Verified Algorithms for Solving Markov Decision Processes

Maximilian Schäffeler and Mohammad Abdulaziz

December 28, 2021

Abstract

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

Contents

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

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

1 Value Iteration

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

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

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

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

lemma *vi-rel-dec*:
  assumes $l \neq 0$ $\mathcal{L}_b\ v \neq \nu_b\text{-}opt$
  shows $\lceil log\ (1\ /\ l)\ (dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt) - c\rceil < \lceil log\ (1\ /\ l)\ (dist\ v\ \nu_b\text{-}opt) - c\rceil$
⟨*proof*⟩

lemma *dist-$\mathcal{L}_b$-lt-dist-opt*: $dist\ v\ (\mathcal{L}_b\ v) \leq 2 * dist\ v\ \nu_b\text{-}opt$
⟨*proof*⟩

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

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

termination
⟨*proof*⟩

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*: $(1 - l) * dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (\mathcal{L}_b\ v)$
  ⟨*proof*⟩

lemma *dist-$\mathcal{L}_b$-opt-eps*:
  assumes $eps > 0$ $2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1-l)$
  shows $dist\ (\mathcal{L}_b\ v)\ \nu_b\text{-}opt < eps\ /\ 2$
⟨*proof*⟩

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

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

lemma *value-iteration-error*:
  assumes $eps > 0$
  shows $dist\ (value\text{-}iteration\ eps\ v)\ \nu_b\text{-}opt < eps\ /\ 2$
  ⟨*proof*⟩

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

definition $vi\text{-}policy\ eps\ v = find\text{-}policy\ (value\text{-}iteration\ eps\ v)$

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

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

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

lemma
  assumes $vi\ v\ (Suc\ n) \leq vi\ v\ n$
  shows $vi\ v\ (Suc\ n + m) \leq vi\ v\ (n + m)$
⟨*proof*⟩

lemma
  assumes $vi\ v\ n \leq vi\ v\ (Suc\ n)$
  shows $vi\ v\ (n + m) \leq vi\ v\ (Suc\ n + m)$
⟨*proof*⟩

lemma $vi\ v \longrightarrow \nu_b\text{-}opt$
  ⟨*proof*⟩

lemma $(\lambda n.\ dist\ (vi\ v\ (Suc\ n))\ (vi\ v\ n)) \longrightarrow 0$
  ⟨*proof*⟩

end

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

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

lemma *find-policy-error-bound*:
assumes $eps > 0$ $2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1-l)$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (find\text{-}policy\ (\mathcal{L}_b\ v))))\ \nu_b\text{-}opt < eps$
⟨*proof*⟩

lemma *vi-policy-opt*:
assumes $0 < eps$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (vi\text{-}policy\ eps\ v)))\ \nu_b\text{-}opt < eps$
⟨*proof*⟩

lemma *lemma-6-3-1-d*:
assumes $eps > 0$
assumes $2 * l * dist\ (vi\ v\ (Suc\ n))\ (vi\ v\ n) < eps * (1-l)$
shows $dist\ (vi\ v\ (Suc\ n))\ \nu_b\text{-}opt < eps\ /\ 2$
⟨*proof*⟩

end

context *MDP-act* **begin**

definition $find\text{-}policy'\ (v :: 's \Rightarrow_b real)\ s = arb\text{-}act\ (opt\text{-}acts\ v\ s)$

definition $vi\text{-}policy'\ eps\ v = find\text{-}policy'\ (value\text{-}iteration\ eps\ v)$

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*⟩

lemma *vi-policy'-opt*:
assumes $eps > 0$ $l > 0$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (vi\text{-}policy'\ eps\ v)))\ \nu_b\text{-}opt < eps$
⟨*proof*⟩

end
end

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

begin

2 Policy Iteration

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

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

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

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

context *MDP-act*
begin

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

definition $\textit{policy-step}\ d = \textit{policy-improvement}\ d\ (\textit{policy-eval}\ d)$

function $\textit{policy-iteration} :: ('s \Rightarrow 'a) \Rightarrow ('s \Rightarrow 'a)$ **where**
  $\textit{policy-iteration}\ d = ($
  $\textit{let}\ d' = \textit{policy-step}\ d\ \textit{in}$
  $\textit{if}\ d = d' \vee \neg \textit{is-dec-det}\ d\ \textit{then}\ d\ \textit{else policy-iteration}\ d')$
  $\langle \textit{proof} \rangle$

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*: $\textit{is-dec-det}\ (\textit{policy-improvement}\ d\ v)$
  $\langle \textit{proof} \rangle$

lemma *policy-improvement-is-dec-det*: $d \in D_D \Longrightarrow \textit{policy-improvement}\ d\ v \in D_D$
  $\langle \textit{proof} \rangle$

lemma *policy-improvement-improving*:
  assumes $d \in D_D$

shows ν*-improving* v (*mk-dec-det* (*policy-improvement* d v))
⟨*proof*⟩

lemma *eval-policy-step-L*:
assumes *is-dec-det* d
shows L (*mk-dec-det* (*policy-step* d)) (*policy-eval* d) $= \mathcal{L}_b$ (*policy-eval* d)
⟨*proof*⟩

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*⟩

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* (*mk-dec-det* d)) $= \nu_b$*-opt*
⟨*proof*⟩

end

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

locale *MDP-PI-finite* = *MDP-act* A K r l *arb-act*
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*⟩

lemma *finite-rel*: *finite* $\{(u, v)$. *is-dec-det* $u \wedge$ *is-dec-det* $v \wedge \nu_b$ (*mk-stationary-det* u) $>$
ν_b (*mk-stationary-det* v)$\}$
⟨*proof*⟩

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*⟩

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*⟩

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)
⟨*proof*⟩

lemma *pi-pi*[*simp*]: $d \in D_D \Longrightarrow$ *policy-step* (*policy-iteration* d) = *policy-iteration* d
⟨*proof*⟩

lemma *policy-iteration-correct*:
$d \in D_D \Longrightarrow \nu_b$ (*mk-stationary-det* (*policy-iteration* d)) = ν_b-*opt*
⟨*proof*⟩
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-*$\mathcal{L}$
⟨*proof*⟩

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

lemma *apply-nu-inv-mat*:
(*id-blinfun* $-$ l $*_R$ $\mathcal{P}_1$ d) v = *Bfun* ($\lambda i.$ ((*nu-inv-mat* d) $*v$ (*vec-lambda* v)) \$ i)
⟨*proof*⟩

lemma *bounded-linear-vec-lambda*: *bounded-linear* ($\lambda x.$ *vec-lambda* (x :: $'s \Rightarrow_b$ *real*))

