lambda x.\ x \in A\ s$) (*?d* s)
using *d-GS-least-eq A-ne A-fin MDP-reward-Util.arg-max-on-in*
by (*auto simp*: *LeastI-ex finite-is-arg-max*)
qed *fastforce*
also have ... = ($\bigsqcup a \in A\ s.$ (*r-det*$_b$ (*?d*($s := a$)) + $l *_R$ ($\mathcal{P}_U$ (*?d*($s := a$)) v + $\mathcal{P}_L$ (*?d*($s := a$)) (*GS-inv* (*?d*($s := a$)) v))) s)
using *GS-inv-rec* **by** *auto*
also have ... = ($\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$ ($\mathcal{P}_U$ (*?d*($s := a$)) v + $\mathcal{P}_L$ (*?d*($s := a$)) (*GS-inv* (*?d*($s := a$)) v)) s)
by *auto*
also have ... = ($\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$ ($\mathcal{P}_U$ (*?d*($s := a$)) v + $\mathcal{P}_L$ (*?d*($s := a$)) (*GS-inv* *?d* v)) s)
proof –
have $\mathcal{P}_L$ (*?d*($s := a$)) (*GS-inv* (*?d*($s := a$)) v) s = $\mathcal{P}_L$ (*?d*($s := a$)) (*GS-inv* (*?d*) v) s **for** a
by (*fastforce intro*!: *GS-indep-high-states strict-lower-triangularD*[*OF* $\mathcal{P}_L$-*strict-lower*, *of s* - - (*?d*($s := a$))])
thus *?thesis*
by *auto*
qed
also have ... = ($\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$ $\mathcal{P}_1$ (*mk-dec-det* (*?d*($s := a$))) (*bfun-if* ($\lambda s'.\ s' < s$) (*GS-inv* *?d* v) v) s)
proof –
have (*bfun-if* ($\lambda s'.\ s' < s$) *0* v + *bfun-if* (($\leq$) s) *0* (*GS-inv* *?d* v)) = (*bfun-if* ($\lambda s'.\ s' < s$) (*GS-inv* *?d* v) v)
by (*auto simp*: *bfun-if.rep-eq*)
thus *?thesis*
by (*auto simp*: $\mathcal{P}_L$.*rep-eq* $\mathcal{P}_U$.*rep-eq* *blinfun.add-right*[*symmetric*] *apply-bfun-plus*)
qed
also have ... = ($\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$ $\mathcal{P}_1$ (*mk-dec-det* (*?d*($s := a$))) (*bfun-if* ($\lambda s'.\ s' < s$) (*GS*.$\mathcal{L}_b$-*split* v) v) s)
using $\mathcal{L}_b$-*split-eq-GS-inv* **by** *presburger*
also have ... = ($\bigsqcup a \in A\ s.\ r\ (s,\ a) + l *$ *measure-pmf.expectation* (K (s, a)) (*bfun-if* ($\lambda s'.\ s' < s$) (*GS*.$\mathcal{L}_b$-*split* v) v))
using $\mathcal{P}_1$-*det* **by** *auto*
finally show *?thesis*.
qed

lemma $\mathcal{L}_b$*-split-GS-iter*:
assumes $\bigwedge s'.\ s' < s \Longrightarrow v'\ s' = GS.\mathcal{L}_b\text{-}split\ v\ s'$ $\bigwedge s'.\ s' \geq s \Longrightarrow v'\ s' = v\ s'$
shows $GS.\mathcal{L}_b\text{-}split\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v'\ s)$
unfolding $\mathcal{L}_b$*-split-GS* **using** *assms*[*symmetric*] **by** (*auto simp*: *bfun-if.rep-eq cong*: *if-cong*)

function *GS-rec-upto* **where**
GS-rec-upto n v s = (
if $n \leq s$
then v
else GS-rec-upto n ($v(s := (\bigsqcup a \in A\ s.\ r\ (s, a) + l * measure\text{-}pmf.expectation\ (K\ (s, a))\ v)))\ (Suc\ s)$)
by *auto*
termination
by (*relation Wellfounded.measure* ($\lambda(n,v,s).\ n - s$)) *auto*

lemmas *GS-rec-upto.simps*[*simp del*]

lemma *GS-rec-upto-ge*:
assumes $s' \geq n$
shows *GS-rec-upto n v s* $s' = v\ s'$
using *assms*
by (*induction s arbitrary*: s' *rule*: *GS-rec-upto.induct*) (*fastforce simp add*: *GS-rec-upto.simps*)

lemma *GS-rec-upto-less*:
assumes $s > s'$
shows *GS-rec-upto n v s* $s' = v\ s'$
using *assms*
by (*induction s arbitrary*: s' *rule*: *GS-rec-upto.induct*) (*auto simp*: *GS-rec-upto.simps*)

lemma *GS-rec-upto-eq*:
assumes $s < n$
shows *GS-rec-upto n v s s* = $(\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
using *assms*
proof (*induction n v s rule*: *GS-rec-upto.induct*)
case (*1 n v s*)
then show *?case*
using *GS-rec-upto-less* **by** (*cases Suc s* < *n*) (*auto simp add*: *GS-rec-upto.simps*)
qed

lemma *GS-rec-upto-Suc*:
assumes $s' < n$
shows *GS-rec-upto* (*Suc n*) *v s* s' = *GS-rec-upto n v s* s'
using *assms*
proof (*induction n v s arbitrary*: s' *rule*: *GS-rec-upto.induct*)

case (*1 n v s*)
then show *?case*
using *GS-rec-upto-less* **by** (*fastforce simp*: *GS-rec-upto.simps*)
qed

lemma *GS-rec-upto-Suc'*:
assumes $s \leq n$
shows *GS-rec-upto* (*Suc n*) *v s n* = ($\bigsqcup a \in A\ n.\ L_a\ a$ (*GS-rec-upto n v s*) *n*)
using *assms*
proof (*induction n v s rule*: *GS-rec-upto.induct*)
case (*1 n v s*)
then show *?case*
by (*fastforce simp*: *not-less-eq-eq GS-rec-upto.simps*)
qed

lemma *GS-rec-upto-correct*:
assumes $s < n$
shows *GS.*$\mathcal{L}_b$*-split v s* = *GS-rec-upto n v 0 s*
using *assms*
proof (*induction n arbitrary*: *s*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
then show *?case*
proof (*cases* $s < n$)
case *True*
thus *?thesis*
using *Suc.IH* **by** (*auto simp*: *GS-rec-upto-Suc*)
next
case *False*
hence $s = n$
using *Suc* **by** *auto*
thus *?thesis*
using *Suc.IH GS-rec-upto-ge* **by** (*auto simp*: *GS-rec-upto-Suc'* *intro*: $\mathcal{L}_b$*-split-GS-iter*)
qed
qed

end
end
theory *GS-Code*
imports
Code-Setup
../Splitting-Methods-Fin
HOL−Library.Code-Target-Numeral
HOL−Data-Structures.Array-Braun

begin

context *MDP-nat-disc* **begin**

lemma $\mathcal{L}_b$*-split-zero*:
  assumes $\bigwedge s.\ s \geq states \Longrightarrow apply\text{-}bfun\ v\ s = 0$
  shows $GS.\mathcal{L}_b\text{-}split\ v\ s = GS\text{-}rec\text{-}upto\ states\ v\ 0\ s$
proof (*cases* $s < states$)
  case *True*
  then show *?thesis* **using** *GS-rec-upto-correct* **by** *auto*
next
  case *False*
  have *aux*: $s \geq states \Longrightarrow apply\text{-}bfun\ (GS.\mathcal{L}_b\text{-}split\ v)\ s = 0$ **for** *s*
  proof (*induction s rule*: *less-induct*)
    case (*less x*)
    have $r\ (x,\ a) = 0$ **if** $a \in A\ x$ **for** *a*
      by (*simp add*: *less.prems reward-zero-outside*)
   moreover have *measure-pmf.expectation* (K (x, a)) ((*bfun-if* ($\lambda s'$.
$s' < x$) ($GS.\mathcal{L}_b$*-split v*) *v*)) = *0* **for** *a*
      using *K-closed-compl assms less*
    by (*fastforce simp*: *bfun-if.rep-eq intro*!: *AE-pmfI integral-eq-zero-AE*)
    ultimately show *?case*
      by (*auto simp*: *A-ne* $\mathcal{L}_b$*-split-GS*)
  qed
  then show *?thesis*
    by (*metis False GS-rec-upto-ge assms not-less*)
qed
end

context *MDP-Code* **begin**

function $GS\text{-}iter\text{-}aux :: nat \Rightarrow {'tv} \Rightarrow real \Rightarrow ({'tv} \times real)$ **where**
  GS-iter-aux s v md = (
  if $s \geq states$
  then (*v*, *md*)
  else (
    let vs-old = *v-lookup v s*;
        vs-new = $\mathcal{L}$*-GS-code* (*s-lookup mdp s*) *v*;
        vs-diff = *abs* (*vs-old* − *vs-new*);
        v' = *v-update s vs-new v*
    in
        GS-iter-aux (*Suc s*) v' (*max md vs-diff*)))
  by *auto*
termination
  by (*relation Wellfounded.measure* ($\lambda(n,\ \text{-}).\ states - n$)) *auto*

definition *GS-iter v* = *GS-iter-aux 0 v 0*

lemmas *GS-iter-aux.simps*[*simp del*]

lemma *GS-iter-aux-fst-correct*:
assumes *v-len v = states v-invar v*
shows $s < states \longrightarrow$ *v-lookup* (*fst* (*GS-iter-aux n v md*)) *s* = *MDP.GS-rec-upto states* (*V-Map.map-to-bfun v*) *n s* $\wedge$ *v-invar* (*fst* (*GS-iter-aux n v md*))
using *assms* **unfolding** *GS-iter-def*
proof (*induction n v md rule*: *GS-iter-aux.induct*)
case (*1 s v md*)
show *?case*
unfolding *GS-iter-aux.simps*[*of s*] *MDP.GS-rec-upto.simps*[*of - - s*]
apply (*auto simp add*: *1.prems assms*(*1*) *intro*!: *v-lookup-map-to-bfun*)
apply (*simp add*: *1* $\mathcal{L}$-*GS-code-correct*)
using *1.IH*
apply (*smt* (*verit*) *1.IH 1.prems*(*1*) *1.prems*(*2*) *Sup.SUP-cong V-Map.invar-update V-Map.map-to-bfun-update* $\mathcal{L}$-*GS-code-correct linorder-le-less-linear v-len-update*)
by (*auto simp add*: *1.IH* $\mathcal{L}$-*GS-code-correct 1 V-Map.map-to-bfun-update v-lookup-map-to-bfun v-len-update V-Map.invar-update*)
qed

lemma *snd-GS-iter-aux-correct*:
assumes *v-len v = states v-invar v*
shows *snd* (*GS-iter-aux n v md*) = *Max* (*Set.insert md* (($\lambda s.$ *abs* (*MDP.GS-rec-upto states* (*V-Map.map-to-bfun v*) *n s* − (*V-Map.map-to-bfun v*) *s*)) ‘ {*n*..<*states*}))
using *assms* **unfolding** *GS-iter-def*
proof (*induction n v md rule*: *GS-iter-aux.induct*)
case (*1 s' v md*)
{
assume *s'-le*: $s' < states$
have *snd* (*GS-iter-aux s' v md*) = (*snd* (*GS-iter-aux* (*Suc s'*) (*v-update s'* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s'*) *v*) *v*) (*max md* |*v-lookup v s'* − $\mathcal{L}$-*GS-code* (*s-lookup mdp s'*) *v*|)))
unfolding *GS-iter-aux.simps*[*of s'*]
using *s'-le*
by *auto*
also have ... = *Max* (*Set.insert* (*max md* |*v-lookup v s'* − $\mathcal{L}$-*GS-code* (*s-lookup mdp s'*) *v*|)
(($\lambda s.$ |*MDP.GS-rec-upto states* (*apply-bfun* (*V-Map.map-to-bfun* (*v-update s'* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s'*) *v*) *v*))) (*Suc s'*) *s* − *apply-bfun* (*V-Map.map-to-bfun* (*v-update s'* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s'*) *v*) *v*)) *s*|) ‘ {*Suc s'*..<*states*}))
apply (*subst 1.IH*)
subgoal using *s'-le* **by** *auto*
using *s'-le v-len-update*
by (*auto simp add*: *1.prems V-Map.invar-update s'-le*)
also have ... = *Max* (*Set.insert* (*max md* |*v-lookup v s'* −

$\mathcal{L}$-GS-code (*s-lookup mdp* s') *v*|)
((*λs.* |*MDP.GS-rec-upto states* ((*V-Map.map-to-bfun v*)(s' := ($\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v*))) (*Suc* s') *s* − *apply-bfun* (*V-Map.map-to-bfun* (*v-update* s' ($\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v*) *v*)) *s*|) ' {*Suc* s'..<*states*}))
using *1.prems*(*1*) *1.prems*(*2*) *V-Map.map-to-bfun-update* s'*-le* **by** *presburger*
also have ... = *Max* (*Set.insert* (*max md* |*v-lookup v* s' − $\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v*|)
((*λs.* |*MDP.GS-rec-upto states* ((*V-Map.map-to-bfun v*)(s' := $\bigsqcup$ *a*∈*MDP-A* s'. *MDP.*L_a *a* (*V-Map.map-to-bfun v*) s')) (*Suc* s') *s* − ((*V-Map.map-to-bfun v*) (s' := $\bigsqcup$ *a*∈*MDP-A* s'. *MDP.*L_a *a* (*V-Map.map-to-bfun v*) s')) *s* |) ' {*Suc* s'..<*states*}))
using *1.prems*(*1*) *1.prems*(*2*) *MDP.SUP-*L_a*-eq-det MDP.*$\mathcal{L}_b$*-eq-SUP-*L_a *V-Map.map-to-fun-update* $\mathcal{L}$*-code-correct* $\mathcal{L}$*-code-lookup map-to-bfun-eq-fun* **by** *auto*

also have ... = *Max* (*Set.insert* (*max md* |*v-lookup v* s' − $\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v*|)
((*λs.* |*MDP.GS-rec-upto states* ((*V-Map.map-to-bfun v*)(s' := $\bigsqcup$ *a*∈*MDP-A* s'. *MDP.*L_a *a* (*V-Map.map-to-bfun v*) s')) (*Suc* s') *s* − *V-Map.map-to-bfun v s* |) ' {*Suc* s'..<*states*}))
using *1.prems*(*1*) *1.prems*(*2*) *MDP.SUP-*L_a*-eq-det MDP.*$\mathcal{L}_b$*-eq-SUP-*L_a *V-Map.map-to-fun-update* $\mathcal{L}$*-code-correct* $\mathcal{L}$*-code-lookup map-to-bfun-eq-fun* **by** *auto*
also have ... = *Max* (*Set.insert* (*max md* |*v-lookup v* s' − $\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v*|)
((*λs.* |*MDP.GS-rec-upto states* (*apply-bfun* (*V-Map.map-to-bfun v*)) s' *s* − *V-Map.map-to-bfun v s*|) ' {*Suc* s'..<*states*}))
using s'*-le MDP.GS-rec-upto.simps*[*symmetric*, *of states* s' (*apply-bfun* (*V-Map.map-to-bfun v*))]
by *presburger*
also have ... = *max md* (*Max* (*Set.insert* (|*v-lookup v* s' − $\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v*|)
((*λs.* |*MDP.GS-rec-upto states* (*apply-bfun* (*V-Map.map-to-bfun v*)) s' *s* − *V-Map.map-to-bfun v s*|) ' {*Suc* s'..<*states*})))
proof −
have ∗: *Max* (*Set.insert* (*max x y*) *X*) = *max x* (*Max* (*Set.insert y X*)) **if** *finite X* **for** *X x y*
by (*metis Max-insert Max-singleton max.assoc that*)
thus *?thesis*
by *blast*
qed
also have ... = *max md* (*Max* (*Set.insert* (|$\mathcal{L}$*-GS-code* (*s-lookup mdp* s') *v* − *V-Map.map-to-bfun v* s'|)
((*λs.* |*MDP.GS-rec-upto states* (*apply-bfun* (*V-Map.map-to-bfun v*)) s' *s* − *V-Map.map-to-bfun v s*|) ' {*Suc* s'..<*states*})))
by (*smt* (*verit*, *best*) *1.prems*(*1*) *1.prems*(*2*) *v-lookup-map-to-bfun* s'*-le*)
also have ... = *max md* (*Max* ((*λs.* (|*MDP.GS-rec-upto states*

(*apply-bfun* (*V-Map.map-to-bfun v*)) s' s − *V-Map.map-to-bfun v s*|))
‘ $\{s'\}$ ∪
((λ*s*. |*MDP.GS-rec-upto states* (*apply-bfun* (*V-Map.map-to-bfun*
v)) s' s − *V-Map.map-to-bfun v s*|) ‘ {*Suc* s'..<*states*})))
proof −
have ∗ :*ℒ-GS-code* (*s-lookup mdp* s') *v* = *MDP.GS-rec-upto states*
(*apply-bfun* (*V-Map.map-to-bfun v*)) s' s'
apply (*subst MDP.GS-rec-upto-eq*)
using *s′-le*
apply *blast*
using *ℒ-GS-code-correct 1 s′-le*
by *presburger*
show *?thesis*
by (*auto simp*: ∗)
qed
also have ... = *max md* (*Max* ((λ*s*. |*MDP.GS-rec-upto states*
(*apply-bfun* (*V-Map.map-to-bfun v*)) s' s − *V-Map.map-to-bfun v s*|)
‘ ($\{s'\}$ ∪ {*Suc* s'..<*states*})))
unfolding *atLeastLessThan-def lessThan-def*
by *auto*
also have ... = *max md* (*Max* ((λ*s*. |*MDP.GS-rec-upto states*
(*apply-bfun* (*V-Map.map-to-bfun v*)) s' s − *V-Map.map-to-bfun v s*|)
‘ ({s'..<*states*})))
proof −
have ($\{s'\}$ ∪ {*Suc* s'..<*states*}) = {s'..<*states*}
using *s′-le*
by *auto*
thus *?thesis* **by** *auto*
qed
finally have *snd* (*GS-iter-aux* s' *v md*) = *max md* (*MAX s*∈{s'..<*states*}.
|*MDP.GS-rec-upto states* ((*V-Map.map-to-bfun v*)) s' s − (*V-Map.map-to-bfun*
v) *s*|).
}
thus *?case*
apply (*cases* s' < *states*)
apply *auto*
using *1.prems*(*1*) *1.prems*(*2*) *GS-iter-aux-fst-correct assms*(*1*)
apply (*simp add*: *1.prems*(*1*) *1.prems*(*2*) *GS-iter-aux-fst-correct*
assms(*1*))
by (*simp add*: *GS-iter-aux.simps*)
qed

lemma *invar-GS-iter-aux*: *v-len v* = *states* ⟹ *v-invar v* ⟹ *v-invar*
(*fst* (*GS-iter-aux n v md*))
by (*metis GS-iter-aux.simps GS-iter-aux-fst-correct fst-conv linorder-not-le*)

lemma *invar-GS-iter*: *v-len v* = *states* ⟹ *v-invar v* ⟹ *v-invar* (*fst*
(*GS-iter v*))

using *invar-GS-iter-aux GS-iter-def* **by** *auto*

lemma *len-GS-iter-aux*[*simp*]: *v-invar v* $\Longrightarrow$ *v-len v* = *states* $\Longrightarrow$ *v-len* (*fst* (*GS-iter-aux n v md*)) = *states*
proof (*induction n v md rule*: *GS-iter-aux.induct*)
case (*1 s v md*)
have *2*: *v-len* (*v-update s* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s*) *v*) *v*) = *v-len v* **if** *s* < *states*
using *1 that v-len-update* **by** *blast*
have *v-len* (*fst* (*GS-iter-aux* (*Suc s*) (*v-update s* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s*) *v*) *v*) (*max md* |*v-lookup v s* − $\mathcal{L}$-*GS-code* (*s-lookup mdp s*) *v*|))) = *v-len v* **if** *s* < *states*
unfolding *2*[*OF that*, *symmetric*]
using *1*(*2*,*3*) *that*
apply (*subst 1.IH*[*OF* -])
apply *auto*
using *V-Map.invar-update 2* **by** *force+*
thus *?case*
by (*metis 1.prems*(*2*) *GS-iter-aux.elims fst-conv less-eq-Suc-le not-less-eq-eq*)
qed

lemma *len-GS-iter*[*simp*]: *v-invar v* $\Longrightarrow$ *v-len v* = *states* $\Longrightarrow$ *v-len* (*fst* (*GS-iter v*)) = *v-len v*
using *len-GS-iter-aux GS-iter-def* **by** *auto*

lemma *GS-iter-aux-correct′*:
assumes *v-len v* = *states v-invar v*
shows *apply-bfun* (*V-Map.map-to-bfun* (*fst* (*GS-iter-aux 0 v md*))) *s* = *MDP.GS-rec-upto states* (*V-Map.map-to-bfun v*) *0 s*
proof (*cases s* < *states*)
case *True*
then show *?thesis*
using *assms*
by (*metis GS-iter-aux-fst-correct len-GS-iter-aux v-lookup-map-to-bfun*)
next
case *False*
then show *?thesis*
by (*simp add*: *MDP.GS-rec-upto-ge V-Map.map-to-bfun.rep-eq assms*(*1*) *assms*(*2*))
qed

lemma *GS-iter-aux-correct″*:
assumes *v-len v* = *states v-invar v*
shows *V-Map.map-to-bfun* (*fst* (*GS-iter v*)) = *MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)
apply (*rule bfun-eqI*)
unfolding *V-Map.map-to-bfun.rep-eq*

apply *auto*
apply (*simp add*: *GS-iter-aux-fst-correct GS-iter-def MDP.GS-rec-upto-correct assms*(*1*) *assms*(*2*))
apply (*simp add*: *GS-iter-def assms*(*1*) *assms*(*2*) *invar-GS-iter-aux*)
by (*metis GS-iter-aux-correct′ GS-iter-def MDP.*$\mathcal{L}_b$*-split-zero V-Map.map-to-bfun.rep-eq assms*(*1*) *assms*(*2*) *linorder-not-less*)

lemma *snd-GS-iter-correct′*:
assumes *v-len v* = *states v-invar v*
shows *snd* (*GS-iter v*) = *dist* (*V-Map.map-to-bfun* (*fst* (*GS-iter v*))) (*V-Map.map-to-bfun v*)
proof −
have *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) = *0* **if** $x \geq$ *states* **for** *x*
by (*metis GS-iter-aux-correct′′ V-Map.map-to-bfun.rep-eq assms*(*1*) *assms*(*2*) *dist-eq-0-iff leD len-GS-iter that*)
hence ($\bigsqcup$*x*. *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*)) =
($\bigsqcup$*x*∈{*0..*<*Suc states*}. *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*)) ⊔ ($\bigsqcup$*x*∈{*Suc states..*}. *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*))
apply (*subst cSUP-union*[*symmetric*])
apply *auto*
by (*simp add*: *ivl-disj-un-one*(*8*))
also have ... = *max 0* ($\bigsqcup$(*Set.insert 0* ((λ*x*. *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*)) ‘ {*0..*< *states*})))
proof −
have *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) = *0* **if** *x* ∈ {*Suc states..*} **for** *x*
apply *auto*
using *MDP.*$\mathcal{L}_b$*-split-zero*
by (*meson Suc-leD* ‹⋀*x*. *states* ≤ *x* ⟹ *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) = *0*› *atLeast-iff dist-eq-0-iff that*)
thus *?thesis*
using *sup-real-def*
by (*simp add*: ‹⋀*x*. *states* ≤ *x* ⟹ *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*) = *0*› *atLeast0-lessThan-Suc*)
qed
also have ... = *max 0* (*Max* (*Set.insert 0* ((λ*x*. *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split* (*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*)) ‘ {*0..*< *states*})))
by (*auto simp*: *cSup-eq-Max*)
also have ... = (*Max* (*Set.insert 0* ((λ*x*. *dist* (*apply-bfun* (*MDP.GS.*$\mathcal{L}_b$*-split*

(*V-Map.map-to-bfun v*)) *x*) (*apply-bfun* (*V-Map.map-to-bfun v*) *x*)) ‘ {*0..< states*})))
by *auto*
also have ... = *snd* (*GS-iter v*)
unfolding *GS-iter-def*
apply (*subst snd-GS-iter-aux-correct*)
apply (*simp add*: *assms*)
apply (*simp add*: *assms*)
apply (*auto simp*: *dist-real-def*)
apply (*subst MDP.*$\mathcal{L}_b$*-split-zero*)
apply (*simp add*: *V-Map.map-to-bfun.rep-eq assms*(*1*))
by (*auto simp*: *dist-real-def*)
finally show *?thesis*
by (*simp add*: *GS-iter-aux-correct''* *assms*(*1*) *assms*(*2*) *dist-bfun.rep-eq*)
qed

lemma *GS-iter-aux-correct*:
assumes *s* < *states v-len v* = *states v-invar v*
shows *v-lookup* (*fst* (*GS-iter-aux n v eps*)) *s* = *MDP.GS-rec-upto states* (*V-Map.map-to-bfun v*) *n s*
using *GS-iter-aux-fst-correct assms*(*1*) *assms*(*2*) *assms*(*3*) **by** *blast*

definition *find-policy-code-aux-upt* (*v*::'*tv*) *n* = (
fold (λ*s* (*d*,*v*). *let* (*d'*, *v'*) = *find-policy-state-code-aux' v s in*
(*d-update s d' d*, *v-update s v' v*)) [*0..<n*] (*d-empty*, *v*))

lemma *find-policy-code-aux-upt-Suc*:
find-policy-code-aux-upt v (*Suc s*) = (
let (*d*, *v*) = (*find-policy-code-aux-upt v s*) *in*
(*d-update s* ((*fst* (*find-policy-state-code-aux' v s*))) *d*, *v-update s* (*snd* (*find-policy-state-code-aux' v s*)) *v*))
unfolding *find-policy-code-aux-upt-def*
by (*auto simp*: *case-prod-beta*)

definition *find-policy-code-aux v* = *find-policy-code-aux-upt v states*
definition *find-policy-code v* = *fst* (*find-policy-code-aux v*)

lemma *d-invar-find-policy-code-aux-upt*: *D-Map.invar* (*fst* (*find-policy-code-aux-upt v n*))
by (*induction n*) (*auto simp*: *D-Map.map-specs case-prod-beta find-policy-code-aux-upt-def*)

lemma *v-len-invar-find-policy-code-aux-upt*: $n \le j \Longrightarrow$ *v-len v* = $j \Longrightarrow$ *v-invar v* $\Longrightarrow$ *v-len* (*snd* (*find-policy-code-aux-upt v n*)) = $j \wedge$ *v-invar* (*snd* (*find-policy-code-aux-upt v n*))
apply (*induction n arbitrary*: *v*)
apply (*simp add*: *find-policy-code-aux-upt-def*)
apply (*simp add*: *case-prod-beta find-policy-code-aux-upt-def*)

```
  apply (subst V-Map.invar-update)
    apply blast
   apply simp
  using Suc-le-lessD v-len-update by presburger

lemma assumes s < states v-invar v v-len v ≥ states
  shows
   d-lookup (fst (find-policy-code-aux v)) s = d-lookup (fst (find-policy-code-aux-upt
v (Suc s))) s
   v-lookup (snd (find-policy-code-aux v)) s = v-lookup (snd (find-policy-code-aux-upt
v (Suc s))) s
  unfolding find-policy-code-aux-def
  using assms
proof (induction states arbitrary: v)
  case (Suc states)
  {
    case 1
    show ?case
    proof (cases s = states)
    next
      case False
      then show ?thesis
        using 1 less-Suc-eq
        apply (subst find-policy-code-aux-upt-Suc)
      by (auto simp: case-prod-beta D-Map.map-update[OF d-invar-find-policy-code-aux-upt]
Suc(1)[symmetric])
    qed auto
  next
    case 2
    then show ?case
    proof (cases s = states)
    next
      case False
      then show ?thesis
        using 2 less-Suc-eq
        apply (subst find-policy-code-aux-upt-Suc)
        apply (auto simp: case-prod-beta Suc(2)[symmetric])
      by (metis False Suc-leD dec-induct v-len-invar-find-policy-code-aux-upt
v-lookup-update)
    qed auto
  }
qed auto

lemma find-policy-code-invar: D-Map.invar (find-policy-code v)
  unfolding find-policy-code-def find-policy-code-aux-def
 by (induction states) (auto simp: find-policy-code-aux-upt-def D-Map.map-specs
case-prod-unfold)

lemma find-policy-code-notin:
```

assumes $s \geq states$ **shows** *d-lookup* (*find-policy-code v*) $s = None$
using *assms d-invar-find-policy-code-aux-upt*
unfolding *find-policy-code-def find-policy-code-aux-def*
by (*induction states*) (*auto simp*: *find-policy-code-aux-upt-def case-prod-beta D-Map.map-specs*)

lemma *find-policy-code-in*:
assumes $s < states$ **shows** $\exists x.$ *d-lookup* (*find-policy-code v*) $s =$ *Some x*
using *assms*
unfolding *find-policy-code-def find-policy-code-aux-def*
proof (*induction states*)
case *0*
then show *?case*
by *simp*
next
case (*Suc states*)
then show *?case*
using *d-invar-find-policy-code-aux-upt*
by (*auto simp*: *find-policy-code-aux-upt-Suc case-prod-beta D-Map.map-specs*)
qed

lemma *GS-iter-aux-fold*: *fst* (*GS-iter-aux s v md*) = *fold* (λs *v. v-update s* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s*) *v*) *v*) [*s..<states*] *v*
proof (*induction s v md arbitrary*: *rule*: *GS-iter-aux.induct*)
case (*1 s v*)
have *aux*: $s < states \Longrightarrow$ [*s..<states*] = *s*#[*Suc s..<states*]
using *upt-conv-Cons* **by** *presburger*
show *?case*
by (*subst GS-iter-aux.simps*) (*auto simp*: *1 aux*)
qed

lemma *find-policy-state-code-aux′-eq-*$\mathcal{L}$*-GS-code*:
assumes *v-len v = states v-invar v* $s < states$
shows *snd* (*find-policy-state-code-aux′ v s*) = $\mathcal{L}$-*GS-code* (*s-lookup mdp s*) *v*
using *assms*
by (*auto simp*: $\mathcal{L}$-*GS-code-correct cSup-eq-Max find-policy-state-code-aux′-eq′*)

lemma *snd-find-policy-code-aux-upt*:
assumes *v-len v = states v-invar v*
shows (*snd* (*find-policy-code-aux-upt v states*)) = *fst* (*GS-iter-aux 0 v md*)
proof –
have *fst* (*GS-iter-aux 0 v md*) = *fold* (λs *v. v-update s* ($\mathcal{L}$-*GS-code* (*s-lookup mdp s*) *v*) *v*) [*0..<states*] *v*
unfolding *GS-iter-aux-fold* **..**
also have . . . = *fold* (λs *v. v-update s* (*snd* (*find-policy-state-code-aux′*

v s)) *v*) [*0..<states*] *v*
using *find-policy-state-code-aux′-eq-$\mathcal{L}$-GS-code assms*
by (*auto simp*: *V-Map.invar-update v-len-update intro*!: *fold-cong′*[**where** *P* = *λv. v-len v* = *states* ∧ *v-invar v*])
also have ... = (*snd* (*find-policy-code-aux-upt v states*))
unfolding *find-policy-code-aux-upt-def*
by (*induction states*) (*auto simp add*: *split-def*)
finally show *?thesis*..
qed

lemma *GS-rec-upto-Suc*: *MDP.GS-rec-upto* (*Suc n*) *v 0* = (*MDP.GS-rec-upto n v 0*)(*n* := ($\bigsqcup$ *a*∈*MDP-A n. MDP.*L_a *a* (*MDP.GS-rec-upto n v 0*) *n*))
proof –
have *s* ≠ *n* ⟹ *MDP.GS-rec-upto* (*Suc n*) *v 0 s* = *MDP.GS-rec-upto n v 0 s* **for** *s*
using *MDP.GS-rec-upto-Suc MDP.GS-rec-upto-ge*
by (*metis Suc-leI le-neq-implies-less not-less*)
moreover have *s* = *n* ⟹ *MDP.GS-rec-upto* (*Suc n*) *v 0 s* = ($\bigsqcup$ *a*∈*MDP-A n. MDP.*L_a *a* (*MDP.GS-rec-upto n v 0*) *n*) **for** *s*
using *MDP.GS-rec-upto-Suc′* **by** *auto*
ultimately show *?thesis*
by *auto*
qed

lemma *keys-fst-find-policy-code-aux-upt*: *s* ≤ *states* ⟹ *D-Map.keys* (*fst* (*find-policy-code-aux-upt v s*)) = {*0..<s*}
using *d-invar-find-policy-code-aux-upt find-policy-code-aux-upt-def*
by (*induction s arbitrary*: *v*) (*auto simp*: *find-policy-code-aux-upt-Suc case-prod-beta*)

lemma *keys-fst-find-policy-code-aux*: *D-Map.keys* (*fst* (*find-policy-code-aux v*)) = {*0..<states*}
using *keys-fst-find-policy-code-aux-upt find-policy-code-aux-def*
by *force*

lemma *find-policy-code-ge*: *s* ≥ *states* ⟹ *D-Map.map-to-fun* (*find-policy-code v*) *s* = *0*
using *find-policy-code-notin find-policy-code-def*
by (*auto simp*: *D-Map.map-to-fun-def*)

lemma *find-policy-code-aux-upt-zero*[*simp*]: *find-policy-code-aux-upt v 0* = (*d-empty*, *v*)
unfolding *find-policy-code-aux-upt-def*
by *auto*

lemma *GS-rec-upto-zero*[*simp*]: *MDP.GS-rec-upto 0 v n* = *v*
by (*auto simp*: *MDP.GS-rec-upto.simps*)

lemma *keys-find-policy-code-aux-upt*:*n* < *states* ⟹ *v-invar v* ⟹

v-len v = states $\Longrightarrow$ *v-len* (*snd* (*find-policy-code-aux-upt v n*)) = *states*
apply (*induction n arbitrary*: *v*)
apply (*auto simp*: *case-prod-beta find-policy-code-aux-upt-Suc*)
by (*metis Suc-lessD less-or-eq-imp-le v-len-invar-find-policy-code-aux-upt v-len-update*)

lemma *split-eq-GS-rec-upto-Sup*:
MDP.GS.$\mathcal{L}_b$*-split v s* = ($\bigsqcup$ *a*$\in$*MDP-A s. MDP.*L_a *a* (*MDP.GS-rec-upto s* (*apply-bfun v*) *0*) *s*)
using *MDP.GS-rec-upto-correct MDP.GS-rec-upto-ge MDP.*$\mathcal{L}_b$*-split-GS-iter*[*symmetric*, *of - - v*] **by** *auto*

lemma *split-eq-GS-rec-upto-is-arg-max*:
assumes *is-arg-max* (λ*a. MDP.*L_a *a* (*MDP.GS-rec-upto s* (*apply-bfun v*) *0*) *s*) (λ*a. a* $\in$ *MDP-A s*) *a*
shows *MDP.GS.*$\mathcal{L}_b$*-split v s* = *MDP.*L_a *a* (*MDP.GS-rec-upto s* (*apply-bfun v*) *0*) *s*
using *arg-max-SUP*[*OF assms*] *split-eq-GS-rec-upto-Sup*
by *auto*

lemma *MDP.GS-rec-upto n* (*apply-bfun v*) *0 s* = (*if s* < *n then MDP.GS.*$\mathcal{L}_b$*-split v s else v s*)
using *MDP.GS-rec-upto-correct MDP.GS-rec-upto-ge*
by *auto*

lemma *GS-rec-upto-eq-*$\mathcal{L}_b$*-split'*: *MDP.GS-rec-upto n* (*apply-bfun v*) *0* = (λ*s. if s* < *n then MDP.GS.*$\mathcal{L}_b$*-split v s else v s*)
using *MDP.GS-rec-upto-correct MDP.GS-rec-upto-ge not-le*
by *auto*

lemma *snd-find-policy-code-aux-upt-correct*:
assumes *v-len v* = *states v-invar v n* $\le$ *states*
shows *V-Map.map-to-fun* (*snd* (*find-policy-code-aux-upt v n*)) = *MDP.GS-rec-upto n* (*V-Map.map-to-fun v*) *0*
using *assms*
proof (*induction n*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
have *V-Map.map-to-fun* (*snd* (*find-policy-code-aux-upt v* (*Suc n*))) *n* = *snd* (*find-policy-state-code-aux'* (*snd* (*find-policy-code-aux-upt v n*)) *n*)
unfolding *find-policy-code-aux-upt-Suc*
using *Suc*(*3*) *Suc*(*2*)
apply (*auto simp*: *case-prod-unfold V-Map.map-to-fun-update*)
apply (*subst V-Map.map-to-fun-update*)
using *v-len-invar-find-policy-code-aux-upt*