⟨*proof*⟩

lemma *bounded-linear-vec-lambda-blinfun*:
fixes f :: $('s \Rightarrow_b \mathit{real}) \Rightarrow_L ('s \Rightarrow_b \mathit{real})$
shows *bounded-linear* ($\lambda v.$ *vec-lambda* (*apply-bfun* (*blinfun-apply* f (*bfun.Bfun* ((\$) v)))))
⟨*proof*⟩

lemma *invertible-nu-inv-max*: *invertible* (*nu-inv-mat* d)
⟨*proof*⟩

end

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

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

lemma $\mathcal{L}$*-fin-eq-det*: $\mathcal{L}\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
⟨*proof*⟩

lemma $\mathcal{L}_b$*-fin-eq-det*: $\mathcal{L}_b\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
⟨*proof*⟩

sublocale *MDP-PI-finite* A K r l $\lambda X.$ *Least* ($\lambda x.\ x \in X$)
⟨*proof*⟩

end
end

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

3 Modified Policy Iteration

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

3.1 The Advantage Function B

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

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

lemma *B-eq-$\mathcal{L}$*: $B\ v\ s = \mathcal{L}\ v\ s - v\ s$
⟨*proof*⟩

B is a bounded function.

lift-definition $B_b :: ('s \Rightarrow_b real) \Rightarrow 's \Rightarrow_b real$ **is** B
⟨*proof*⟩

lemma *B_b-eq-$\mathcal{L}_b$*: $B_b\ v = \mathcal{L}_b\ v - v$
⟨*proof*⟩

lemma *$\mathcal{L}_b$-eq-SUP-L_a*: $\mathcal{L}_b\ v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
⟨*proof*⟩

3.2 Optimization of the Value Function over Multiple Steps

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

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

lemma *U-zero* [*simp*]: $U\ 0\ v = v$
⟨*proof*⟩

lemma *U-one-eq-$\mathcal{L}$*: $U\ 1\ v\ s = \mathcal{L}\ v\ s$
⟨*proof*⟩

lift-definition $U_b :: nat \Rightarrow ('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow_b real)$ **is** U
⟨*proof*⟩

lemma *U_b-contraction*: $dist\ (U_b\ m\ v)\ (U_b\ m\ u) \le l\ \widehat{}\ m * dist\ v\ u$
⟨*proof*⟩

lemma *U_b-conv*:
$\exists!v.\ U_b\ (Suc\ m)\ v = v$
$(\lambda n.\ (U_b\ (Suc\ m)\ ^{\frown\frown}\ n)\ v) \longrightarrow (THE\ v.\ U_b\ (Suc\ m)\ v = v)$
⟨*proof*⟩

lemma *U_b-convergent*: $convergent\ (\lambda n.\ (U_b\ (Suc\ m)\ ^{\frown\frown}\ n)\ v)$
⟨*proof*⟩

lemma *U_b-mono*:

assumes $v \le u$
shows $U_b\ m\ v \le U_b\ m\ u$
⟨*proof*⟩

lemma U_b*-le-*$\mathcal{L}_b$: $U_b\ m\ v \le (\mathcal{L}_b\ \widehat{}\widehat{}\ m)\ v$
⟨*proof*⟩

lemma *L-iter-le-*U_b:
assumes $d \in D_R$
shows $(L\ d\widehat{}\widehat{}m)\ v \le U_b\ m\ v$
⟨*proof*⟩

lemma *lim-*U_b: $lim\ (\lambda n.\ (U_b\ (Suc\ m)\ \widehat{}\widehat{}\ n)\ v) = \nu_b\text{-}opt$
⟨*proof*⟩

lemma U_b*-tendsto*: $(\lambda n.\ (U_b\ (Suc\ m)\ \widehat{}\widehat{}\ n)\ v) \longrightarrow \nu_b\text{-}opt$
⟨*proof*⟩

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

lemma *dist-*U_b*-opt*: $dist\ (U_b\ m\ v)\ \nu_b\text{-}opt \le l\widehat{}m * dist\ v\ \nu_b\text{-}opt$
⟨*proof*⟩

3.3 Expressing a Single Step of Modified Policy Iteration

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

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

lemma *W-eq-L-iter*:
assumes ν*-improving* $v\ d$
shows $W\ d\ m\ v = (L\ d\widehat{}\widehat{}m)\ v$
⟨*proof*⟩

lemma *W-le-*U_b:
assumes $v \le u$ ν*-improving* $v\ d$
shows $W\ d\ m\ v \le U_b\ m\ u$
⟨*proof*⟩

lemma *W-ge-*$\mathcal{L}_b$:
assumes $v \le u\ 0 \le B_b\ u$ ν*-improving* $u\ d'$

shows $\mathcal{L}_b\ v \le W\ d'\ (Suc\ m)\ u$
$\langle proof \rangle$

lemma B_b-*le*:
assumes ν-*improving v d*
shows $B_b\ v + (l *_R \mathcal{P}_1\ d - \textit{id-blinfun})\ (u - v) \le B_b\ u$
$\langle proof \rangle$

lemma $\mathcal{L}_b$-*W-ge*:
assumes $u \le \mathcal{L}_b\ u$ ν-*improving u d*
shows $W\ d\ m\ u \le \mathcal{L}_b\ (W\ d\ m\ u)$
$\langle proof \rangle$

3.4 Computing the Bellman Operator over Multiple Steps

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

lemma *sum-telescope'*: $(\sum i{\le}k.\ f\ (Suc\ i) - f\ i\ ) = f\ (Suc\ k) - (f\ 0$ $::\ 'c\ ::\ \textit{ab-group-add})$
$\langle proof \rangle$

lemma *L-pow-eq*:
assumes ν-*improving v (mk-dec-det d)*
shows *L-pow v d m* $= v + (\sum i \le m.\ ((l *_R \mathcal{P}_1\ (\textit{mk-dec-det}\ d))\hat{}\hat{}i))$ $(B_b\ v)$
$\langle proof \rangle$

lemma *L-pow-eq-W*:
assumes $d \in D_D$
shows *L-pow v (policy-improvement d v) m = W (mk-dec-det (policy-improvement d v)) (Suc m) v*
$\langle proof \rangle$

lemma *L-pow*-$\mathcal{L}_b$-*mono-inv*:
assumes $d \in D_D$ $v \le \mathcal{L}_b\ v$
shows *L-pow v (policy-improvement d v) m* $\le \mathcal{L}_b$ *(L-pow v (policy-improvement d v) m)*
$\langle proof \rangle$

3.5 The Modified Policy Iteration Algorithm

context
fixes $d0 :: 's \Rightarrow 'a$
fixes $v0 :: 's \Rightarrow_b real$
fixes $m :: nat \Rightarrow ('s \Rightarrow_b real) \Rightarrow nat$

assumes *d0*: $d0 \in D_D$
begin

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

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

definition $mpi\text{-}val\ n = snd\ (mpi\ n)$
definition $mpi\text{-}pol\ n = fst\ (mpi\ n)$

lemma *mpi-pol-zero*[*simp*]: $mpi\text{-}pol\ 0 = policy\text{-}improvement\ d0\ v0$
⟨*proof*⟩

lemma *mpi-pol-Suc*: $mpi\text{-}pol\ (Suc\ n) = policy\text{-}improvement\ (mpi\text{-}pol\ n)\ (mpi\text{-}val\ (Suc\ n))$
⟨*proof*⟩

lemma *mpi-pol-is-dec-det*: $mpi\text{-}pol\ n \in D_D$
⟨*proof*⟩

lemma ν*-improving-mpi-pol*: $\nu\text{-}improving\ (mpi\text{-}val\ n)\ (mk\text{-}dec\text{-}det\ (mpi\text{-}pol\ n))$
⟨*proof*⟩

lemma *mpi-val-zero*[*simp*]: $mpi\text{-}val\ 0 = v0$
⟨*proof*⟩

lemma *mpi-val-Suc*: $mpi\text{-}val\ (Suc\ n) = L\text{-}pow\ (mpi\text{-}val\ n)\ (mpi\text{-}pol\ n)\ (m\ n\ (mpi\text{-}val\ n))$
⟨*proof*⟩

lemma *mpi-val-eq*: $mpi\text{-}val\ (Suc\ n) =$
$mpi\text{-}val\ n + (\sum i \le m\ n\ (mpi\text{-}val\ n).\ (l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ (mpi\text{-}pol\ n)))\ \widehat{\ }\widehat{\ }\ i)\ (B_b\ (mpi\text{-}val\ n))$
⟨*proof*⟩

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 $mpi\text{-}val\ (Suc\ n) = \mathcal{L}_b\ (mpi\text{-}val\ n)$
⟨*proof*⟩

3.6 Convergence Proof

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

fun w **where**
$w\ 0 = v0\ |$
$w\ (Suc\ n) = U_b\ (Suc\ (m\ n\ (mpi\text{-}val\ n)))\ (w\ n)$

lemma $dist\text{-}\nu_b\text{-}opt$: $dist\ (w\ (Suc\ n))\ \nu_b\text{-}opt \leq l * dist\ (w\ n)\ \nu_b\text{-}opt$
⟨*proof*⟩

lemma $dist\text{-}\nu_b\text{-}opt\text{-}n$: $dist\ (w\ n)\ \nu_b\text{-}opt \leq l\hat{\ }n * dist\ v0\ \nu_b\text{-}opt$
⟨*proof*⟩

lemma $w\text{-}conv$: $w \longlongrightarrow \nu_b\text{-}opt$
⟨*proof*⟩

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 $mpi\text{-}conv$:
assumes $v0 \leq \mathcal{L}_b\ v0$
shows $mpi\text{-}val \longrightarrow \nu_b\text{-}opt$ **and** $\bigwedge n.\ mpi\text{-}val\ n \leq mpi\text{-}val\ (Suc\ n)$
⟨*proof*⟩

3.7 ϵ-Optimality

This gives an upper bound on the error of MPI.

lemma $mpi\text{-}pol\text{-}eps\text{-}opt$:
assumes $2 * l * dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) < eps * (1 - l)$
$eps > 0$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (mpi\text{-}pol\ n)))\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) \leq eps\ /\ 2$
⟨*proof*⟩

lemma $mpi\text{-}pol\text{-}opt$:
assumes $2 * l * dist\ (mpi\text{-}val\ n)\ (\mathcal{L}_b\ (mpi\text{-}val\ n)) < eps * (1 - l)$
$eps > 0$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (mpi\text{-}pol\ n)))\ (\nu_b\text{-}opt) < eps$
⟨*proof*⟩

lemma $mpi\text{-}val\text{-}term\text{-}ex$:
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*⟩
end

3.8 Unbounded MPI

context
fixes $eps\ \delta :: real$ **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 (*policy-improvement d v*, *v*)
  else mpi-algo (*policy-improvement d v*) (*L-pow v* (*policy-improvement d v*) (*m 0 v*)) (*λn. m* (*Suc n*)))
  ⟨*proof*⟩

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

fun *mpi′* **where**
  mpi′ d v 0 m = (*policy-improvement d v*, *v*) |
  mpi′ d v (*Suc n*) *m* = (
  let d′ = *policy-improvement d v*; *v′* = *L-pow v d′* (*m 0 v*) *in mpi′ d′ v′ n* (*λn. m* (*Suc n*)))

lemma *mpi-Suc′*:
  assumes $d \in D_D$
  shows *mpi d v m* (*Suc n*) = *mpi* (*policy-improvement d v*) (*L-pow v* (*policy-improvement d v*) (*m 0 v*)) (*λa. m* (*Suc a*)) *n*
  ⟨*proof*⟩

lemma
  assumes $d \in D_D$
  shows *mpi d v m n* = *mpi′ d v n m*
  ⟨*proof*⟩

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*⟩

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

lemma *mpi-algo-def′*:
  assumes $d \in D_D$ $v \le \mathcal{L}_b\ v$ *eps* > *0*
  shows *mpi-algo d v m* = *mpi-alg-rec d v m*
  ⟨*proof*⟩

lemma *mpi-algo-eq-mpi*:
  assumes $d \in D_D$ $v \le \mathcal{L}_b\ v$ *eps* > *0*
  shows *mpi-algo d v m* = *mpi d v m* (*LEAST n. 2 ∗ l ∗ dist* (*mpi-val d v m n*) ($\mathcal{L}_b$ (*mpi-val d v m n*)) < *eps* ∗ (*1* − *l*))
⟨*proof*⟩

lemma *mpi-algo-opt*:
  assumes $v0 \leq \mathcal{L}_b\ v0\ eps > 0\ d \in D_D$
  shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (fst\ (mpi\text{-}algo\ d\ v0\ m))))\ \nu_b\text{-}opt < eps$
$\langle proof \rangle$

end

3.9 Initial Value Estimate *v0-mpi*

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

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

lift-definition $v0\text{-}mpi_b :: 's \Rightarrow_b real$ **is** *v0-mpi*
  $\langle proof \rangle$

lemma $v0\text{-}mpi_b\text{-}le\text{-}\mathcal{L}_b$: $v0\text{-}mpi_b \leq \mathcal{L}_b\ v0\text{-}mpi_b$
$\langle proof \rangle$

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

definition $mpi\text{-}user\ eps\ m = ($
  $if\ eps \leq 0\ then\ undefined\ else\ mpi\text{-}algo\ eps\ (\lambda x.\ arb\text{-}act\ (A\ x))\ v0\text{-}mpi_b\ m)$

lemma *mpi-user-eq*:
  assumes $eps > 0$
  shows $mpi\text{-}user\ eps = mpi\text{-}alg\text{-}rec\ eps\ (\lambda x.\ arb\text{-}act\ (A\ x))\ v0\text{-}mpi_b$
  $\langle proof \rangle$

lemma *mpi-user-opt*:
  assumes $eps > 0$
  shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (fst\ (mpi\text{-}user\ eps\ n))))\ \nu_b\text{-}opt < eps$
  $\langle proof \rangle$

end

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

4 Matrices

proposition *scalar-matrix-assoc'*:
fixes C :: ('*b*::*real-algebra-1*)$^\frown$'*m*$^\frown$'*n*
shows $k *_R (C ** D) = C ** (k *_R D)$
⟨*proof*⟩