apply *auto*
by (*metis Suc.prems*(*3*) *Suc-leD Suc-le-lessD assms*(*1*))+
also have ... = (*MAX a*∈*MDP-A n. MDP.*L_a *a* (*apply-bfun* (*V-Map.map-to-bfun* (*snd* (*find-policy-code-aux-upt v n*)))) *n*)
using *keys-find-policy-code-aux-upt Suc v-len-invar-find-policy-code-aux-upt*
by (*smt* (*verit*, *ccfv-SIG*) *Suc-leD Suc-le-lessD Sup.SUP-cong find-policy-state-code-aux′-eq′ snd-conv*)
also have ... = ($\bigsqcup$ *a*∈*MDP-A n. MDP.*L_a *a* (*apply-bfun* (*V-Map.map-to-bfun* (*snd* (*find-policy-code-aux-upt v n*)))) *n*)
using *Suc*
by (*auto simp*: *cSup-eq-Max*[*symmetric*] *V-Map.map-to-fun-def V-Map.map-to-bfun.rep-eq*)
also have ... = ($\bigsqcup$ *a*∈*MDP-A n. MDP.*L_a *a* ((*V-Map.map-to-fun* (*snd* (*find-policy-code-aux-upt v n*)))) *n*)
using *Suc.prems*
by (*auto simp*: *map-to-bfun-eq-fun*)
also have ... = *MDP.GS-rec-upto* (*Suc n*) (*V-Map.map-to-fun v*) *0 n*
using *MDP.GS-rec-upto-Suc′ Suc*
by *auto*
finally have *V-Map.map-to-fun* (*snd* (*find-policy-code-aux-upt v* (*Suc n*))) *n* = *MDP.GS-rec-upto* (*Suc n*) (*V-Map.map-to-fun v*) *0 n*.
moreover have *V-Map.map-to-fun* (*snd* (*find-policy-code-aux-upt v* (*Suc n*))) *s* = *MDP.GS-rec-upto* (*Suc n*) (*V-Map.map-to-fun v*) *0 s* **if** *s* ≠ *n* **for** *s*
unfolding *find-policy-code-aux-upt-Suc*
using *Suc Suc-lessD that*
apply (*auto simp*: *case-prod-beta V-Map.map-to-fun-update GS-rec-upto-Suc*)
by (*metis Suc-leD Suc-le-lessD V-Map.invar-update V-Map.map-to-fun-def v-len-invar-find-policy-code-aux-upt v-len-update v-lookup-update*)
ultimately show *?case*
by *fastforce*
qed

lemma *GS-inv-eq-L*: *apply-bfun* (*MDP.GS-inv d v*) *s* = *MDP.L* (*MDP.mk-dec-det d*) ((*bfun-if* ((≤) *s*) *v* (*MDP.GS-inv d v*))) *s*
using *MDP.GS-inv-elem-eq MDP.L-def* **by** *presburger*

lemma *GS-inv-eq-*L_a: *MDP.GS-inv d v s* = *MDP.*L_a (*d s*) (*bfun-if* ((≤) *s*) *v* (*MDP.GS-inv d v*)) *s*
using *GS-inv-eq-L MDP.L-eq-*L_a*-det* **by** *presburger*

lemma *is-arg-max-*L_a*-GS-inv*:
is-arg-max (λ*a. MDP.*L_a *a* (*bfun-if* ((≤) *s*) *v* (*MDP.GS-inv d v*)) *s*) (λ*a. a* ∈ *MDP-A s*) *a*
⟷ *is-arg-max* (λ*a.* (*MDP.GS-inv* (*d*(*s* := *a*)) *v s*)) (λ*a. a* ∈ *MDP-A s*) *a*
proof −
have ∗: *s′* < *s* ⟹ *MDP.GS-inv* (*d*(*s* := *a*)) *v s′* = *MDP.GS-inv d*

v s′ **for** *s′ a*
using *MDP.GS-indep-high-states* **by** *fastforce*
show *?thesis*
unfolding *GS-inv-eq-L_a* **by** (*fastforce simp*: *bfun-if.rep-eq* ∗ *cong*: *if-cong*)
qed

lemma *GS-rec-upto-eq-$\mathcal{L}_b$-split′′*: *MDP.GS-rec-upto s* (*apply-bfun v*) *0* = *bfun-if* ((≤) *s*) *v* (*MDP.GS.$\mathcal{L}_b$-split v*)
by (*fastforce simp*: *MDP.GS-rec-upto-ge bfun-if.rep-eq MDP.GS-rec-upto-correct not-le*)

lemma *GS-inv-GS-least-eq-split*: *MDP.GS-inv* (*MDP.d-GS-least v*) *v* = *MDP.GS.$\mathcal{L}_b$-split v*
using *arg-max-SUP*[*OF MDP.d-GS-least-is-arg-max*]
by (*auto simp*: *MDP.GS.$\mathcal{L}_b$-split.rep-eq MDP.GS.$\mathcal{L}$-split-def MDP.GS-inv-def*[*symmetric*])

lemma *is-arg-max-L_a-GS-inv-d-GS-least*:
is-arg-max (λ*a*. *MDP.L_a a* (*MDP.GS-rec-upto s* (*apply-bfun v*) *0*) *s*) (λ*a*. *a* ∈ *MDP-A s*) *a*
⟷ *is-arg-max* (λ*a*. (*MDP.GS-inv* ((*MDP.d-GS-least v*)(*s* := *a*)) *v s*)) (λ*a*. *a* ∈ *MDP-A s*) *a*
by (*auto simp*: *GS-inv-GS-least-eq-split GS-rec-upto-eq-$\mathcal{L}_b$-split′′ is-arg-max-L_a-GS-inv*[*symmetric*])

lemma *d-GS-least-ge*: *s* ≥ *states* ⟹ *MDP.d-GS-least* (*V-Map.map-to-bfun v*) *s* = *0*
by (*subst MDP.d-GS-least-eq*) (*auto intro*!: *Least-equality simp*: *is-arg-max-linorder MDP-A-def*)

lemma *fst-find-policy-code-aux-upt-correct*:
assumes *v-len v* = *states v-invar v n* ≤ *states s* < *n*
shows *D-Map.map-to-fun* (*fst* (*find-policy-code-aux-upt v n*)) *s* = *least-arg-max* (λ*a*. *MDP.L_a a* (*MDP.GS-rec-upto s* (*V-Map.map-to-fun v*) *0*) *s*) (λ*a*. *a* ∈ *MDP-A s*)
using *assms*
proof (*induction n arbitrary*: *s*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
have *D-Map.map-to-fun* (*fst* (*find-policy-code-aux-upt v* (*Suc n*))) *n* = *fst* (*find-policy-state-code-aux′* (*snd* (*find-policy-code-aux-upt v n*)) *n*)
using *d-invar-find-policy-code-aux-upt Suc*
by (*auto simp*: *find-policy-code-aux-upt-Suc case-prod-unfold D-Map.map-to-fun-update*)
also have . . . = *least-arg-max* (λ*a*. *MDP.L_a a* (*apply-bfun* (*V-Map.map-to-bfun* (*snd* (*find-policy-code-aux-upt v n*)))) *n*) (λ*a*. *a* ∈ *MDP-A n*)
using *Suc keys-find-policy-code-aux-upt*

apply (*auto simp*: *find-policy-state-code-aux'-eq'*)
apply (*subst find-policy-state-code-aux'-eq'*)
using *Suc-le-lessD* **apply** *presburger+*
apply (*meson Suc-leD v-len-invar-find-policy-code-aux-upt*)
by *force*
also have ... = *least-arg-max* ($\lambda a.$ *MDP*.L_a *a* (*MDP*.*GS-rec-upto n* (*V-Map.map-to-fun v*) *0*) *n*) ($\lambda a.$ $a \in$ *MDP-A n*)
using *Suc map-to-bfun-eq-fun*
by (*auto simp*: *snd-find-policy-code-aux-upt-correct*)
finally have *D-Map.map-to-fun* (*fst* (*find-policy-code-aux-upt v* (*Suc n*))) *n* = *least-arg-max* ($\lambda a.$ *MDP*.L_a *a* (*MDP*.*GS-rec-upto n* (*V-Map.map-to-fun v*) *0*) *n*) ($\lambda a.$ $a \in$ *MDP-A n*).
moreover have *D-Map.map-to-fun* (*fst* (*find-policy-code-aux-upt v* (*Suc n*))) *s* = *least-arg-max* ($\lambda a.$ *MDP*.L_a *a* (*MDP*.*GS-rec-upto s* (*V-Map.map-to-fun v*) *0*) *s*) ($\lambda a.$ $a \in$ *MDP-A s*) **if** $s < n$ **for** *s*
using *that d-invar-find-policy-code-aux-upt Suc*
by (*auto simp*: *find-policy-code-aux-upt-Suc case-prod-unfold D-Map.map-to-fun-update*)
ultimately show *?case*
using *Suc* **by** (*cases* $s = n$) *auto*
qed

lemma *GS-iter'-correct*:
assumes *v-len v* = *states v-invar v*
shows *D-Map.map-to-fun* (*find-policy-code v*) = (*MDP*.*d-GS-least* (*V-Map.map-to-bfun v*))
proof −
have *D-Map.map-to-fun* (*find-policy-code v*) *s* = (*MDP*.*d-GS-least* (*V-Map.map-to-bfun v*)) *s* **if** $s \geq$ *states* **for** *s*
using *find-policy-code-ge d-GS-least-ge that*
by *auto*
moreover have *D-Map.map-to-fun* (*find-policy-code v*) *s* = (*MDP*.*d-GS-least* (*V-Map.map-to-bfun v*)) *s* **if** $s <$ *states* **for** *s*
using *that assms*
proof (*induction s rule*: *less-induct*)
case (*less x*)
show *?case*
unfolding *find-policy-code-def find-policy-code-aux-def*
using *less assms*
by (*auto intro*!: *least-arg-max-cong'* *simp*: *MDP*.*d-GS-least-eq fst-find-policy-code-aux-upt-correct map-to-bfun-eq-fun is-arg-max-*L_a*-GS-inv-d-GS-least*[*symmetric*] *least-arg-max-def*[*symmetric*])
qed
ultimately show *?thesis*
using *not-le* **by** *blast*
qed

partial-function (*tailrec*) *GS-code-aux* **where**
GS-code-aux v eps = (
let (v', *md*) = *GS-iter v in*

```
    if (2 * l) * md < eps * (1 − l)
    then v′
    else GS-code-aux v′ eps)

lemmas GS-code-aux.simps[code]

definition GS-code v eps = (if l = 0 ∨ eps ≤ 0 then fst (GS-iter v)
else GS-code-aux v eps)


lemma GS-code-aux-correct-aux:
  assumes eps > 0 v-invar v v-len v = states l ≠ 0
  shows V-Map.map-to-fun (GS-code-aux v eps) = MDP.gs-iteration
eps (V-Map.map-to-bfun v)
  ∧ v-len (GS-code-aux v eps) = states ∧ v-invar (GS-code-aux v eps)
  using assms
proof (induction eps (V-Map.map-to-bfun v) arbitrary: v rule: MDP.gs-iteration.induct)
  case (1 eps)
   have *:2 * l * snd (GS-iter v) < eps * (1 − l) ⟷ 2 * l * dist
(MDP.GS.ℒ_b-split (V-Map.map-to-bfun v)) (V-Map.map-to-bfun v) <
eps * (1 − l)
    using GS-iter-aux-correct′′
    by (auto simp: snd-GS-iter-correct′ 1)

  thus ?case
  proof (cases 2 * l * snd (GS-iter v) < eps * (1 − l))
    case True
    then show ?thesis
      unfolding GS-code-aux.simps[of v]
      apply (simp add: case-prod-beta)
      apply (subst MDP.gs-iteration.simps)
      apply (auto simp: case-prod-beta *)
              apply (metis 1.prems(2) 1.prems(3) GS-iter-aux-correct′′
map-to-bfun-eq-fun)
      using 1.prems(1) apply (auto simp: dist-commute)
       apply (simp add: 1.prems(2) 1.prems(3))
      using 1.prems(2) 1.prems(3) invar-GS-iter by blast
  next
    case False
    then show ?thesis
      unfolding GS-code-aux.simps[of v]
      apply (simp add: case-prod-beta)
      apply (subst MDP.gs-iteration.simps)
      apply (auto simp: case-prod-beta *)
      using 1.prems(1) apply (auto simp: dist-commute)
    by (auto simp add: 1.hyps 1.prems(2) 1.prems(3) GS-iter-aux-correct′′
assms(4) invar-GS-iter)
  qed
qed
```

lemma *GS-code-aux-correct*:
assumes *eps* > *0* *v-invar v v-len v* = *states l* ≠ *0*
shows *V-Map.map-to-fun* (*GS-code-aux v eps*) = *MDP.gs-iteration eps* (*V-Map.map-to-bfun v*)
using *assms GS-code-aux-correct-aux* **by** *auto*

lemma *GS-code-aux-keys*:
assumes *eps* > *0* *v-invar v v-len v* = *states l* ≠ *0*
shows *v-len* (*GS-code-aux v eps*) = *states*
using *assms GS-code-aux-correct-aux* **by** *auto*

lemma *GS-code-aux-invar*:
assumes *eps* > *0* *v-invar v v-len v* = *states l* ≠ *0*
shows *v-invar* (*GS-code-aux v eps*)
using *assms GS-code-aux-correct-aux* **by** *auto*

lemma *GS-code-correct*:
assumes *eps* > *0* *v-invar v v-len v* = *states*
shows *V-Map.map-to-fun* (*GS-code v eps*) = *MDP.gs-iteration eps* (*V-Map.map-to-bfun v*)
proof (*cases l* = *0*)
case *True*
then show *?thesis*
using *assms invar-GS-iter GS-iter-aux-correct''*
unfolding *GS-code-def MDP.gs-iteration.simps*[*of - V-Map.map-to-bfun v*]
by (*fastforce simp*: *map-to-bfun-eq-fun*)
next
case *False*
then show *?thesis*
using *assms*
by (*auto simp add*: *GS-code-def GS-code-aux-correct MDP.gs-iteration.simps*)
qed

definition *GS-policy-code v eps* = *find-policy-code* (*GS-code v eps*)

lemma *GS-policy-code-correct*:
assumes *eps* > *0* *v-invar v v-len v* = *states*
shows *D-Map.map-to-fun* (*GS-policy-code v eps*) = *MDP.vi-gs-policy eps* (*V-Map.map-to-bfun v*)
proof −
have *aux*: *V-Map.map-to-bfun* (*GS-code v eps*) = (*MDP.gs-iteration eps* (*V-Map.map-to-bfun v*))
using *GS-code-correct*[*OF assms*] *assms*(*2*) *map-to-bfun-eq-fun* **by** *auto*
have *D-Map.map-to-fun* (*GS-policy-code v eps*) = *MDP.d-GS-least* (*V-Map.map-to-bfun* (*GS-code v eps*))

unfolding *GS-code-def GS-policy-code-def MDP.vi-gs-policy-def*
proof (*subst GS-iter′-correct*)
show *v-len* (*if* $l = 0 \lor eps \leq 0$ *then fst* (*GS-iter v*) *else GS-code-aux v eps*) = *states*
using *assms len-GS-iter GS-code-aux-keys assms* **by** *presburger*
qed (*auto simp*: *assms GS-code-aux-invar invar-GS-iter*)
also have ... = *MDP.d-GS-least* (*MDP.gs-iteration eps* (*V-Map.map-to-bfun v*))
using *GS-code-correct*[*of eps v*] *assms* **by** (*auto simp*: *aux*)
finally show *?thesis* **unfolding** *MDP.vi-gs-policy-def* **by** *auto*
qed

end

lemma *inorder-empty*: *Tree2.inorder am* = [] $\Longrightarrow$ *am* = $\langle\rangle$
using *Tree2.inorder.elims* **by** *blast*

context *MDP-nat-disc*
begin

lemma *dist-opt-bound-*$\mathcal{L}_b$*-split*: *dist v* ν_b*-opt* $\leq$ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) / ($1 - l$)
using *contraction-*$\mathcal{L}$*-split-dist*
by (*simp add*: *mult.commute mult-imp-le-div-pos*)

lemma *cert-*$\mathcal{L}_b$*-split*:
assumes $\varepsilon \geq 0$ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) / ($1 - l$) $\leq \varepsilon$
shows *dist v* ν_b*-opt* $\leq \varepsilon$
using *assms dist-opt-bound-*$\mathcal{L}_b$*-split order-trans* **by** *auto*

definition *check-value-GS eps v* $\longleftrightarrow$ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) / ($1 - l$) $\leq$ *eps*

definition *gs-policy-bound-error v* = (
let v′ = (*GS.*$\mathcal{L}_b$*-split v*); *err* = ($2 * l$) $*$ *dist v v′* / ($1 - l$) *in*
(*err*, *d-GS-least v′*))

lemma $\mathcal{L}_b$*-split-eq-L-opt*: *GS.*$\mathcal{L}_b$*-split v* = *GS.L-split* (*d-GS-least v*) *v*
by (*simp add*: *GS-inv-def* $\mathcal{L}_b$*-split-eq-GS-inv*)

lemma *L-split-fix-*ν:
assumes $d \in D_D$
assumes *GS.L-split d v* = *v*
shows $v = \nu_b$ (*mk-stationary-det d*)
proof –
have *r-dec*$_b$ (*mk-dec-det d*) = (*id-blinfun* $-$ $l *_R \mathcal{P}_1$ (*mk-dec-det d*))

v
  using *GS-inv-rec*[*of d v*]
  unfolding *GS-inv-def assms*(*2*) $\mathcal{P}_1$*-sum-lower-upper*
  by (*auto simp*: *blinfun.bilinear-simps algebra-simps*)
 hence $v = (\sum t.\ (l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d))^{\frown\frown}t)\ (r\text{-}dec_b\ (mk\text{-}dec\text{-}det\ d))$
  using *inv-norm-le*$'$(*2*)[*OF norm-*$\mathcal{P}_1$*-l-less*] **by** *auto*
 thus $v = \nu_b\ (mk\text{-}stationary\ (mk\text{-}dec\text{-}det\ d))$
  by (*auto simp*: ν*-stationary blincomp-scaleR-right*)
qed

lemma
 assumes $gs\text{-}policy\text{-}bound\text{-}error\ v = (err,\ d)$
 shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ d))\ \nu_b\text{-}opt \le err$
proof (*cases* $l = 0$)
 case *True*
 hence $gs\text{-}policy\text{-}bound\text{-}error\ v = (0,\ d\text{-}GS\text{-}least\ (GS.\mathcal{L}_b\text{-}split\ v))$
  unfolding *gs-policy-bound-error-def* **by** *auto*
 have $GS.\mathcal{L}_b\text{-}split\ v = GS.\mathcal{L}_b\text{-}split\ \nu_b\text{-}opt$
  by (*auto simp*: $GS.\mathcal{L}_b$*-split.rep-eq R-GS-def GS.*$\mathcal{L}$*-split-def simp del*: $GS.\mathcal{L}_b$*-split-fix intro*!: *bfun-eqI*)
   (*simp add*: *True*)
 hence $GS.\mathcal{L}_b\text{-}split\ v = \nu_b\text{-}opt$
  by *auto*
 hence $\nu_b\ (mk\text{-}stationary\text{-}det\ (d\text{-}GS\text{-}least\ (GS.\mathcal{L}_b\text{-}split\ v))) = \nu_b\text{-}opt$
  using $GS.\mathcal{L}_b$*-split-fix GS-inv-def Q-GS-def R-GS-def True* $\mathcal{L}_b$*-split-eq-GS-inv* ν*-stationary-inv*
  by *force*
 then show *?thesis*
  using *assms* **unfolding** *gs-policy-bound-error-def*
  by (*auto simp*: *True*)
next
 case *False*
 then show *?thesis*
 proof (*cases* $GS.\mathcal{L}_b\text{-}split\ v = v$)
  case *True*
  have *v-opt*: $v = \nu_b\text{-}opt$
   using $GS.\mathcal{L}_b$*-lim*(*1*) $GS.\mathcal{L}_b$*-split-fix True* **by** *blast*
  have $*$: $(\nu_b\ (mk\text{-}stationary\text{-}det\ d) = v) = (GS.L\text{-}split\ d\ v = v)$ **if** $d \in D_D$ **for** $d\ v$
   using *that L-split-fix-*ν *GS.L-split-fix* **by** *auto*

  have $GS.L\text{-}split\ (d\text{-}GS\text{-}least\ \nu_b\text{-}opt)\ \nu_b\text{-}opt = \nu_b\text{-}opt$
   using $GS.\mathcal{L}_b$*-split-fix* $\mathcal{L}_b$*-split-eq-L-opt* **by** *auto*
  hence $\nu_b\ (mk\text{-}stationary\text{-}det\ (d\text{-}GS\text{-}least\ (GS.\mathcal{L}_b\text{-}split\ v))) = \nu_b\text{-}opt$
   using *d-GS-least-is-dec* **by** (*auto simp*: *v-opt* $*$)
  then show *?thesis*
   using *assms* **unfolding** *gs-policy-bound-error-def*

by (*auto simp*: *True*)
next
case *False*
hence *1*: *dist v* (*GS*.$\mathcal{L}_b$-*split v*) > *0*
by *fastforce*
hence *2* ∗ *l* ∗ *dist v* (*GS*.$\mathcal{L}_b$-*split v*) > *0*
using ‹*l* ≠ *0*› *zero-le-disc* **by** (*simp add*: *less-le*)
hence *err* > *0*
using *assms* **unfolding** *gs-policy-bound-error-def* **by** *auto*
hence *dist* (ν_b (*mk-stationary-det* (*d-GS-least* (*GS*.$\mathcal{L}_b$-*split v*))))
ν_b-*opt* < *err′* **if** *err* < *err′* **for** *err′*
using *that assms*
unfolding *gs-policy-bound-error-def*
by (*auto simp*: *pos-divide-less-eq*[*symmetric*] *intro*: *find-policy-error-bound-gs*)
then show *?thesis*
using *assms* **unfolding** *gs-policy-bound-error-def* **by** *force*
qed
qed

end

context *MDP-Code*
begin
definition *gs-policy-bound-error-code v* = (
let v′ = *fst* (*GS-iter v*);
d = *if states* = *0 then 0 else* (*MAX s* ∈ {..< *states*}. *dist* (*v-lookup*
v s) (*v-lookup v′ s*));
err = (*2* ∗ *l*) ∗ *d* / (*1* − *l*) *in*
(*err*, *find-policy-code v′*))

lemma
assumes *v-len v* = *states v-invar v*
shows *D-Map.map-to-fun* (*snd* (*gs-policy-bound-error-code v*)) = *snd*
(*MDP.gs-policy-bound-error* (*V-Map.map-to-bfun v*))
unfolding *MDP.gs-policy-bound-error-def gs-policy-bound-error-code-def*
by (*simp add*: *GS-iter′-correct GS-iter-aux-correct″ assms invar-GS-iter*)

lemma
assumes *v-len v* = *states v-invar v*
shows (*fst* (*gs-policy-bound-error-code v*)) = *fst* (*MDP.gs-policy-bound-error*
(*V-Map.map-to-bfun v*))
proof−
have *dist-zero-ge*: *dist* ((*V-Map.map-to-bfun v*) *x*) ((*V-Map.map-to-bfun*
(*fst* (*GS-iter v*))) *x*) = *0* **if** *x* ≥ *states* **for** *x*
using *assms that*
by (*auto simp*: *V-Map.map-to-bfun.rep-eq split*: *option.splits*)
have *univ*: *UNIV* = {*0*..<*states*} ∪ {*states*..} **by** *auto*
let *?d* = λ*x*. *dist* ((*V-Map.map-to-bfun v*) *x*) ((*V-Map.map-to-bfun*

(*fst* (*GS-iter v*))) *x*)

have *fin*: *finite* (*range* (λ*x*. *?d x*))
by (*auto simp*: *dist-zero-ge univ Set.image-Un Set.image-constant*[*of states*])
have *r*: *range* (λ*x*. *?d x*) = *?d* ‘ {..<*states*} ∪ *?d* ‘ {*states*..}
by *force*
hence *Sup* (*range ?d*) = *Max* (*range ?d*)
using *fin cSup-eq-Max* **by** *blast*
also have ... = (*if states* = *0 then* (*Max* (*?d* ‘ {*states*..})) *else max* (*Max* (*?d* ‘ {..<*states*})) (*Max* (*?d* ‘ {*states*..})))
using *r fin* **by** (*auto intro*: *Max-Un*)
also have ... = (*if states* = *0 then 0 else max* (*Max* (*?d* ‘ {..<*states*})) *0*)
using *dist-zero-ge*
by (*auto simp*: *Set.image-constant*[*of states*] *cSup-eq-Max*[*symmetric*, *of* (λ-. *0*) ‘ {*states*..}])
also have ... = (*if states* = *0 then 0 else* (*Max* (*?d* ‘ {..<*states*})))
by (*auto intro*!: *max-absorb1 max-geI*)
finally have *1*: *Sup* (*range ?d*) = (*if states* = *0 then 0 else* (*Max* (*?d* ‘ {..<*states*}))).
thus *?thesis*
unfolding *MDP.gs-policy-bound-error-def gs-policy-bound-error-code-def dist-bfun-def*
using *assms GS-iter′-correct GS-iter-aux-correct″ invar-GS-iter*
apply *auto*
using *GS-iter-aux-correct GS-iter-def MDP.GS-rec-upto-correct V-Map.map-to-fun-def map-to-bfun-eq-fun* **by** *auto*
qed

end

global-interpretation *GS-Code*: *MDP-Code*

IArray.sub λ*n x arr*. *IArray* ((*IArray.list-of arr*)[*n*:= *x*]) *IArray.length IArray IArray.list-of* λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder rbt

MDP.transitions (*Rep-Valid-MDP mdp*) *MDP.states* (*Rep-Valid-MDP mdp*)

starray-get λ*i x arr*. *starray-set arr i x starray-length starray-of-list* λ*arr*. *starray-foldr* (λ*x xs*. *x* # *xs*) *arr* [] λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder rbt

MDP.disc (*Rep-Valid-MDP mdp*)

for *mdp states l*
defines *GS-code* = *GS-Code.GS-code*
  and *find-policy-code* = *GS-Code.find-policy-code*
  and *GS-policy-code* = *GS-Code.GS-policy-code*
  and *GS-code-aux* = *GS-Code.GS-code-aux*
  and *check-dist* = *GS-Code.check-dist*
  and *GS-iter* = *GS-Code.GS-iter*
  and *GS-iter-aux* = *GS-Code.GS-iter-aux*
  and *$\mathcal{L}$-GS-code* = *GS-Code.$\mathcal{L}$-GS-code*
  and L_a*-code* = *GS-Code.*L_a*-code*
  and *a-lookup'* = *GS-Code.a-lookup'*
  and *d-lookup'* = *GS-Code.d-lookup'*
  and *v0* = *GS-Code.v0*
  and *find-policy-code-aux* = *GS-Code.find-policy-code-aux*
  and *find-policy-code-aux-upt* = *GS-Code.find-policy-code-aux-upt*
  and *find-policy-state-code-aux'* = *GS-Code.find-policy-state-code-aux'*
  and *find-policy-state-code-aux* = *GS-Code.find-policy-state-code-aux*
  and *entries* = *M.entries*
  and *from-list* = *M.from-list*
  and *arr-tabulate* = *starray-Array.arr-tabulate*

and *v-map-from-list* = *GS-Code.v-map-from-list*
and *gs-policy-bound-error-code* = *GS-Code.gs-policy-bound-error-code*
  using *Rep-Valid-MDP*
  by *unfold-locales*
  (*fastforce simp*: *pmf-of-list-wf-def Ball-set-list-all*[*symmetric*] *case-prod-beta is-MDP-def M.invar-def M.entries-def M.is-empty-def RBT-Set.empty-def length-0-conv*[*symmetric*])+

lemmas *entries-def*[*unfolded M.entries-def*, *code*]
lemmas *from-list-def*[*unfolded M.from-list-def*, *code*]
lemmas *arr-tabulate-def*[*unfolded starray-Array.arr-tabulate-def*, *code*]

end
theory *GS-Code-Export-Float*
  imports
    GS-Code
    Code-Real-Approx-By-Float-Fix
begin

export-code
  v-map-from-list
  to-valid-MDP MDP GS-policy-code v0 gs-policy-bound-error-code

RBT-Map.update nat-map-from-list assoc-list-to-MDP RBT-Set.empty nat-pmf-of-list pmf-of-list
nat-of-integer Ratreal int-of-integer inverse-divide Tree2.inorder integer-of-nat
in *SML* **module-name** *GS-Code-Float* **file-prefix** *GS-Code-Float*

end
theory *GS-Code-Export-Rat*
imports
GS-Code
begin

export-code
quotient-of ord-real-inst.less-eq-real gs-policy-bound-error-code
plus-real-inst.plus-real minus-real-inst.minus-real v0 to-valid-MDP MDP RBT-Map.update
Rat.of-int divide divide-rat-inst.divide-rat divide-real-inst.divide-real nat-map-from-list
assoc-list-to-MDP nat-pmf-of-list RBT-Set.empty GS-policy-code pmf-of-list nat-of-integer Ratreal int-of-integer
inverse-divide Tree2.inorder integer-of-nat v-map-from-list
in *SML* **module-name** *GS-Code-Rat* **file-prefix** *GS-Code-Rat*
end

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

15 Modified Policy Iteration

locale *MDP-MPI* = *MDP-att-*$\mathcal{L}$ *A K r l* + *MDP-act-disc arb-act A K r l*
for A **and** K :: $'s$ :: *countable* $\times$ $'a$ :: *countable* $\Rightarrow$ $'s$ *pmf* **and** *r l arb-act*
begin

15.1 The Advantage Function B

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

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

lemma *cSUP-plus*:
assumes $X \neq \{\}$ *bdd-above* $(f`X)$
shows $(\bigsqcup x \in X.\ f\ x + c) = (\bigsqcup x \in X.\ f\ x) + (c{::}\textit{real})$

proof (*rule antisym*)
 show ($\bigsqcup x{\in}X.\ f\ x + c) \le \bigsqcup\ (f\ `\ X) + c$
  using *assms* **by** (*fastforce intro*: *cSUP-least cSUP-upper*)
 show $\bigsqcup\ (f\ `\ X) + c \le (\bigsqcup x{\in}X.\ f\ x + c)$
  unfolding *le-diff-eq*[*symmetric*]
  using *assms*
  by (*intro cSUP-least*) (*auto simp add*: *algebra-simps bdd-above-def intro*!: *cSUP-upper2 intro*: *add-left-mono*)+
qed

lemma *cSUP-minus*:
 assumes $X \neq \{\}$ *bdd-above* ($f`X$)
 shows $(\bigsqcup x \in X.\ f\ x - c) = (\bigsqcup x \in X.\ f\ x) - (c{::}real)$
 using *cSUP-plus*[*OF assms*, *of* $- c$] **by** *auto*

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*)
 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 *?thesis*
  unfolding $*$ *$\mathcal{L}$-def* **using** *ex-dec* **by** (*fastforce intro*!: *cSUP-minus*)
qed

B is a bounded function.

lift-definition B_b :: $('s \Rightarrow_b real) \Rightarrow 's \Rightarrow_b real$ **is** B
 unfolding *B-eq-$\mathcal{L}$* **using** *$\mathcal{L}$-bfun* **by** (*auto intro*: *Bounded-Functions.minus-cont*)

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*

15.2 Optimization of the Value Function over Multiple Steps

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

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

lemma $U_b\text{-}le\text{-}\mathcal{L}_b$: $U_b\ m\ v \le (\mathcal{L}_b\ \widehat{}\widehat{}\ m)\ v$
proof −
have $U_b\ m\ v\ s = (\bigsqcup d \in D_R.\ (L\ d\widehat{}\widehat{}\ m)\ v\ s)$ **for** $m\ v\ s$
by $(auto\ simp\colon L\text{-}iter\ U_b.rep\text{-}eq\ \mathcal{L}_b.rep\text{-}eq\ U\text{-}def\ \mathcal{L}\text{-}def)$
thus *?thesis*
using $L\text{-}iter\text{-}le\text{-}\mathcal{L}_b\ ex\text{-}dec$ **by** $(fastforce\ intro!\colon cSUP\text{-}least)$
qed

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

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

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

lemma U_b-*fix-unique*: $U_b\ (Suc\ m)\ v = v \longleftrightarrow v = \nu_b$-*opt*
  using *theI'*[*OF* U_b-*conv*(*1*)] U_b-*conv*(*1*)
  by (*auto simp*: *LIMSEQ-unique*[*OF* U_b-*tendsto* U_b-*conv*(*2*)[*of m*]])

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

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

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

using *L-iter* **by** *auto*
finally show *?thesis* **.**
qed

lemma U_b*-ge*: $d \in D_R \Longrightarrow U_b\ m\ u \geq \nu_b$*-fin* (*mk-stationary* d) m + (($l *_R \mathcal{P}_1\ d$) $\widehat{\ }\widehat{\ }$ m) u
using ν*-improving-D-MR 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*)

lemma *W-le-*U_b:
assumes $v \leq u$ ν*-improving* $v\ d$
shows $W\ d\ m\ v \leq U_b\ m\ u$
using *assms*
by (*fastforce simp*: *W-eq-L-iter intro*!: *order.trans*[*OF L-iter-le-*U_b U_b*-mono*])

lemma *W-ge-*$\mathcal{L}_b$:
assumes $v \leq u$ $0 \leq B_b\ u$ ν*-improving* $u\ d'$
shows $\mathcal{L}_b\ v \leq W\ d'$ (*Suc* m) u
proof −
have $\mathcal{L}_b\ v \leq u + B_b\ u$
using *assms*(*1*) $\mathcal{L}_b$*-mono* B_b*-eq-*$\mathcal{L}_b$ **by** *auto*
also have ... $\leq W\ d'$ (*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-blinfun*) ($u - v$) $\leq B_b\ u$
proof −
have *r-dec*$_b$ d + ($l *_R \mathcal{P}_1\ d$ − *id-blinfun*) $u \leq B_b\ u$
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-dec*$_b$ d + ($l *_R \mathcal{P}_1\ d$ − *id-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

15.4 Computing the Bellman Operator over Multiple Steps

definition *L-pow* $v\ d\ m$ = (L (*mk-dec-det* d) $\widehat{\ }\widehat{\ }$ m) v

lemma *L-pow-eq*:
fixes d **defines** $d' \equiv$ *mk-dec-det d*
assumes ν*-improving* $v\ d'$
shows *L-pow* $v\ d\ m = v + (\sum i < m.\ ((l *_R \mathcal{P}_1\ d')^{\frown\frown} i))\ (B_b\ v)$
using *L-pow-def W-def W-eq-L-iter assms* **by** *presburger*

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*)) $m\ v$
using *assms policy-improvement-improving* **by** (*auto simp*: *W-eq-L-iter L-pow-def*)

lemma *find-policy′-is-dec-det*: *is-dec-det* (*find-policy′ v*)
using *find-policy′-def is-dec-det-def some-opt-acts-in-A* **by** *presburger*

lemma *find-policy′-improving*: ν*-improving v* (*mk-dec-det* (*find-policy′ v*))
using ν*-improving-opt-acts find-policy′-def* **by** *presburger*

lemma *L-pow-eq-W′*: *L-pow v* (*find-policy′ v*) $m = W$ (*mk-dec-det* (*find-policy′ v*)) $m\ v$
using *find-policy′-improving* **by** (*auto simp*: *W-eq-L-iter L-pow-def*)

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 $0 \le ((l *_R \mathcal{P}_1\ d)\ ^{\frown\frown}\ m)\ (B_b\ u)$
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 $\ldots = ((l *_R \mathcal{P}_1\ d)^{\frown\frown} 0 + (\sum i < m.\ (l *_R \mathcal{P}_1\ d)^{\frown\frown}(Suc\ i)))\ (B_b\ u) - (\sum i < m.\ (l *_R \mathcal{P}_1\ d)^{\frown\frown}\ i)\ (B_b\ u)$
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-blinfun*$)\ o_L\ (\sum i < m.\ (l *_R \mathcal{P}_1\ d)^{\frown\frown} i))\ (B_b\ u)$
by (*auto simp*: *blinfun.bilinear-simps sum-subtractf*)
also have $\ldots = B_b\ u + (l *_R \mathcal{P}_1\ d -$ *id-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