4.1 Nonnegative Matrices

lemma *nonneg-matrix-nonneg* [*dest*]: $0 \leq m \Longrightarrow 0 \leq m \$ i \$ j$
⟨*proof*⟩

lemma *matrix-mult-mono*:
assumes $0 \leq E$ $0 \leq C$ (E :: *real*$^\frown$'*c*$^\frown$'*c*) $\leq B$ $C \leq D$
shows $E ** C \leq B ** D$
⟨*proof*⟩

lemma *nonneg-matrix-mult*: $0 \leq$ (C :: ('*b*::{*field*, *ordered-ring*})$^\frown$-$^\frown$-) $\Longrightarrow 0 \leq D \Longrightarrow 0 \leq C ** D$
⟨*proof*⟩

lemma *zero-le-mat-iff* [*simp*]: $0 \leq$ *mat* (x :: '*c* :: {*zero*, *order*}) $\longleftrightarrow$ $0 \leq x$
⟨*proof*⟩

lemma *nonneg-mat-ge-zero*: $0 \leq Q \Longrightarrow 0 \leq v \Longrightarrow 0 \leq Q *v$ v (v :: *real*$^\frown$'*c*)
⟨*proof*⟩

lemma *nonneg-mat-mono*: $0 \leq Q \Longrightarrow u \leq v \Longrightarrow Q *v$ $u \leq Q *v$ v (v :: *real*$^\frown$'*c*)
⟨*proof*⟩

lemma *nonneg-mult-imp-nonneg-mat*:
assumes $\bigwedge v.\ v \geq 0 \Longrightarrow X *v\ v \geq 0$
shows $X \geq$ (*0* :: *real* $^\frown$ - $^\frown$-)
⟨*proof*⟩

lemma *nonneg-mat-iff*:
($X \geq$ (*0* :: *real* $^\frown$ - $^\frown$-)) $\longleftrightarrow$ ($\forall v.\ v \geq 0 \longrightarrow X *v\ v \geq 0$)
⟨*proof*⟩

lemma *mat-le-iff*: ($X \leq Y$) $\longleftrightarrow$ ($\forall x{\geq}0.$ (X::*real*$^\frown$-$^\frown$-) $*v\ x \leq Y *v\ x$)
⟨*proof*⟩

4.2 Matrix Powers

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

matpow-Suc: $matpow\ A\ (Suc\ n) = (matpow\ A\ n) ** A$

lemma *nonneg-matpow*: $0 \leq X \Longrightarrow 0 \leq matpow\ (X :: real\ \hat{}\ \text{-}\ \hat{}\ \text{-})\ i$
⟨*proof*⟩

lemma *matpow-mono*: $0 \leq C \Longrightarrow C \leq D \Longrightarrow matpow\ (C :: real\hat{}\text{-}\hat{}\text{-})\ n \leq matpow\ D\ n$
⟨*proof*⟩

lemma *matpow-scaleR*: $matpow\ (c *_R (X :: 'b :: real\text{-}algebra\text{-}1\hat{}\text{-}\hat{}\text{-}))\ n = (c\hat{}n) *_R (matpow\ X)\ n$
⟨*proof*⟩

lemma *matrix-vector-mult-code'*: $(X *v\ x)\ \$\ i = (\sum j{\in}UNIV.\ X\ \$\ i\ \$\ j * x\ \$\ j)$
⟨*proof*⟩

lemma *matrix-vector-mult-mono*: $(0{::}real\hat{}\text{-}\hat{}\text{-}) \leq X \Longrightarrow 0 \leq v \Longrightarrow X \leq Y \Longrightarrow X *v\ v \leq Y *v\ v$
⟨*proof*⟩

4.3 Triangular Matrices

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

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

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

lemma *stlI*: *strict-lower-triangular-mat* $X \Longrightarrow$ *lower-triangular-mat* X
⟨*proof*⟩

lemma *lower-triangular-mat-mat*: *lower-triangular-mat* $(mat\ x)$
⟨*proof*⟩

lemma *lower-triangular-mult*:
assumes *lower-triangular-mat* X *lower-triangular-mat* Y
shows *lower-triangular-mat* $(X ** Y)$
⟨*proof*⟩

lemma *lower-triangular-pow*:
assumes *lower-triangular-mat* X
shows *lower-triangular-mat* $(matpow\ X\ i)$
⟨*proof*⟩

lemma *lower-triangular-suminf*:
assumes $\bigwedge i.$ *lower-triangular-mat* $(f\ i)$ *summable* $(f :: nat \Rightarrow$ $'b::real\text{-}normed\text{-}vector\hat{}\text{-}\hat{}\text{-})$
shows *lower-triangular-mat* $(\sum i.\ f\ i)$
⟨*proof*⟩

lemma *lower-triangular-pow-eq*:
assumes *lower-triangular-mat* X *lower-triangular-mat* Y $\bigwedge s'.\ s' \leq s \Longrightarrow row\ s'\ X = row\ s'\ Y$ $s' \leq s$
shows $row\ s'\ (matpow\ X\ i) = row\ s'\ (matpow\ Y\ i)$
⟨*proof*⟩

lemma *lower-triangular-mat-mult*:
assumes *lower-triangular-mat* M $\bigwedge i.\ i \leq j \Longrightarrow v\ \$\ i = v'\ \$\ i$
shows $(M *v\ v)\ \$\ j = (M *v\ v')\ \$\ j$
⟨*proof*⟩

4.4 Inverses

lemma *matrix-inv*:
assumes *invertible* M
shows *matrix-inv-left*: $matrix\text{-}inv\ M ** M = mat\ 1$
and *matrix-inv-right*: $M ** matrix\text{-}inv\ M = mat\ 1$
⟨*proof*⟩

lemma *matrix-inv-unique*:
fixes $A::'a::\{semiring\text{-}1\}\hat{}'n\hat{}'n$
assumes *AB*: $A ** B = mat\ 1$ **and** *BA*: $B ** A = mat\ 1$
shows $matrix\text{-}inv\ A = B$
⟨*proof*⟩

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

5 Bounded Linear Functions and Matrices

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

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

lemma *plus-vec-eq*: $(\chi\ i.\ f\ i\ + g\ i) = (\chi\ i.\ f\ i) + (\chi\ i.\ g\ i)$

$\langle proof \rangle$

lemma *matrix-to-blinfun-mult*: *matrix-to-blinfun* m (v :: $'c$::*finite* $\Rightarrow_b$ *real*) i = (m $*v$ (χ i. v i)) \$ i
$\langle proof \rangle$

lemma *blinfun-to-matrix-mult*: (*blinfun-to-matrix* f $*v$ (χ i. *apply-bfun* v i)) \$ i = f v i
$\langle proof \rangle$

lemma *blinfun-to-matrix-mult'*: (*blinfun-to-matrix* f $*v$ v) \$ i = f (*Bfun* (λi. v \$ i)) i
$\langle proof \rangle$

lemma *blinfun-to-matrix-mult''*: (*blinfun-to-matrix* f $*v$ v) = (χ i. f (*Bfun* (λi. v \$ i)) i)
$\langle proof \rangle$

lemma *matrix-to-blinfun-inv*: *matrix-to-blinfun* (*blinfun-to-matrix* f) = f
$\langle proof \rangle$

lemma *blinfun-to-matrix-add*: *blinfun-to-matrix* (f + g) = *blinfun-to-matrix* f + *blinfun-to-matrix* g
$\langle proof \rangle$

lemma *blinfun-to-matrix-diff*: *blinfun-to-matrix* (f − g) = *blinfun-to-matrix* f − *blinfun-to-matrix* g
$\langle proof \rangle$

lemma *blinfun-to-matrix-scaleR*: *blinfun-to-matrix* (c $*_R$ f) = c $*_R$ *blinfun-to-matrix* f
$\langle proof \rangle$

lemma *matrix-to-blinfun-add*:
matrix-to-blinfun ((f :: *real*^-^-) + g) = *matrix-to-blinfun* f + *matrix-to-blinfun* g
$\langle proof \rangle$

lemma *matrix-to-blinfun-diff*:
matrix-to-blinfun ((f :: *real*^-^-) − g) = *matrix-to-blinfun* f − *matrix-to-blinfun* g
$\langle proof \rangle$

lemma *matrix-to-blinfun-scaleR*:
matrix-to-blinfun (c $*_R$ (f :: *real*^-^-)) = c $*_R$ *matrix-to-blinfun* f
$\langle proof \rangle$

lemma *matrix-to-blinfun-comp*: *matrix-to-blinfun* ((m:: *real*^-^-) $**$

n) = (*matrix-to-blinfun m*) o_L (*matrix-to-blinfun n*)
⟨*proof*⟩

lemma *blinfun-to-matrix-comp*: *blinfun-to-matrix* (f o_L g) = (*blinfun-to-matrix* f) $**$ (*blinfun-to-matrix g*)
⟨*proof*⟩

lemma *blinfun-to-matrix-id*: *blinfun-to-matrix id-blinfun* = *mat 1*
⟨*proof*⟩

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

lemma *matrix-to-blinfun-inv*$_L$:
assumes *invertible m*
shows *matrix-to-blinfun* (*matrix-inv* (m :: *real*^-^-)) = *inv*$_L$ (*matrix-to-blinfun* m)
invertible$_L$ (*matrix-to-blinfun m*)
⟨*proof*⟩

lemma *blinfun-to-matrix-inverse*:
assumes *invertible*$_L$ X
shows *invertible* (*blinfun-to-matrix* (X :: ($'b$::*finite* $\Rightarrow_b$ *real*) $\Rightarrow_L$ $'c$::*finite* $\Rightarrow_b$ *real*))
blinfun-to-matrix (*inv*$_L$ X) = *matrix-inv* (*blinfun-to-matrix* X)
⟨*proof*⟩

lemma *blinfun-to-matrix-inv*[*simp*]: *blinfun-to-matrix* (*matrix-to-blinfun* f) = f
⟨*proof*⟩

lemma *invertible-invertible*$_L$*-I*: *invertible* (*blinfun-to-matrix* f) $\Longrightarrow$ *invertible*$_L$ f
invertible$_L$ (*matrix-to-blinfun* X) $\Longrightarrow$ *invertible* (X :: *real*^-^-)
⟨*proof*⟩

lemma *bounded-linear-blinfun-to-matrix*: *bounded-linear* (*blinfun-to-matrix* :: ($'a$ $\Rightarrow_b$ *real*) $\Rightarrow_L$ ($'b$ $\Rightarrow_b$ *real*) $\Rightarrow$ *real*^$'a$^$'b$)
⟨*proof*⟩

lemma *summable-blinfun-to-matrix*:
assumes *summable* (f :: *nat* $\Rightarrow$ ($'c$::*finite* $\Rightarrow_b$ -) $\Rightarrow_L$ ($'c$ $\Rightarrow_b$ -))
shows *summable* (λi. *blinfun-to-matrix* (f i))
⟨*proof*⟩

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

lemma *nonneg-blinfun-mono*: *nonneg-blinfun* $Q \Longrightarrow u \leq v \Longrightarrow Q\ u \leq Q\ v$
⟨*proof*⟩

lemma *nonneg-blinfun-nonneg*: *nonneg-blinfun* $Q \Longrightarrow 0 \leq v \Longrightarrow 0 \leq Q\ v$
⟨*proof*⟩

lemma *nonneg-id-blinfun*: *nonneg-blinfun id-blinfun*
⟨*proof*⟩

lemma *norm-nonneg-blinfun-one*:
assumes $0 \leq$ *blinfun-to-matrix* X
shows *norm* $X =$ *norm* (*blinfun-apply* X *1*)
⟨*proof*⟩

lemma *matrix-le-norm-mono*:
assumes $0 \leq$ (*blinfun-to-matrix* C)
and (*blinfun-to-matrix* C) $\leq$ (*blinfun-to-matrix* D)
shows *norm* $C \leq$ *norm* D
⟨*proof*⟩

lemma *blinfun-to-matrix-matpow*: *blinfun-to-matrix* (X ^^ i) = *matpow* (*blinfun-to-matrix* X) i
⟨*proof*⟩

lemma *nonneg-blinfun-iff*: *nonneg-blinfun* $X \longleftrightarrow (\forall v \geq 0.\ X\ v \geq 0)$
⟨*proof*⟩

lemma *blinfun-apply-mono*: (*0*::*real*^-^-) $\leq$ *blinfun-to-matrix* $X \Longrightarrow 0 \leq v \Longrightarrow$ *blinfun-to-matrix* $X \leq$ *blinfun-to-matrix* $Y \Longrightarrow X\ v \leq Y\ v$
⟨*proof*⟩

end

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

6 Value Iteration using Splitting Methods

6.1 Regular Splittings for Matrices and Bounded Linear Functions

definition *is-splitting-mat* $X\ Q\ R \longleftrightarrow$

$X = Q - R \wedge \mathit{invertible}\ Q \wedge 0 \leq \mathit{matrix\text{-}inv}\ Q \wedge 0 \leq R$

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

lemma *is-splitting-blin-def'*: *is-splitting-blin* $X\ Q\ R \longleftrightarrow$
$X = Q - R \wedge \mathit{invertible}_L\ Q \wedge \mathit{nonneg\text{-}blinfun}\ (\mathit{inv}_L\ Q) \wedge \mathit{nonneg\text{-}blinfun}\ R$
⟨*proof*⟩

lemma *is-splitting-blinD*[*dest*]:
assumes *is-splitting-blin* $X\ Q\ R$
shows $X = Q - R$ $\mathit{invertible}_L\ Q$ $\mathit{nonneg\text{-}blinfun}\ (\mathit{inv}_L\ Q)$ $\mathit{nonneg\text{-}blinfun}\ R$
⟨*proof*⟩