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*

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

15.5 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 = (*find-policy' v0*, *v0*) |
mpi (*Suc n*) =
(*let* (*d*, *v*) = *mpi n*; *v'* = *L-pow v d* (*Suc* (*m n v*)) *in*
(*find-policy' v'*, *v'*))

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

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

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

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*

15.6 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-val n*))) (*w n*)

lemma *dist*-ν_b-*opt*: *dist* (*w* (*Suc n*)) ν_b-*opt* ≤ *l* ∗ *dist* (*w n*) ν_b-*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*) ν_b-*opt* ≤ *l*^*n* ∗ *dist v0* ν_b-*opt*
 by (*induction n*) (*fastforce simp*: *algebra-simps intro*: *order.trans*[*OF dist*-ν_b-*opt*] *mult-left-mono*)+

lemma *w-conv*: *w* ⟶ ν_b-*opt*
proof −
 have (λ*n*. *l*^*n* ∗ *dist v0* ν_b-*opt*) ⟶ *0*
 using *LIMSEQ-realpow-zero* **by** (*cases v0* = ν_b-*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 $v0 \le \mathcal{L}_b\ v0$
shows *mpi-val* $\longrightarrow \nu_b$*-opt* **and** $\bigwedge n.$ *mpi-val n* $\le$ *mpi-val* (*Suc n*)
proof –
define *y* **where** *y n* = $(\mathcal{L}_b$ ^^ $n)$ *v0* **for** *n*
have *aux*: *mpi-val n* $\le \mathcal{L}_b$ (*mpi-val n*) $\wedge$ *mpi-val n* $\le$ *mpi-val* (*Suc n*) $\wedge$ *y n* $\le$ *mpi-val n* $\wedge$ *mpi-val n* $\le$ *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-val* (*Suc n*) = *W* (*mk-dec-det* (*mpi-pol n*)) (*Suc* (*m n* (*mpi-val n*))) (*mpi-val n*)
using ν*-improving-mpi-pol mpi-val-Suc W-eq-L-iter L-pow-def* **by** *auto*
hence $*$: *mpi-val* (*Suc n*) $\le \mathcal{L}_b$ (*mpi-val* (*Suc n*))
using *Suc.IH* $\mathcal{L}_b$*-W-ge* ν*-improving-mpi-pol* **by** *presburger*
moreover have *mpi-val* (*Suc n*) $\le$ *mpi-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*) $\le$ *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*) $\le$ *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* $\le$ *mpi-val* (*Suc n*) **for** *n*
by *auto*
have *y* $\longrightarrow \nu_b$*-opt*
using $\mathcal{L}_b$*-lim y-def* **by** *presburger*
thus *mpi-val* $\longrightarrow \nu_b$*-opt*
using *aux* **by** (*auto intro*: *tendsto-bfun-sandwich*[*OF - w-conv*])
qed

15.7 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 ... $\leq$ (*l*/ (*1*–*l*)) $*$ (*eps* $*$ (*1* – *l*) / (*2* $*$ *l*))
using $**$ **by** (*fastforce intro*!: *mult-left-mono simp*: *divide-nonneg-pos*)
also have ... = *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 $\exists n.\ 2 * l * dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) < eps * (1 - l)$
proof −
have $(\lambda n.\ dist\ (mpi\text{-}val\ n)\ \nu_b\text{-}opt) \longrightarrow 0$
using *mpi-conv*(*1*)[*OF assms*(*1*)] *tendsto-dist-iff*
by *blast*
hence $(\lambda n.\ dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n))) \longrightarrow 0$
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

15.8 Unbounded MPI

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

function (*domintros*) *mpi-algo* **where** *mpi-algo d v m* = (
if $2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1 - l)$
then (*find-policy′ v*, *v*)
else mpi-algo (*find-policy′ v*) (*L-pow v* (*find-policy′ v*) (*Suc* (*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 = (*find-policy′ v*, *v*) |
mpi′ d v (*Suc n*) *m* = (
let d′ = *find-policy′ v*; *v′* = *L-pow v d′* (*Suc* (*m 0 v*)) *in mpi′ d′ v′ n* ($\lambda n.$ *m* (*Suc n*)))

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

lemma
  assumes $d \in D_D$
  shows $mpi\ v\ m\ n = mpi'\ d\ v\ n\ m$
  using *assms*
proof (*induction n arbitrary*: *d v m rule*: *nat.induct*)
  case (*Suc nat*)
  thus *?case*
   using *find-policy'-is-dec-det* **by** (*fastforce simp*: *Let-def mpi-Suc'*[*OF Suc*(*2*)])
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** $n = (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ v\ m\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ v\ m\ n)) < eps * (1 - l))$ (**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 $2 * l * dist\ (mpi\text{-}val\ v\ m\ 0)\ (\mathcal{L}_b\ (mpi\text{-}val\ v\ m\ 0)) < eps * (1 - l)$
      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 $?d = find\text{-}policy'\ v$
    have $Suc\ n = Suc\ (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ v\ m\ (Suc\ n))\ (\mathcal{L}_b\ (mpi\text{-}val\ v\ m\ (Suc\ n))) < eps * (1 - l))$
      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 $n = (LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ v\ m\ (Suc\ n))\ (\mathcal{L}_b\ (mpi\text{-}val\ v\ m\ (Suc\ n))) < eps * (1 - l))$
      by *auto*
    hence *n-eq*: $n =$
    $(LEAST\ n.\ 2 * l * dist\ (mpi\text{-}val\ (L\text{-}pow\ v\ ?d\ (Suc\ (m\ 0\ v)))\ (\lambda a.$

m (Suc a)) n) ($\mathcal{L}_b$ ($mpi\text{-}val$ ($L\text{-}pow$ v $?d$ (Suc (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* ($?d$, $L\text{-}pow\ v\ ?d\ (Suc\ (m\ 0\ v))$, $\lambda a.$ m (Suc a))
apply (*rule Suc.IH*[*OF n-eq* ‹$0 < eps$›])
using *Suc.prems is-dec-det-pi L-pow-*$\mathcal{L}_b$*-mono-inv' find-policy'-is-dec-det* **by** *auto*
ultimately show *?case*
using *mpi-algo.domintros* **by** *blast*
qed
qed

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

lemma *mpi-algo-def'*:
assumes $d \in D_D$ $v \leq \mathcal{L}_b\ v$ $eps > 0$
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-def''*:
assumes $d \in D_D$ $v \leq \mathcal{L}_b\ v$ $eps > 0$
shows *mpi-algo d v m* = (
let $v' = \mathcal{L}_b\ v$; d' = *find-policy'* v *in*
if $2 * l * dist\ v\ v' < eps * (1 - l)$
then (d', v)
else mpi-algo d' ($L\text{-}pow$ v' d' ((m 0 v))) ($\lambda n.$ m (Suc n)))
proof –
have ν*-improving* v (*mk-dec-det* (*find-policy'* v))
using ν*-improving-opt-acts find-policy'-def* **by** *presburger*
hence *aux*: $L\text{-}pow$ ($\mathcal{L}_b$ v) (*find-policy'* v) n = $L\text{-}pow$ v (*find-policy'* v) (Suc n) **for** n
using ‹$d \in D_D$› ν*-improving-imp-*$\mathcal{L}_b$
by (*auto simp*: *funpow-swap1 L-pow-def*)
show *?thesis*
unfolding *mpi-algo-def'*[*OF assms*] *Let-def aux*[*symmetric*] **by** *auto*
qed

lemma *mpi-algo-eq-mpi*:
assumes $d \in D_D$ $v \leq \mathcal{L}_b\ v$ $eps > 0$
shows *mpi-algo d v m* = *mpi v m* (*LEAST* $n.$ $2 * l * dist$ (*mpi-val*

v m n) ($\mathcal{L}_b$ (*mpi-val v m n*)) $<$ *eps* $*$ (*1* $-$ *l*))
proof $-$
define *n* **where** *n* = (*LEAST n.* *2* $*$ *l* $*$ *dist* (*mpi-val v m n*) ($\mathcal{L}_b$ (*mpi-val 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*: $\neg$ (*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 v m n'*) ($\mathcal{L}_b$ (*mpi-val 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* (*find-policy' v*) (*L-pow v* (*find-policy' v*) (*Suc* (*m 0 v*))) (λ*n. m* (*Suc n*)) *n*)
using *Suc*(*3*) *mpi-Suc'* **by** (*auto simp*: *mpi-val-def*)
hence *mpi-algo d v m* = *mpi 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' find-policy'-is-dec-det*
by *fastforce*
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* = λ*n.* *2* $*$ *l* $*$ *dist* (*mpi-val v0 m n*) ($\mathcal{L}_b$ (*mpi-val v0 m n*)) $<$ *eps* $*$ (*1* $-$ *l*)
let *?n* = *Least ?P*
have *mpi-algo d v0 m* = *mpi 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

15.9 Initial Value Estimate *v0-mpi*

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

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

lift-definition $v0\text{-}mpi_b$ :: $'s \Rightarrow_b real$ **is** *v0-mpi*
  by (*auto simp*: *v0-mpi-def*)

lemma $v0\text{-}mpi_b\text{-}le\text{-}\mathcal{L}_b$: $v0\text{-}mpi_b \le \mathcal{L}_b\ v0\text{-}mpi_b$
proof (*rule less-eq-bfunI*)
  fix *x*
  have *bounded-r'*: *bounded* (($\lambda a.\ r\ (x, a)$) ‘ *A x*) **for** *x*
    using *r-bounded'*
    unfolding *bounded-def*
    by *simp* (*meson UNIV-I*)
  have ∗: ($\bigsqcap a{\in}A\ x.\ r\ (x, a)$) $\le r\ (x,a)$ **if** $a \in A\ x$ **for** *a x*
    using *bounded-r' that*
    by (*auto intro*!: *cInf-lower bounded-imp-bdd-below*)
  have ∗∗∗∗: $r\ (s,a) \le r_M$ **for** *s a*
    using *abs-le-iff abs-r-le-*r_M **by** *blast*
  have ∗∗: *bounded* (*range* ($\lambda s'.\ \bigsqcap a{\in}A\ s'.\ r\ (s', a)$))
    using *abs-r-le-*r_M *ex-dec-det is-dec-det-def A-ne*
      by (*auto simp add*: *minus-le-iff abs-le-iff intro*!: *cINF-greatest order.trans*[*OF* ∗] *boundedI*[*of* - r_M])
  have *r-min* $\le r\ (s, a)$ **if** $a \in A\ s$ **for** *s a*
    using *r-bounded' that* ∗∗
    by (*auto intro*!: *bounded-imp-bdd-below cInf-lower2*[*OF* - ∗])

    hence *r-min* $\le (1{-}l) * r\ (s, a) + l *$ *r-min* **if** $a \in A\ s$ **for** *s a*
      using *disc-lt-one zero-le-disc that* **by** (*meson order-less-imp-le order-refl segment-bound-lemma*)
  hence *r-min* / (*1* − *l*) $\le$ ((*1*−*l*) ∗ *r* (*s*, *a*) + *l* ∗ *r-min*) / (*1* − *l*) **if** $a \in A\ s$ **for** *s a*
    using *order-less-imp-le*[*OF disc-lt-one*] *that* **by** (*auto intro*!: *divide-right-mono*)
  hence *r-min* / (*1* − *l*) $\le r\ (s, a)$ + (*l* ∗ *r-min*) / (*1* − *l*) **if** $a \in A\ s$ **for** *s a*
    using *disc-lt-one that* **by** (*auto simp*: *add-divide-distrib*)
  hence *r-min* / (*1* − *l*) $\le L_a$ (*arb-act* (*A x*)) (λs. *r-min* / (*1* − *l*)) *x*
    using *A-ne arb-act-in* **by** *auto*
  moreover have *bdd-above* (($\lambda a.\ L_a\ a$ (λs. *r-min* / (*1* − *l*)) *x*) ‘ *A x*)
    using *r-bounded*
   by (*fastforce simp*: *bounded-def intro*!: *bounded-imp-bdd-above bounded-plus-comp*)
  ultimately show $v0\text{-}mpi_b\ x \le \mathcal{L}_b\ v0\text{-}mpi_b\ x$
   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*)

qed

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

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

lemma *mpi-user-eq*:
assumes *eps* > *0*
shows *mpi-user eps* = *mpi-alg-rec eps* (λ*x. 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 *MPI-Code*
imports
Code-Setup
../Modified-Policy-Iteration
HOL−Library.Code-Target-Numeral
begin

sublocale *MDP-nat-disc* ⊆ *MDP-MPI*
by *unfold-locales*

context *MDP-Code* **begin**

definition *d0* = *D-Map.from-list′* (λ*s. fst* (*hd* (*a-inorder* (*s-lookup mdp s*)))) [*0..*<*states*]

definition *r-min-code* =
min 0 (*MIN s* ∈ *set* [*0..*<*states*]. *MIN* (-, *r*, -) ∈ *set* (*a-inorder* (*s-lookup mdp s*)). *r*)

definition *v0-code* = *V-Map.arr-tabulate* (λ*s. r-min-code* / (*1* − *l*)) *states*

definition *d0-code* = *D-Map.from-list′* (λ*s. fst* (*hd* (*a-inorder* (*s-lookup mdp s*)))) [*0..*<*states*]

definition *find-policy-L-code v* =
fold (λ*s* (*d'*, *v'*).
let (*ds*, *vs*) = *find-policy-state-code-aux' v s in*
(*d-update s ds d'*, *v-update s vs v'*)) [*0..<states*] (*d-empty*, *V-Map.arr-tabulate* (λ-. *0*) *states*)

definition *find-policy-L-code' v* =
fold (λ*s* (*d'*, *v'*).
let (*ds*, *vs*) = *find-policy-state-code-aux' v s in*
(*d-update s ds d'*, *v-update s vs v'*)) [*0..<states*] (*d-empty*, *v*)

lemma *fold-prod*: *fold* (λ*x* (*a1*, *a2*). (*f x a1*, *g x a2*)) *xs* (*z1*, *z2*) =
(*fold f xs z1*, *fold g xs z2*)
by (*induction xs arbitrary*: *z1 z2*) *auto*

lemma *s-lookup-entries-eq*:
assumes *s* < *states*
shows {(*a*, *r*, *pmf-of-list k*) | *a r k*. (*a*, *r*, *k*) ∈ *A-Map.entries* (*s-lookup mdp s*)}
= {(*a*, *MDP-r* (*s*,*a*), *MDP-K* (*s*,*a*)) | *a* . *a* ∈ *MDP-A s*}
proof −
have ∃ *k*. *MDP-K* (*s*, *a*) = *pmf-of-list k* ∧ (*a*, *MDP-r* (*s*, *a*), *k*) ∈ *A-Map.entries* (*s-lookup mdp s*)
if *a* ∈ *MDP-A s* **for** *a*
by (*metis a-map-entries-lookup fst-sa-lookup'-eq assms prod.collapse snd-sa-lookup'-eq that*)
thus *?thesis*
using *entries-A-eq-K assms entries-A-eq-r*
by (*auto simp*: *a-inorderD*(*1*))
qed

lemma *a-lookup-entries*: *A-Map.invar m* ⟹ *kv* ∈ *A-Map.entries m* ⟹ *a-lookup' m* (*fst kv*) = *snd kv*
by (*metis A-Map.inorder-lookup-Some a-lookup'-def option.case*(*2*) *prod.collapse*)

lemma *a-inorder-eq-MDP-A*: *x* < *states* ⟹ *fst* ‘ *set* (*a-inorder* (*s-lookup mdp x*)) = *MDP-A x*
using *A-Map.keys-def MDP-A-def* **by** *presburger*

lemma *find-policy-L-code-split*:
assumes *v-len v* = *states v-invar v*
shows *fst* (*find-policy-L-code v*) = *vi-find-policy-code v*
⋀*i*. *i* < *states* ⟹ *v-lookup* (*snd* (*find-policy-L-code v*)) *i* = *v-lookup* (ℒ*-code v*) *i*
v-len (*snd* (*find-policy-L-code v*)) = *states*
v-invar (*snd* (*find-policy-L-code v*))
proof (*goal-cases*)

have ∗∗: $(x,y) \in$ *A-Map.entries* ((*s-lookup mdp i*)) $\Longrightarrow$ (*a-lookup'* (*s-lookup mdp i*) *x*) = *y*
if $i <$ *states* **for** *i x y*
by (*simp add*: *A-Map.inorder-lookup-Some a-lookup'-def invar-s-lookup that*)

have ∗: *find-policy-L-code v* =
(*vi-find-policy-code v*,
fold (λ*s. v-update s* (*snd* (*find-policy-state-code-aux' v s*))) [*0..<states*]

(*V-Map.arr-tabulate* (λ-*.0*) *states*))
unfolding *find-policy-L-code-def vi-find-policy-code-def*
by (*simp add*: *foldl-conv-fold case-prod-beta fold-prod D-Map.from-list'-def*)

have ∗∗:
v-lookup (*fold* (λ*s. v-update s* (*snd* (*find-policy-state-code-aux' v s*))) [*0..<states*] *v0*) *i* =
v-lookup (ℒ-*code v*) *i*
if $i <$ *states* **for** *i*
unfolding ℒ-*code-def* ℒ-*GS-code-def V-Map.arr-tabulate-def*
using *V-Map.invar-array v0-correct*
using *A-Map.is-empty-def A-Map.invar-def A-Map.entries-def*
using *ne-s-lookup invar-s-lookup a-lookup-entries*
using *that*
by (*auto simp*: *fold-max-eq-arg-max' image-image case-prod-beta find-policy-state-code-aux-eq*
V-Map.lookup-array v-lookup-fold)
case *1*
thus *?case* **using** ∗ **by** *auto*

case *3*
show *?case*
unfolding ∗
by (*auto simp*: *V-len-fold*)
case *4*
show *?case*
unfolding ∗
by (*auto simp*: *V-invar-fold*)
case (*2 i*) **thus** *?case*
using ∗∗
by (*auto simp*: ∗ *v0-def*)
qed

definition *L-code d v* =
V-Map.arr-tabulate (λ*s.* L_a-*code* (*a-lookup'* (*s-lookup mdp s*) (*d-lookup' d s*)) *v*) *states*

lemma *L-code-correct*:
assumes $s <$ *states v-len v* = *states v-invar v*

D-Map.keys d = MDP.state-space D-Map.invar d ($\bigwedge$*s. s < states* $\Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*)
shows
v-lookup (*L-code d v*) *s = MDP.L* (*MDP.mk-dec-det* (*D-Map.map-to-fun d*)) (*V-Map.map-to-bfun v*) *s*
using *assms*
unfolding *L-code-def MDP.L-eq-*L_a*-det*
by (*auto simp*: *map-to-fun-lookup L-GS-code-correct′*)

lemma *L-code-invar*: *v-invar* (*L-code d v*)
by (*simp add*: *L-code-def*)

lemma *L-code-keys*:
assumes *v-len v = states v-invar v*
D-Map.keys d = MDP.state-space D-Map.invar d ($\bigwedge$*s. s < states* $\Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*)
shows *v-len* (*L-code d v*) *= states*
by (*simp add*: *L-code-def*)

definition *L-pow-code v d m =* (*L-code d* $\widehat{\ }\widehat{\ }$ *m*) *v*

lemma *L-pow-code-Suc*: *L-pow-code v d* (*Suc m*) *= L-code d* (*L-pow-code v d m*)
by (*auto simp*: *L-pow-code-def*)

lemma *L-code-to-bfun*:
assumes *v-len v = states v-invar v*
D-Map.keys d = MDP.state-space D-Map.invar d ($\bigwedge$*s. s < states* $\Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*)
shows *V-Map.map-to-bfun* (*L-code d v*) *=*
MDP.L (*MDP.mk-dec-det* (*D-Map.map-to-fun d*)) (*V-Map.map-to-bfun v*)
proof (*rule bfun-eqI*)
fix *s*
show (*V-Map.map-to-bfun* (*L-code d v*)) *s =*
(*MDP.L* (*MDP.mk-dec-det* (*D-Map.map-to-fun d*)) (*V-Map.map-to-bfun v*)) *s*
proof (*cases s < states*)
case *True*
then show *?thesis*
using *L-code-correct assms*
by (*auto simp*: *L-code-def v-lookup-map-to-bfun*)
next
case *False*
then show *?thesis*
using *assms*
by (*subst MDP.L-zero*) (*auto simp*: *L-code-def V-Map.map-to-bfun.rep-eq split*: *option.splits*)
qed

qed

lemma *L-pow-code-correct*:
assumes *v-len v* = *states v-invar v*
D-Map.keys d = *MDP.state-space D-Map.invar d* ($\bigwedge$*s. s* < *states* $\Longrightarrow$ *d-lookup′ d s* ∈ *MDP-A s*)
shows
v-len (*L-pow-code v d m*) = *states*
v-invar (*L-pow-code v d m*)
V-Map.map-to-bfun (*L-pow-code v d m*) = ((*MDP.L-pow* (*V-Map.map-to-bfun v*) ((*D-Map.map-to-fun d*))) *m*)
using *assms*
proof (*induction m arbitrary*: *v*)
case (*Suc m*)
{
case *3*
then show *?case*
using *Suc*
by (*auto simp*: *L-pow-code-def L-code-to-bfun MDP.L-pow-def*)
}
qed (*auto simp add*: *L-pow-code-def L-code-to-bfun L-code-def MDP.L-pow-def*)

partial-function (*tailrec*) *mpi-partial-code* **where**
mpi-partial-code eps d v m =
(*let* (*d′*, *v′*) = *find-policy-L-code v in* (
if l = *0* ∨ *check-dist v v′ eps*
then (*d′*, *v*)
else mpi-partial-code eps d′ (*L-pow-code v′ d′ m*) *m*))

lemmas *mpi-partial-code.simps*[*code*]

lemma *vi-find-policy-code-correct′*:
assumes *v-len v-code* = *states v-invar v-code*
shows *d-lookup* (*vi-find-policy-code v-code*) *s* = (
if s < *states then Some* (*MDP.find-policy′* (*V-Map.map-to-bfun v-code*) *s*) *else None*)
using *assms vi-find-policy-code-correct*[*of v-code s*] *d-invar-vi-find-policy-code*
using *d-keys-vi-find-policy-code D-Map.lookup-None-set-inorder*[*of vi-find-policy-code v-code s*]
unfolding *MDP.find-policy′-def D-Map.map-to-fun-def*
by (*auto simp*: *least-arg-max-def MDP.is-opt-act-def vi-find-policy-code-notin split*: *option.splits*)

lemma L_a*-equiv*: (L_a*-code* (*a-lookup′* (*s-lookup mdp s*) (*d-lookup′ d s*)) *v*) = (L_a*-code* (*a-lookup′* (*s-lookup mdp s*) (*d-lookup′ d s*)) *v′*)
if $\bigwedge$*i. i* < *states* $\Longrightarrow$ *v-lookup v i* = *v-lookup v′ i s* < *states v-len v* = *states v-len v′* = *states v-invar v v-invar v′*
D-Map.keys d = *MDP.state-space D-Map.invar d* ($\bigwedge$*s. s* < *states*

$\Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*)
for *s v v′ d*
proof −
have *V-Map.map-to-bfun v* = *V-Map.map-to-bfun v′*
using *that*
by (*auto simp*: *V-Map.map-to-bfun.rep-eq*)
moreover have ∗: L_a*-code* (*a-lookup′* (*s-lookup mdp s*) (*d-lookup′ d s*)) *v* = *MDP*.L_a (*d-lookup′ d s*) (*apply-bfun* (*V-Map.map-to-bfun v*)) *s*
using *that snd-sa-lookup′-eq pmf-of-list-wf-mdp set-list-pmf-in-states*[*of s* (*d-lookup′ d s*)]
by (*subst* L_a*-code-correct*[*of s - -* (*d-lookup′ d s*)]) (*fastforce simp add*: *fst-sa-lookup′-eq*)+
ultimately show *?thesis*
unfolding ∗
using *that snd-sa-lookup′-eq pmf-of-list-wf-mdp set-list-pmf-in-states*[*of s d-lookup′ d s*]
by (*subst* L_a*-code-correct*[*of s - -* (*d-lookup′ d s*)]) (*auto simp add*: *fst-sa-lookup′-eq*)
qed

lemma *L-code-equiv*: *v-lookup* (*L-code d v*) *i* = *v-lookup* (*L-code d v′*) *i*
if $\bigwedge$*i*. *i* < *states* $\Longrightarrow$ *v-lookup v i* = *v-lookup v′ i i* < *states D-Map.keys d* = *MDP.state-space D-Map.invar d* ($\bigwedge$*s*. *s* < *states* $\Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*)
v-len v = *states v-len v′* = *states v-invar v v-invar v′*
unfolding *L-code-def*
using *that*
by (*auto intro*!: L_a*-equiv*)

lemma *L-pow-code-equiv*: *v-lookup* (*L-pow-code v d m*) *i* = *v-lookup* (*L-pow-code v′ d m*) *i* **if** $\bigwedge$*i*. *i* < *states* $\Longrightarrow$ *v-lookup v i* = *v-lookup v′ i i* < *states*
D-Map.keys d = *MDP.state-space D-Map.invar d* ($\bigwedge$*s*. *s* < *states* $\Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*) *v-len v* = *states v-len v′* = *states v-invar v v-invar v′*
for *v v′ d i m*
using *that L-code-invar*
proof (*induction m arbitrary*: *v v′ i*)
case *0*
then show *?case* **by** (*simp add*: *L-pow-code-def*)
next
case (*Suc m*)
thus *?case*
unfolding *L-pow-code-Suc*
using *L-pow-code-correct L-code-equiv*
by *presburger*
qed

lemma *map-to-bfun-snd-find-policy-L-code*:
assumes *v-len v-code* = *states v-invar v-code*
shows *V-Map.map-to-bfun* (*snd* (*find-policy-L-code v-code*)) = *V-Map.map-to-bfun*($\mathcal{L}$*-code v-code*)
using *invar-*$\mathcal{L}$*-code*
by (*auto simp*: *V-Map.map-to-bfun.rep-eq assms find-policy-L-code-split*)

lemma *mpi-partial-code-correct*:
fixes *eps d-code v-code m-code*

assumes *MDP.mpi-algo-dom eps* (*d*, *v*, *m*)
assumes *v* = *V-Map.map-to-bfun v-code*
assumes *d* = *D-Map.map-to-fun d-code*
assumes *m* = (λ(*a*::*nat*) (*b*:: *nat* $\Rightarrow_b$ *real*). *m-code*)
assumes *eps* > *0*
assumes *d* $\in$ *MDP.*D_D
assumes *v* $\leq$ *MDP.*$\mathcal{L}_b$ *v*
assumes *v-invar v-code*
assumes *v-len v-code* = *states*
shows
D-Map.map-to-fun (*fst* (*mpi-partial-code eps d-code v-code m-code*))
= *fst* (*MDP.mpi-algo eps d v m*)
V-Map.map-to-bfun (*snd* (*mpi-partial-code eps d-code v-code m-code*))
= *snd* (*MDP.mpi-algo eps d v m*)
proof −
have *MDP.mpi-algo eps d v m* = (*D-Map.map-to-fun* (*fst* (*mpi-partial-code eps d-code v-code m-code*)),
V-Map.map-to-bfun (*snd* (*mpi-partial-code eps d-code v-code m-code*)))
using *assms*
proof (*induction d v m arbitrary*: *d-code v-code m-code rule*: *MDP.mpi-algo.pinduct*)
case (*1 d v m*)
then show *?case*
proof (*cases l* = *0*)
case *True*
have $*$: *mpi-partial-code eps d-code v-code m-code* = (*let* (d', v') = *find-policy-L-code v-code in* (d', *v-code*)) **for** *v-code*
using *True mpi-partial-code.simps* **by** *presburger*
have *MDP.mpi-algo eps* (*D-Map.map-to-fun d-code*) (*V-Map.map-to-bfun v-code*) (λ*a b*. *m-code*) = (*MDP.find-policy'* *v*, *v*)
using *1 True MDP.mpi-algo.psimps*
by *auto*
also have ... = (*D-Map.map-to-fun* (*fst* (*mpi-partial-code eps d-code v-code m-code*)), *V-Map.map-to-bfun* (*snd* (*mpi-partial-code eps d-code v-code m-code*)))
using *1.prems*
by (*auto simp*: $*$ *case-prod-beta vi-find-policy-correct find-policy-L-code-split*)
finally show *?thesis*
unfolding *1*

by *auto*
next
case *False*
hence *check-dist v-code* ($\mathcal{L}$*-code v-code*) *eps* $\longleftrightarrow$ *dist v* (*MDP*.$\mathcal{L}_b$ *v*) $<$ (*eps* $*$ (*1* $-$ *l*)) / (*2* $*$ *l*)
using *1 invar-*$\mathcal{L}$*-code assms*(*6*) $\mathcal{L}$*-code-correct*$'$
by (*auto simp*: *check-dist-correct*)
hence $*$: *check-dist v-code* ($\mathcal{L}$*-code v-code*) *eps* $\longleftrightarrow$ *2* $*$ *l* $*$ *dist v* (*MDP*.$\mathcal{L}_b$ *v*) $<$ *eps* $*$ (*1* $-$ *l*)
using *zero-le-disc-locale False*
by (*auto simp*: *algebra-simps less-divide-eq*)
then show *?thesis*
proof (*cases check-dist v-code* ($\mathcal{L}$*-code v-code*) *eps*)
case *True*
hence *2* $*$ *l* $*$ *dist v* (*MDP*.$\mathcal{L}_b$ *v*) $<$ *eps* $*$ (*1* $-$ *l*)
using $*$ **by** *auto*
hence $*$: *MDP.mpi-algo eps d v m* = (*MDP.find-policy*$'$ *v*, *v*)
by (*simp add*: *MDP.mpi-algo.domintros MDP.mpi-algo.psimps*)
moreover have $**$:
(*mpi-partial-code eps d-code v-code m-code*) = (*fst* (*find-policy-L-code v-code*), *v-code*)
using *1.prems True False*
by (*simp add*: *mpi-partial-code.simps check-dist-def find-policy-L-code-split case-prod-beta*)
ultimately show *?thesis*
using *1.prems*
by (*simp add*: *find-policy-L-code-split vi-find-policy-correct*)
next
case *False*
hence *not-check*: $\neg$*2* $*$ *l* $*$ *dist v* (*MDP*.$\mathcal{L}_b$ *v*) $<$ *eps* $*$ (*1* $-$ *l*)
using $*$ **by** *auto*

have *d-in-A*: $\bigwedge$*s*. *s* $<$ *states* $\Longrightarrow$ *d-lookup*$'$ (*vi-find-policy-code v-code*) *s* $\in$ *MDP-A s*
unfolding *d-lookup*$'$*-def*
using *1.prems MDP.find-policy*$'$*-is-dec-det MDP.is-dec-det-def*

by (*auto simp*: *vi-find-policy-code-correct*$'$)
have *aux*: *V-Map.map-to-bfun* (*L-pow-code v-code* (*vi-find-policy-code v-code*) (*Suc m-code*)) =
V-Map.map-to-bfun (*L-pow-code* ($\mathcal{L}$*-code v-code*) (*vi-find-policy-code v-code*) *m-code*)
proof $-$
have $**$: *i* $<$ *states* $\Longrightarrow$ *v-lookup* (*L-code* (*vi-find-policy-code v-code*) *v-code*) *i* = *v-lookup* ($\mathcal{L}$*-code v-code*) *i* **for** *i*
using *d-in-A d-invar-vi-find-policy-code d-keys-vi-find-policy-code*
using *1.prems*(*7,8*) *MDP*.ν*-improving-imp-*$\mathcal{L}_b$[*OF MDP.find-policy*$'$*-improving*]
by (*auto simp*: *L-code-correct* $\mathcal{L}$*-code-correct vi-find-policy-correct*)

have ∗: *V-Map.map-to-bfun* (*L-pow-code v-code* (*vi-find-policy-code v-code*) (*Suc m-code*)) =
V-Map.map-to-bfun (*L-pow-code* (*L-code* (*vi-find-policy-code v-code*) *v-code*) (*vi-find-policy-code v-code*) *m-code*)
by (*simp add*: *L-pow-code-def funpow-swap1*)
show *?thesis*
unfolding ∗
by (*auto intro*!: *bfun-eqI L-pow-code-equiv simp*: *L-pow-code-correct*(*1*,*2*)
d-invar-vi-find-policy-code d-keys-vi-find-policy-code
L-code-keys L-code-invar invar-ℒ-code keys-ℒ-code
V-Map.map-to-bfun.rep-eq ∗∗ *1.prems*(*7*,*8*) *d-in-A*)
qed
have *MDP.mpi-algo eps d v m* = *MDP.mpi-algo eps* (*D-Map.map-to-fun d-code*) (*V-Map.map-to-bfun v-code*) (λ*a b*. *m-code*)
using *1* **by** *auto*
also have . . . =
MDP.mpi-algo eps (*MDP.find-policy′ v*) (*MDP.L-pow v* (*MDP.find-policy′ v*) (*Suc m-code*)) *m*
using *1 not-check* **by** (*auto simp*: *MDP.mpi-algo.psimps*)
also have . . . = *MDP.mpi-algo eps* (*D-Map.map-to-fun* (*vi-find-policy-code v-code*)) (*MDP.L-pow* (*V-Map.map-to-bfun v-code*) (*D-Map.map-to-fun* (*vi-find-policy-code v-code*)) (*Suc m-code*)) *m*
using *1* **by** (*auto simp*: *vi-find-policy-correct*[*symmetric*])
also have . . . = *MDP.mpi-algo eps* (*D-Map.map-to-fun* (*vi-find-policy-code v-code*)) (*V-Map.map-to-bfun* (*L-pow-code v-code* (*vi-find-policy-code v-code*) (*Suc m-code*))) *m*
using *1 L-pow-code-correct*(*3*) *d-in-A d-invar-vi-find-policy-code d-keys-vi-find-policy-code*
by *auto*
also have . . . = *MDP.mpi-algo eps* (*D-Map.map-to-fun* (*vi-find-policy-code v-code*)) (*V-Map.map-to-bfun* (*L-pow-code* (*ℒ-code v-code*) (*vi-find-policy-code v-code*) *m-code*)) *m*
using *aux* **by** *auto*
also have . . . = (*let* (*d′*, *v′*) = (*mpi-partial-code eps* (*vi-find-policy-code v-code*) (*L-pow-code* (*ℒ-code v-code*) (*vi-find-policy-code v-code*) *m-code*) *m-code*) *in*
(*D-Map.map-to-fun d′*, *V-Map.map-to-bfun v′*))
proof −
have
[*simp*]: *v-invar* (*L-pow-code* (*ℒ-code v-code*) (*vi-find-policy-code v-code*) *m-code*)
and [*simp*]: *v-len* (*L-pow-code* (*ℒ-code v-code*) (*vi-find-policy-code v-code*) *m-code*) = *states*
and *L-pow-code-eq*:
MDP.L-pow (*V-Map.map-to-bfun v-code*) (*MDP.find-policy′* (*V-Map.map-to-bfun v-code*)) (*Suc m-code*) = *V-Map.map-to-bfun* (*L-pow-code* (*ℒ-code v-code*) (*vi-find-policy-code v-code*) *m-code*)
using *d-in-A keys-ℒ-code invar-ℒ-code 1 d-keys-vi-find-policy-code d-invar-vi-find-policy-code L-pow-code-correct*

by (*auto simp*: *aux*[*symmetric*] *vi-find-policy-correct*)
show *?thesis*
unfolding *Let-def case-prod-beta*
using *MDP.find-policy′-is-dec-det not-check 1.prems*(*6*)
by (*subst 1*(*2*)[*symmetric*]) (*auto simp*: *1.prems L-pow-code-eq*[*symmetric*] *vi-find-policy-correct intro*!: *MDP.L-pow-*$\mathcal{L}_b$*-mono-inv′*)
qed
also have ... = *MDP.mpi-algo eps* (*MDP.find-policy′ v*) (*MDP.L-pow v* (*MDP.find-policy′ v*) (*Suc* (*m 0 v*))) (λ*a*. *m* (*Suc a*))
unfolding *Let-def case-prod-beta*
using ‹*l*≠ *0*› *not-check*
using *MDP.find-policy′-is-dec-det d-invar-vi-find-policy-code d-keys-vi-find-policy-code*
using *MDP.L-pow-*$\mathcal{L}_b$*-mono-inv′ vi-find-policy-correct*
using *1.prems L-pow-code-correct d-in-A invar-*$\mathcal{L}$*-code keys-*$\mathcal{L}$*-code*
by (*auto simp*: *1*(*2*)[*symmetric*] *aux*[*symmetric*])
also have ... = (*D-Map.map-to-fun* (*fst* (*mpi-partial-code eps d-code v-code m-code*)), *V-Map.map-to-bfun* (*snd* (*mpi-partial-code eps d-code v-code m-code*)))
proof −
have ∗: *MDP.L-pow* (*V-Map.map-to-bfun v-code*) (*MDP.find-policy′* (*V-Map.map-to-bfun v-code*)) (*Suc m-code*) =
V-Map.map-to-bfun (*L-pow-code* (*snd* (*find-policy-L-code v-code*)) (*fst* (*find-policy-L-code v-code*)) *m-code*)
using *d-keys-vi-find-policy-code d-invar-vi-find-policy-code d-in-A*
using *1.prems L-pow-code-correct aux invar-*$\mathcal{L}$*-code map-to-bfun-snd-find-policy-L-code vi-find-policy-correct*
by (*auto simp*: *find-policy-L-code-split*)
show *?thesis*
unfolding *mpi-partial-code.simps*[*of - - v-code*]
using *not-check False 1.prems*
using *d-in-A d-invar-vi-find-policy-code d-keys-vi-find-policy-code find-policy-L-code-split MDP.L-pow-*$\mathcal{L}_b$*-mono-inv′* ∗[*symmetric*]
using *MDP.find-policy′-is-dec-det*
by (*auto simp*: *case-prod-beta check-dist-def 1*(*2*)[*symmetric*] *L-pow-code-correct vi-find-policy-correct*)
qed
finally show *MDP.mpi-algo eps d v m* = (*D-Map.map-to-fun* (*fst* (*mpi-partial-code eps d-code v-code m-code*)), *V-Map.map-to-bfun* (*snd* (*mpi-partial-code eps d-code v-code m-code*)))
by *auto*
qed
qed
qed
thus *D-Map.map-to-fun* (*fst* (*mpi-partial-code eps d-code v-code m-code*)) = *fst* (*MDP.mpi-algo eps d v m*) *V-Map.map-to-bfun* (*snd* (*mpi-partial-code eps d-code v-code m-code*)) = *snd* (*MDP.mpi-algo*

eps d v m)
  using *assms*
  by (*auto simp*: *MDP.termination-mpi-algo*)
qed

lemma *d-map-to-fun-from-list′*: *D-Map.map-to-fun* (*D-Map.from-list′ f xs*) *a* = (*if* $a \in$ *set xs then f a else 0*)
 by (*simp add*: *d-lookup′-def map-to-fun-lookup map-to-fun-notin*)

definition *MPI-code eps m* =
 (*if eps* $\leq$ *0 then undefined else*
  let (*d*, *v*) = *mpi-partial-code eps d0-code v0-code m in d*)

lemma *d0-code-is-dec-det*: *MDP.is-dec-det* (*D-Map.map-to-fun d0-code*)
 unfolding *d0-code-def A-Map.keys-def MDP.is-dec-det-def MDP-A-def*
 using *MDP-A-outside ne-s-lookup A-Map.is-empty-def*
 by (*auto split*: *option.splits simp*: *d-map-to-fun-from-list′*)

lemma *Min-cong*: *finite X* $\Longrightarrow$ *X* $\neq$ {} $\Longrightarrow$ ($\bigwedge$*x*. *x* $\in$ *X* $\Longrightarrow$ *f x* = *g x*) $\Longrightarrow$ (*MIN x* $\in$ *X*. *f x*) = (*MIN x* $\in$ *X*. *g x*)
 by *force*

lemma *r-min-code-correct*:
 assumes *states* > *0*
 shows *r-min-code* = *MDP.r-min*
proof −
 have *bounded-r′*: *bounded* ((λ*a*. *MDP-r* (*x*, *a*)) ‘ *MDP-A x*) **for** *x*
  using *MDP.r-bounded′*
  unfolding *bounded-def*
  by *simp* (*meson UNIV-I*)
 have ∗: ($\bigsqcap$*a*∈*MDP-A x*. *MDP-r* (*x*, *a*)) $\leq$ *MDP-r* (*x*,*a*) **if** *a* $\in$ *MDP-A x* **for** *a x*
  using *bounded-r′ that*
  by (*auto intro*!: *cInf-lower bounded-imp-bdd-below*)
 have ∗∗∗∗: *MDP-r* (*s*,*a*) $\leq$ *MDP.r*$_M$ **for** *s a*
  using *abs-le-iff MDP.abs-r-le-r*$_M$ **by** *blast*
 have ∗∗: *bounded* (*range* (λ*s′*. $\bigsqcap$*a*∈*MDP-A s′*. *MDP-r* (*s′*, *a*)))
  using *MDP.abs-r-le-r*$_M$ *MDP.ex-dec-det MDP.is-dec-det-def MDP.A-ne*
   by (*auto simp add*: *minus-le-iff abs-le-iff intro*!: *cINF-greatest order.trans*[*OF* ∗] *boundedI*[*of - MDP.r*$_M$])
 have *MDP.r-min* $\leq$ *MDP-r* (*s*, *a*) **if** *a* $\in$ *MDP-A s* **for** *s a*
  using *MDP.r-bounded′ that* ∗∗
  by (*auto intro*!: *bounded-imp-bdd-below cInf-lower2*[*OF -* ∗])
 have *bdd*: *bdd-below* ((λ*x*. $\bigsqcap$*a*∈*MDP-A x*. *MDP-r* (*x*, *a*)) ‘ {*states*..})
  using ∗∗ *bounded-real* **by** (*auto intro*!: *bounded-imp-bdd-below*)
 have ($\bigsqcap$*x*. ($\bigsqcap$*a*∈*MDP-A x*. *MDP-r* (*x*, *a*))) = ($\bigsqcap$*x*∈{*0*..<*states*} ∪ {*states*..}. ($\bigsqcap$*a*∈*MDP-A x*. *MDP-r* (*x*, *a*)))
  by (*simp add*: *ivl-disj-un-one*(*8*))
 also have . . . = *min* ($\bigsqcap$*x*∈{*0*..<*states*}. ($\bigsqcap$*a*∈*MDP-A x*. *MDP-r*

$(x, a)))$ $(\bigsqcap x\in\{states..\}.$ $(\bigsqcap a\in MDP\text{-}A\ x.\ MDP\text{-}r\ (x, a)))$
    using *bdd*
    by (*auto simp add*: *image-Un cInf-union-distrib inf-min assms*)
  also have ... $=$ *min* $(\bigsqcap x\in\{0..<states\}.$ $(\bigsqcap a\in MDP\text{-}A\ x.\ MDP\text{-}r$
$(x, a)))$ $(\bigsqcap x\in\{states..\}.$ $(\bigsqcap a\in MDP\text{-}A\ x.\ 0))$
    using *MDP-r-zero-notin-states* **by** *auto*
  also have ... $=$ *min* $(\bigsqcap x\in\{0..<states\}.$ $(\bigsqcap a\in MDP\text{-}A\ x.\ MDP\text{-}r$
$(x, a)))$ *0*
    by *auto*
  also have ... $=$ *min* (*MIN* $x\in\{0..<states\}$. (*MIN* $a\in MDP\text{-}A$ *x*.
MDP-r (*x*, *a*))) *0*
    using *assms*
    by (*simp add*: *cInf-eq-Min*)
  also have ... $=$ *r-min-code*
    unfolding *r-min-code-def*
     using *assms A-Map.is-empty-def ne-s-lookup A-Map.entries-def*
entries-A-eq-r
   by (*auto simp*: *case-prod-beta MDP-A-def A-Map.keys-def min.commute*
image-image
        intro!: *Min-cong cong*[*of min 0*, *OF refl*])
  finally show *?thesis*..
qed

lemma *v0-code-correct*: $s < states \Longrightarrow$ *v-lookup v0-code s* $=$ (*MDP.v0-mpi*$_b$
s)
  unfolding *v0-code-def MDP.v0-mpi*$_b$*.rep-eq* *MDP.v0-mpi-def*
 by (*auto simp add*: *not-less MDP-r-zero-notin-states r-min-code-correct*)

lemma *v0-invar*: *v-invar v0-code*
  by (*simp add*: *v0-code-def*)

lemma *v0-keys*: *v-len v0-code* $=$ *states*
  by (*simp add*: *v0-code-def*)

lemma L_a*-indep-notin*:
  assumes $s < states$
  shows *MDP.*L_a *d* (*apply-bfun v*) *s* $=$ *MDP.*L_a *d* (*bfun-if* ($\lambda s.\ s <$
states) *v u*) *s*
proof −
  have *measure-pmf.expectation* (*MDP-K* (*s*, *d*)) *v* $=$
    measure-pmf.expectation (*MDP-K* (*s*, *d*)) ($\lambda s.$ *if* $s < states$ *then v*
s else u s)
    using *MDP-K-closed assms*
    by (*auto intro*!: *AE-pmfI integral-cong-AE simp*: *subset-eq*)
  thus *?thesis*
    by (*auto simp*: *bfun-if.rep-eq*)
qed

lemma $\mathcal{L}_b$*-indep-notin*: $s < states \Longrightarrow$ *MDP.*$\mathcal{L}_b$ *v s* $=$ *MDP.*$\mathcal{L}_b$ (*bfun-if*

($\lambda s.\ s < states$) v u) s
  unfolding *MDP*.$\mathcal{L}_b$-*eq-SUP-L*$_a{}'$
  using L_a-*indep-notin* **by** *presburger*

lemma
  v0-code-inc-$\mathcal{L}_b$:
  V-Map.map-to-bfun v0-code $\leq$ *MDP*.$\mathcal{L}_b$ (*V-Map.map-to-bfun v0-code*)
proof (*rule less-eq-bfunI*)
  fix x
  show (*V-Map.map-to-bfun v0-code*) $x \leq$ (*MDP*.$\mathcal{L}_b$ (*V-Map.map-to-bfun v0-code*)) x
  proof (*cases* $x < states$)
    case *True*
    have (*V-Map.map-to-bfun v0-code*) x = *MDP.v0-mpi*$_b$ x
      using *True v0-keys*
        by (*simp add*: *True V-Map.map-to-bfun.rep-eq v0-code-correct v0-invar*)
    also have ... $\leq$ *MDP*.$\mathcal{L}_b$ *MDP.v0-mpi*$_b$ x
      using *MDP.v0-mpi*$_b$-*le*-$\mathcal{L}_b$ **by** *blast*
    also have ... = *MDP*.$\mathcal{L}_b$ ((*bfun-if* ($\lambda s.\ s < states$) (*V-Map.map-to-bfun v0-code*)) (*MDP.v0-mpi*$_b$)) x
      using *v0-invar*
    by (*auto simp*: *apply-bfun-inverse bfun-if-def V-Map.map-to-bfun.rep-eq v0-code-correct MDP.L-def v0-keys MDP*.$\mathcal{L}$-*def cong*: *if-cong*)
    also have ... = *MDP*.$\mathcal{L}_b$ (*V-Map.map-to-bfun v0-code*) x
      using *True* $\mathcal{L}_b$-*indep-notin* **by** *presburger*
    finally show *?thesis*.
  next
    case *False*
    then show *?thesis*
    by (*simp add*: *MDP*.$\mathcal{L}_b$-*zero v0-code-def V-Map.map-to-bfun.rep-eq*)
  qed
qed

lemma
  fixes *eps m-code*
  defines *d-opt-code* $\equiv$ (*MPI-code eps m-code*)
  defines $m \equiv (\lambda(a{::}nat)\ (b{::}\ nat \Rightarrow_b real).\ m\text{-}code)$
  assumes $eps > 0$
  defines $v \equiv$ *V-Map.map-to-bfun v0-code*
  defines $d \equiv$ *D-Map.map-to-fun d0-code*
  defines $m \equiv (\lambda(a{::}nat)\ (b{::}\ nat \Rightarrow_b real).\ m\text{-}code)$
  shows
    D-Map.map-to-fun d-opt-code = *fst* (*MDP.mpi-algo eps d v m*)
  unfolding *d-def v-def m-def d-opt-code-def MPI-code-def*
  using *assms d0-code-is-dec-det v0-code-inc*-$\mathcal{L}_b$ *v0-invar MDP.termination-mpi-algo*
  by (*auto simp*: *v0-keys case-prod-beta intro*!: *mpi-partial-code-correct*(*1*))
end

global-interpretation *MPI-Code*: *MDP-Code*

IArray.sub λ*n x arr*. *IArray* ((*IArray.list-of arr*)[*n*:= *x*]) *IArray.length*
IArray IArray.list-of λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder
rbt

MDP.transitions (*Rep-Valid-MDP mdp*) *MDP.states* (*Rep-Valid-MDP*
mdp)

starray-get λ*i x arr*. *starray-set arr i x* *starray-length starray-of-list*
λ*arr*. *starray-foldr* (λ*x xs*. *x* # *xs*) *arr* [] λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder
rbt

MDP.disc (*Rep-Valid-MDP mdp*)
for *mdp*
defines *MPI-code* = *MPI-Code.MPI-code*
  and *a-lookup'* = *MPI-Code.a-lookup'*
  and *d-lookup'* = *MPI-Code.d-lookup'*

and *check-dist* = *MPI-Code.check-dist*

and *entries* = *M.entries*
and *from-list'* = *M.from-list'*

and *mpi-partial-code* = *MPI-Code.mpi-partial-code*
and L_a*-code* = *MPI-Code.*L_a*-code*
and *L-pow-code* = *MPI-Code.L-pow-code*
and *L-code* = *MPI-Code.L-code*

and *find-policy-state-code-aux'* = *MPI-Code.find-policy-state-code-aux'*
and *find-policy-state-code-aux* = *MPI-Code.find-policy-state-code-aux*
and *find-policy-L-code* = *MPI-Code.find-policy-L-code*

and *r-min-code* = *MPI-Code.r-min-code*
and *v0-code* = *MPI-Code.v0-code*
and *d0-code* = *MPI-Code.d0-code*
and *arr-tabulate* = *starray-Array.arr-tabulate*
  using *Rep-Valid-MDP*
  by *unfold-locales*
  (*auto simp*: *pmf-of-list-wf-def Ball-set-list-all*[*symmetric*] *case-prod-beta*
is-MDP-def
    RBT-Set.empty-def M.invar-def empty-def M.entries-def M.is-empty-def

length-0-conv[*symmetric*])

lemmas *entries-def*[*unfolded M.entries-def*, *code*]
lemmas *from-list′-def*[*unfolded M.from-list′-def*, *code*]
lemmas *arr-tabulate-def*[*unfolded starray-Array.arr-tabulate-def*, *code*]

end
theory *MPI-Code-Export-Float*
 imports
  MPI-Code
  Code-Real-Approx-By-Float-Fix
begin

export-code
 to-valid-MDP MDP MPI-code v0-code
 RBT-Map.update nat-map-from-list assoc-list-to-MDP RBT-Set.empty nat-pmf-of-list pmf-of-list
 nat-of-integer Ratreal int-of-integer inverse-divide Tree2.inorder integer-of-nat
 in *SML* **module-name** *MPI-Code-Float* **file-prefix** *MPI-Code-Float*

end
theory *MPI-Code-Export-Rat*
 imports
  MPI-Code
begin

export-code
 ord-real-inst.less-eq-real quotient-of
 plus-real-inst.plus-real minus-real-inst.minus-real to-valid-MDP MDP RBT-Map.update
 Rat.of-int divide divide-rat-inst.divide-rat divide-real-inst.divide-real nat-map-from-list
 assoc-list-to-MDP nat-pmf-of-list RBT-Set.empty MPI-code pmf-of-list nat-of-integer Ratreal int-of-integer
 inverse-divide Tree2.inorder integer-of-nat
 in *SML* **module-name** *MPI-Code-Rat* **file-prefix** *MPI-Code-Rat*
end
theory *Blinfun-To-Matrix*
 imports
  Jordan-Normal-Form.Matrix
  Perron-Frobenius.HMA-Connect
  MDP−Rewards.Blinfun-Util
begin
unbundle *no vec-syntax*
hide-const *Finite-Cartesian-Product.vec*
hide-type *Finite-Cartesian-Product.vec*

15.10.1 Gauss Seidel is a Regular Splitting

abbreviation *mat-inv m* ≡ *the* (*mat-inverse m*)

lemma *all-imp-Max*:
  assumes *finite X X* ≠ {} ∀ *x* ∈ *X*. *P* (*f x*)
  shows *P* (*MAX x* ∈ *X*. *f x*)
proof −
  have (*MAX x* ∈ *X*. *f x*) ∈ *f* ‘ *X*
    using *assms*
    by *auto*
  thus *?thesis*
    using *assms* **by** *force*
qed

lemma *vec-add*: *Matrix.vec n* (λ*i*. *f i* + *g i*) = *Matrix.vec n f* + *Matrix.vec n g*
  by *auto*

lemma *vec-scale*: *Matrix.vec n* (λ*i*. *r* ∗ *f i*) = *r* $\cdot_v$ (*Matrix.vec n f*)
  by *auto*

lift-definition *bfun-mat* :: *real mat* ⇒ (*nat* $\Rightarrow_b$ *real*) ⇒ (*nat* $\Rightarrow_b$ *real*)
is (λ*m v i*.
    if i < *dim-row m then* (*m* $*_v$ (*Matrix.vec* (*dim-col m*) (*apply-bfun v*))) $ *i else 0*)
proof
  fix *m* :: *real mat* **and** *v*
  have *norm*(*if i* < *dim-row m then* (*m* $*_v$ *Matrix.vec* (*dim-col m*) (*apply-bfun v*)) \$*v i else 0*) ≤
    (*if dim-row m* = *0 then 0 else* (*MAX i* ∈ {*0*..<*dim-row m*}. |(*m* $*_v$ *Matrix.vec* (*dim-col m*) (*apply-bfun v*)) \$*v i*|)) **for** *i*
    by (*force simp*: *Max-ge-iff*)
  thus *bounded* (*range* (λ*i*. *if i* < *dim-row m then* (*m* $*_v$ *Matrix.vec* (*dim-col m*) (*apply-bfun v*)) \$*v i else 0*))
    by (*blast intro*!: *boundedI*)
qed

definition *blinfun-to-mat m n* (*f* :: (*nat* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*nat* $\Rightarrow_b$ -)) =
  Matrix.mat m n (λ(*i*, *j*). *f* (*Bfun* (λ*k*. *if j* = *k then 1 else 0*)) *i*)

lemma *bounded-mult*:
  assumes *bounded* ((*f* :: ′*c* ⇒ *real*) ‘ *X*) *bounded* (*g* ‘ *X*)
  shows *bounded* ((λ*x*. *f x* ∗ *g x*) ‘ *X*)
proof −
  obtain *a b* :: *real* **where** ∀ *x*∈*X*. *norm* (*f x*) ≤ *a* ∀ *x*∈*X*. *norm* (*g x*) ≤ *b*
    using *assms* **by** (*auto simp*: *bounded-iff*)
  hence *norm* (*f x* ∗ *g x*) ≤ *a* ∗ *b* **if** *x* ∈ *X* **for** *x*

using *that* **by** (*auto simp*: *abs-mult intro*!: *mult-mono*)
thus *?thesis*
by (*fastforce intro*!: *boundedI*)
qed

lift-definition *mat-to-blinfun* :: *real mat* ⇒ (*nat* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*nat* $\Rightarrow_b$ *real*) **is** *bfun-mat*
proof
show *bfun-mat m* (*x* + *y*) = *bfun-mat m x* + *bfun-mat m y* **for** *m x y*
by (*auto simp*: *vec-add bfun-mat.rep-eq scalar-prod-add-distrib*[*of - dim-col m*])
show *bfun-mat m* ($x *_R y$) = $x *_R$ *bfun-mat m y* **for** *m x y*
by (*auto simp*: *vec-scale bfun-mat.rep-eq*)
have *aux*: *0* ≤ *Max* (*abs* ‘ *elements-mat* (*m*::*real mat*)) **if** *0* < *dim-row m 0* < *dim-col m* **for** *m*
using *that* **by** (*auto intro*: *all-imp-Max abs-le-norm-bfun simp*: *elements-mat-def*)

have *1*: |∑ *i* = *0*..<*dim-col* (*m*::*real mat*). *m* $$ (*n*, *i*) ∗ *apply-bfun x i*| ≤ (∑ *i* = *0*..<*dim-col m*. |*m* $$ (*n*, *i*) ∗ *apply-bfun x i*|) **for** *x m n*
by (*rule sum-abs*)
have *2*: (∑ *i* = *0*..<*dim-col m*. |(*m*::*real mat*) $$ (*n*, *i*) ∗ *apply-bfun x i*|) ≤ (∑ *i* = *0*..<*dim-col m*. *Max* (*abs* ‘ *elements-mat m*) ∗ |*apply-bfun x i*|) **if** *n* < *dim-row m* **for** *x m n*
unfolding *abs-mult elements-mat-def* **using** *that* **by** (*fastforce intro*!: *mult-right-mono sum-mono Max-ge*)
have *3*: (∑ *i* = *0*..<*dim-col m*. *Max* (*abs* ‘ *elements-mat m*) ∗ |*apply-bfun x i*|) ≤ (∑ *i* = *0*..<*dim-col m*. *Max* (*abs* ‘ *elements-mat m*) ∗ *norm x*) **if** *n* < *dim-row m* **for** *x m n*
using *that aux* **by** (*intro sum-mono*) (*auto intro*!: *mult-left-mono abs-le-norm-bfun*)

have *4*: (∑ *i* = *0*..<*dim-col* (*m*::*real mat*). *Max* (*abs* ‘ *elements-mat m*) ∗ *norm* (*x* :: (- $\Rightarrow_b$ -))) = *norm x* ∗ *dim-col m* ∗ *Max* (*abs* ‘ (*elements-mat m*)) **if** *n* < *dim-row m* **for** *n x m*
using *that* **by** *auto*

have |∑ *i* = *0*..<*dim-col* (*m*::*real mat*). *m* $$ (*n*, *i*) ∗ *apply-bfun x i*| ≤ *norm x* ∗ *dim-col m* ∗ *Max* (*abs* ‘ (*elements-mat m*)) **if** *n* < *dim-row m* **for** *x m n*
using *order.trans*[*OF order.trans*[*OF 1 2*[*OF that*]] *3*] *that* **unfolding** *4*[*OF that*] **by** *auto*
hence *norm* (*bfun-mat m x*) ≤ *norm x* ∗ (*if* (*dim-col m* = *0* ∨ *dim-row m* = *0*) *then 0 else dim-col m* ∗ *Max* (*abs* ‘ (*elements-mat m*))) **for** *m x*
using *aux*
by (*auto intro*!: *cSup-least bfun-eqI simp*: *norm-bfun-def′*[*of bfun-mat - -*] *bfun-mat.rep-eq scalar-prod-def mult.assoc*)

thus $\exists K. \forall x.$ *norm* (*bfun-mat m x*) $\leq$ *norm x* $*$ *K* **for** *m*
by *auto*
qed

lemma *mat-to-blinfun-mult*: *mat-to-blinfun m* (*v* :: *nat* $\Rightarrow_b$ *real*) *i* = *bfun-mat m v i*
by (*simp add*: *mat-to-blinfun.rep-eq*)

lemma *blinfun-to-mat-add-scale*: *blinfun-to-mat n m* (*v* + *b* $*_R$ *u*) = *blinfun-to-mat n m v* + *b* $\cdot_m$ (*blinfun-to-mat n m u*)
unfolding *blinfun-to-mat-def blinfun.add-left blinfun.scaleR-left*
by *auto*

lemma *mat-scale-one*[*simp*]: *1* $\cdot_m$ (*m*::*real mat*) = *m*
unfolding *smult-mat-def*
by (*auto simp*: *map-mat-def mat-eq-iff*)

lemma *blinfun-to-mat-add*: (*blinfun-to-mat n m* (*v* + *u*) :: *real mat*) = *blinfun-to-mat n m v* + (*blinfun-to-mat n m u*)
using *blinfun-to-mat-add-scale*[**where** *b* = *1*]
by *auto*

lemma *blinfun-to-mat-sub*: (*blinfun-to-mat n m* (*v* − *u*) :: *real mat*) = *blinfun-to-mat n m v* − *blinfun-to-mat n m u*
using *blinfun-to-mat-add-scale*[**where** *b* = −*1*]
by *auto*

lemma *blinfun-to-mat-zero*[*simp*]: *blinfun-to-mat n m 0* = 0_m *n m*
by (*auto simp*: *blinfun-to-mat-def*)

lemma *blinfun-to-mat-scale*: (*blinfun-to-mat n m* (*r* $*_R$ *v*) :: *real mat*) = *r* $\cdot_m$ (*blinfun-to-mat n m v*)
using *blinfun-to-mat-add-scale*[**where** *v* = *0*, **where** *b* = *r*]
by (*auto simp add*: *blinfun-to-mat-def*)

lemma *Bfun-if*[*simp*]: *apply-bfun* (*bfun.Bfun* (λ*k*. *if b k then a else c*)) = (λ*k*. *if b k then a else c*)
by (*auto intro*!: *Bfun-inverse*)

lemma *blinfun-to-mat-correct*: *blinfun-to-mat* (*dim-row v*) (*dim-col v*) (*mat-to-blinfun v*) = *v*
unfolding *blinfun-to-mat-def mat-to-blinfun.rep-eq bfun-mat.rep-eq*
by (*auto simp*: *mult-mat-vec-def Matrix.mat-eq-iff scalar-prod-def if-distrib cong*: *if-cong*)

lemma *blinfun-to-mat-id*: *blinfun-to-mat n n id-blinfun* = 1_m *n*
by (*auto simp*: *blinfun-to-mat-def*)

lemma *nonneg-mult-vec-mono*:
assumes 0_m (*dim-row X*) (*dim-col X*) $\leq$ *X v* $\leq$ *u dim-vec v* = *dim-col X*
shows $X *_v$ (*v* :: *real vec*) $\leq X *_v u$
using *assms*
unfolding *Matrix.less-eq-mat-def Matrix.less-eq-vec-def*
by (*auto simp*: *Matrix.scalar-prod-def intro*!: *sum-mono mult-left-mono*)

unbundle *no vec-syntax*

lemma *nonneg-blinfun-mat*: *nonneg-blinfun* (*mat-to-blinfun M*) $\longleftrightarrow$ (0_m (*dim-row M*) (*dim-col M*) $\leq$ *M*)
proof
assume *nonneg-blinfun* (*mat-to-blinfun M*)
hence $v \geq 0 \Longrightarrow 0 \leq$ *mat-to-blinfun M v* **for** *v* **unfolding** *nonneg-blinfun-def* **by** *auto*
hence *aux*: $v \geq 0 \Longrightarrow 0 \leq$ *bfun-mat M v* **for** *v* **unfolding** *mat-to-blinfun.rep-eq* **by** *auto*
hence *aux*: ($\bigwedge x$. *apply-bfun v x* $\geq$ *0*) $\Longrightarrow 0 \leq$ *bfun-mat M v x* **for** *v x* **unfolding** *less-eq-bfun-def* **by** *auto*

have $0 \leq$ *M* \$\$ (*i*, *j*) **if** *i* < *dim-row M* **and** *j* < *dim-col M* **for** *i j*
using *aux*[*of Bfun* (λk. *if k* = *j then 1 else 0*)] *that*
unfolding *bfun-mat.rep-eq*
by (*auto cong*: *if-cong simp*: *Matrix.mult-mat-vec-def scalar-prod-def if-distrib*)
thus 0_m (*dim-row M*) (*dim-col M*) $\leq$ *M*
unfolding *less-eq-mat-def* **by** *auto*
next
assume (0_m (*dim-row M*) (*dim-col M*) $\leq$ *M*)
hence $0 \leq$ *M* \$\$ (*i*,*j*) **if** *i* < *dim-row M* **and** *j* < *dim-col M* **for** *i j*
unfolding *less-eq-mat-def* **using** *that* **by** *auto*
thus *nonneg-blinfun* (*mat-to-blinfun M*)
unfolding *nonneg-blinfun-def mat-to-blinfun.rep-eq less-eq-bfun-def bfun-mat.rep-eq*
by (*auto simp*: *scalar-prod-def intro*!: *sum-nonneg*)
qed

lemma *mat-row-sub*: *X* $\in$ *carrier-mat n m* $\Longrightarrow$ *Y* $\in$ *carrier-mat n m* $\Longrightarrow$ *i* < *n* $\Longrightarrow$ *Matrix.row* (*X* − *Y*) *i* = *Matrix.row X i* − *Matrix.row Y i*
unfolding *Matrix.row-def* **by** *auto*

lemma *mat-to-blinfun-sub*: *X* $\in$ *carrier-mat n m* $\Longrightarrow$ *Y* $\in$ *carrier-mat n m* $\Longrightarrow$ *mat-to-blinfun* (*X* − *Y*) = *mat-to-blinfun X* − *mat-to-blinfun Y*
by (*auto intro*!: *blinfun-eqI simp*: *minus-scalar-prod-distrib*[*of - m*] *mat-row-sub mat-to-blinfun.rep-eq blinfun.diff-left bfun-mat.rep-eq*)

definition *inverse-mats* C D $\longleftrightarrow$ ($\exists$ n. C $\in$ *carrier-mat* n n $\wedge$ D $\in$ *carrier-mat* n n) $\wedge$ *inverts-mat* C D $\wedge$ *inverts-mat* D C

lemma *inverse-mats-sym*: *inverse-mats* C D $\Longrightarrow$ *inverse-mats* D C
  unfolding *inverse-mats-def* **by** *auto*

lemma *inverse-mats-unique*:
  assumes *inverse-mats* C D *inverse-mats* C E **shows** $D = E$
proof −
  have $D = D * (C * E)$
    using *assms* **unfolding** *inverse-mats-def inverts-mat-def* **by** *auto*
  also have $\ldots = (D * C) * E$
    using *assms* **unfolding** *inverse-mats-def* **by** *auto*
  finally show *?thesis*
    using *assms* **unfolding** *inverse-mats-def inverts-mat-def* **by** *auto*
qed

definition *inverse-mat* D = (*THE* E. *inverse-mats* D E)

lemma *invertible-mat-iff-inverse*: *invertible-mat* M $\longleftrightarrow$ ($\exists$ N. *inverse-mats* M N)
proof
  show *invertible-mat* M $\Longrightarrow$ $\exists$ N. *inverse-mats* M N
  unfolding *invertible-mat-def inverts-mat-def inverse-mats-def*
  by (*metis carrier-matI index-mult-mat*(*3*) *index-one-mat*(*3*) *square-mat.elims*(*2*))
  show $\exists$ N. *inverse-mats* M N $\Longrightarrow$ *invertible-mat* M
    unfolding *inverse-mats-def invertible-mat-def*
    by (*metis carrier-matD*(*1*) *carrier-matD*(*2*) *square-mat.elims*(*1*))
qed

lemma *mat-inverse-eq-inverse-mat*:
  assumes D $\in$ *carrier-mat* n n *invertible-mat* (D :: *real mat*)
  shows (*mat-inverse* D) = *Some* (*inverse-mat* D)
proof (*cases mat-inverse* D)
  case *None*
  have D $\notin$ *Units* (*ring-mat* *TYPE*(*real*) n n)
    by (*simp add*: *assms*(*1*) *None mat-inverse*)
  hence $x * D \neq 1_m$ n $\vee$ $D * x \neq 1_m$ n **if** x∈*carrier-mat* n n **for** x
    using *assms that* **by** (*simp add*: *Units-def ring-mat-simps*)
  hence ¬*invertible-mat* D
    unfolding *invertible-mat-iff-inverse*
   by (*metis assms*(*1*) *carrier-matD*(*1*) *inverse-mats-def inverts-mat-def*)
  then show *?thesis*
    using *assms* **by** *auto*
next
  case (*Some* a)
  hence *inverse-mats* D a
     using *assms*(*1*) *mat-inverse*(*2*) **unfolding** *inverse-mats-def inverts-mat-def* **by** *auto*

thus *?thesis*
unfolding *inverse-mat-def local.Some*
using *inverse-mats-unique*
by (*auto intro*: *HOL.the1-equality*[*symmetric*])
qed

lemma *invertible-inverse-mats*:
assumes *invertible-mat M*
shows *inverse-mats M* (*inverse-mat M*)
by (*metis assms inverse-mat-def inverse-mats-unique invertible-mat-iff-inverse theI-unique*)

definition *bfun-to-vec n v* = *Matrix.vec n* (*apply-bfun v*)

lemma *blinfun-to-mat-mult*:
(*blinfun-to-mat n m A*) $*_v$ (*bfun-to-vec m v*) = *bfun-to-vec n* (*A* (*bfun-if* ($\lambda i.$ $i < m$) *v 0*))
proof −
have ($\sum$ *ia* = *0..<m.* (*A* (*bfun.Bfun* ($\lambda k.$ *if ia* = *k then 1 else 0*))) *i* $*$ *v ia*) = (*A* (*bfun-if* ($\lambda k.$ $k < m$) *v 0*)) *i* **for** *i*
proof −
have ($\sum$ *ia* = *0..<m.* (*A* (*bfun.Bfun* ($\lambda k.$ *if ia* = *k then 1 else 0*))) *i* $*$ *v ia*) =
($\sum$ *ia* = *0..<m.* (*A* (*v ia* $*_R$ *bfun.Bfun* ($\lambda k.$ *if ia* = *k then 1 else 0*)) *i*))
by (*auto intro*!: *sum.cong simp add*: *blinfun.scaleR-right*)
also have ... = ($\sum$ *ia* = *0..<m.* (*A* (*v ia* $*_R$ *bfun.Bfun* ($\lambda k.$ *if ia* = *k then 1 else 0*)))) *i*
by (*induction m*) *auto*
also have ... = (*A* ($\sum$ *ia* = *0..<m.* (*v ia* $*_R$ *bfun.Bfun* ($\lambda k.$ *if ia* = *k then 1 else 0*)))) *i*
unfolding *blinfun.sum-right* **by** *auto*
also have ... = *blinfun-apply A* (*bfun-if* ($\lambda k.$ $k < m$) *v 0*) *i*
proof −
have ($\sum$ *ia* = *0..<m.* (*v ia* $*_R$ *bfun.Bfun* ($\lambda k.$ *if ia* = *k then 1 else 0*))) =
($\sum$ *i* = *0..<m.* *bfun-if* ($\lambda k.$ $k = i$) *v 0*)
by (*auto simp*: *bfun-if.rep-eq intro*!: *sum.cong*)
also have ... = *bfun-if* ($\lambda k.$ $k < m$) *v 0*
by (*induction m arbitrary*: *v*) (*auto simp add*: *bfun-eqI bfun-if.rep-eq*)
finally show *?thesis* **by** *auto*
qed
finally show *?thesis*.
qed
thus *?thesis*
unfolding *blinfun-to-mat-def bfun-to-vec-def mult-mat-vec-def scalar-prod-def*
by *auto*
qed

lemma *Max-geI*:
  assumes *finite X* (y::-:: *linorder*) $\in X$ $x \leq y$ **shows** $x \leq$ *Max X*
  using *assms Max-ge-iff* **by** *auto*

lift-definition *vec-to-bfun* :: *real vec* $\Rightarrow$ (*nat* $\Rightarrow_b$ *real*) **is**
  $\lambda v\ i.$ *if* $i <$ *dim-vec v then v* \$ *i else 0*
proof –
  fix *n* **and** *v* :: *real vec*
  show ($\lambda i.$ *if* $i < n$ *then v* \$ *i else 0*) $\in$ *bfun*
    by (*rule bfun-normI*[*of - if n* = *0 then 0 else Max* (*abs* ‘ ((\$) *v*) ‘
{..<*n*})]) (*auto intro*!: *Max-geI*)
qed

lemma *vec-to-bfun-to-vec*[*simp*]: *bfun-to-vec* (*dim-vec v*) (*vec-to-bfun v*) = *v*
  by (*auto simp*: *bfun-to-vec-def vec-to-bfun.rep-eq*)

lemma *bfun-to-vec-to-bfun*[*simp*]: *vec-to-bfun* (*bfun-to-vec m v*) = *bfun-if* ($\lambda i.$ $i < m$) *v 0*
  by (*auto simp*: *bfun-to-vec-def vec-to-bfun.rep-eq bfun-if.rep-eq*)

lemma *bfun-if-vec-to-bfun*[*simp*]: (*bfun-if* ($\lambda i.$ $i <$ *dim-vec v*) (*vec-to-bfun v*) *0*) = *vec-to-bfun v*
  by (*auto simp*: *bfun-if.rep-eq vec-to-bfun.rep-eq*)

lemma *blinfun-to-mat-mult*$'$:
  shows (*blinfun-to-mat n* (*dim-vec v*) *A*) $*_v$ *v* = *bfun-to-vec n* (*blinfun-apply A* (*vec-to-bfun v*))
  using *blinfun-to-mat-mult*[*of n dim-vec v A vec-to-bfun v*]
  by *auto*