6.2 Splitting Methods for MDPs

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

lemma *inv-Q-mono*: $d \in D_D \Longrightarrow u \leq v \Longrightarrow (\mathit{inv}_L\ (Q\ d))\ u \leq (\mathit{inv}_L\ (Q\ d))\ v$
⟨*proof*⟩

lemma *splitting-eq*: $d \in D_D \Longrightarrow Q\ d - R\ d = (\mathit{id\text{-}blinfun} - l *_R \mathcal{P}_1\ (\mathit{mk\text{-}dec\text{-}det}\ d))$
⟨*proof*⟩

lemma *Q-nonneg*: $d \in D_D \Longrightarrow 0 \leq v \Longrightarrow 0 \leq \mathit{inv}_L\ (Q\ d)\ v$
⟨*proof*⟩

lemma *Q-invertible*: $d \in D_D \Longrightarrow \mathit{invertible}_L\ (Q\ d)$
⟨*proof*⟩

lemma *R-nonneg*: $d \in D_D \Longrightarrow 0 \leq v \Longrightarrow 0 \leq R\ d\ v$
⟨*proof*⟩

lemma *R-mono*: $d \in D_D \Longrightarrow u \leq v \Longrightarrow (R\ d)\ u \leq (R\ d)\ v$
⟨*proof*⟩

lemma *QR-nonneg*: $d \in D_D \Longrightarrow 0 \leq v \Longrightarrow 0 \leq (inv_L\ (Q\ d)\ o_L\ R\ d)\ v$
⟨*proof*⟩

lemma *QR-mono*: $d \in D_D \Longrightarrow u \leq v \Longrightarrow (inv_L\ (Q\ d)\ o_L\ R\ d)\ u \leq (inv_L\ (Q\ d)\ o_L\ R\ d)\ v$
⟨*proof*⟩

lemma *norm-QR-less-one*: $d \in D_D \Longrightarrow norm\ (inv_L\ (Q\ d)\ o_L\ R\ d) < 1$
⟨*proof*⟩

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

6.3 Discount Factor *QR-disc*

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

lemma *QR-le-QR-disc*: $d \in D_D \Longrightarrow norm\ (inv_L\ (Q\ d)\ o_L\ (R\ d)) \leq QR\text{-}disc$
⟨*proof*⟩

lemma *a-nonneg*: $0 \leq QR\text{-}disc$
⟨*proof*⟩

6.4 Bellman-Operator

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

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

lemma *$\mathcal{L}$-split-bfun-aux*:
assumes $d \in D_D$
shows $norm\ (L\text{-}split\ d\ v) \leq (\bigsqcup d \in D_D.\ norm\ (inv_L\ (Q\ d))) * r_M + norm\ v$
⟨*proof*⟩

lift-definition $\mathcal{L}_b\text{-}split :: ('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow_b real)$ **is** *$\mathcal{L}$-split*
⟨*proof*⟩

lemma *$\mathcal{L}_b$-split-def'*: $\mathcal{L}_b\text{-}split\ v\ s = (\bigsqcup d{\in}D_D.\ L\text{-}split\ d\ v\ s)$
⟨*proof*⟩

lemma $\mathcal{L}_b$-*split-contraction*: *dist* ($\mathcal{L}_b$-*split v*) ($\mathcal{L}_b$-*split u*) $\leq$ *QR-disc* $*$ *dist v u*
⟨*proof*⟩

lemma $\mathcal{L}_b$-*lim*:
  $\exists!v.$ $\mathcal{L}_b$-*split v* = *v*
  ($\lambda n.$ ($\mathcal{L}_b$-*split* $\widehat{\ }\widehat{\ }$ *n*) *v*) $\longrightarrow$ (*THE v.* $\mathcal{L}_b$-*split v* = *v*)
  ⟨*proof*⟩

lemma $\mathcal{L}_b$-*split-tendsto-opt*: ($\lambda n.$ ($\mathcal{L}_b$-*split* $\widehat{\ }\widehat{\ }$ *n*) *v*) $\longrightarrow$ ν_b-*opt*
⟨*proof*⟩

lemma $\mathcal{L}_b$-*split-fix*[*simp*]: $\mathcal{L}_b$-*split* ν_b-*opt* = ν_b-*opt*
  ⟨*proof*⟩

lemma *dist*-$\mathcal{L}_b$-*split-opt-eps*:
   assumes *eps* > *0 2* $*$ *QR-disc* $*$ *dist v* ($\mathcal{L}_b$-*split v*) < *eps* $*$ (*1*−*QR-disc*)
  shows *dist* ($\mathcal{L}_b$-*split v*) ν_b-*opt* < *eps* / *2*
⟨*proof*⟩

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*⟩

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*⟩

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* ($\lambda d.$ *L-split d* ($\mathcal{L}_b$-*split v*) *s*) ($\lambda d.$ $d \in D_D$) *d*
  shows *dist* (ν_b (*mk-stationary-det d*)) ν_b-*opt* < *eps*
⟨*proof*⟩
end

context *MDP-ord* **begin**
lemma *inv-one-sub-Q'*:
  fixes *Q* :: *'c* :: *banach* $\Rightarrow_L$ *'c*
  assumes *onorm-le*: *norm* (*id-blinfun* − *Q*) < *1*
  shows inv_L *Q* = ($\sum$ *i.* (*id-blinfun* − *Q*)$\widehat{\ }\widehat{\ }$*i*)
  ⟨*proof*⟩

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

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

6.5 Gauss Seidel Splitting

6.5.1 Definition of Upper and Lower Triangular Matrices

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

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

lemma *P-dec-elem*: *P-dec d \$ i \$ j* = *pmf* (*K* (*i*, *d i*)) *j*
  ⟨*proof*⟩

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

lemma *nonneg-P-dec*: *0* $\leq$ *P-dec d*
  ⟨*proof*⟩

lemma *nonneg-P-upper*: *0* $\leq$ *P-upper d*
  ⟨*proof*⟩

lemma *nonneg-P-lower*: *0* $\leq$ *P-lower d*
  ⟨*proof*⟩

lemma *nonneg-*$\mathcal{P}_L$: *nonneg-blinfun* ($\mathcal{P}_L$ *d*)
  ⟨*proof*⟩

lemma *nonneg-*$\mathcal{P}_1$: *nonneg-blinfun* ($\mathcal{P}_1$ *d*)
  ⟨*proof*⟩

lemma *norm-*$\mathcal{P}_L$*-le*: *norm* ($\mathcal{P}_L$ *d*) $\leq$ *norm* ($\mathcal{P}_1$ (*mk-dec-det d*))
  ⟨*proof*⟩

lemma *norm-*$\mathcal{P}_L$*-le-one*: *norm* ($\mathcal{P}_L$ *d*) $\leq$ *1*
  ⟨*proof*⟩

lemma *norm-*$\mathcal{P}_L$*-less-one*: *norm* (*l* $*_R$ $\mathcal{P}_L$ *d*) $<$ *1*

$\langle proof \rangle$

lemma $\mathcal{P}_L$*-le-*$\mathcal{P}_1$: $0 \le v \Longrightarrow \mathcal{P}_L\ d\ v \le \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d)\ v$
$\langle proof \rangle$

lemma $\mathcal{P}_U$*-le-*$\mathcal{P}_1$: $0 \le v \Longrightarrow \mathcal{P}_U\ d\ v \le \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d)\ v$
$\langle proof \rangle$

lemma *row-P-upper-indep*: $d\ s = d'\ s \Longrightarrow row\ s\ (P\text{-}upper\ d) = row\ s\ (P\text{-}upper\ d')$
$\langle proof \rangle$

lemma *row-P-lower-indep*: $d\ s = d'\ s \Longrightarrow row\ s\ (P\text{-}lower\ d) = row\ s\ (P\text{-}lower\ d')$
$\langle proof \rangle$

lemma *triangular-mat-P-upper*: *upper-triangular-mat* (*P-upper d*)
$\langle proof \rangle$

lemma *slt-P-lower*: *strict-lower-triangular-mat* (*P-lower d*)
$\langle proof \rangle$

lemma *lt-P-lower*: *lower-triangular-mat* (*P-lower d*)
$\langle proof \rangle$

6.5.2 Gauss Seidel is a Regular Splitting

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

lemma *splitting-gauss*: $is\text{-}splitting\text{-}blin\ (id\text{-}blinfun - l *_R \mathcal{P}_1\ (mk\text{-}dec\text{-}det\ d))\ (Q\text{-}GS\ d)\ (R\text{-}GS\ d)$
$\langle proof \rangle$

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

abbreviation $Q\text{-}mat\ d \equiv blinfun\text{-}to\text{-}matrix\ (Q\text{-}GS\ d)$
abbreviation $R\text{-}mat\ d \equiv blinfun\text{-}to\text{-}matrix\ (R\text{-}GS\ d)$

lemma *Q-mat-def*: $Q\text{-}mat\ d = mat\ 1 - l *_R P\text{-}lower\ d$
$\langle proof \rangle$

lemma *R-mat-def*: $R\text{-}mat\ d = l *_R P\text{-}upper\ d$
$\langle proof \rangle$

lemma *triangular-mat-R*: *upper-triangular-mat* (*R-mat d*)
$\langle proof \rangle$

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

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

lemma *inv-Q-mat-suminf*: *matrix-inv* (*Q-mat d*) = ($\sum$ *k*. (*matpow* (*l* $*_R$ (*P-lower d*)) *k*))
⟨*proof*⟩

lemma *lt-Q-inv*: *lower-triangular-mat* (*matrix-inv* (*Q-mat d*))
⟨*proof*⟩

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

lemma *inv-Q-mat-indep*:
assumes $\bigwedge i.\ i \le s \Longrightarrow d\ i = d'\ i$ $i \le s$
shows *row i* (*matrix-inv* (*Q-mat d*)) = *row i* (*matrix-inv* (*Q-mat d'*))
⟨*proof*⟩

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

lemma *GS-indep-high-states*:
assumes $\bigwedge s'.\ s' \le s \Longrightarrow d\ s' = d'\ s'$
shows *GS-inv d v* \$ *s* = *GS-inv d' v* \$ *s*
⟨*proof*⟩

This recursive definition mimics the computation of the GS iteration.

lemma *GS-inv-rec*: *GS-inv d v* = *r-vec d* + *l* $*_R$ (*P-upper d* $*v$ *v* + *P-lower d* $*v$ (*GS-inv d v*))
⟨*proof*⟩

lemma *is-am-GS-inv-extend*:
assumes $\bigwedge s.\ s < k \Longrightarrow$ *is-arg-max* (λd. *GS-inv d v* \$ *s*) ($\lambda d.\ d \in D_D$) *d*
and *is-arg-max* (λa. *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* (λd. *GS-inv d v* \$ *s*) ($\lambda d.\ d \in D_D$) (*d* (*k* := *a*))
⟨*proof*⟩

lemma *is-arg-max-GS-le*:
$\exists d.\ \forall s \le k.$ *is-arg-max* (λd. *GS-inv d v* \$ *s*) ($\lambda d.\ d \in D_D$) *d*
⟨*proof*⟩

lemma *ex-is-arg-max-GS*:

$\exists\, d.\ \forall\, s.\ \textit{is-arg-max}\ (\lambda d.\ \textit{GS-inv}\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
⟨*proof*⟩

function *GS-rec-fun* **where**
$\textit{GS-rec-fun}\ v\ s = (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l * ($
$(\sum s' < s.\ \textit{pmf}\ (K\ (s,a))\ s' * (\textit{GS-rec-fun}\ v\ s')) +$
$(\sum s' \in \{s'.\ s \leq s'\}.\ \textit{pmf}\ (K\ (s,a))\ s' * v\ s')))$
⟨*proof*⟩
termination
⟨*proof*⟩

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

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

lemma *GS-rec-fun-elem*: $\textit{GS-rec-fun}\ v\ s = (\bigsqcup a \in A\ s.\ \textit{GS-rec-elem}\ v\ s\ a)$
⟨*proof*⟩

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

lemma *GS-rec-def′*: $\textit{GS-rec}\ v\ \$\ s = (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l * ($
$(\sum s' < s.\ \textit{pmf}\ (K\ (s,a))\ s' * (\textit{GS-rec}\ v\ \$\ s')) +$
$(\sum s' \in \{s'.\ s \leq s'\}.\ \textit{pmf}\ (K\ (s,a))\ s' * v\ \$\ s')))$
⟨*proof*⟩

lemma *GS-rec-eq*: $\textit{GS-rec}\ v\ \$\ s = (\bigsqcup a \in A\ s.\ r\ (s,\ a) + l * ($
$(\textit{P-lower}\ (d(s := a)) *v\ (\textit{GS-rec}\ v))\ \$\ s + (\textit{P-upper}\ (d(s := a)) *v\ v)\ \$\ s))$
⟨*proof*⟩
definition $\textit{GS-rec-step}\ d\ v \equiv \textit{r-vec}\ d + l *_R (\textit{P-lower}\ d *v\ \textit{GS-rec}\ v + \textit{P-upper}\ d *v\ v)$

lemma *GS-rec-eq′*: $\textit{GS-rec}\ v\ \$\ s = (\bigsqcup a \in A\ s.\ \textit{GS-rec-step}\ (d(s{:=}\ a))\ v\ \$\ s)$
⟨*proof*⟩

lemma *GS-rec-eq-vec*:
$\textit{GS-rec}\ v\ \$\ s = (\bigsqcup d{\in}D_D.\ \textit{GS-rec-step}\ d\ v\ \$\ s)$
⟨*proof*⟩

lift-definition $\textit{GS-rec-fun}_b :: ('s \Rightarrow_b \textit{real}) \Rightarrow ('s \Rightarrow_b \textit{real})$ **is** *GS-rec-fun*
⟨*proof*⟩