lemma *blinfun-to-mat-mult*$''$:
  assumes m = *dim-vec v*
  shows (*blinfun-to-mat n m A*) $*_v$ *v* = *bfun-to-vec n* (*blinfun-apply A* (*vec-to-bfun v*))
  using *blinfun-to-mat-mult*$'$ *assms*
  by *auto*

lemma *matrix-eqI*:
  fixes *A* :: *real mat*
  assumes $\bigwedge v.$ $v \in$ *carrier-vec m* $\Longrightarrow$ $A *_v v = B *_v v$ $A \in$ *carrier-mat n m* $B \in$ *carrier-mat n m*
  shows $A = B$
proof –
  have $A *_v$ *Matrix.vec m* ($\lambda j.$ *if* $i = j$ *then 1 else 0*) = *Matrix.col A i* $B *_v$ *Matrix.vec m* ($\lambda j.$ *if* $i = j$ *then 1 else 0*) = *Matrix.col B i*
      if $i < m$ **for** *i*
    using *assms that*

by (*auto simp*: *mult-mat-vec-def scalar-prod-def if-distrib cong*: *if-cong*)
thus *?thesis*
using *assms*
by (*auto intro*: *Matrix.mat-col-eqI*)
qed

lemma *blinfun-to-mat-in-carrier*[*simp*]: *blinfun-to-mat m p A* $\in$ *carrier-mat m p*
unfolding *blinfun-to-mat-def*
by *auto*

lemma *blinfun-to-mat-dim-col*[*simp*]: *dim-col* (*blinfun-to-mat m p A*) = *p*
unfolding *blinfun-to-mat-def*
by *auto*

lemma *blinfun-to-mat-dim-row*[*simp*]: *dim-row* (*blinfun-to-mat m p A*) = *m*
unfolding *blinfun-to-mat-def*
by *auto*

lemma *bfun-to-vec-carrier*[*simp*]: *bfun-to-vec m v* $\in$ *carrier-vec m*
by (*simp add*: *bfun-to-vec-def*)

lemma *vec-cong*: ($\bigwedge i.\ i < n \Longrightarrow f\ i = g\ i$) $\Longrightarrow$ *vec n f* = *vec n g*
by *auto*

lemma *mat-to-blinfun-compose*:
assumes *dim-col A* = *dim-row B*
shows (*mat-to-blinfun A* o_L *mat-to-blinfun B*) = *mat-to-blinfun* (*A* $*$ *B*)
proof (*intro blinfun-eqI bfun-eqI*)
fix *i x*
show *apply-bfun* (*blinfun-apply* (*mat-to-blinfun A* o_L *mat-to-blinfun B*) *i*) *x* = *apply-bfun* (*blinfun-apply* (*mat-to-blinfun* (*A* $*$ *B*)) *i*) *x*
proof (*cases x* < *dim-row A*)
case *True*
have (*mat-to-blinfun A* o_L *mat-to-blinfun B*) *i x* = (*bfun-mat A* (*bfun-mat B i*)) *x*
by (*auto simp*: *mat-to-blinfun.rep-eq*)
also have ... = *Matrix.row A x* $\cdot$ *vec* (*dim-col A*) (λ*ia*. *Matrix.row B ia* $\cdot$ *vec* (*dim-col B*) *i*)
using *assms* **by** (*auto simp*: *bfun-mat.rep-eq True cong*: *vec-cong*)
also have ... = *vec* (*dim-col B*) (λ*j*. *Matrix.row A x* $\cdot$ *col B j*) $\cdot$ *vec* (*dim-col B*) *i*
using *assms* **by** (*auto simp add*: *carrier-vecI assoc-scalar-prod*[*of - dim-col A*])
also have ... = *Matrix.row* (*A* $*$ *B*) *x* $\cdot$ *vec* (*dim-col B*) (*apply-bfun*

i)
  by (*subst Matrix.row-mult*) (*auto simp add*: *assms True*)
 also have . . . = *mat-to-blinfun* ($A * B$) *i x*
  by (*simp add*: *mat-to-blinfun.rep-eq bfun-mat.rep-eq True*)
 finally show *?thesis*.
qed (*simp add*: *bfun-mat.rep-eq mat-to-blinfun-mult*)
qed

lemma *blinfun-to-mat-compose*:
 fixes A B :: (*nat* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*nat* $\Rightarrow_b$ *real*)
 assumes
  $\bigwedge v\ v'\ j.$ ($\bigwedge i.\ i < m \Longrightarrow$ *apply-bfun* v i = *apply-bfun* v' i) $\Longrightarrow j <$ $n \Longrightarrow A\ v\ j = A\ v'\ j$
 shows *blinfun-to-mat n m A* $*$ *blinfun-to-mat m p B* = *blinfun-to-mat* *n p* (A o_L B)
proof (*rule matrix-eqI*[*of p - - n*])
 fix v :: *real vec*
 assume *v*[*simp*, *intro*]: $v \in$ *carrier-vec p*
 hence *blinfun-to-mat n m A* $*$ *blinfun-to-mat m p B* $*_v$ v = *bfun-to-vec* *n* (A (*vec-to-bfun* (*blinfun-to-mat m p B* $*_v$ v)))
  by (*auto simp*: *blinfun-to-mat-mult''* *blinfun-to-mat-mult assoc-mult-mat-vec*[*of* - *n m* - *p*])
 also have . . . = *bfun-to-vec n* (A (*vec-to-bfun* (*bfun-to-vec m* (B (*vec-to-bfun v*)))))
  using *v* **by** (*fastforce simp*: *blinfun-to-mat-mult''* *dest*!: *carrier-vecD*)+
 also have . . . = *bfun-to-vec n* ((A o_L B) (*vec-to-bfun v*))
  unfolding *bfun-to-vec-to-bfun*
  using *assms* **by** (*fastforce simp add*: *bfun-if.rep-eq bfun-to-vec-def* *intro*!: *vec-cong*)
 also have . . . = *blinfun-to-mat n p* (A o_L B) $*_v$ v
  using *v* **by** (*fastforce simp*: *blinfun-to-mat-mult''* *dest*!: *carrier-vecD*)+
 finally show *blinfun-to-mat n m A* $*$ *blinfun-to-mat m p B* $*_v$ v = *blinfun-to-mat n p* (A o_L B) $*_v$ v.
qed *auto*

lemma *invertible-mat-dims*: *invertible-mat A* $\Longrightarrow$ *dim-col A* = *dim-row* *A*
 by (*simp add*: *invertible-mat-def*)

lemma *invertible-mat-square*: *invertible-mat A* $\Longrightarrow$ *square-mat A*
 by (*simp add*: *invertible-mat-dims*)

lemma *inverse-mat-dims*:
 assumes *invertible-mat A*
 shows *dim-col* (*inverse-mat A*) = *dim-col A* *dim-row* (*inverse-mat* *A*) = *dim-row A*
 using *assms inverse-mats-def invertible-inverse-mats* **by** *auto*

lemma *inverse-mat-mult*[*simp*]:

assumes *invertible-mat* A
shows *inverse-mat* $A * A = 1_m$ (*dim-row* A) $A *$ *inverse-mat* $A = 1_m$ (*dim-row* A)
using *assms invertible-inverse-mats*[*OF assms*] *inverse-mat-dims*
unfolding *inverts-mat-def inverse-mats-def*
by *auto*

lemma *invertible-mult*:
assumes *invertible-mat* m *dim-vec* $a =$ *dim-col* m *dim-vec* $b =$ *dim-col* m
shows $a = b \longleftrightarrow m *_v a = m *_v b$
proof −
have (*inverse-mat* $m * m$) $*_v a = a$ (*inverse-mat* $m * m$) $*_v b = b$
by (*metis assms carrier-vec-dim-vec inverse-mat-mult*(*1*) *invertible-mat-dims one-mult-mat-vec*)+
thus *?thesis*
by (*metis assms assoc-mult-mat-vec carrier-mat-triv carrier-vec-dim-vec inverse-mat-dims*(*1*) *invertible-mat-dims*)
qed

lemma *inverse-mult-iff*:
assumes *invertible-mat* m
and *dim-vec* $v =$ *dim-col* m *dim-vec* $b =$ *dim-row* m
shows $v =$ *inverse-mat* $m *_v b \longleftrightarrow m *_v v = b$
proof −
have $v =$ *inverse-mat* $m *_v b \longleftrightarrow m *_v v = m *_v$ (*inverse-mat* m $*_v b$)
using *invertible-mult* **by** (*metis assms dim-mult-mat-vec inverse-mat-dims*(*2*) *invertible-mat-dims*)
also have ... $\longleftrightarrow m *_v v = (m *$ *inverse-mat* $m) *_v b$
by (*subst assoc-mult-mat-vec*[*of - dim-col m dim-col m - dim-col m*])
(*auto simp add*: *assms carrier-vecI inverse-mat-dims invertible-mat-dims*)
also have ... $\longleftrightarrow m *_v v = b$
by (*metis assms*(*1*) *assms*(*3*) *carrier-vecI inverse-mat-mult*(*2*) *one-mult-mat-vec*)
finally show *?thesis*.
qed

lemma *inverse-blinfun-to-mat*:
fixes A :: (*nat* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*nat* $\Rightarrow_b$ *real*)
assumes *invertible*$_L$ A
assumes ($\bigwedge v\ v'\ j.$ ($\bigwedge i.\ i < m \Longrightarrow$ *apply-bfun* $v\ i =$ *apply-bfun* $v'\ i$) $\Longrightarrow j < m \Longrightarrow (A\ v)\ j = (A\ v')\ j$)
assumes ($\bigwedge v\ v'\ j.$ ($\bigwedge i.\ i < m \Longrightarrow$ *apply-bfun* $v\ i =$ *apply-bfun* $v'\ i$) $\Longrightarrow j < m \Longrightarrow (inv_L\ A\ v)\ j = (inv_L\ A\ v')\ j$)
shows *blinfun-to-mat* m m (inv_L A) = (*inverse-mat* (*blinfun-to-mat* m m A)) *invertible-mat* (*blinfun-to-mat* m m A)

proof −
 have $*$: *blinfun-to-mat m m A* $*$ *blinfun-to-mat m m* (*inv*$_L$ *A*) = 1_m *m*
  by (*subst blinfun-to-mat-compose*) (*fastforce simp*: *blinfun-to-mat-id assms*(*1*) *intro*: *assms*(*2*))+
 moreover have $**$: *blinfun-to-mat m m* (*inv*$_L$ *A*) $*$ *blinfun-to-mat m m A* = 1_m *m*
  by (*subst blinfun-to-mat-compose*) (*fastforce simp*: *blinfun-to-mat-id assms*(*1*) *intro*: *assms*(*3*))+
 ultimately have $***$: *invertible-mat* (*blinfun-to-mat m m A*)
  by (*metis blinfun-to-mat-dim-col blinfun-to-mat-dim-row invertible-mat-def inverts-mat-def square-mat.elims*(*1*))

 have *inverse-mats* (*blinfun-to-mat m m A*) (*blinfun-to-mat m m* (*inv*$_L$ *A*))
  unfolding *inverse-mats-def inverts-mat-def* **using** $*$ $**$ **by** *force*
 hence *inverse-mat* (*blinfun-to-mat m m A*) = *blinfun-to-mat m m* (*inv*$_L$ *A*)
  using $***$ *inverse-mats-unique invertible-inverse-mats* **by** *blast*
 thus *blinfun-to-mat m m* (*inv*$_L$ *A*) = *inverse-mat* (*blinfun-to-mat m m A*) *invertible-mat* (*blinfun-to-mat m m A*)
  using ‹*invertible-mat* (*blinfun-to-mat m m A*)› **by** *auto*
qed

end
theory *Policy-Iteration-Fin*
 imports
  Policy-Iteration
  MDP-fin
  Blinfun-To-Matrix
begin

context *MDP-nat-disc* **begin**

lemma *finite-*D_D[*simp*]: *finite* D_D
proof −
 let *?set* = $\{d.\ \forall x :: nat.\ (x \in \{0..<states\} \longrightarrow d\ x \in (\bigcup s\in\{0..<states\}.\ A\ s)) \land (x \notin \{0..<states\} \longrightarrow d\ x = 0)\}$
 have *finite* $(\bigcup s<states.\ A\ s)$
  using *A-fin* **by** *auto*
 hence *finite ?set*
  by (*intro finite-set-of-finite-funs*) *auto*
 moreover have $D_D \subseteq$ *?set*
  unfolding *is-dec-det-def*
  using *A-outside*
  by (*auto simp*: *not-less*)
 ultimately show *?thesis*
  using *finite-subset*
  by *auto*

qed

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

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

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 ¬ (d = x ∨ ¬ *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

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*)

lemma ν_b-*zero-notin*: $s \geq$ *states* $\Longrightarrow \nu_b$ p s = *0*
  using ν-*zero-notin* **unfolding** ν_b.*rep-eq* **by** *auto*

lemma *r-dec*$_b$-*zero-notin*: $s \geq$ *states* $\Longrightarrow$ *r-dec*$_b$ d s = *0*
  using *reward-zero-outside*
  by *auto*

lemma ν_b-*eq-inv*: ν_b (*mk-stationary* d) = *inv*$_L$ (*id-blinfun* − $l *_R \mathcal{P}_1$ d) (*r-dec*$_b$ d)
  using ν-*stationary-inv*.

lemma ν_b-*eq-bfun-if*: ν_b (*mk-stationary* d) = *bfun-if* ($\lambda i.\ i <$ *states*) (ν_b (*mk-stationary* d)) *0*
  using ν_b-*zero-notin* **by** (*auto simp*: *bfun-if.rep-eq*)

lemma ν_b*-vec-aux*: ((*1*$_m$ *states*) − *l* $\cdot_m$ (*blinfun-to-mat states states* ($\mathcal{P}_1$ *d*))) $*_v$ *bfun-to-vec states* (ν_b (*mk-stationary d*)) = *bfun-to-vec states* (*r-dec*$_b$ *d*)
proof −
  let *?to-mat* = *blinfun-to-mat states states*
  let *?to-vec* = *bfun-to-vec states*

 have ((*1*$_m$ *states*) − *l* $\cdot_m$ (*?to-mat* ($\mathcal{P}_1$ *d*))) $*_v$ *?to-vec* (ν_b (*mk-stationary d*)) =
    ((*1*$_m$ *states*) − *?to-mat* (*l* $*_R$ ($\mathcal{P}_1$ *d*))) $*_v$ *?to-vec* (ν_b (*mk-stationary d*))
    using *blinfun-to-mat-scale* **by** *fastforce*
  also have ... = (*?to-mat id-blinfun* − *?to-mat* (*l* $*_R$ ($\mathcal{P}_1$ *d*))) $*_v$ *?to-vec* (ν_b (*mk-stationary d*))
    using *blinfun-to-mat-id* **by** *presburger*
  also have ... = *?to-mat* (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ *d*) $*_v$ *?to-vec* (ν_b (*mk-stationary d*))
    using *blinfun-to-mat-sub* **by** *presburger*
 also have ... = *?to-vec* ((*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ *d*) ((ν_b (*mk-stationary d*))))
    unfolding *blinfun-to-mat-mult* **using** ν_b*-eq-bfun-if* **by** *auto*
  also have ... = *?to-vec* (*r-dec*$_b$ *d*)
  by (*metis L-*ν*-fix-iff L-def blinfun.diff-left blinfun.scaleR-left diff-eq-eq id-blinfun.rep-eq*)
  finally show *?thesis*.
qed

lemma *summable-geom-*$\mathcal{P}_1$: *summable* (λk. (($l *_R \mathcal{P}_1$ *d*)^^*k*))
  using *summable-inv-Q norm-*$\mathcal{P}_1$
  by (*metis add-diff-cancel-left′ diff-add-cancel norm-*$\mathcal{P}_1$*-l-less*)

lemma *summable-geom-*$\mathcal{P}_1$*′*: *summable* (λk. (($l *_R \mathcal{P}_1$ *d*)^^*k*) *v*) **for** *v*
  using *summable-geom-*$\mathcal{P}_1$ *tendsto-blinfun-apply*
  unfolding *summable-def sums-def*
  by (*fastforce simp*: *blinfun.sum-left*)

lemma *summable-geom-*$\mathcal{P}_1$*″*: *summable* (λk. (($l *_R \mathcal{P}_1$ *d*)^^*k*) *v s*) **for** *v s*
  using *summable-geom-*$\mathcal{P}_1$*′ bfun-tendsto-apply-bfun*
    unfolding *summable-def sums-def*
    by (*fastforce simp*: *sum-apply-bfun*)

lemma *K-closed′*: *s*<*states* $\Longrightarrow$ *j* $\in$ *set-pmf* (*K* (*s*, *a*)) $\Longrightarrow$ *j* < *states*
  by (*meson K-closed atLeastLessThan-iff basic-trans-rules*(*31*))

lemma $\mathcal{P}_1$*-indep*:
  assumes ($\bigwedge$*i*. *i* < *states* $\Longrightarrow$ *apply-bfun v i* = *apply-bfun v′ i*) *j* < *states*

shows $(l *_R \mathcal{P}_1\ d)\ v\ j = (l *_R \mathcal{P}_1\ d)\ v'\ j$
using *assms K-closed*$'$[*OF assms*(*2*)]
by (*auto simp*: *blinfun.scaleR-left* $\mathcal{P}_1$*.rep-eq K-st-def intro*!: *integral-cong-AE AE-pmfI*)

lemma *inv*$_L$*-indep*:
assumes $\bigwedge i.\ i < states \Longrightarrow apply\text{-}bfun\ v\ i = apply\text{-}bfun\ v'\ i\ j < states$
shows $((inv_L\ (id\text{-}blinfun - l *_R \mathcal{P}_1\ d))\ v)\ j = ((inv_L\ (id\text{-}blinfun - l *_R \mathcal{P}_1\ d))\ v')\ j$
proof −
have $((l *_R \mathcal{P}_1\ d)$ ^^ $n)\ v\ j = ((l *_R \mathcal{P}_1\ d)$ ^^ $n)\ v'\ j$ **for** n
using *assms* $\mathcal{P}_1$*-indep* **by** (*induction n arbitrary*: *j v v*$'$) *fastforce*+
thus *?thesis*
using *summable-geom-*$\mathcal{P}_1$ *summable-geom-*$\mathcal{P}_1{}'$
by (*auto simp*: *inv*$_L$*-inf-sum blinfun-apply-suminf*[*symmetric*] *suminf-apply-bfun*)
qed

lemma *vec-*ν_b: *bfun-to-vec states* (ν_b (*mk-stationary d*)) =
inverse-mat ((*1*$_m$ *states*) − $l\ \cdot_m$ (*blinfun-to-mat states states* ($\mathcal{P}_1$ *d*))) $*_v$ (*bfun-to-vec states* (*r-dec*$_b$ *d*))
proof −
have *bfun-to-vec states* (ν_b (*mk-stationary d*)) = *bfun-to-vec states* (*inv*$_L$ (*id-blinfun* − $l *_R \mathcal{P}_1$ *d*) (*r-dec*$_b$ *d*))
using ν_b*-eq-inv* **by** *force*
also have ... = *bfun-to-vec states* (*inv*$_L$ (*id-blinfun* − $l *_R \mathcal{P}_1$ *d*) (*bfun-if* ($\lambda i.\ i < states$) (*r-dec*$_b$ *d*) *0*))
using *r-dec*$_b$*-zero-notin*
by (*subst bfun-if-eq*) *auto*
also have ... = *blinfun-to-mat states states* (*inv*$_L$ (*id-blinfun* − $l *_R \mathcal{P}_1$ *d*)) $*_v$ (*bfun-to-vec states* (*r-dec*$_b$ *d*))
using *blinfun-to-mat-mult*..
also have ... = *inverse-mat* (*blinfun-to-mat states states* (*id-blinfun* − $l *_R \mathcal{P}_1$ *d*)) $*_v$ (*bfun-to-vec states* (*r-dec*$_b$ *d*))
using *inv*$_L$*-indep* $\mathcal{P}_1$*-indep*
by (*fastforce simp add*: *inverse-blinfun-to-mat invertible*$_L$*-inf-sum blinfun.diff-left*)+
finally show *?thesis*
using *blinfun-to-mat-id blinfun-to-mat-scale blinfun-to-mat-sub* **by** *presburger*
qed

lemma *invertible-*ν_b*-mat*: *invertible-mat* ((*1*$_m$ *states*) − $l\ \cdot_m$ (*blinfun-to-mat states states* ($\mathcal{P}_1$ *d*)))
proof −
have *invertible-mat* (*blinfun-to-mat states states* ((*id-blinfun* − $l *_R \mathcal{P}_1$ *d*)))

using $\mathcal{P}_1$*-indep* inv_L*-indep*
by (*fastforce simp*: $invertible_L$*-inf-sum blinfun.diff-left intro*!: *inverse-blinfun-to-mat*(*2*))+
thus *?thesis*
by (*auto simp*: *blinfun-to-mat-id blinfun-to-mat-sub blinfun-to-mat-scale*)
qed

lemma *mat-cong*:
assumes ($\bigwedge i\ j.\ i < n \Longrightarrow j < m \Longrightarrow f\ i\ j = g\ i\ j$)
shows *Matrix.mat n m* (λ(*i*, *j*). *f i j*) = *Matrix.mat n m* (λ(*i*,*j*). *g i j*)
using *assms* **by** *auto*

lemma $\mathcal{P}_1$*-mat*: (*Matrix.mat states states* (λ(*s*, *s'*). *pmf* (*K* (*s*, *d s*)) *s'*)) = *blinfun-to-mat states states* ($\mathcal{P}_1$ (*mk-dec-det d*))
proof −
have *pmf* (*K* (*s*, *d s*)) *s'* = *measure-pmf.expectation* (*K* (*s*, *d s*)) (λ*k*. *if s'* = *k then 1 else 0*)
if *s* < *states s'* < *states* **for** *s s'*
by (*auto simp*: *integral-measure-pmf-real*[*of* {*s'*}] *split*: *if-splits*)
thus *?thesis*
by (*auto simp*: *blinfun-to-mat-def* $\mathcal{P}_1$*.rep-eq K-st-def mk-dec-det-def bind-return-pmf*)
qed

lemma *vec-*ν_b*'*: *bfun-to-vec states* (ν_b (*mk-stationary-det d*)) =
inverse-mat ((*1*$_m$ *states*) − *l* $\cdot_m$ (*Matrix.mat states states* (λ(*s*, *s'*). *pmf* (*K* (*s*, *d s*)) *s'*))) $*_v$
(*vec states* (λ*i*. *r* (*i*, *d i*)))
unfolding *vec-*ν_b **using** $\mathcal{P}_1$*-mat* **by** (*auto simp*: *bfun-to-vec-def*)

lemma *vec-*ν_b*''*:
assumes *s* < *states*
shows (ν_b (*mk-stationary-det d*)) *s* =
(*inverse-mat* ((*1*$_m$ *states*) − *l* $\cdot_m$ (*Matrix.mat states states* (λ(*s*, *s'*). *pmf* (*K* (*s*, *d s*)) *s'*))) $*_v$
(*vec states* (λ*i*. *r* (*i*, *d i*)))) \$ *s*
using *vec-*ν_b*'* *assms* **unfolding** *bfun-to-vec-def* **by** (*metis index-vec*)

lemma *invertible-*ν_b*-mat'*:
invertible-mat (*1*$_m$ *states* − *l* $\cdot_m$ *Matrix.mat states states* (λ(*s*, *y*). *pmf* (*K* (*s*, *d s*)) *y*))
using *invertible-*ν_b*-mat* $\mathcal{P}_1$*-mat* **by** *presburger*

lemma *dim-vec-*ν_b: *dim-vec* (*inverse-mat* ((*1*$_m$ *states*) −
l $\cdot_m$ (*Matrix.mat states states* (λ(*s*, *s'*). *pmf* (*K* (*s*, *d s*)) *s'*))) $*_v$
(*vec states* (λ*i*. *r* (*i*, *d i*)))) = *states*
by (*simp add*: *inverse-mat-dims*(*2*) *invertible-*ν_b*-mat'*)

end
end
theory *PI-Code*
  imports
    ../Policy-Iteration-Fin
    HOL−Library.Code-Target-Numeral
    Jordan-Normal-Form.Matrix-Impl
    Code-Setup
begin
context *MDP-Code* **begin**

definition *policy-eval-code d* =
  inverse-mat (1_m *states* −
  $l \cdot_m$ (*Matrix.mat states states* (λ(*s*, *s*′). *pmf* (*MDP-K* (*s*, *d-lookup*′ *d s*)) *s*′))) $*_v$
  (*vec states* (λ*i*. *MDP-r* (*i*, *d-lookup*′ *d i*)))

lemma *d-lookup*′*-eq-map-to-fun*: *D-Map.invar d* ⟹ *s* ∈ *D-Map.keys d* ⟹ *d-lookup*′ *d s* = *D-Map.map-to-fun d s*
  using *D-Map.lookup-None-set-inorder*
  by (*auto simp*: *D-Map.map-to-fun-def d-lookup*′*-def split*: *option.splits*)

lemma *policy-eval-correct*:
  assumes *D-Map.keys d* = {*0..<states*} *D-Map.invar d s* < *states*
  shows *policy-eval-code d* $*v s* = *MDP*.$\nu_b$ (*MDP.mk-stationary-det* (*D-Map.map-to-fun d*)) *s*
  unfolding *policy-eval-code-def MDP.vec*-ν_b′′[*OF assms*(*3*)]
  using *assms d-lookup*′*-eq-map-to-fun*
  by (*auto cong*: *vec-cong MDP.mat-cong*)

definition *transition-vecs* =
  Matrix.vec states (λ*s*. *M.from-list* (*map* (λ(*a*, -, *ps*). (*a*,
    Matrix.vec states (λ*s*′. ∑*x* ← *ps*. *if fst x* = *s*′ *then snd x else 0*)))
(*a-inorder* (*s-lookup mdp s*))))

lemma *transition-vecs-correct*:
  assumes *s* < *states a* ∈ *MDP-A s s*′ < *states*
  shows *M.lookup*′ (*transition-vecs* $*v s*) *a* $*v s*′ = *pmf* (*MDP-K* (*s*,*a*)) *s*′
proof −
  have ∗: *Matrix.vec states* (λ*s*′. ∑*x*←*snd* (*a-lookup*′ (*s-lookup mdp s*) *a*). *if fst x* = *s*′ *then snd x else 0*) $*v s*′ = *pmf* (*pmf-of-list* (*snd* (*a-lookup*′ (*s-lookup mdp s*) *a*))) *s*′
    by (*auto simp*: *pmf-pmf-of-list assms pmf-of-list-wf-mdp sum-list-map-filter*′)
  have ∗∗:
  M.lookup′ (*M.from-list* (*map* (λ(*a*, -, *ps*). (*a*, *Matrix.vec states* (λ*s*′. ∑*x*←*ps*. *if fst x* = *s*′ *then snd x else 0*))) (*a-inorder* (*s-lookup mdp s*)))) *a* $*v s*′ =
  pmf (*pmf-of-list* (*snd* (*a-lookup*′ (*s-lookup mdp s*) *a*))) *s*′

unfolding $*$[*symmetric*]
using *a-map-entries-lookup*[*OF assms*(*1*,*2*)] *A-Map.distinct-inorder invar-s-lookup*[*OF assms*(*1*)]
by (*subst M.lookup′-from-list-distinct*) (*force simp*: *comp-def case-prod-beta A-Map.entries-def*[*symmetric*] *intro*!: *imageI*)+
show *?thesis*
unfolding *transition-vecs-def MDP-K-def*
using *assms a-lookup-None-notin-A sa-lookup-eq*(*2*) *snd-sa-lookup′-eq*

by (*auto split*: *option.splits simp*: $**$)
qed

lemma *policy-eval-code*: *policy-eval-code d* =
the (*mat-inverse* ((*1*$_m$ *states*) −
l $\cdot_m$ (*Matrix.mat states states* (λ(*s*, *s′*). *pmf* (*MDP-K* (*s*, *d-lookup′ d s*)) *s′*)))) $*_v$
(*vec states* (λ*i*. *MDP-r* (*i*, *d-lookup′ d i*)))
unfolding *policy-eval-code-def*
by (*subst mat-inverse-eq-inverse-mat*) (*auto simp*: *MDP.invertible-*ν_b*-mat′*)

definition *one-st* = *1*$_m$ *states*
definition *k-mat d* = *Matrix.mat states states* (λ(*s*, *y*). *pmf* (*MDP-K* (*s*, *d-lookup′ d s*)) *y*)

definition *k-mat′ d m* = (
Matrix.mat-of-row-fun states states (λ*i*. *M.lookup′* (*m* \$*v i*) (*d-lookup′ d i*)))

lemma *invertible-imp-inv-ex*: *invertible-mat m* ⟹ ∃ *x*∈*carrier-mat* (*dim-row m*) (*dim-row m*). *x* $*$ *m* = *1*$_m$ (*dim-row m*) ∧ *m* $*$ *x* = *1*$_m$ (*dim-row m*)
by (*metis carrier-matD*(*1*) *inverse-mat-mult inverse-mats-def invertible-inverse-mats*)

lemma *policy-eval-code′*:
fixes *d*
defines *m* ≡ (*one-st* − *l* $\cdot_m$ *Matrix.mat states states* (λ(*s*, *y*). *pmf* (*MDP-K* (*s*, *d-lookup′ d s*)) *y*))
shows *policy-eval-code d* = *snd* (*gauss-jordan m* (*1*$_m$ *states*)) $*_v$ (*vec states* (λ*i*. *MDP-r* (*i*, *d-lookup′ d i*)))
proof −
have *m*: *m* ∈ *carrier-mat states states*
using *assms* **by** *fastforce*
hence *fst* (*gauss-jordan m* (*1*$_m$ *states*)) = *1*$_m$ *states*
using *MDP.invertible-*ν_b*-mat′*[*of d-lookup′ d*, *unfolded m-def*[*symmetric*] *one-st-def*[*symmetric*]]
using *m invertible-imp-inv-ex*[*of m*]
by (*auto simp*: *ring-mat-simps Units-def intro*!: *gauss-jordan-inverse-other-direction*[*of - - states - states*])

thus *?thesis*
unfolding *policy-eval-code mat-inverse-def*
by (*auto split*: *if-splits simp*: *one-st-def m-def case-prod-beta*)
qed

lemma *policy-eval-code''*[*code*]:
fixes *d*
defines $m \equiv$ (*one-st* $-$ $l \cdot_m$ ((*k-mat d*)))
shows *policy-eval-code d* = *snd* (*gauss-jordan m one-st*) $*_v$ (*vec states* (λi. *MDP-r* (*i*, *d-lookup' d i*)))
unfolding *m-def policy-eval-code' k-mat-def one-st-def* **by** (*simp add*: *mat-code*)

definition *policy-eval-code' d m* = *snd* (*gauss-jordan* (*one-st* $-$ $l \cdot_m$ ((*k-mat' d m*))) *one-st*) $*_v$ (*vec states* (λi. *MDP-r* (*i*, *d-lookup' d i*)))

lemma *dim-policy-eval-code*: *dim-vec* (*policy-eval-code d*) = *states*
by (*simp add*: *policy-eval-code-def MDP.invertible-ν_b-mat' inverse-mat-dims*(*2*))

lemma *policy-eval-correct'*:
assumes *D-Map.keys d* = {*0..<states*} *D-Map.invar d*
shows *vec-to-bfun* (*policy-eval-code d*) = *MDP.ν_b* (*MDP.mk-stationary-det* (*D-Map.map-to-fun d*))
using *policy-eval-correct assms dim-policy-eval-code MDP.ν_b-zero-notin*
by (*auto simp*: *vec-to-bfun.rep-eq*)

definition *pi-find-policy-state-code-aux' d v s* = (
let (*d'*, *v'*) = *find-policy-state-code-aux' v s in*
if L_a*-code* (*a-lookup'* (*s-lookup mdp s*) *d*) *v* = *v' then d else d'*)

definition *pi-find-policy-code d v* =
D-Map.from-list' (λs. *pi-find-policy-state-code-aux'* (*d-lookup' d s*) *v s*) [*0..<states*]

lemma *vi-find-policy-code-invar*: *D-Map.invar* (*pi-find-policy-code d v*)
unfolding *pi-find-policy-code-def* **by** *simp*

lemma *keys-vi-find-policy-code-aux-upt*: *D-Map.keys* (*pi-find-policy-code d v*) = {*0..<states*}
unfolding *pi-find-policy-code-def* **by** *simp*

lemma *find-policy-state-code-aux'-in-acts*:
assumes *s* < *states v-len v* = *states v-invar v*
shows *fst* (*find-policy-state-code-aux' v s*) $\in$ *MDP-A s*
using *MDP.A-ne MDP.A-fin assms least-arg-max-prop*[*of* λx. $x \in$ *MDP-A s*]
by (*fastforce simp*: *find-policy-state-code-aux'-eq'*)

lemma *pi-find-policy-state-code-aux′-correct*:
assumes $s < states$ *D-Map.invar d v-len v* $=$ *states v-invar v D-Map.keys d* $=$ *MDP.state-space d-lookup′ d s* $\in$ *MDP-A s*
shows *pi-find-policy-state-code-aux′* (*d-lookup′ d s*) *v s* $=$ *MDP.policy-improvement* (*D-Map.map-to-fun d*) (*V-Map.map-to-bfun v*) *s*
proof (*cases is-arg-max* (λa. *MDP.*L_a *a* (*apply-bfun* (*V-Map.map-to-bfun v*)) *s*) (λa. $a \in$ *MDP-A s*) (*D-Map.map-to-fun d s*))
case *True*
hence *aux*: L_a*-code* (*a-lookup′* (*s-lookup mdp s*) (*d-lookup′ d s*)) *v* $=$ *snd* (*find-policy-state-code-aux′ v s*)
using *MDP.A-fin*
by (*subst L-GS-code-correct′*) (*fastforce intro*!: *Max-eqI*[*symmetric*] *simp*: *assms find-policy-state-code-aux′-eq′ d-lookup′-eq-map-to-fun split*: *option.splits*)+
then show *?thesis*
proof −
have *pi-find-policy-state-code-aux′* (*d-lookup′ d s*) *v s* $=$ *d-lookup′ d s*
unfolding *pi-find-policy-state-code-aux′-def*
by (*simp add*: *aux case-prod-unfold*)
thus *?thesis*
using *True*
by (*fastforce simp*: *assms MDP.policy-improvement-def d-lookup′-eq-map-to-fun split*: *option.splits*)
qed
next
case *False*
hence L_a*-code* (*a-lookup′* (*s-lookup mdp s*) (*d-lookup′ d s*)) *v* $<$ (*MAX* $a \in$ *MDP-A s. MDP.*L_a *a* (*apply-bfun* (*V-Map.map-to-bfun v*)) *s*)
using *False assms* **by** (*auto simp*: *L-GS-code-correct′ is-arg-max-linorder not-le map-to-fun-lookup Max-gr-iff*)
thus *?thesis*
unfolding *pi-find-policy-state-code-aux′-def MDP.policy-improvement-def*
using *False*
by (*auto simp*: *assms find-policy-state-code-aux′-eq′ least-arg-max-def MDP.is-opt-act-def*)
qed

lemma *pi-find-policy-code-correct*:
assumes *v-len v* $=$ *states D-Map.keys d* $=$ *MDP.state-space v-invar v D-Map.invar d* $\bigwedge s.\ s < states \Longrightarrow$ *d-lookup′ d s* $\in$ *MDP-A s*
shows *D-Map.map-to-fun* ((*pi-find-policy-code d v*)) *s* $=$ *MDP.policy-improvement* (*D-Map.map-to-fun d*) (*V-Map.map-to-bfun v*) *s*
using *assms*
proof (*cases* $s < states$)
case *True*
then show *?thesis*
unfolding *pi-find-policy-code-def*

by (*simp add*: *assms pi-find-policy-state-code-aux'-correct D-Map.map-to-fun-def*)
next
case *False*
then show *?thesis*
using *keys-vi-find-policy-code-aux-upt assms vi-find-policy-code-invar is-arg-max-const MDP.A-outside*
by (*auto intro*!: *Least-equality simp*: *map-to-fun-notin MDP.policy-improvement-def MDP.is-opt-act-def*)
qed

definition *eq-policy d1 d2* = ($\forall$ *x*<*states*. *d-lookup d1 x* = *d-lookup d2 x*)
definition *policy-step-code d* = (
let v = *policy-eval-code d in*
pi-find-policy-code d (*V-Map.arr-tabulate* ((\$*v*) *v*) *states*))

definition *policy-step-code' d m* = (
let v = *policy-eval-code' d m in*
pi-find-policy-code d (*V-Map.arr-tabulate* ((\$*v*) *v*) *states*))

partial-function (*tailrec*) *PI-code-aux'* **where**
PI-code-aux' d m = (
let d' = *policy-step-code' d m in*
if eq-policy d d'
then d
else PI-code-aux' d' m)

partial-function (*tailrec*) *PI-code-aux* **where**
PI-code-aux d = (
let d' = *policy-step-code d in*
if eq-policy d d'
then d
else PI-code-aux d')

lemma *fold-policy-eval-update-eq*:
assumes *s* < *states D-Map.keys d* = *MDP.state-space D-Map.invar d*
shows *v-lookup* (*V-Map.arr-tabulate* (λ*x*. *policy-eval-code d* \$*v x*) *states*) *s* = (*MDP.policy-eval* (*D-Map.map-to-fun d*) *s*)
using *assms*
by (*auto simp*: *v-lookup-fold policy-eval-correct MDP.policy-eval-def*)

lemma *fold-policy-eval-update-eq'*:
assumes *D-Map.keys d* = *MDP.state-space D-Map.invar d*
shows *V-Map.map-to-bfun* (*V-Map.arr-tabulate* (λ*x*. (*policy-eval-code d* \$*v x*)) *states*) =
(*MDP.policy-eval* (*D-Map.map-to-fun d*))
proof (*rule bfun-eqI*)
fix *s*

```
 show (V-Map.map-to-bfun (V-Map.arr-tabulate (($v) (policy-eval-code
d)) states)) s =
  (MDP.policy-eval (D-Map.map-to-fun d)) s
  proof (cases s < states)
    case True
    then show ?thesis
     by (auto simp: V-Map.map-to-bfun.rep-eq assms policy-eval-correct
MDP.policy-eval-def)
  next
    case False
    then show ?thesis
        by (auto simp: MDP.policy-eval-def V-Map.map-to-bfun.rep-eq
MDP.ν_b-zero-notin)
  qed
qed

lemmas PI-code-aux.simps[code]
lemmas PI-code-aux'.simps[code]

lemmas MDP.policy-iteration.simps[simp del]

definition is-dec-det-code d ⟷
 D-Map.keys d = {0..<states} ∧ D-Map.invar d ∧ (∀ s ∈ set [0..<states].
a-lookup (s-lookup mdp s) (d-lookup' d s) ≠ None)

lemma [code]: is-dec-det-code d ⟷
  (map fst (d-inorder d)) = [0..<states] ∧ D-Map.invar d ∧ (∀ s ∈ set
[0..<states]. a-lookup (s-lookup mdp s) (d-lookup' d s) ≠ None)
proof −
  have D-Map.invar d ⟹ fst ‘ set (d-inorder d) = {0..<n} ⟹ (map
fst (d-inorder d)) = [0..<n] for n
    by (simp add: D-Map.invar-def strict-sorted-equal)
  moreover have D-Map.invar d ⟹ map fst (d-inorder d) = [0..<n]
⟹ fst ‘ set (d-inorder d) = {0..<n} for n
    using image-set[of fst d-inorder d]
    by auto
  ultimately show ?thesis
    unfolding D-Map.keys-def is-dec-det-code-def
    by blast
qed

definition PI-code d0 = (if ¬ is-dec-det-code d0 then d0 else PI-code-aux
d0)

lemma k-mat-eq': is-dec-det-code d ⟹ k-mat d = k-mat' d transi-
tion-vecs
  unfolding k-mat-def k-mat'-def Matrix.mat-eq-iff
 by (auto simp: is-dec-det-code-def intro!: transition-vecs-correct[symmetric]
intro: a-lookup-some-in-A)
```

lemma *policy-eval-code-eq′*: *is-dec-det-code d* $\Longrightarrow$ *policy-eval-code d* = *policy-eval-code′ d transition-vecs*
  unfolding *policy-eval-code″ policy-eval-code′-def*
  using *k-mat-eq′*
  by *force*

lemma *policy-step-code-eq′*: *is-dec-det-code d* $\Longrightarrow$ *policy-step-code d* = *policy-step-code′ d transition-vecs*
  unfolding *policy-step-code-def policy-step-code′-def*
  using *policy-eval-code-eq′* **by** *presburger*

lemma *policy-step-code-correct*:
  assumes *D-Map.keys d* = *MDP.state-space D-Map.invar d* ($\bigwedge$*s. s* < *states* $\Longrightarrow$ *d-lookup′ d s* ∈ *MDP-A s*)
  shows *D-Map.map-to-fun* (*policy-step-code d*) = (*MDP.policy-step* (*D-Map.map-to-fun d*))
  unfolding *policy-step-code-def MDP.policy-step-def*
  by (*auto simp*: *fold-policy-eval-update-eq′ pi-find-policy-code-correct assms*)

lemma *PI-code-aux-correct-aux*:
  assumes *D-Map.invar d D-Map.keys d* = {*0..*<*states*} ($\bigwedge$*s. s* < *states* $\Longrightarrow$ *d-lookup′ d s* ∈ *MDP-A s*)
  shows *D-Map.map-to-fun* (*PI-code-aux d*) = *MDP.policy-iteration* (*D-Map.map-to-fun d*)
    ∧ (*is-dec-det-code d* $\longrightarrow$ *PI-code-aux d* = *PI-code-aux′ d transition-vecs*)
  using *assms*
proof (*induction* (*D-Map.map-to-fun d*) *arbitrary*: *d rule*: *MDP.policy-iteration.induct*)
  case *1*
  show *?case*
  proof (*cases eq-policy d* (*policy-step-code d*))
    case *True*
   hence ∗: *D-Map.map-to-fun d s* = (*MDP.policy-step* (*D-Map.map-to-fun d*)) *s* **for** *s*
    proof (*cases s* < *states*)
      case *True*
      then show *?thesis*
      using *True vi-find-policy-code-invar 1* ‹*eq-policy d* (*policy-step-code d*)›
      by (*auto simp*: *D-Map.map-to-fun-def eq-policy-def policy-step-code-correct*[*symmetric*] *policy-step-code-def*)
    next
      case *False*
      hence *MDP.policy-step* (*D-Map.map-to-fun d*) *s* = *0*
      by (*auto simp*: *1 MDP.policy-improvement-def is-arg-max-linorder MDP.policy-step-def MDP-A-def map-to-fun-notin*)
      then show *?thesis*

using *1 D-Map.lookup-some-set-key False*
by (*auto simp*: *D-Map.map-to-fun-def split*: *option.splits*)
qed
have *D-Map.map-to-fun* (*PI-code-aux d*) = *D-Map.map-to-fun d*
by (*simp add*: *PI-code-aux.simps policy-step-code-correct True*)
thus *?thesis*
using ∗ *MDP.policy-iteration.simps*[*of D-Map.map-to-fun d*] *True*
by (*fastforce simp*: *policy-step-code-eq′ PI-code-aux′.simps*[*of d*] *PI-code-aux.simps*[*of d*])
next
case *False*
then obtain *s* **where** *s*: *s* < *states d-lookup d s* ≠ *d-lookup* (*policy-step-code d*) *s*
unfolding *eq-policy-def policy-step-code-def*
using *1*(*2*,*3*) *D-Map.lookup-notin-keys keys-vi-find-policy-code-aux-upt vi-find-policy-code-invar*
by (*auto simp*: *d-lookup′-def*)
have *invar-step*: *D-Map.invar* (*policy-step-code d*)
by (*simp add*: *policy-step-code-def vi-find-policy-code-invar*)
have *keys-step*: *D-Map.keys* (*policy-step-code d*) = *D-Map.keys d*
by (*simp add*: *1 keys-vi-find-policy-code-aux-upt policy-step-code-def*)
have ∗: *D-Map.map-to-fun d s* ≠ (*MDP.policy-step* (*D-Map.map-to-fun d*)) *s*
using *D-Map.lookup-in-keys*[*OF invar-step*] *D-Map.lookup-notin-keys*[*OF invar-step*] *s*(*2*) *keys-step invar-step 1*(*2*−*4*)
by (*fastforce dest*: *D-Map.lookup-None-set-inorder*[*OF* ‹*D-Map.invar d*›] *D-Map.lookup-some-set-key*[*OF* ‹*D-Map.invar d*›]
split: *option.splits simp*: *policy-step-code-correct*[*symmetric*] *D-Map.map-to-fun-def*)
have ∗∗: *MDP.is-dec-det* (*D-Map.map-to-fun d*)
using *1* **by** (*auto simp*: *MDP.is-dec-det-def MDP-A-def map-to-fun-lookup map-to-fun-notin*)
have *lookup′*: *s* < *states* ⟹ *d-lookup′* (*policy-step-code d*) *s* ∈ *MDP-A s* **for** *s*
using *1*(*2*−*4*) *keys-step invar-step MDP.is-dec-det-pi*
by (*force simp*: *MDP.is-dec-det-def policy-step-code-correct d-lookup′-eq-map-to-fun MDP.policy-step-def*)
have *D-Map.map-to-fun* (*PI-code-aux d*) = *D-Map.map-to-fun* (*PI-code-aux* (*policy-step-code d*))
by (*simp add*: *PI-code-aux.simps policy-step-code-correct False*)
also have ... = *MDP.policy-iteration* (*D-Map.map-to-fun* (*policy-step-code d*))
using *1*(*2*−*4*) ∗ ∗∗ *policy-step-code-correct lookup′ invar-step keys-step*
by (*intro conjunct1*[*OF 1*(*1*)]) (*auto*)
also have ... = *MDP.policy-iteration* (*MDP.policy-step* (*D-Map.map-to-fun d*))
using *1* **by** (*auto simp*: *policy-step-code-correct*)
finally have *aux*: *D-Map.map-to-fun* (*PI-code-aux d*) = *MDP.policy-iteration*

(*D-Map.map-to-fun d*)
unfolding *PI-code-aux.simps*[*of d*] *PI-code-aux′.simps*[*of d*]
using ∗∗ **by** (*auto simp*: *MDP.policy-iteration.simps*)
thus *?thesis*
proof −
have *d*: *is-dec-det-code d*
unfolding *is-dec-det-code-def*
using *1 a-lookup-None-notin-A*
by (*metis atLeastLessThan-iff set-upt*)
hence *is-dec-det-code* (*policy-step-code d*)
by (*metis a-lookup-None-notin-A atLeastLessThan-iff invar-step is-dec-det-code-def keys-step lookup′ set-upt*)
hence *PI-code-aux* (*policy-step-code d*) = *PI-code-aux′* (*policy-step-code d*) *transition-vecs*
using ∗ ∗∗ *1 invar-step keys-step lookup′ policy-step-code-correct*
by *metis*
hence *PI-code-aux d* = *PI-code-aux′ d transition-vecs*
unfolding *PI-code-aux.simps*[*of d*] *PI-code-aux′.simps*[*of d*]
using *policy-step-code-eq′*[*OF d*]
by *auto*
thus *?thesis*
using ∗∗ *aux*
by *fastforce*
qed
qed
qed

lemma *PI-code-correct*:
assumes *D-Map.invar d D-Map.keys d* = *MDP.state-space* (⋀*s. s* < *states* ⟹ *d-lookup′ d s* ∈ *MDP-A s*)
shows *D-Map.map-to-fun* (*PI-code d*) = *MDP.policy-iteration* (*D-Map.map-to-fun d*)
proof −
have *is-dec-det-code d*
unfolding *is-dec-det-code-def*
using *a-lookup-None-notin-A assms*
by (*fastforce simp*: *not-Some-eq*[*symmetric*])
thus *?thesis*
using *assms*
by (*auto simp*: *PI-code-def conjunct1*[*OF PI-code-aux-correct-aux*])
qed

lemma [*code*]: *PI-code d0* = (*if* ¬ *is-dec-det-code d0 then d0 else PI-code-aux′ d0 transition-vecs*)
using *conjunct2*[*OF PI-code-aux-correct-aux*[*of d0*]]
unfolding *PI-code-def is-dec-det-code-def*
using *a-lookup-some-in-A*
by *force*

definition *d0* = *D-Map.from-list'* ($\lambda s.$ *fst* (*hd* (*a-inorder* (*s-lookup* *mdp s*)))) [*0..<states*]

end

lemma *inorder-empty*: *Tree2.inorder am* = [] $\Longrightarrow$ *am* = $\langle\rangle$
using *Tree2.inorder.elims* **by** *blast*

global-interpretation *PI-Code*: *MDP-Code*

IArray.sub $\lambda n\ x\ arr.$ *IArray* ((*IArray.list-of arr*)[*n*:= *x*]) *IArray.length* *IArray IArray.list-of* λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder *rbt*

MDP.transitions (*Rep-Valid-MDP mdp*) *MDP.states* (*Rep-Valid-MDP* *mdp*)

starray-get $\lambda i\ x\ arr.$ *starray-set arr i x starray-length starray-of-list* $\lambda arr.$ *starray-foldr* ($\lambda x\ xs.\ x$ # *xs*) *arr* [] λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder *rbt*

MDP.disc (*Rep-Valid-MDP mdp*)
for *mdp*
defines *PI-code* = *PI-Code.PI-code*
and *PI-code-aux* = *PI-Code.PI-code-aux*
and L_a*-code* = *PI-Code.*L_a*-code*
and *a-lookup'* = *PI-Code.a-lookup'*
and *d-lookup'* = *PI-Code.d-lookup'*
and *find-policy-state-code-aux'* = *PI-Code.find-policy-state-code-aux'*
and *find-policy-state-code-aux* = *PI-Code.find-policy-state-code-aux*
and *entries* = *M.entries*
and *from-list'* = *M.from-list'*
and *pi-find-policy-code* = *PI-Code.pi-find-policy-code*
and *pi-find-policy-state-code-aux'* = *PI-Code.pi-find-policy-state-code-aux'*
and *policy-eval-code* = *PI-Code.policy-eval-code*
and *is-dec-det-code* = *PI-Code.is-dec-det-code*
and *policy-step-code* = *PI-Code.policy-step-code*
and *eq-policy* = *PI-Code.eq-policy*
and *MDP-r* = *PI-Code.MDP-r*
and *MDP-K* = *PI-Code.MDP-K*
and *d0* = *PI-Code.d0*
and *k-mat* = *PI-Code.k-mat*

and *one-st* = *PI-Code.one-st*
and *arr-tabulate* = *starray-Array.arr-tabulate*
and *transition-vecs* = *PI-Code.transition-vecs*
and *M-from-list* = *M.from-list*
and *M-lookup′* = *M.lookup′*
and *M-keys* = *M.keys*
and *M-invar* = *M.invar*

and *PI-code-aux′* = *PI-Code.PI-code-aux′*
and *policy-step-code′* = *PI-Code.policy-step-code′*
and *policy-eval-code′* = *PI-Code.policy-eval-code′*
and *k-mat′* = *PI-Code.k-mat′*

using *Rep-Valid-MDP is-MDP-def*
by *unfold-locales*
(*fastforce simp*: *Ball-set-list-all*[*symmetric*] *case-prod-beta pmf-of-list-wf-def is-MDP-def M.invar-def empty-def M.entries-def M.is-empty-def length-0-conv*[*symmetric*])+

lemmas *arr-tabulate-def*[*unfolded starray-Array.arr-tabulate-def*, *code*]
lemmas *entries-def*[*unfolded M.entries-def*, *code*]
lemmas *from-list′-def*[*unfolded M.from-list′-def*, *code*]

lemmas *M-from-list-def*[*unfolded M.from-list-def*, *code*]
lemmas *M-lookup′-def*[*unfolded M.lookup′-def*, *code*]
lemmas *M-keys-def*[*unfolded M.keys-def*, *code*]
lemmas *M-invar-def*[*unfolded M.invar-def*, *code*]

lift-definition *mat-of-row-fun-code* :: *nat* ⇒ *nat* ⇒ (*nat* ⇒ *′a vec-impl*) ⇒ *′a mat-impl* **is**
λ *nr nc f*. (*nr*, *nc*,
let m = *IArray.of-fun* (λ *i. snd* (*Rep-vec-impl* (*f i*))) *nr in*
if ∀ *i*<*nr*. *IArray.length* (*IArray.sub m i*) = *nc*
then m else Code.abort (*STR ′′set-fold-cfc RBT-set*: *ccompare* = *None′′*)
(λ-. *IArray.of-fun* (λ *i. IArray.of-fun* (λ *j. vec-index-impl* (*f i*) *j*) *nc*) *nr*))
by *auto*

lift-definition *vec-to-vec-impl* :: *′a vec* ⇒ *′a vec-impl* **is**
λ*v. vec-of-fun* (*dim-vec v*) (($) *v*)**.**

lemma *vec-impl-eqI*: *snd* (*Rep-vec-impl v*) = *snd* (*Rep-vec-impl u*) ⟹ *fst* (*Rep-vec-impl v*) = *fst* (*Rep-vec-impl u*) ⟹ *v* = *u*
by (*meson Rep-vec-impl-inject prod-eq-iff*)

lemma *vec-impl-exhaust*: (⋀*v*. *P* (*Abs-vec-impl* (*IArray.length v*, *v*))) ⟹ *P u*
by (*auto intro*: *Abs-vec-impl-induct*)

lemma *vec-to-vec-impl-code*[*code*]: *vec-to-vec-impl* (*vec-impl v*) = *v*
proof −
  have *vec-to-vec-impl* (*vec-impl* (*Abs-vec-impl* (*length v*, *IArray v*)))
= *Abs-vec-impl* (*length v*, *IArray v*) **for** *v*
  proof −
    have *vec-to-vec-impl* (*vec-impl* (*Abs-vec-impl* (*length v*, *IArray v*)))
= *vec-of-fun* (*length v*) (($*v*) (*vec-impl* (*Abs-vec-impl* (*length v*, *IArray v*))))
      unfolding *vec-to-vec-impl.abs-eq*
      by (*metis dim-vec-of-list vec-of-list vec-of-list-impl.abs-eq*)
    also have ... = *Abs-vec-impl* (*length v*, *IArray* (*map* (($*v*) (*Abs-vec* (*length v*, *Matrix.mk-vec* (*length v*) ((!!) (*IArray v*))))) [*0..<length v*]))
     by (*subst vec-impl.abs-eq*) (*auto simp*: *eq-onp-same-args vec-of-fun-def*
)
    also have ... = *Abs-vec-impl* (*length v*, *IArray* (*map* (*Matrix.mk-vec* (*length v*) ((!!) (*IArray v*))) [*0..<length v*]))
      by (*subst vec-index.abs-eq*) (*auto simp*: *eq-onp-same-args*)
    also have ... = *Abs-vec-impl* (*length v*, *IArray* (*map* (((!!) (*IArray v*))) [*0..<length v*]))
        by (*metis IArray.sub-def Matrix.mk-vec-def list-of.simps un-def-vec*)
    also have ... = *Abs-vec-impl* (*length v*, *IArray v*)
      by (*simp add*: *list.map-cong map-nth*)
    finally show *?thesis*.
  qed
  hence *vec-to-vec-impl* (*vec-impl* (*Abs-vec-impl* (*IArray.length v*, *v*)))
= *Abs-vec-impl* (*IArray.length v*, *v*) **for** *v*
    unfolding *IArray.length-def* **using** *list-of.simps iarray.exhaust* **by** *metis*
  thus *?thesis*
    by (*rule vec-impl-exhaust*)
qed

lemma *dim-row-mat-of-row-fun-code*[*simp*]: *dim-row* (*mat-impl* (*mat-of-row-fun-code nr nc f*)) = *nr*
  by (*simp add*: *dim-row-code dim-row-impl.abs-eq eq-onp-same-args mat-of-row-fun-code.abs-eq*)

lemma *dim-col-mat-of-row-fun-code*[*simp*]: *dim-col* (*mat-impl* (*mat-of-row-fun-code nr nc f*)) = *nc*
  by (*simp add*: *dim-col-code dim-col-impl.abs-eq eq-onp-same-args mat-of-row-fun-code.abs-eq*)

lemma *mat-of-row-fun-code*[*code*]: *mat-of-row-fun nr nc f* =
  mat-impl (*mat-of-row-fun-code nr nc* (λ*i*. *vec-to-vec-impl* (*f i*)))
proof −
  have *index-mat-impl* (*mat-of-row-fun-code nr nc* (λ*i*. *vec-to-vec-impl* (*f i*))) (*i*, *j*) = *f i* $*v j* **if** *i* < *nr j* < *nc* **for** *i j*

proof (*cases* ∀ *i*<*nr*. *length* (*IArray.list-of* (*snd* (*Rep-vec-impl* (*vec-to-vec-impl* (*f i*))))) = *nc*)
case *True*
then show *?thesis*
using *that*
unfolding *mat-of-row-fun-code.abs-eq vec-to-vec-impl.rep-eq*
by (*auto simp*: *index-mat-impl.abs-eq eq-onp-same-args vec-of-fun.rep-eq*)
next
case *False*
then show *?thesis*
using *that*
unfolding *mat-of-row-fun-code.abs-eq vec-to-vec-impl.rep-eq*
using *list-of-vec-index*[*of f i j*] *list-of-vec-map*[*of f i*]
by (*auto simp*: *index-mat-impl.abs-eq eq-onp-same-args vec-of-fun.rep-eq vec-index-impl.rep-eq*)
qed
thus *?thesis*
by (*auto simp*: *index-mat-code*)
qed
end
theory *PI-Code-Export-Float*
imports
PI-Code
Code-Real-Approx-By-Float-Fix
begin

The code generation for Gaussian elimination and pmfs conflicts.

code-datatype *set RBT-set Complement Collect-set Set-Monad DList-set*

lemmas *List.subset-code*(*1*)[*code*] *List.in-set-member*[*code*]

lemma [*code*]: *finite* (*set xs*) = *True* **by** *auto*

lemma *set-fold-cfc-code*[*code*]:
set-fold-cfc f b (*set* (*xs* :: ′*c*::*ccompare list*)) =
(*case ID ccompare of None* ⇒ *Code.abort STR* ′′*set-fold-cfc RBT-set*: *ccompare* = *None*′′ (λ-. *set-fold-cfc f b* (*set xs*))
| *Some* (*x* :: ′*c* ⇒ ′*c* ⇒ *order*) ⇒ *fold* (*comp-fun-commute-apply f*) (*remdups xs*) *b*)
unfolding *set-fold-cfc.rep-eq*
by (*auto split*: *option.splits simp*: *comp-fun-commute.comp-fun-commute comp-fun-commute-def*′
intro!: *comp-fun-commute-on.fold-set-fold-remdups*[*of set xs*] *Finite-Set.comp-fun-commute-on.intro*)

export-code
d0 to-valid-MDP MDP RBT-Map.update nat-map-from-list assoc-list-to-MDP RBT-Set.empty PI-code
nat-pmf-of-list pmf-of-list nat-of-integer Ratreal int-of-integer in-

verse-divide Tree2.inorder
integer-of-nat
in *SML* **module-name** *PI-Code-Float* **file-prefix** *PI-Code-Float*

end
theory *PI-Code-Export-Rat*
imports
PI-Code
begin

code-datatype *set RBT-set Complement Collect-set Set-Monad DList-set*

lemmas *List.subset-code*(*1*)[*code*] *List.in-set-member*[*code*]

lemma *finite-set-code*[*code*]: *finite* (*set xs*) = *True* **by** *auto*

lemma *set-fold-cfc-code*[*code*]:
set-fold-cfc f b (*set* (*xs* :: *'c*::*ccompare list*)) =
(*case ID ccompare of None* ⇒ *Code.abort STR ''set-fold-cfc RBT-set*: *ccompare* = *None''* (λ-. *set-fold-cfc f b* (*set xs*))
| *Some* (*x* :: *'c* ⇒ *'c* ⇒ *order*) ⇒ *fold* (*comp-fun-commute-apply f*) (*remdups xs*) *b*)
unfolding *set-fold-cfc.rep-eq*
by (*auto split*: *option.splits simp*: *comp-fun-commute.comp-fun-commute comp-fun-commute-def'*
intro!: *comp-fun-commute-on.fold-set-fold-remdups*[*of set xs*] *Finite-Set.comp-fun-commute-on.intro*)

export-code
ord-real-inst.less-eq-real quotient-of
plus-real-inst.plus-real minus-real-inst.minus-real d0 to-valid-MDP MDP RBT-Map.update
Rat.of-int divide divide-rat-inst.divide-rat divide-real-inst.divide-real nat-map-from-list
assoc-list-to-MDP nat-pmf-of-list RBT-Set.empty PI-code pmf-of-list nat-of-integer
Ratreal int-of-integer inverse-divide Tree2.inorder integer-of-nat
in *SML* **module-name** *PI-Code-Rat* **file-prefix** *PI-Code-Rat*

end
theory *Backward-Induction*
imports *MDP−Rewards.MDP-reward*
begin

locale *MDP-reward-fin* = *discrete-MDP A K*
for
A **and**
K :: *'s* ::*countable* × *'a* ::*countable* ⇒ *'s pmf* +
fixes

  $r :: ('s \times 'a) \Rightarrow$ *real* **and**
  r-fin $:: 's \Rightarrow$ *real* **and**
  $N ::$ *nat*
assumes
  r-fin-bounded: *bounded* (*range r-fin*) **and**
  r-bounded-fin: *bounded* (*range r*)
begin

interpretation *MDP-reward A K r 1*
  rewrites $1 * (x::real) = x$ **and** $\bigwedge x.(1::real)\hat{}(x::nat)=1$
  using *r-bounded-fin*
  by *unfold-locales* (*auto simp*: *algebra-simps*)

definition $\nu N\ p\ s = (\int t.\ (\sum i<N.\ r\ (t\ !!\ i)) +\ (r\text{-}fin\ (fst(t\ !!\ N)))$
$\partial\mathcal{T}\ p\ s)$

lemma *measurable-r-fin-nth* [*measurable*]: $(\lambda t.\ r\text{-}fin\ ((t\ !!\ i))) \in borel\text{-}measurable$
S
  by *measurable*

lemma *integrable-r-fin-nth* [*simp*]: *integrable* $(\mathcal{T}\ p\ s)\ (\lambda t.\ r\text{-}fin\ (fst(t$
$!!\ i)))$
  using *bounded-range-subset*[*OF r-fin-bounded*]
  by (*auto simp*: *range-composition*[*of r-fin*])

lemma νN-*eq*: $\nu N\ p\ s = (\sum i < N.\ measure\text{-}pmf.expectation\ (Pn'\ p$
$s\ i)\ r) + measure\text{-}pmf.expectation\ (Xn'\ p\ s\ N)\ r\text{-}fin$
proof –
  have $\nu N\ p\ s = (\int t.\ (\sum i<N.\ r\ (t\ !!\ i))\ \partial\mathcal{T}\ p\ s) + (\int t.\ (r\text{-}fin\ (fst(t$
$!!\ N)))\ \partial\mathcal{T}\ p\ s)$
    unfolding νN-*def*
    by (*auto intro*: *Bochner-Integration.integral-add*)
  moreover have $(\int t.\ (\sum i<N.\ r\ (t\ !!\ i))\ \partial\mathcal{T}\ p\ s) = (\sum i < N.$
$measure\text{-}pmf.expectation\ (Pn'\ p\ s\ i)\ r)$
    using ν-*fin-Suc* ν-*fin-eq-Pn* **by** *force*
  moreover have $(\int t.\ (r\text{-}fin\ (fst(t\ !!\ N)))\ \partial\mathcal{T}\ p\ s) = measure\text{-}pmf.expectation$
$(Xn'\ p\ s\ N)\ r\text{-}fin$
    by (*auto simp*: Xn'-Pn' Pn'-*eq*-$\mathcal{T}$ *integral-distr*)
  ultimately show *?thesis* **by** *auto*
qed

function νN-*eval* **where**
  $\nu N\text{-}eval\ p\ h\ s = ($
    if length $h = N$ *then r-fin s else*
    if length $h > N$ *then 0 else*
      $measure\text{-}pmf.expectation\ (p\ h\ s)\ (\lambda a.\ r\ (s,a) +$
        $measure\text{-}pmf.expectation\ (K\ (s,a))\ (\lambda s'.\ \nu N\text{-}eval\ p\ (h@[(s,a)])$
$s')))$
  by *auto*

termination
by (*relation Wellfounded.measure* (λ(-,*h*,*s*). N − *length h*)) *auto*

lemmas *abs-disc-eq*[*simp del*]
lemmas ν*N-eval.simps*[*simp del*]

lemma *pmf-bounded-integrable*: *bounded* (*range* (*f*::- ⇒ *real*)) ⟹ *integrable* (*measure-pmf p*) *f*
using *bounded-norm-le-SUP-norm*[*of f*]
by (*intro measure-pmf.integrable-const-bound*[*of* - ⨆*x*. |*f x*|]) *auto*

lemma *abs-boundedD*[*dest*]: (⋀*x*. |*f x*| ≤ (*c*::*real*)) ⟹ *bounded* (*range f*)
using *bounded-real* **by** *auto*

lemma *abs-integral-le*[*intro*]: (⋀*x*. |*f x*| ≤ (*c*::*real*)) ⟹ *abs* (*measure-pmf.expectation p f*) ≤ *c*
by (*fastforce intro*!: *pmf-bounded-integrable abs-boundedD measure-pmf.integral-le-const order.trans*[*OF integral-abs-bound*])

lemma *abs-*ν*N-eval-le*: |ν*N-eval p h s*| ≤ (N − *length h*) ∗ r_M + (⨆*s*. |*r-fin s*|)
proof (*induction* (N − *length h*) *arbitrary*: *h s*)
case *0*
then show *?case*
using *r-fin-bounded*
by (*auto simp*: ν*N-eval.simps intro*!: *bounded-imp-bdd-above cSUP-upper2*)
next
case (*Suc x*)
have N > *length h*
using *Suc*(*2*) **by** *linarith*
hence *Suc-le*: *Suc* (*length h*) ≤ N
by *auto*
have ∗: |ν*N-eval p* (*h* @ [(*s*, *a*)]) *s'*| ≤ *real* (N − *length h* − *1*) ∗ r_M + (⨆*s*. |*r-fin s*|) **for** *a s'*
using *Suc.hyps*(*1*)[*of h* @[(*s*,*a*)]] *Suc.hyps*(*2*)
by (*auto simp*: *of-nat-diff*[*OF Suc-le*] *algebra-simps*)
hence ∗∗: |*measure-pmf.expectation* (*p h s*) (λ*a*. *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-eval p* (*h* @ [(*s*, *a*)])))|
≤ *real* (N − *length h* − *1*) ∗ r_M + (⨆*s*. |*r-fin s*|)
using *Suc* **by** *auto*
have |*measure-pmf.expectation* (*p h s*) (λ*a*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-eval p* (*h* @ [(*s*, *a*)])))|
≤ |*measure-pmf.expectation* (*p h s*) (λ*a*. *r* (*s*, *a*)) + *measure-pmf.expectation* (*p h s*) (λ*a*. *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-eval p* (*h* @ [(*s*, *a*)])))|
using *abs-r-le-*r_M
by (*subst Bochner-Integration.integral-add*) (*auto intro*!: *abs-boundedD*

$*$ *pmf-bounded-integrable*)
also have ... $\leq$ |*measure-pmf.expectation* (*p h s*) ($\lambda a.$ *r* (*s*, *a*))| + | *measure-pmf.expectation* (*p h s*) ($\lambda a.$ *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-eval p* (*h* @ [(*s*, *a*)])))|
by *auto*
also have ... $\leq$ r_M + | *measure-pmf.expectation* (*p h s*) ($\lambda a.$ *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-eval p* (*h* @ [(*s*, *a*)])))|
using *abs-r-le-*r_M **by** *auto*
also have ... $\leq$ r_M + (*N* $-$ *length h* $-$ *1*) $*$ r_M + ($\bigsqcup$ *s*. |*r-fin s*|)
using $**$ **by** *force*
also have ... $\leq$ (*N* $-$ *length h*) $*$ r_M + ($\bigsqcup$ *s*. |*r-fin s*|)
using *Suc Suc-le* **by** (*auto simp*: *of-nat-diff algebra-simps*)
finally show *?case*
using ν*N-eval.simps* ‹*length h* < *N*› **by** *force*
qed

lemma *abs-*ν*N-eval-le*$'$: |ν*N-eval p h s*| $\leq$ *N* $*$ r_M + ($\bigsqcup$ *s*. |*r-fin s*|)
by (*simp add*: *mult-left-mono* r_M*-nonneg algebra-simps order.trans*[*OF abs-*ν*N-eval-le*[*of p h s*]])

lemma *measure-pmf-expectation-bind*:
assumes *bounded* (*range f*)
shows *measure-pmf.expectation* (*p* $\gg=$ *q*) (*f*::- $\Rightarrow$ *real*) = *measure-pmf.expectation p* ($\lambda x.$ *measure-pmf.expectation* (*q x*) *f*)
unfolding *measure-pmf-bind*
using *assms measure-pmf-in-subprob-space*
by (*fastforce intro*!: *Giry-Monad.integral-bind*[*of - count-space UNIV* $\bigsqcup$ *x*. |*f x*|] *bounded-imp-bdd-above cSUP-upper*)+

lemma *Pn*$'$*-shift*: *bounded* (*range* (*f* :: - $\Rightarrow$ *real*)) $\Longrightarrow$ *measure-pmf.expectation* (*p h s*)
($\lambda a.$ *measure-pmf.expectation* (*K* (*s*, *a*))
($\lambda s'.$ *measure-pmf.expectation* (*Pn*$'$ ($\lambda h'.$ *p* ((*h* @ (*s*, *a*)# *h*$'$))) *s*$'$ *n*) *f*))
= *measure-pmf.expectation* (*Pn*$'$ ($\lambda h'.$ *p* (*h* @ *h*$'$)) *s* (*Suc n*)) *f*
unfolding *PSuc*$'$ π*-Suc-def K0*$'$*-def*
by (*auto simp*: *measure-pmf-expectation-bind*)

lemma *bounded-r-snd*$'$: *bounded* (($\lambda a.$ *r* (*s*, *a*)) ‘ *X*)
using *r-bounded*$'$ *image-image*
by *metis*

lemma *bounded-r-snd*: *bounded* (*range* ($\lambda a.$ *r* (*s*, *a*)))
using *bounded-r-snd*$'$**.**

lemma ν*N-eval-eq*: *length h* $\leq$ *N* $\Longrightarrow$ ν*N-eval p h s* =
($\sum$ *i* $\in${*length h*..< *N*}.
measure-pmf.expectation (*Pn*$'$ ($\lambda h'.$ *p* (*h*@*h*$'$)) *s* (*i* $-$ *length h*)) *r*) + *measure-pmf.expectation* (*Xn*$'$ ($\lambda h'.$ *p* (*h*@*h*$'$)) *s* (*N* $-$ *length h*)) *r-fin*

proof (*induction* $N -$ *length h arbitrary*: *h s*)
  case *0*
  then show *?case*
    using *νN-eval.simps* **by** *auto*
next
  case (*Suc x*)
  hence *length h* $<$ *N*
    by *auto*
  hence
    νN-eval p h s $=$
      measure-pmf.expectation (*p h s*) (*λa. r* (*s,a*) $+$
        measure-pmf.expectation (*K* (*s,a*)) (*λs′. νN-eval p* (*h*@[(*s,a*)])
s′))
    by (*auto simp*: *νN-eval.simps*[*of p h*] *split*: *if-splits*)
  also have ... $=$
    measure-pmf.expectation (*p h s*) (*λa. r* (*s,a*)) $+$ *measure-pmf.expectation*
(*p h s*) (*λa. measure-pmf.expectation* (*K* (*s,a*)) (*λs′. νN-eval p* (*h*@[(*s,a*)])
s′))
    using *abs-νN-eval-le′ bounded-r-snd*
   by (*fastforce simp*: *bounded-real intro*!: *Bochner-Integration.integral-add*
pmf-bounded-integrable abs-integral-le)
  also have ... $=$
    ($\sum$ *i* $=$ *length h*..$<$*N. measure-pmf.expectation* (*Pn′* (*λh′. p* (*h* @
h′)) *s* (*i* $-$ *length h*)) *r*) $+$ *measure-pmf.expectation* (*Xn′* (*λh′. p* (*h* @
h′)) *s* (*N* $-$ *length h*)) *r-fin*
  proof –
   have *measure-pmf.expectation* (*p h s*) (*λa. measure-pmf.expectation*
(*K* (*s,a*)) (*λs′. νN-eval p* (*h*@[(*s,a*)]) *s′*)) $=$
    measure-pmf.expectation (*p h s*)
     (*λa. measure-pmf.expectation* (*K* (*s, a*))
        (*λs′.* ($\sum$ *i* $=$ *length* (*h* @ [(*s, a*)])..$<$*N. measure-pmf.expectation*
(*Pn′* (*λh′. p* ((*h* @ [(*s, a*)]) @ *h′*)) *s′* (*i* $-$ *length* (*h* @ [(*s, a*)]))) *r*) $+$
                measure-pmf.expectation (*Xn′* (*λh′. p* ((*h* @ [(*s, a*)]) @
h′)) *s′* (*N* $-$ *length* (*h* @ [(*s, a*)]))) *r-fin*))
      using *Suc* ‹*length h* $<$ *N*›
      by *auto*
    also have ... $=$
    measure-pmf.expectation (*p h s*)
     (*λa. measure-pmf.expectation* (*K* (*s, a*))
          (*λs′.* ($\sum$ *i* $=$ *length h* $+$ *1*..$<$*N. measure-pmf.expectation*
(*Pn′* (*λh′. p* ((*h* @ [(*s, a*)]) @ *h′*)) *s′* (*i* $-$ *length h* $-$ *1*)) *r*) $+$
                measure-pmf.expectation (*Xn′* (*λh′. p* ((*h* @ [(*s, a*)]) @
h′)) *s′* (*N* $-$ *length h* $-$ *1*)) *r-fin*))
      using *Suc* ‹*length h* $<$ *N*› *K0′-def*
      by *auto*
    also have ... $=$
    measure-pmf.expectation (*p h s*)
     (*λa. measure-pmf.expectation* (*K* (*s, a*))
          (*λs′.* ($\sum$ *i* $=$ *length h* $+$ *1*..$<$*N. measure-pmf.expectation*

$(Pn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @ $h'))$ s' $(i$ $-$ *length* h $-$ $1))$ $r))$ $+$
measure-pmf.expectation $(K$ $(s,$ $a))$
$(\lambda s'.$ *measure-pmf.expectation* $(Xn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @
$h'))$ s' $(N$ $-$ *length* h $-$ $1))$ *r-fin*$))$
using *abs-exp-r-le r-fin-bounded*
by (*fastforce intro*!: *Bochner-Integration.integral-cong*[*OF refl*]
Bochner-Integration.integral-add
pmf-bounded-integrable Bochner-Integration.integrable-sum
simp: *bounded-real*)+
also have . . . =
measure-pmf.expectation $(p$ h $s)$
$(\lambda a.$ *measure-pmf.expectation* $(K$ $(s,$ $a))$
$(\lambda s'.$ $(\sum i$ $=$ *length* h $+$ $1..<N.$ *measure-pmf.expectation*
$(Pn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @ $h'))$ s' $(i$ $-$ *length* h $-$ $1))$ $r)))$ $+$
measure-pmf.expectation $(p$ h $s)$ $(\lambda a.$ *measure-pmf.expectation* $(K$
$(s,$ $a))$ $(\lambda s'.$ *measure-pmf.expectation*
$(Xn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @ $h'))$ s' $(N$ $-$ *length* h $-$ $1))$ *r-fin*$))$
using *abs-r-le-*r_M *r-fin-bounded*
by (*fastforce intro*!:
Bochner-Integration.integral-add Bochner-Integration.integrable-sum
pmf-bounded-integrable
abs-integral-le order.trans[*OF sum-abs*] *order.trans*[*OF sum-bounded-above*[*of*
- - r_M]] *simp*: *bounded-real*)
also have . . . = *measure-pmf.expectation* $(p$ h $s)$ $(\lambda a.$ $(\sum i$ $=$
length h $+$ $1..<N.$ *measure-pmf.expectation* $(K$ $(s,$ $a))$
$(\lambda s'.$ *measure-pmf.expectation* $(Pn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @
$h'))$ s' $(i$ $-$ *length* h $-$ $1))$ $r)))$ $+$
measure-pmf.expectation $(p$ h $s)$ $(\lambda a.$ *measure-pmf.expectation* $(K$
$(s,$ $a))$ $(\lambda s'.$ *measure-pmf.expectation*
$(Xn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @ $h'))$ s' $(N$ $-$ *length* h $-$ $1))$ *r-fin*$))$
using *abs-r-le-*r_M
by (*subst Bochner-Integration.integral-sum*) (*auto intro*!: *pmf-bounded-integrable*
boundedI[*of* - r_M] *abs-integral-le*)
also have . . . = $(\sum i$ $=$ *length* h $+$ $1..<N.$ *measure-pmf.expectation*
$(p$ h $s)$ $(\lambda a.$ *measure-pmf.expectation* $(K$ $(s,$ $a))$
$(\lambda s'.$ *measure-pmf.expectation* $(Pn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @
$h'))$ s' $(i$ $-$ *length* h $-$ $1))$ $r)))$ $+$
measure-pmf.expectation $(p$ h $s)$ $(\lambda a.$ *measure-pmf.expectation* $(K$
$(s,$ $a))$ $(\lambda s'.$ *measure-pmf.expectation*
$(Xn'$ $(\lambda h'.$ p $((h$ @ $[(s,$ $a)])$ @ $h'))$ s' $(N$ $-$ *length* h $-$ $1))$ *r-fin*$))$
using *abs-r-le-*r_M
by (*subst Bochner-Integration.integral-sum*) (*auto intro*!: *pmf-bounded-integrable*
boundedI[*of* - r_M] *abs-integral-le*)
also have . . . =
$(\sum i$ $=$ *length* h $+$ $1..<N.$ (*measure-pmf.expectation* $(Pn'$ $(\lambda h'.$ p
$(h$ @ $h'))$ s $(i$ $-$ *length* $h)))$ $r)$ $+$
measure-pmf.expectation $(Xn'$ $(\lambda h'.$ p $(h$ @ $h'))$ s $(N$ $-$ *length*
$h))$ *r-fin*
using *r-bounded r-fin-bounded* ‹*length* h $<$ N›

by (*auto simp add*: *Pn′-shift Xn′-Pn′ Suc-diff-Suc range-composition*)
finally show *?thesis*
unfolding *sum.atLeast-Suc-lessThan*[*OF* ‹*length h* < *N*›] *r-dec-eq-r-K0*
by *auto*
qed
finally show *?case* **.**
qed

lemma νN-*eval-correct*: νN-*eval p* [] *s* = νN *p s*
using *lessThan-atLeast0*
by (*auto simp*: νN-*eq* νN-*eval-eq*)

lift-definition νN_b :: ($'s$, $'a$) *pol* $\Rightarrow$ $'s \Rightarrow_b$ *real* **is** νN
using *r-fin-bounded*
by (*intro bfun-normI*[*of* - $r_M * N + (\bigsqcup x.\ |r\text{-}fin\ x|)$]])
(*auto simp add*: νN-*eq* r_M-*def r-bounded bounded-abs-range intro*!: *add-mono*
order.trans[*OF integral-abs-bound*] *pmf-bounded-integrable lemma-4-3-1*
order.trans[*OF sum-abs*] *order.trans*[*OF abs-triangle-ineq*] *order.trans*[*OF sum-bounded-above*[*of* - - r_M]])

definition νN-*opt s* = ($\bigsqcup p \in \Pi_{HR}.\ \nu N\ p\ s$)
definition νN-*eval-opt h s* = ($\bigsqcup p \in \Pi_{HR}.\ \nu N$-*eval p h s*)

function νN-*opt-eqn* **where**
νN-*opt-eqn h s* = (
if length h = *N then r-fin s else*
if length h > *N then 0 else*
$\bigsqcup a \in A\ s.$ (*r* (*s*,*a*) +
measure-pmf.expectation (*K* (*s*,*a*)) ($\lambda s'$. νN-*opt-eqn* (*h*@[(*s*,*a*)]) s')))
by *auto*

termination
by (*relation Wellfounded.measure* (λ(*h*,*s*). *N* − *length h*)) *auto*

lemmas νN-*opt-eqn.simps*[*simp del*]

lemma *abs*-νN-*opt-eqn-le*: |νN-*opt-eqn h s*| ≤ (*N* − *length h*) ∗ r_M + ($\bigsqcup s.\ |r\text{-}fin\ s|$)
proof (*induction* (*N* − *length h*) *arbitrary*: *h s*)
case *0*
then show *?case*
using *r-fin-bounded*
by (*auto simp*: νN-*opt-eqn.simps intro*!: *bounded-imp-bdd-above cSUP-upper2*)
next
case (*Suc x*)
have *N* > *length h*

using *Suc*(*2*) **by** *linarith*
have ∗: $|\nu N$-*opt-eqn* (*h* @ [(*s*, *a*)]) $s'| \le$ *real* (*N* − *length h* − *1*) ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|) **for** *a* s'
using *Suc*(*1*)[*of* (*h* @ [(*s*, *a*)])] *Suc*(*2*)
by *auto*
hence |*measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)]))|
≤ *real* (*N* − *length h* − *1*) ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|) **for** *a*
using *Suc* **by** *auto*
hence ∗∗: r_M + |*measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)]))|
≤ *real* (*N* − *length h*) ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|) **for** *a*
using *Suc*
by (*auto simp*: *of-nat-diff algebra-simps*)
hence ∗: |*r* (*s*, *a*)| + |*measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)]))| ≤ *real* (*N* − *length h*) ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|) **for** *a*
using *abs-r-le-*r_M
by (*meson add-le-cancel-right order.trans*)
hence ∗: |*r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)]))| ≤ *real* (*N* − *length h*) ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|) **for** *a*
using *order.trans*[*OF abs-triangle-ineq*] **by** *auto*
have |νN-*opt-eqn h s*| = |$\bigsqcup$ *a*∈*A s*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)]))|
unfolding νN-*opt-eqn.simps*[*of h*] **using** ‹*length h* < *N*›
by *auto*
also have ... ≤ |$\bigsqcup$ *a*∈*A s*. *measure-pmf.expectation* (*return-pmf a*) (*λa*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)])))|
by *auto*
also have ... ≤ ($\bigsqcup$ *a*∈*A s*. | *measure-pmf.expectation* (*return-pmf a*) (*λa*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h* @ [(*s*, *a*)])))|)
using ‹*length h* < *N*› *A-ne* ∗
by (*auto intro*!: *boundedI abs-cSUP-le*)
also have ... ≤ *real* (*N* − *length h*) ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|)
using ∗ *A-ne*
by (*auto intro*!: *cSUP-least*)
finally show *?case*.
qed

lemma *abs-*νN*-opt-eqn-le*$'$: |νN-*opt-eqn h s*| ≤ *N* ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|)
by (*simp add*: *mult-left-mono* r_M*-nonneg algebra-simps order.trans*[*OF abs-*νN*-opt-eqn-le*[*of h s*]])

lemma *abs-*νN*-eval-opt-le*$'$: |νN-*eval-opt h s*| ≤ *N* ∗ r_M + ($\bigsqcup$ *s*. |*r-fin s*|)
unfolding νN-*eval-opt-def*
using *policies-ne abs-*νN*-eval-le*$'$

by (*auto intro*!: *order.trans*[*OF abs-cSUP-le*] *boundedI cSUP-least*)

lemma *exp-νN-eval-opt-le*: $|$*measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval-opt* *h*)$| \leq N * r_M + (\bigsqcup s.\ |$*r-fin s*$|)$
by (*metis abs-νN-eval-opt-le$'$ abs-integral-le*)

lemma *bounded-exp-νN-eval-opt*: (*bounded* (($\lambda a.$ *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval-opt* (*h a*))) ‘ *X*))
using *exp-νN-eval-opt-le*
by (*auto intro*!: *boundedI*)

lemma *bounded-r-exp-νN-eval-opt*: (*bounded* (($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval-opt* (*h a*))) ‘ *X*))
using *bounded-exp-νN-eval-opt r-bounded abs-r-le-r_M*
by (*intro bounded-plus-comp*) (*auto intro*!: *boundedI*)

lemma *integrable-r-exp-νN-eval-opt*: (*integrable* (*measure-pmf q*) (($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval-opt* (*h a*)))))
using *bounded-r-exp-νN-eval-opt pmf-bounded-integrable* **by** *blast*

lemma *exp-νN-eval-le*: $|$*measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval p h*)$| \leq N * r_M + (\bigsqcup s.\ |$*r-fin s*$|)$
by (*metis abs-νN-eval-le$'$ abs-integral-le*)

lemma *bounded-exp-νN-eval*: (*bounded* (($\lambda a.$ *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval p* (*h a*))) ‘ *X*))
using *exp-νN-eval-le*
by (*auto intro*!: *boundedI*)

lemma *bounded-r-exp-νN-eval*: (*bounded* (($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval p* (*h a*))) ‘ *X*))
using *bounded-exp-νN-eval r-bounded abs-r-le-r_M*
by (*intro bounded-plus-comp*) (*auto intro*!: *boundedI*)

lemma *integrable-r-exp-νN-eval*: (*integrable* (*measure-pmf q*) (($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-eval p* (*h a*)))))
using *bounded-r-exp-νN-eval pmf-bounded-integrable* **by** *blast*

lemma *exp-νN-opt-eqn-le*: $|$*measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-opt-eqn* *h*)$| \leq N * r_M + (\bigsqcup s.\ |$*r-fin s*$|)$
by (*metis abs-νN-opt-eqn-le$'$ abs-integral-le*)

lemma *bounded-exp-νN-opt-eqn*: (*bounded* (($\lambda a.$ *measure-pmf.expectation* (*K* (*s*, *a*)) (*νN-opt-eqn* (*h a*))) ‘ *X*))
using *exp-νN-opt-eqn-le*
by (*auto intro*!: *boundedI*)

lemma *bounded-r-exp-νN-opt-eqn*: (*bounded* (($\lambda a.$ *r* (*s*, *a*) + *mea-*

sure-pmf.expectation (*K* (*s*, *a*)) (ν*N-opt-eqn* (*h a*))) ‘ *X*))
using *bounded-exp-νN-opt-eqn r-bounded abs-r-le-r$_M$*
by (*intro bounded-plus-comp*) (*auto intro*!: *boundedI*)

lemma *integrable-r-exp-νN-opt-eqn*: (*integrable* (*measure-pmf q*) ((λ*a*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-opt-eqn* (*h a*)))))
using *bounded-r-exp-νN-opt-eqn pmf-bounded-integrable* **by** *blast*

lemma *νN-eval-le-opt-eqn*: $p \in \Pi_{HR} \Longrightarrow$ *νN-eval p h s* $\leq$ *νN-opt-eqn h s*
proof (*induction p h s rule*: *νN-eval.induct*)
case (*1 p h s*)
have *νN-eval p* (*h* @ [(*s*, *a*)]) $s' \leq$ *νN-opt-eqn* (*h* @[(*s*,*a*)]) s' **if** *length h* $< N$ **for** *a* s'
using *that 1* **by** *fastforce*
hence $*$: *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-eval p* (*h* @ [(*s*, *a*)])) $\leq$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (ν*N-opt-eqn* (*h* @ [(*s*, *a*)])) **if** *length h* $< N$ **for** *a*
using *abs-νN-eval-le$'$ abs-νN-opt-eqn-le$'$ that*
by (*fastforce intro*!: *integral-mono pmf-bounded-integrable simp*: *bounded-real*)
have $**$: $a \in$ *set-pmf* (*p h s*) $\Longrightarrow a \in A\ s$ **for** *a*
using *1 is-dec-def is-policy-def* **by** *blast*
then show *?case*
unfolding *νN-eval.simps*[*of p h*] *νN-opt-eqn.simps*[*of h*]
using *integrable-r-exp-νN-eval bounded-r-exp-νN-eval bounded-r-exp-νN-opt-eqn* $*$
by (*auto simp*: *set-pmf-not-empty intro*!: *order.trans*[*OF lemma-4-3-1*] *cSUP-mono bexI bounded-imp-bdd-above*)
qed

lemma *νN-eval-le-opt*: $p \in \Pi_{HR} \Longrightarrow$ *νN-eval-opt h s* $\geq$ *νN-eval p h s*
unfolding *νN-eval-opt-def*
using *bounded-subset-range*[*OF abs-boundedD*[*OF abs-νN-eval-le$'$*]]
by (*force intro*!: *cSUP-upper abs-boundedD bounded-imp-bdd-above*)

lemma *νN-opt-eqn-bounded*[*simp*, *intro*]: *bounded* ((ν*N-opt-eqn h*) ‘ *X*)
by (*meson Blinfun-Util.bounded-subset abs-νN-opt-eqn-le$'$ abs-boundedD subset-UNIV*)

lemma *νN-eval-opt-bounded*[*simp*, *intro*]: *bounded* ((ν*N-eval-opt h*) ‘ *X*)
by (*meson Blinfun-Util.bounded-subset abs-νN-eval-opt-le$'$ abs-boundedD subset-UNIV*)

lemma *νN-eval-bounded*[*simp*, *intro*]: *bounded* ((ν*N-eval p h*) ‘ *X*)
by (*meson Blinfun-Util.bounded-subset abs-νN-eval-le$'$ abs-boundedD subset-UNIV*)

lemma νN-*opt-ge*: *length* $h \leq N \Longrightarrow \nu N$-*opt-eqn* h $s \geq \nu N$-*eval-opt* h
s
proof (*induction* N − *length* h *arbitrary*: h s)
  case *0*
  then show *?case*
    unfolding νN-*eval-opt-def* νN-*opt-eqn.simps*[*of* h]
    using *policies-ne*
    by (*subst* νN-*eval-eq*) *auto*
next
  case (*Suc* x)
  hence *length* $h < N$
    by *linarith*
  {
    fix p **assume** $p \in \Pi_{HR}$
    have νN-*eval* p h s = *measure-pmf.expectation* (p h s) ($\lambda a.$ (r (s,a)
+
        measure-pmf.expectation (K (s,a)) ($\lambda s'.$ νN-*eval* p (h@[(s,a)])
s')))
      unfolding νN-*eval.simps*[*of* p h]
      using ‹*length* $h < N$›
      by *auto*
    also have ... $\leq$ ($\bigsqcup a \in A\ s.$ (r (s,a) +
        measure-pmf.expectation (K (s,a)) ($\lambda s'.$ νN-*eval* p (h@[(s,a)])
s')))
    using ‹$p \in \Pi_{HR}$› *is-dec-def is-policy-def bounded-r-snd′ bounded-exp-*νN*-eval*
    by (*auto intro*!: *lemma-4-3-1 bounded-plus-comp pmf-bounded-integrable*
simp: *r-bounded′*)
    also have ... $\leq$ ($\bigsqcup a \in A\ s.$ (r (s,a) +
        measure-pmf.expectation (K (s,a)) ($\lambda s'.$ νN-*opt-eqn* (h@[(s,a)])
s')))
    proof −
      have $a \in A\ s \Longrightarrow$
        r (s,a) + *measure-pmf.expectation* (K (s,a)) (νN-*eval* p (h @
[(s,a)])) $\leq$
          r (s,a) + *measure-pmf.expectation* (K (s,a)) (νN-*eval-opt*
(h@[(s,a)])) **for** a
        using *abs-boundedD*[*OF abs-*νN*-eval-opt-le′*] *abs-boundedD*[*OF*
abs-νN*-eval-le′*]
        using νN-*eval-le-opt* ‹$p \in \Pi_{HR}$›
        by (*force intro*!: *integral-mono pmf-bounded-integrable*)
      moreover have $a \in A\ s \Longrightarrow$
          r (s,a) + *measure-pmf.expectation* (K (s,a)) (νN-*eval-opt*
(h@[(s,a)])) $\leq$
          r (s,a) + *measure-pmf.expectation* (K (s,a)) (νN-*opt-eqn*
(h@[(s,a)])) **for** a
        using νN-*eval-le-opt-eqn policies-ne Suc*
      by (*auto intro*!: *integral-mono pmf-bounded-integrable cSUP-least*)
      ultimately show *?thesis*

using *A-ne bounded-imp-bdd-above bounded-r-exp-νN-opt-eqn*
by (*fastforce intro*!: *cSUP-mono*)+
qed
also have ... = *νN-opt-eqn h s*
unfolding *νN-opt-eqn.simps*[*of h*]
using ‹*length h* < *N*›
by *auto*
finally have *νN-opt-eqn h s* ≥ *νN-eval p h s*.
}
then show *?case*
unfolding *νN-eval-opt-def*
using *policies-ne*
by (*auto intro*!: *cSUP-least*)
qed

lemma *Sup-wit-ex*:
assumes (*d* ::*real*)> *0*
assumes *X* ≠ {}
assumes *bdd-above* (*f* ‘ *X*)
shows ∃ *x* ∈ *X*. ($\bigsqcup$ *x* ∈ *X*. *f x*) < *f x* + *d*
proof −
have ∃ *x* ∈*X*. ($\bigsqcup$ *x* ∈ *X*. *f x*) − *d* < *f x*
using *assms*
by (*auto simp*: *less-cSUP-iff*[*symmetric*])
thus *?thesis*
by *force*
qed

lemma *νN-opt-eqn-markov*: *length h* ≤ *N* ⟹ *length h* = *length h′* ⟹ *νN-opt-eqn h* = *νN-opt-eqn h′*
proof (*induction N* − *length h arbitrary*: *h h′*)
case *0*
then show *?case*
by (*auto simp*: *νN-opt-eqn.simps*)
next
case (*Suc x*)
{
fix *s*
have *νN-opt-eqn h s* = ($\bigsqcup$ *a* ∈ *A s*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K*(*s*,*a*)) (*νN-opt-eqn* (*h*@[(*s*,*a*)])))
using *Suc* **by** (*fastforce simp*: *νN-opt-eqn.simps*)
also have ... = ($\bigsqcup$ *a* ∈ *A s*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K*(*s*,*a*)) (*νN-opt-eqn* (*h′*@[(*s*,*a*)])))
using *Suc*
by (*auto intro*!: *SUP-cong Bochner-Integration.integral-cong Suc*(*1*)[*THEN cong*])
also have ... = *νN-opt-eqn h′ s*
using *Suc νN-opt-eqn.simps* **by** *fastforce*

finally have νN-*opt-eqn h s* = νN-*opt-eqn h' s* **.**
}
thus *?case* **by** *auto*
qed

lemma νN-*opt-le*:
fixes *eps* :: *real*
assumes *eps* > *0*
shows $\exists\, p \in \Pi_{MD}.\ \forall\, h\ s.$ *length h* $\leq N \longrightarrow \nu N$-*eval* (*mk-markovian-det p*) *h s* + *real* (*N* − *length h*) ∗ *eps* ≥ νN-*opt-eqn h s*
proof −
define *p* **where** *p* = (*λn s. if n* ≥ *N then SOME a. a* ∈ *A s else SOME a. a* ∈ *A s* ∧
r (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*replicate n* (*s*, *SOME a. a* ∈ *A s*) @ [(*s*,*a*)])) + *eps* > νN-*opt-eqn* (*replicate n* (*s*,*SOME a. a* ∈ *A s*)) *s*)
have ∗: ∃ *a* . *a* ∈ *A s* ∧
r (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* (*h*@[(*s*,*a*)])) + *eps* > νN-*opt-eqn h s*
if *length h* < *N*
for *h s*
using *that Sup-wit-ex*[*OF assms A-ne*, *unfolded Bex-def*] *bounded-imp-bdd-above bounded-r-exp-*νN*-opt-eqn*
by (*auto simp*: νN-*opt-eqn.simps*)
hence ∗∗: ∃ *a* . *a* ∈ *A s* ∧
r (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*, *a*)) (νN-*opt-eqn* ((*replicate n* (*s*,*SOME a. a* ∈ *A s*))@[(*s*,*a*)])) + *eps* > νN-*opt-eqn* (*replicate n* (*s*,*SOME a. a* ∈ *A s*)) *s*
if *n* < *N* **for** *n s*
using *that* **by** *simp*
have *p-prop*: *p n s* ∈ *A s* ∧ *r* (*s*, *p n s*) + *measure-pmf.expectation* (*K* (*s*, *p n s*)) (νN-*opt-eqn* ((*replicate n* (*s*,*SOME a. a* ∈ *A s*))@[(*s*,*p n s*)])) + *eps* > νN-*opt-eqn* ((*replicate n* (*s*,*SOME a. a* ∈ *A s*))) *s*
if *n* < *N* **for** *n s*
using *someI-ex*[*OF* ∗∗[*OF that*], *of s*] *that*
by (*auto simp*: *p-def*)
hence *p-prop'*: *p* (*length h*) *s* ∈ *A s* ∧ *r* (*s*, *p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s*, *p* (*length h*) *s*)) (νN-*opt-eqn* (*h*@[(*s*,*p* (*length h*) *s*)])) + *eps* > νN-*opt-eqn h s*
if *length h* < *N* **for** *h s*
using *that*
by (*auto simp*:
νN-*opt-eqn-markov*[*of h* (*replicate* (*length h*) (*s*,*SOME a. a* ∈ *A s*))]
νN-*opt-eqn-markov*[*of* (*h*@[(*s*,*p* (*length h*) *s*)]) (*replicate* (*length h*) (*s*, *SOME a. a* ∈ *A s*) @ [(*s*, *p* (*length h*) *s*)])])
have *p n s* ∈ *A s* **for** *n s*
using *SOME-is-dec-det is-dec-det-def p-def p-prop* **by** *auto*
hence *p*:*p* ∈ Π_{MD}

using *is-dec-det-def* **by** *force*
{
fix *h s p*
assume $p \in \Pi_{MD}$
and
p: $\bigwedge h\ s.$ *length h* $<$ *N* $\Longrightarrow$ *r* (*s*, *p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s*, *p* (*length h*) *s*)) (ν*N-opt-eqn* (*h*@[(*s*,*p* (*length h*) *s*)])) + *eps* $>$ ν*N-opt-eqn h s*
have *length h* $\leq$ *N* $\Longrightarrow$ ν*N-eval* (*mk-markovian-det p*) *h s* + *real* (*N* $-$ *length h*) $*$ *eps* $\geq$ ν*N-opt-eqn h s*
proof (*induction N* $-$ *length h arbitrary*: *h s*)
case *0*
hence $*$: *length h* = *N*
by *auto*
thus *?case*
by (*auto simp*: ν*N-opt-eqn.simps* ν*N-eval.simps*)
next
case (*Suc x*)
hence $*$: *length h* $<$ *N*
by *auto*
have ν*N-opt-eqn h s* $-$ *real* (*N* $-$ *length h*) $*$ *eps* $<$ *r* (*s*, *p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s*, *p* (*length h*) *s*)) (ν*N-opt-eqn* (*h*@[(*s*,*p* (*length h*) *s*)])) $-$ *real* (*N* $-$ *length h*) $*$ *eps* + *eps*
using *p*[*OF* $*$, *of s*] **by** *auto*
also have ... = *r* (*s*, *p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s*, *p* (*length h*) *s*)) (ν*N-opt-eqn* (*h*@[(*s*,*p* (*length h*) *s*)])) $-$ *real* (*N* $-$ *length h* $-$ *1*) $*$ *eps*
proof $-$
have *real* (*N* $-$ *length h* $-$ *1*) = *real* (*N* $-$ *length h*) $-$ *1*
using $*$ **by** (*auto simp*: *algebra-simps*)
thus *?thesis*
by *algebra*
qed
also have ... = *r* (*s*, *p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s*, *p* (*length h*) *s*)) ($\lambda s'$. ν*N-opt-eqn* (*h*@[(*s*,*p* (*length h*) *s*)]) s' $-$ *real* (*N* $-$ *length h* $-$ *1*) $*$ *eps*)
by (*subst Bochner-Integration.integral-diff*) (*auto intro*: *pmf-bounded-integrable*)

also have ... $\leq$ *r* (*s*, *p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s*, *p* (*length h*) *s*)) (ν*N-eval* (*mk-markovian-det p*) (*h*@[(*s*,*p* (*length h*) *s*)]))
using *Suc*(*1*)[*of h*@[-]] *Suc* $*$
by (*intro add-mono integral-mono pmf-bounded-integrable bounded-minus-comp*) (*auto simp*: *algebra-simps*)
also have ... = ν*N-eval* (*mk-markovian-det p*) *h s*
using *Suc*
by (*auto simp*: *mk-markovian-det-def* ν*N-eval.simps*)
finally show *?case*
by *auto*

qed
}
thus *?thesis*
using *p p-prop'* **by** *blast*
qed

lemma ν*N-opt-le'*:
fixes *eps* :: *real*
assumes *eps* > *0*
shows $\exists\, p \in \Pi_{MD}.\ \forall\, h\ s.$ *length h* $\leq N \longrightarrow \nu$*N-eval* (*mk-markovian-det p*) *h s* + *eps* $\geq \nu$*N-opt-eqn h s*
proof −
obtain *p* **where** $p{\in}\Pi_{MD}$ **and** $\bigwedge h\ s.$ *length h* $\leq N \Longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval* (*mk-markovian-det p*) *h s* + *real* (*N* − *length h*) ∗ (*eps*/*N*)
using ν*N-opt-le*[*of eps* / *N*] ν*N-opt-le assms*
by (*cases N* = *0*) *force+*
hence ∗∗: $\bigwedge h\ s.$ *length h* $\leq N \Longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval* (*mk-markovian-det p*) *h s* + *eps* − ((*eps* ∗ *length h*) / *N*)
using *assms*
by (*cases N* = *0*) (*auto simp*: *algebra-simps of-nat-diff intro*: *add-increasing*)
moreover have ∗:*eps* ∗ *real* (*length h*) / *N* $\geq$ *0* **for** *h*
using *assms* **by** *auto*
ultimately have $\bigwedge h\ s.$ *length h* $\leq N \Longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval* (*mk-markovian-det p*) *h s* + *eps*
by (*auto intro*!: *order.trans*[*OF* ∗∗])
thus *?thesis*
using ‹$p \in \Pi_{HD}$› **by** *blast*
qed

lemma *mk-det-preserves*: $p \in \Pi_{HD} \Longrightarrow$ (*mk-det p*) $\in \Pi_{HR}$
unfolding *is-policy-def mk-det-def*
by (*auto simp*: *is-dec-def is-dec-det-def*)

lemma *mk-markovian-det-preserves*: $p \in \Pi_{MD} \Longrightarrow$ (*mk-markovian-det p*) $\in \Pi_{HR}$
unfolding *is-policy-def mk-markovian-det-def*
by (*auto simp*: *is-dec-def is-dec-det-def*)

lemma ν*N-opt-eq*:
assumes *length h* $\leq N$
shows ν*N-opt-eqn h s* = ν*N-eval-opt h s*
proof −
{
fix *eps* :: *real*
assume *0* < *eps*
hence $\exists\, p{\in}\Pi_{HR}.\ \forall\, h\ s.$ *length h* $\leq N \longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval p h s* + *eps*

using *mk-markovian-det-preserves* ν*N-opt-le*$'$[*of eps*]
by *auto*
then obtain *p* **where** $p \in \Pi_{HR}$ **and** $**$: *length h* $\leq N \Longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval p h s* + *eps* **for** *h s*
by *auto*
hence *length h* $\leq N \Longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval-opt h s* + *eps* **for** *h s*
using ν*N-eval-le-opt*[*of p*]
by (*auto intro*: *order.trans*[*OF* $**$])
}
hence *length h* $\leq N \Longrightarrow \nu$*N-opt-eqn h s* $\leq \nu$*N-eval-opt h s*
by (*meson field-le-epsilon*)
thus *?thesis*
using ν*N-opt-ge assms antisym* **by** *auto*
qed

lemma ν*N-opt-eqn-correct*: ν*N-opt s* = ν*N-opt-eqn* [] *s*
using ν*N-eval-correct* ν*N-eval-opt-def* ν*N-opt-def* ν*N-opt-eq* **by** *force*

lemma *thm-4-3-4*:
assumes *eps* $\geq$ *0* *p* $\in \Pi_{MD}$
and $\bigwedge$*h s. length h* $< N \Longrightarrow$ *r* (*s, p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s, p* (*length h*) *s*)) (ν*N-opt-eqn* (*h*@[(*s, p* (*length h*) *s*)])) + *eps*
$\geq$ ($\bigsqcup$ *a* $\in$ *A s. r* (*s, a*) + *measure-pmf.expectation* (*K* (*s,a*)) (ν*N-opt-eqn* (*h*@[(*s, a*)])))
shows $\bigwedge$*h s. length h* $\leq N \Longrightarrow \nu$*N-eval* (*mk-markovian-det p*) *h s* + (*N* $-$ *length h*) $*$ *eps* $\geq \nu$*N-opt-eqn h s*
$\bigwedge$*s.* ν*N* (*mk-markovian-det p*) *s* + *N* $*$ *eps* $\geq \nu$*N-opt s*
proof $-$
show ν*N-eval* (*mk-markovian-det p*) *h s* + (*N* $-$ *length h*) $*$ *eps* $\geq \nu$*N-opt-eqn h s* **if** *length h* $\leq N$ **for** *h s*
using *assms that*
proof (*induction N* $-$ *length h arbitrary*: *h s*)
case *0*
then show *?case*
using ν*N-eval.simps* ν*N-opt-eqn.simps* **by** *force*
next
case (*Suc x*)
have ν*N-opt-eqn h s* = ($\bigsqcup$ *a* $\in$ *A s. r* (*s, a*) + *measure-pmf.expectation* (*K* (*s,a*)) (ν*N-opt-eqn* (*h*@[(*s, a*)])))
using *Suc.hyps*(*2*) ν*N-opt-eqn.simps* **by** *fastforce*
also have ... $\leq$ *r* (*s, p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s, p* (*length h*) *s*)) (ν*N-opt-eqn* (*h*@[(*s, p* (*length h*) *s*)])) + *eps*
using *Suc.hyps*(*2*) *Suc.prems*(*3*)
by *simp*
also have ... $\leq$ *r* (*s, p* (*length h*) *s*) + *measure-pmf.expectation* (*K* (*s, p* (*length h*) *s*)) ($\lambda s'$. ν*N-eval* (*mk-markovian-det p*) (*h*@[(*s, p* (*length h*) *s*)]) *s'* +

$(N - \mathit{length}\ (h@[(s,p\ (\mathit{length}\ h)\ s)])) * \mathit{eps}) + \mathit{eps}$
using *Suc*(*1*)[*of* (*h*@[(*s*,*p* (*length h*) *s*)])] *Suc.hyps*(*2*) *assms*
by (*auto intro*!: *integral-mono pmf-bounded-integrable bounded-plus-comp*)
also have $\ldots = r\ (s,\ p\ (\mathit{length}\ h)\ s) + \mathit{measure\text{-}pmf.expectation}\ (K\ (s,\ p\ (\mathit{length}\ h)\ s))\ (\nu N\text{-}\mathit{eval}\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ (h@[(s,\ p\ (\mathit{length}\ h)\ s)])) + (N - \mathit{length}\ h) * \mathit{eps}$
using *Suc*
by (*subst Bochner-Integration.integral-add*) (*auto simp*: *of-nat-diff left-diff-distrib distrib-right intro*!: *pmf-bounded-integrable*)
also have $\ldots = \nu N\text{-}\mathit{eval}\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ h\ s + (N - \mathit{length}\ h) * \mathit{eps}$
using *Suc*
by (*auto simp add*: ν*N-eval.simps mk-markovian-det-def*)
finally show *?case*.
qed
from *this*[*of* []] **show** $\nu N\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ s + N * \mathit{eps} \geq \nu N\text{-}\mathit{opt}\ s$ **for** *s*
using ν*N-eval-correct* ν*N-opt-eqn-correct*
by *auto*
qed

lemma ν*N-has-eps-opt-pol*:
assumes $\mathit{eps} > 0$
shows $\exists\, p \in \Pi_{MD}.\ \forall\, s.\ \nu N\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ s + \mathit{eps} \geq \nu N\text{-}\mathit{opt}\ s$
proof –
obtain *p* **where** $p{\in}\Pi_{MD}$ **and**
P: $\bigwedge h\ s.\ \mathit{length}\ h \leq N \Longrightarrow \nu N\text{-}\mathit{opt\text{-}eqn}\ h\ s \leq \nu N\text{-}\mathit{eval}\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ h\ s + \mathit{eps}$
using ν*N-opt-le*′[*of eps*] *assms* **by** *auto*
from *P*[*of* []] **have** $\nu N\text{-}\mathit{opt\text{-}eqn}\ []\ s \leq \nu N\text{-}\mathit{eval}\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ []\ s + \mathit{eps}$ **for** *s*
by *auto*
thus *?thesis*
unfolding ν*N-opt-eqn-correct*
using ν*N-eval-correct* ‹$p \in \Pi_{HD}$› **by** *auto*
qed

lemma ν*N-le-opt*: $p \in \Pi_{HR} \Longrightarrow \nu N\ p\ s \leq \nu N\text{-}\mathit{opt}\ s$
by (*metis* ν*N-eval-correct* ν*N-eval-le-opt-eqn* ν*N-opt-eqn-correct*)

lemma ν*N-has-opt-pol*:
assumes $\bigwedge h\ s.$
$\mathit{length}\ h < N$
$\Longrightarrow \exists\, a \in A\ s.\ r\ (s,\ a) + \mathit{measure\text{-}pmf.expectation}\ (K\ (s,a))\ (\nu N\text{-}\mathit{opt\text{-}eqn}\ (h@[(s,a)]))$
$= (\bigsqcup a \in A\ s.\ r\ (s,\ a) + \mathit{measure\text{-}pmf.expectation}\ (K\ (s,a))\ (\nu N\text{-}\mathit{opt\text{-}eqn}\ (h@[(s,a)])))$
shows $\exists\, p \in \Pi_{MD}.\ \forall\, s.\ \nu N\ (\mathit{mk\text{-}markovian\text{-}det}\ p)\ s = \nu N\text{-}\mathit{opt}\ s$

```
proof −
  define p where p = (λn s. if n ≥ N then SOME a. a ∈ A s else
      SOME a. a ∈ A s ∧
          r (s, a) + measure-pmf.expectation (K (s, a)) (νN-opt-eqn
(replicate n (s,SOME a. a ∈ A s)@[(s,a)])) = νN-opt-eqn (replicate n
(s,SOME a. a ∈ A s)) s
)
  have p-short: p n s = (
      SOME a. a ∈ A s ∧
          r (s, a) + measure-pmf.expectation (K (s, a)) (νN-opt-eqn
(replicate n (s,SOME a. a ∈ A s)@[(s,a)])) = νN-opt-eqn (replicate n
(s,SOME a. a ∈ A s)) s)
    if n < N for n s
    unfolding p-def using that by auto
  have ∗: p n s ∈ A s
    (n < N ⟹ r (s, p n s) + measure-pmf.expectation (K (s,p n s))
(νN-opt-eqn ((replicate n (s,SOME a. a ∈ A s))@[(s,p n s)]))
      = (⨆ a ∈ A s. r (s, a) + measure-pmf.expectation (K (s,a))
(νN-opt-eqn ((replicate n (s,SOME a. a ∈ A s))@[(s,a)])))) for n s
      using someI-ex[OF assms[unfolded Bex-def]] SOME-is-dec-det
      by (auto simp: νN-opt-eqn.simps is-dec-det-def p-def)
    have νN (mk-markovian-det p) s ≥ νN-opt s for s
    proof (intro thm-4-3-4(2)[of 0 p, simplified])
      show ∀ n. is-dec-det (p n)
        using ∗
        by (auto simp: is-dec-det-def)
    next
      {
        fix h :: ('s × 'a) list and s
        assume length h < N
        have (⨆ a∈A s. r (s, a) + measure-pmf.expectation (K (s, a))
(νN-opt-eqn (h @ [(s, a)]))) =
            (⨆ a ∈ A s. r (s, a) + measure-pmf.expectation (K (s,a))
(νN-opt-eqn ((replicate (length h) (s,SOME a. a ∈ A s))@[(s,a)])))
          using ‹length h < N›
           by (auto intro!: SUP-cong Bochner-Integration.integral-cong
νN-opt-eqn-markov[THEN cong])
        also have … = r (s, p (length h) s) + measure-pmf.expectation
(K (s,p (length h) s)) (νN-opt-eqn ((replicate (length h) (s,SOME a. a
∈ A s))@[(s,p (length h) s)]))
          using ∗ ‹length h < N› by presburger
        also have … = r (s, p (length h) s) + measure-pmf.expectation
(K (s,p (length h) s)) (νN-opt-eqn (h@[(s,p (length h) s)]))
          using ‹length h < N›
        by (auto intro!: Bochner-Integration.integral-cong νN-opt-eqn-markov[THEN
cong])
        finally show (⨆ a∈A s. r (s, a) + measure-pmf.expectation (K
(s, a)) (νN-opt-eqn (h @ [(s, a)])))
            ≤ r (s, p (length h) s) + measure-pmf.expectation (K (s, p
```

(*length h*) *s*)) (*νN-opt-eqn* (*h* @ [(*s*, *p* (*length h*) *s*)]))
        by *auto*
    }
  qed
  hence *νN* (*mk-markovian-det p*) *s* = *νN-opt s* **for** *s*
    using *νN-le-opt* ∗(*1*) *mk-markovian-det-preserves*
    by (*simp add*: *is-dec-det-def order-antisym*)
  thus *?thesis*
    using ∗(*1*)
    by (*auto simp*: *is-dec-det-def*)
qed

lemma *ex-Max*: *finite X* ⟹ *X* ≠ {} ⟹ ∃ *x* ∈ *X*. *f x* = *Max* (*f* ‘ *X*)
  by (*metis* (*mono-tags*, *opaque-lifting*) *Max-in empty-is-image finite-imageI imageE*)

lemma *fin-A-imp-opt-pol*:
  assumes ⋀*s*. *finite* (*A s*)
  shows ∃ *p*∈Π_{MD}. ∀ *s*. *νN* (*mk-markovian-det p*) *s* = *νN-opt s*
  using *A-ne assms νN-has-opt-pol*
  by (*fastforce simp*: *cSup-eq-Max intro*!: *ex-Max*)

16 Backward Induction

function *bw-ind-aux* **where**
  bw-ind-aux n s = (
    if n = *N then r-fin s else*
    if n > *N then 0 else*
      ⨆ *a* ∈ *A s*. (*r* (*s*,*a*) +
        measure-pmf.*expectation* (*K* (*s*,*a*)) (λ*s*′. *bw-ind-aux* (*Suc n*) *s*′)))
  by *auto*

termination
  by (*relation Wellfounded*.*measure* (λ(*h*,*s*). *N* − *h*)) *auto*

lemmas *bw-ind-aux*.*simps*[*simp del*]

lemma *bw-ind-aux-eq*: *bw-ind-aux* (*length h*) *s* = *νN-opt-eqn h s*
  by (*induction h s rule*: *νN-opt-eqn*.*induct*)
   (*auto simp*: *bw-ind-aux*.*simps νN-opt-eqn*.*simps split*: *if-splits intro*!: *Bochner-Integration*.*integral-cong SUP-cong*)

fun *bw-ind-aux*′ **where**
  bw-ind-aux′ (*Suc n*) *m* = (
    let m′ = (λ*i s*.
      if i = *n*

then ($\bigsqcup a \in A\ s.\ (r\ (s,a) + measure\text{-}pmf.expectation\ (K\ (s,a))\ (m\ (Suc\ n)))$)
else $m\ i\ s$) *in*
$bw\text{-}ind\text{-}aux'\ n\ m'$) |
$bw\text{-}ind\text{-}aux'\ 0\ m = m$

definition $bw\text{-}ind = bw\text{-}ind\text{-}aux'\ N\ (\lambda i\ s.\ if\ i = N\ then\ r\text{-}fin\ s\ else\ 0)$

lemma $bw\text{-}ind\text{-}aux'\text{-}const[simp]$:
assumes $i \geq n$
shows $bw\text{-}ind\text{-}aux'\ n\ m\ i = m\ i$
using *assms*
proof (*induction n arbitrary*: *m i*)
case *0*
then show *?case* **by** (*auto simp*: *bw-ind-aux′.simps*)
next
case (*Suc n*)
then show *?case*
by *auto*
qed

lemma $bw\text{-}ind\text{-}aux'\text{-}indep$:
assumes $i < n$ **and**
$\bigwedge j.\ j > i \Longrightarrow m\ j = m'\ j$
shows $bw\text{-}ind\text{-}aux'\ n\ m\ i\ s = bw\text{-}ind\text{-}aux'\ n\ m'\ i\ s$
using *assms*
proof (*induction n arbitrary*: *m i m′*)
case *0*
then show *?case*
by *fastforce*
next
case (*Suc n*)
show *?case*
proof (*cases* $i < n$)
case *True*
then show *?thesis*
by (*auto intro*!: *Suc*(*1*) *ext simp*: *Suc*(*2*,*3*))
next
case *False*
then show *?thesis*
using *Suc.prems*(*1*) *less-Suc-eq*
by (*auto simp*: *Suc*)
qed
qed

lemma $bw\text{-}ind\text{-}aux'\text{-}simps'$: $i < n \Longrightarrow bw\text{-}ind\text{-}aux'\ n\ m\ i\ s = (\bigsqcup a \in A\ s.\ (r\ (s,a) + measure\text{-}pmf.expectation\ (K\ (s,a))\ (bw\text{-}ind\text{-}aux'\ n\ m\ (Suc\ i))))$

proof (*induction n arbitrary*: *m i s*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  have *bw-ind-aux′* (*Suc n*) *m i s* = *bw-ind-aux′ n* ($\lambda i\ s.$ *if* $i = n$ *then*
$\bigsqcup a{\in}A\ s.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (m\ (Suc\ n))$
else m i s) *i s*
    by *auto*
  also have ... = ($\bigsqcup a{\in}A\ s.\ r\ (s,\ a) + measure\text{-}pmf.expectation$ (*K*
(*s*, *a*)) ((*bw-ind-aux′* (*Suc n*) *m* (*Suc i*))))
    using *Suc.prems le-less-Suc-eq*
    by (*cases* $n \le i$) (*auto simp*: *Suc.IH bw-ind-aux′-const*)
  finally show *?case*.
qed

lemma *bw-ind-correct*: $n \le N \Longrightarrow$ *bw-ind n* = *bw-ind-aux n*
  unfolding *bw-ind-def*
proof (*induction* $N - n$ *arbitrary*: *n*)
  case *0*
  show *?case*
    using *0*
    by (*subst bw-ind-aux.simps*) (*auto*)
next
  case (*Suc x*)
  thus *?case*
    by (*auto simp*: *bw-ind-aux′-simps′ bw-ind-aux.simps intro*!: *ext*)
qed

definition *bw-ind-pol-gen* (*d* :: *′a set* ⇒ *′a*) *n s* = (
  if $n \ge N$ *then d* (*A s*)
  else
    d ({*a* . *is-arg-max* ($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*,
a)) (*bw-ind-aux* (*Suc n*))) ($\lambda a.$ $a \in A\ s$) *a*}))

lemma *bw-ind-pol-is-arg-max*:
  assumes $\bigwedge X.\ X \ne \{\} \Longrightarrow d\ X \in X$ $\bigwedge s.$ *finite* (*A s*)
  shows *is-arg-max* ($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*,
a)) (*bw-ind-aux* (*Suc n*))) ($\lambda a.$ $a \in A\ s$) (*d* ({*a* . *is-arg-max* ($\lambda a.$ *r* (*s*,
a) + *measure-pmf.expectation* (*K* (*s*, *a*)) (*bw-ind-aux* (*Suc n*))) ($\lambda a.$
$a \in A\ s$) *a*}))
proof −
  let *?s* = {*a*. *is-arg-max* ($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation* (*K*
(*s*, *a*)) (*bw-ind-aux* (*Suc n*))) ($\lambda a.$ $a \in A\ s$) *a*}
  have ‹*d ?s* ∈ *?s*›
   using *assms*(*1*)[*of* {*a*. *is-arg-max* ($\lambda a.$ *r* (*s*, *a*) + *measure-pmf.expectation*
(*K* (*s*, *a*)) (*bw-ind-aux* (*Suc n*))) ($\lambda a.$ $a \in A\ s$) *a*}]
    using *finite-is-arg-max A-ne assms*
    by (*auto simp add*: *finite-is-arg-max*)

thus *?thesis*
by *auto*
qed

lemma *bw-ind-pol-gen*:
assumes $\bigwedge X.\ X \neq \{\} \Longrightarrow d\ X \in X\ \bigwedge s.\ \mathit{finite}\ (A\ s)$
shows *bw-ind-pol-gen* $d \in \Pi_{MD}$
proof –
have $***$:$X \neq \{\} \Longrightarrow X \subseteq Y \Longrightarrow d\ X \in Y$ **for** $X\ Y$
using *assms*
by *auto*
have $\exists\, a.$ *is-arg-max* ($\lambda a.$ r (s, a) + *measure-pmf.expectation* (K (s, a)) (*bw-ind-aux* (*Suc* n))) ($\lambda a.$ $a \in A\ s$) a **for** $n\ s$
using *finite-is-arg-max*[*OF assms*(*2*) *A-ne*]
by *auto*
thus *?thesis*
unfolding *bw-ind-pol-gen-def is-dec-det-def*
by (*force intro*!: $***$)
qed

lemma
assumes $\bigwedge X.\ X \neq \{\} \Longrightarrow d\ X \in X\ \ \bigwedge s.\ \mathit{finite}\ (A\ s)$ *length* $h \leq N$
shows νN-*eval* (*mk-markovian-det* (*bw-ind-pol-gen* d)) $h\ s = \nu N$-*eval-opt* $h\ s$
proof –
have ($\bigwedge h\ s.$ *length* $h < N \Longrightarrow$
($\bigsqcup a{\in}A\ s.$ r (s, a) + *measure-pmf.expectation* (K (s, a)) (νN-*opt-eqn* (h @ [(s, a)])))
$\leq r$ (s, *bw-ind-pol-gen* d (*length* h) s) +
measure-pmf.expectation (K (s, *bw-ind-pol-gen* d (*length* h) s))
(νN-*opt-eqn* (h @ [(s, *bw-ind-pol-gen* d (*length* h) s)])))
using *A-ne bw-ind-pol-is-arg-max*[*OF assms*(*1*,*2*)]
unfolding *bw-ind-aux-eq*[*symmetric*]
by (*auto intro*!: *cSUP-least simp*: *bw-ind-pol-gen-def*)
hence *length* $h \leq N \Longrightarrow \nu N$-*opt-eqn* $h\ s \leq \nu N$-*eval* (*mk-markovian-det* (*bw-ind-pol-gen* d)) $h\ s$ **for** $h\ s$
using *assms bw-ind-pol-gen thm-4-3-4*[*of 0 bw-ind-pol-gen d*, *simplified*]
by *auto*
thus *?thesis*
using νN-*opt-eq* νN-*eval-le-opt assms bw-ind-pol-gen mk-markovian-det-preserves*
by (*auto intro*!: *antisym*)
qed

lemma *bw-ind-aux*$'$-*eq*: $n \leq N \Longrightarrow$ *bw-ind-aux*$'$ N ($\lambda i\ s.$ *if* $i = N$ *then* *r-fin* s *else* 0) $n =$ *bw-ind-aux* n
using *bw-ind-def bw-ind-correct* **by** *presburger*

```
end

end
theory Fin-Code
  imports
    ../Backward-Induction
    Code-Setup
begin

locale MDP-nat-fin = MDP-nat + MDP-reward-fin
begin
end

locale MDP-Code-Fin = MDP-Code-raw +
   R-Fin-Map : Array′ r-fin-lookup :: ′tf ⇒ nat ⇒ real r-fin-update
r-fin-len r-fin-array r-fin-list r-fin-invar +
  V-Map : Array′ v-lookup :: ′tv ⇒ nat ⇒ real v-update v-len v-array
v-list v-invar +
  D-Map : Array′ d-lookup :: ′td ⇒ nat ⇒ nat d-update d-len d-array
d-list d-invar +
  VD-Map : Array′ vd-lookup :: ′tvd ⇒ nat ⇒ (nat × real) vd-update
vd-len vd-array vd-list vd-invar
  for v-lookup v-update v-len v-array v-list v-invar
    and d-lookup d-update d-len d-array d-list  d-invar
    and vd-lookup vd-update vd-len vd-array vd-list  vd-invar
   and r-fin-lookup r-fin-update r-fin-len r-fin-array r-fin-list r-fin-invar
+
  fixes
    N-code :: nat and
    r-fin-code :: ′tf
begin

definition v-map-from-list xs = v-array xs
definition MDP-r-fin s = (if s ≥ states then 0 else r-fin-lookup
r-fin-code s)

lemma bounded-r-fin: bounded (range MDP-r-fin)
  unfolding MDP-r-fin-def
 by (fastforce simp add: nle-le bounded-const finite-nat-set-iff-bounded-le
intro!: finite-imageI)

sublocale MDP: MDP-reward-disc (MDP-A) (MDP-K) (MDP-r) 0
  using bounded-MDP-r
  by unfold-locales auto

sublocale MDP: MDP-act (MDP-A) (MDP-K) λX. LEAST x. x ∈
X
  using MDP.MDP-reward-disc-axioms
  by unfold-locales
```

(*auto intro*: *LeastI2 simp*: *MDP-reward-disc.max-L-ex-def has-arg-max-def finite-is-arg-max*)

sublocale *MDP*: *MDP-nat-fin* λX. *LEAST* x. $x \in X$ (*MDP-A*) (*MDP-K*) *states* (*MDP-r*) *MDP-r-fin N-code*
  using *MDP-K-closed MDP-K-comp-closed MDP-r-zero-notin-states MDP-A-outside bounded-MDP-r bounded-r-fin*
  by *unfold-locales* (*auto intro*: *LeastI2*)

sublocale *V-Map*: *Array-real v-lookup v-update v-len v-array v-list v-invar*
  by *unfold-locales*

sublocale *V-Map*: *Array-zero v-lookup v-update v-len v-array v-list v-invar*
  by *unfold-locales*

sublocale *D-Map*: *Array-zero d-lookup d-update d-len d-array d-list d-invar*
  by *unfold-locales*

definition L_a*-code rp v* = (
    let (*r*, *ps*) = *rp in r* + (*foldl* (λ *acc* (s', *p*). *p* $*$ *v-lookup v* s' + *acc*)) *0 ps*)

lemma L_a*-code-correct*:
  assumes
    s < *states*
    v-len v = *states v-invar v*
    pmf-of-list (*snd rps*) = *MDP-K* (*s*, *a*) *pmf-of-list-wf* (*snd rps*)
    fst ‘ *set* (*snd rps*) ⊆ {*0*..<*states*} *fst rps* = *MDP-r* (*s*, *a*)
  shows L_a*-code rps v* = *MDP-r* (*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*,*a*)) (*V-Map.map-to-bfun v*)
proof −
  have *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*v-lookup v*) = *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*V-Map.map-to-bfun v*)
    using *assms MDP.K-closed*
    by (*force simp*: *V-Map.map-to-bfun.rep-eq split*: *option.splits*
      intro!: *Bochner-Integration.integral-cong-AE AE-pmfI*)
  have *foldl* (λ*acc x*. *f x* + *acc*) *x xs* = ($\sum$ *x*←*xs*. *f x*) + *x* **for** *f xs* **and** *x* :: *real*
    by (*induction xs arbitrary*: *x*) (*auto simp*: *algebra-simps*)
  hence $*$: ($\sum$ *x*←*xs*. *f x*) = *foldl* (λ*acc x*. *f x* + *acc*) (*0*::*real*) *xs* **for** *f xs*
    by (*metis add.right-neutral*)
  have *foldl* (λ*acc* (s', *p*). *p* $*$ *v-lookup v* s' + *acc*) *0* (*snd rps*) = *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*apply-bfun* (*V-Map.map-to-bfun v*))
    unfolding *assms*(*4*)[*symmetric*]

using *assms*(*5*−*7*)
by (*auto intro*!: *foldl-cong simp*: *pmf-of-list-expectation* ∗ *V-Map.map-to-bfun.rep-eq assms*(*2*,*3*))
thus *?thesis*
unfolding L_a-*code-def*
by (*auto simp add*: *assms case-prod-unfold*)
qed

definition *find-policy-state-code-aux v s* =
(*least-arg-max-max-ne* (λ(-, *rsuccs*).
L_a-*code rsuccs v*) ((*a-inorder* (*s-lookup mdp s*))))

definition *find-policy-state-code-aux′ v s* = (
case find-policy-state-code-aux v s of ((*a*, -, -), *v*) ⇒ (*a*, *v*))

definition *vi-find-policy-code* (*v*::′*tv*) = *VD-Map.arr-tabulate* (λ*s*. (*find-policy-state-code-aux′ v s*)) *states*

lemma *find-policy-state-code-aux-eq*:
assumes *s* < *states*
shows *find-policy-state-code-aux′ v s* = (*least-arg-max-max-ne* (λ*a*.
L_a-*code* (*a-lookup′* (*s-lookup mdp s*) *a*) *v*) ((*map fst* (*a-inorder* (*s-lookup mdp s*)))))
unfolding *find-policy-state-code-aux′-def find-policy-state-code-aux-def*
using *assms A-Map.is-empty-def ne-s-lookup*
by(*subst least-arg-max-max-ne-app′*[*symmetric*])
(*auto simp*: *case-prod-unfold a-lookup′-def A-Map.entries-def A-Map.inorder-lookup-Some assms invar-s-lookup*)

lemma *L-GS-code-correct′*:
assumes *s* < *states v-len v* = *states v-invar v a* ∈ *MDP-A s*
shows L_a-*code* (*a-lookup′* (*s-lookup mdp s*) *a*) *v* =
MDP-r(*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*,*a*)) (*V-Map.map-to-bfun v*)
using *pmf-of-list-wf-mdp assms set-list-pmf-in-states*
by (*intro* L_a-*code-correct*)
(*auto simp*: *fst-sa-lookup′-eq*[*symmetric*] *snd-sa-lookup′-eq*)

lemma *find-policy-state-code-aux′-eq′*:
assumes *s* < *states v-len v* = *states v-invar v*
shows *find-policy-state-code-aux′ v s* =
(*least-arg-max* (λ*a*. *MDP-r*(*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*,*a*)) (*V-Map.map-to-bfun v*)) (λ*a*. *a* ∈ *MDP-A s*),
Max ((λ*a*. *MDP-r*(*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*,*a*)) (*V-Map.map-to-bfun v*)) ‘ (*MDP-A s*)))
proof −
have *find-policy-state-code-aux′ v s* = *least-arg-max-max-ne* (λ*a*. L_a-*code* (*a-lookup′* (*s-lookup mdp s*) *a*) *v*) (*map fst* (*a-inorder* (*s-lookup*

mdp s)))
  using *find-policy-state-code-aux-eq assms* **by** *auto*
 also have ‹. . . = (*least-arg-max* (*λa.* L_a*-code* (*a-lookup′* (*s-lookup mdp s*) *a*) *v*) (*List.member* (*map fst* (*a-inorder* (*s-lookup mdp s*)))),
  MAX a∈*set* (*map fst* (*a-inorder* (*s-lookup mdp s*))). L_a*-code* (*a-lookup′* (*s-lookup mdp s*) *a*) *v*)›
  using *A-Map.is-empty-def assms*(*1*) *A-Map.invar-def A-inv-locale S-Map.lookup-in-list s-invar s-len A-ne-locale*
  by (*auto simp*: *fold-max-eq-arg-max′*)
 also have ‹. . . = (*least-arg-max* (*λa. MDP-r*(*s, a*) + *measure-pmf.expectation* (*MDP-K* (*s,a*)) (*V-Map.map-to-bfun v*)) (*List.member* (*map fst* (*a-inorder* (*s-lookup mdp s*)))),
  MAX a∈*set* (*map fst* (*a-inorder* (*s-lookup mdp s*))). *MDP-r*(*s, a*) + *measure-pmf.expectation* (*MDP-K* (*s,a*)) (*V-Map.map-to-bfun v*))›
  using *assms a-inorderD*(*1*) *A-Map.keys-def MDP-A-def*
  by (*auto intro*!: *least-arg-max-cong simp*: *L-GS-code-correct′ in-set-member*[*symmetric*])
 also have ‹. . . = (*least-arg-max* (*λa. MDP-r*(*s, a*) + *measure-pmf.expectation* (*MDP-K* (*s,a*)) (*V-Map.map-to-bfun v*)) (*λa. a* ∈ *MDP-A s*),
  MAX a∈*MDP-A s. MDP-r*(*s, a*) + *measure-pmf.expectation* (*MDP-K* (*s,a*)) (*V-Map.map-to-bfun v*))›
  using *assms A-Map.entries-def A-Map.keys-def A-Map.entries-imp-keys*
  by (*auto intro*!: *least-arg-max-cong′ simp*: *MDP-A-def in-set-member*[*symmetric*])
 finally show *?thesis*.
qed

lemma *vi-find-policy-code-correct*:
 assumes *s* < *states v-len v* = *states v-invar v*
 shows *vd-lookup* (*vi-find-policy-code v*) *s* =
 (*least-arg-max*
  (*λa. MDP-r*(*s, a*) + *measure-pmf.expectation* (*MDP-K* (*s,a*)) (*V-Map.map-to-bfun v*))
  (*λa. a* ∈ *MDP-A s*)
 , *Max* ((*λa. MDP-r*(*s, a*) + *measure-pmf.expectation* (*MDP-K* (*s,a*)) (*V-Map.map-to-bfun v*)) ‘ (*MDP-A s*)))
 unfolding *vi-find-policy-code-def*
 by (*auto simp*: *find-policy-state-code-aux′-eq′ assms*)

fun *bw-ind-aux-code* **where**
 bw-ind-aux-code (*Suc n*) *last-v m-v m-d* = (*let*
  vd = *vi-find-policy-code last-v*;
  v = *V-Map.arr-tabulate* (*λs. snd* (*vd-lookup vd s*)) *states*;
  d = *D-Map.arr-tabulate* (*λs. fst* (*vd-lookup vd s*)) *states in*
  bw-ind-aux-code n v (*last-v* # *m-v*) (*d* # *m-d*)) |
 bw-ind-aux-code 0 last-v m-v m-d = (*last-v* # *m-v*, *m-d*)

definition *bw-ind-code* = *bw-ind-aux-code N-code* (*V-Map.arr-tabulate* (*r-fin-lookup r-fin-code*) *states*) [] []

lemma *bw-ind-aux-code-fst-index*: *i* < *length v0* ⟹ *fst* (*bw-ind-aux-code*

n vl v0 d0) ! (*i* + *n*) =
  (*vl*#*v0*) ! *i*
 by (*induction n arbitrary*: *vl v0 d0 i*) (*auto simp*: *add-Suc*[*symmetric*] *simp del*: *add-Suc*)

lemma *bw-ind-aux-code-fst-index'*: $n \le i \Longrightarrow$ *fst* (*bw-ind-aux-code n vl v0 d0*) ! *i* =
  (*vl*#*v0*) ! (*i* − *n*)
 by (*induction n arbitrary*: *vl v0 d0 i*) *auto*

lemma *bw-ind-aux-code-snd-index'*: $n \le i \Longrightarrow$ *snd* (*bw-ind-aux-code n vl v0 d0*) ! *i* =
  (*d0*) ! (*i* − *n*)
 by (*induction n arbitrary*: *vl v0 d0 i*) *auto*

lemma *bw-ind-code-aux-correct*:
 fixes *n vl v0 d0*
 defines *d* ≡ *snd* (*bw-ind-aux-code n vl v0 d0*)
 defines *v* ≡ *fst* (*bw-ind-aux-code n vl v0 d0*)
 assumes *v-len vl* = *states*
 assumes *v-invar vl*
 assumes ⋀*s*. *s* < *states* ⟹ *m n s* = *v-lookup vl s*
 assumes *s* < *states*
 shows (*i* ≤ *n* ⟶ *v-lookup* (*v* ! *i*) *s* = *MDP.bw-ind-aux' n m i s*) ∧
  (*i* < *n* ⟶ *d-lookup* (*d* ! *i*) *s* = (*least-arg-max*
   (λ*a*. *MDP-r* (*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*MDP.bw-ind-aux' n m* (*Suc i*)))
   (λ*a*. *a* ∈ *MDP-A s*)))
 unfolding *v-def d-def*
 using *assms*
proof (*induction n arbitrary*: *m i v0 d0 vl s*)
 case (*Suc n*)
 show *?case*
 proof (*cases i* = *Suc n*)
  case *True*
  then show *?thesis*
   by (*simp add*: *Suc bw-ind-aux-code-fst-index'*)
 next
  case *False*
  then show *?thesis*
  proof (*cases i* = *n*)
   case *True*
   thus *?thesis*
    using *MDP-K-closed Suc.prems True*
   by (*auto intro*!: *least-arg-max-cong Bochner-Integration.integral-cong-AE AE-pmfI SUP-cong AE-pmfI*
     simp: *cSup-eq-Max*[*symmetric*] *bw-ind-aux-code-snd-index' bw-ind-aux-code-fst-index'*
    subset-eq V-Map.map-to-bfun.rep-eq vi-find-policy-code-correct)

next
case *False*
have ∗: $\bigwedge s.\ s < states \Longrightarrow$
($\bigsqcup a{\in}MDP\text{-}A\ s.\ MDP\text{-}r\ (s,\ a) + measure\text{-}pmf.expectation$ $(MDP\text{-}K\ (s,\ a))\ (m\ (Suc\ n))) =$
v-lookup (*V-Map.arr-tabulate* ($\lambda s.$ *snd* (*vd-lookup* (*vi-find-policy-code vl*) *s*)) *states*) *s*
using *MDP.K-closed*
by (*auto simp*: *subset-eq vi-find-policy-code-correct Suc cSup-eq-Max*[*symmetric*] *V-Map.map-to-bfun.rep-eq*
intro!: *AE-pmfI Bochner-Integration.integral-cong-AE SUP-cong*)
hence *v-lookup* (*fst* (*bw-ind-aux-code* (*Suc n*) *vl v0 d0*) ! *i*) $s =$ $MDP.bw\text{-}ind\text{-}aux'$ (*Suc n*) *m i s* **if** $i \le Suc\ n$
unfolding *bw-ind-aux-code.simps Let-def*
using ‹$i \le Suc\ n$› ‹$i \ne Suc\ n$›
by (*subst Suc*(*1*)[*THEN conjunct1*]) (*auto simp*: *Suc*)
moreover have *d-lookup* (*snd* (*bw-ind-aux-code* (*Suc n*) *vl v0 d0*) ! *i*) $s =$
least-arg-max ($\lambda a.\ MDP\text{-}r\ (s,\ a) + measure\text{-}pmf.expectation$ $(MDP\text{-}K\ (s,\ a))\ (MDP.bw\text{-}ind\text{-}aux'\ (Suc\ n)\ m\ (Suc\ i)))\ (\lambda a.\ a \in MDP\text{-}A\ s)$ **if** $i < Suc\ n$
unfolding *bw-ind-aux-code.simps Let-def*
using ‹$i < Suc\ n$› ‹$i \ne Suc\ n$› ∗ *False*
by (*subst Suc*(*1*)[*THEN conjunct2*]) (*auto simp*: *Suc*)
ultimately show *?thesis*
by *auto*
qed
qed
qed *auto*

lemma *bw-ind-code-correct*:
defines $d \equiv$ *snd bw-ind-code*
defines $v \equiv$ *fst bw-ind-code*
shows $\bigwedge n\ s.\ n \le N\text{-}code \Longrightarrow s < states \Longrightarrow v\text{-}lookup\ (v\ !\ n)\ s =$ $MDP.bw\text{-}ind\ n\ s$
and $\bigwedge n.\ n < N\text{-}code \Longrightarrow s < states \Longrightarrow d\text{-}lookup\ (d\ !\ n)\ s =$ $MDP.bw\text{-}ind\text{-}pol\text{-}gen\ (\lambda X.\ LEAST\ a.\ a \in X)\ n\ s$
proof (*goal-cases*)
case (*1 n s*)
then show *?case*
unfolding *MDP.bw-ind-def*
by (*subst bw-ind-code-aux-correct*[*THEN conjunct1*, *THEN mp*, *symmetric*])
(*auto simp add*: *MDP-r-fin-def bw-ind-code-def v-def*)
next
case (*2 n*)
then show *?case*

unfolding *MDP.bw-ind-pol-gen-def d-def bw-ind-code-def*
by (*subst bw-ind-code-aux-correct*[*THEN conjunct2*])
(*auto simp*: *least-arg-max-def*[*symmetric*] *MDP-r-fin-def MDP.bw-ind-aux′-eq*[*symmetric*])
qed
end

global-interpretation *Fin-Code*:
MDP-Code-Fin

IArray.sub λn *x arr*. *IArray* ((*IArray.list-of arr*)[*n*:= *x*]) *IArray.length*
IArray IArray.list-of λ-. *True*

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup Tree2.inorder
rbt

MDP.transitions (*Rep-Valid-MDP mdp*) *MDP.states* (*Rep-Valid-MDP*
mdp)

starray-get λi *x arr*. *starray-set arr i x starray-length starray-of-list*
λ*arr*. *starray-foldr* (λx *xs*. *x* # *xs*) *arr* [] λ-. *True*

starray-get λi *x arr*. *starray-set arr i x starray-length starray-of-list*
λ*arr*. *starray-foldr* (λx *xs*. *x* # *xs*) *arr* [] λ-. *True*

starray-get λi *x arr*. *starray-set arr i x starray-length starray-of-list*
λ*arr*. *starray-foldr* (λx *xs*. *x* # *xs*) *arr* [] λ-. *True*

starray-get λi *x arr*. *starray-set arr i x starray-length starray-of-list*
λ*arr*. *starray-foldr* (λx *xs*. *x* # *xs*) *arr* [] λ-. *True*

for *mdp r-fin-code N-code*
defines L_a*-code* = *Fin-Code.*L_a*-code*
and *a-lookup′* = *Fin-Code.a-lookup′*
and *v-map-from-list* = *Fin-Code.v-map-from-list*
and *find-policy-state-code-aux′* = *Fin-Code.find-policy-state-code-aux′*
and *find-policy-state-code-aux* = *Fin-Code.find-policy-state-code-aux*
and *entries* = *M.entries*
and *from-list′* = *M.from-list′*
and *from-list* = *M.from-list*
and *bw-ind-code* = *Fin-Code.bw-ind-code*
and *bw-ind-aux-code* = *Fin-Code.bw-ind-aux-code*
and *vi-find-policy-code* = *Fin-Code.vi-find-policy-code*

and *arr-tabulate* = *starray-Array.arr-tabulate*
using *Rep-Valid-MDP*
by *unfold-locales*
(*fastforce simp*: *Ball-set-list-all*[*symmetric*] *case-prod-beta pmf-of-list-wf-def is-MDP-def RBT-Set.empty-def M.invar-def empty-def M.entries-def M.is-empty-def length-0-conv*[*symmetric*])+

lemmas *arr-tabulate-def*[*unfolded starray-Array.arr-tabulate-def*, *code*]
lemmas *entries-def*[*unfolded M.entries-def*, *code*]
lemmas *from-list′-def*[*unfolded M.from-list′-def*, *code*]
lemmas *from-list-def*[*unfolded M.from-list-def*, *code*]

function *tabulate* **where**
tabulate f acc upper n = (
if n < *upper then tabulate f* (*update n* (*f n*) *acc*) *upper* (*Suc n*) *else acc*)
by *auto*
termination
by (*relation Wellfounded.measure* (λ(-, -, *i*,*N*). *i* − *N*)) *auto*

lemma *tabulate-Suc*: *j* ≤ *n′* ⟹ *update n′* (*f n′*) (*tabulate f m n′ j*) = *tabulate f m* (*Suc n′*) *j*
proof (*induction n′* − *j arbitrary*: *m n′ j*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc j*)
then show *?case*
by *auto*
qed

lemma *from-list′-upt* [*code-unfold*]: *from-list′ f* [*0*..<*n*] = *tabulate f empty n 0*
proof −
have *j* ≤ *n* ⟹ *foldl* (λ*acc s*. *update s* (*f s*) *acc*) *m* [*j*..<*n*] = *tabulate f m n j* **for** *m j*
proof (*induction n* − *j arbitrary*: *m n j*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc x*)
then obtain *n′* **where** *n′*: *n* = *Suc n′*
using *diff-le-self Suc-le-D* **by** *metis*
then show *?case*
using *Suc*
by (*auto simp del*: *tabulate.simps simp*: *n′ tabulate-Suc*)
qed
thus *?thesis*
unfolding *from-list′-def M.from-list′-def*

by *auto*
qed

end
theory *Fin-Code-Export-Float*
imports
Fin-Code
Code-Real-Approx-By-Float-Fix
begin

export-code
starray-to-list
to-valid-MDP MDP bw-ind-code v-map-from-list
RBT-Map.update nat-map-from-list assoc-list-to-MDP RBT-Set.empty nat-pmf-of-list pmf-of-list
nat-of-integer Ratreal int-of-integer inverse-divide Tree2.inorder integer-of-nat
in *SML* **module-name** *Fin-Code-Float* **file-prefix** *Fin-Code-Float*

end
theory *Fin-Code-Export-Rat*
imports
Fin-Code
begin

export-code
bw-ind-code starray-to-list
ord-real-inst.less-eq-real quotient-of v-map-from-list
plus-real-inst.plus-real minus-real-inst.minus-real to-valid-MDP MDP RBT-Map.update
Rat.of-int divide divide-rat-inst.divide-rat divide-real-inst.divide-real nat-map-from-list
assoc-list-to-MDP nat-pmf-of-list RBT-Set.empty pmf-of-list nat-of-integer Ratreal int-of-integer
inverse-divide Tree2.inorder integer-of-nat
in *SML* **module-name** *Fin-Code-Rat* **file-prefix** *Fin-Code-Rat*

end

References

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