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

$(\sum s' \in \{s'.\ s \le s'\}.\ pmf\ (K\ (s,a))\ s' * v\ s'))$

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

lemma *GS-rec-fun-eq-GS-iter*:
assumes $\forall s' < s.\ v\text{-}next\ s' = GS\text{-}rec\text{-}fun\ v\ s'$ $\forall s' \in \{s'.\ s \le s'\}.$ $v\text{-}next\ s' = v\ s'$
shows $GS\text{-}rec\text{-}fun\ v\ s = GS\text{-}rec\text{-}iter\ v\text{-}next\ s$
⟨*proof*⟩

lemma *foldl-upd-notin*: $x \notin set\ X \Longrightarrow foldl\ (\lambda f\ y.\ f(y := g\ f\ y))\ c\ X$ $x = c\ x$
⟨*proof*⟩

lemma *foldl-upd-notin'*: $x \notin set\ Y \Longrightarrow foldl\ (\lambda f\ y.\ f(y := g\ f\ y))\ c$ $(X@Y)\ x = foldl\ (\lambda f\ y.\ f(y := g\ f\ y))\ c\ X\ x$
⟨*proof*⟩

lemma *sorted-list-of-set-split*:
assumes *finite X*
shows $sorted\text{-}list\text{-}of\text{-}set\ X = sorted\text{-}list\text{-}of\text{-}set\ \{x \in X.\ x < y\}$ @ $sorted\text{-}list\text{-}of\text{-}set\ \{x \in X.\ y \le x\}$
⟨*proof*⟩

lemma *sorted-list-of-set-split'*:
assumes *finite X*
shows $sorted\text{-}list\text{-}of\text{-}set\ X = sorted\text{-}list\text{-}of\text{-}set\ \{x \in X.\ x \le y\}$ @ $sorted\text{-}list\text{-}of\text{-}set\ \{x \in X.\ y < x\}$
⟨*proof*⟩

lemma *GS-rec-fun-code*: $GS\text{-}rec\text{-}fun\ v\ s = foldl\ (\lambda v\ s.\ v(s := GS\text{-}rec\text{-}iter$ $v\ s))\ v\ (sorted\text{-}list\text{-}of\text{-}set\ \{..s\})\ s$
⟨*proof*⟩

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

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

lemma *GS-rec-eq-elem*: $GS\text{-}rec\ v\ \$\ s = GS\text{-}rec\text{-}fun\ (vec\text{-}nth\ v)\ s$
⟨*proof*⟩

lemma *GS-rec-step-elem*: $GS\text{-}rec\text{-}step\ d\ v\ \$\ s = r\ (s,\ d\ s) + l * ((\sum s' < s.\ pmf\ (K\ (s,\ d\ s))\ s' * GS\text{-}rec\ v\ \$\ s') + (\sum s' \in \{s'.\ s \leq s'\}.\ pmf\ (K\ (s,\ d\ s))\ s' * v\ \$\ s'))$
⟨*proof*⟩

lemma *is-arg-max-GS-rec-step-act*:
assumes $d \in D_D$ $is\text{-}arg\text{-}max\ (\lambda a.\ GS\text{-}rec\text{-}step\ (d'(s := a))\ v\ \$\ s)\ (\lambda a.\ a \in A\ s)\ a$
shows $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (d(s := a))$
⟨*proof*⟩

lemma *is-arg-max-GS-rec-step-act′*:
assumes $d \in D_D$ $is\text{-}arg\text{-}max\ (\lambda a.\ GS\text{-}rec\text{-}step\ (d'(s := a))\ v\ \$\ s)\ (\lambda a.\ a \in A\ s)\ (d\ s)$
shows $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
⟨*proof*⟩

lemma
is-arg-max-GS-rec:
assumes $\bigwedge s.\ is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows $GS\text{-}rec\ v = GS\text{-}rec\text{-}step\ d\ v$
⟨*proof*⟩

lemma
is-arg-max-GS-rec′:
assumes $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows $GS\text{-}rec\ v\ \$\ s = GS\text{-}rec\text{-}step\ d\ v\ \$\ s$
⟨*proof*⟩

lemma
GS-rec-eq-GS-inv:
assumes $\bigwedge s.\ is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows $GS\text{-}rec\ v = GS\text{-}inv\ d\ v$
⟨*proof*⟩

lemma
GS-rec-step-eq-GS-inv:
assumes $\bigwedge s.\ is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows $GS\text{-}rec\text{-}step\ d\ v = GS\text{-}inv\ d\ v$
⟨*proof*⟩

lemma *strict-lower-triangular-mat-mult*:
assumes $strict\text{-}lower\text{-}triangular\text{-}mat\ M$ $\bigwedge i.\ i < j \Longrightarrow v\ \$\ i = v'\ \$\ i$
shows $(M *v\ v)\ \$\ j = (M *v\ v')\ \$\ j$
⟨*proof*⟩

lemma *Q-mat-invertible*: $invertible\ (Q\text{-}mat\ d)$
⟨*proof*⟩

lemma *GS-eq-GS-inv*:
assumes $\bigwedge s.\ s \leq k \Longrightarrow$ *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
assumes $s \leq k$
shows *GS-rec-step* $d\ v\ \$\ s =$ *GS-inv* $d\ v\ \$\ s$
⟨*proof*⟩

lemma *is-arg-max-GS-imp-splitting'*:
assumes $\bigwedge s.\ s \leq k \Longrightarrow$ *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
assumes $s \leq k$
shows *is-arg-max* $(\lambda d.$ *GS-inv* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
⟨*proof*⟩

lemma *is-am-GS-rec-step-indep*:
assumes $d\ s = d'\ s$
assumes *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows *GS-rec* $v\ \$\ s =$ *GS-rec-step* $d'\ v\ \$\ s$
⟨*proof*⟩

lemma *is-am-GS-rec-step-indep'*:
assumes $d\ s = d'\ s$
assumes *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows *GS-rec* $v\ \$\ s =$ *GS-rec-step* $d'\ v\ \$\ s$
⟨*proof*⟩

lemma *is-arg-max-GS-imp-splitting''*:
assumes $\bigwedge s.\ s \leq k \Longrightarrow$ *is-arg-max* $(\lambda d.$ *GS-inv* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
assumes $s \leq k$
shows *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d \wedge$ *GS-inv* $d\ v\ \$\ s =$ *GS-rec* $v\ \$\ s$
⟨*proof*⟩

lemma *is-arg-max-GS-imp-splitting'''*:
assumes $\bigwedge s.\ s \leq k \Longrightarrow$ *is-arg-max* $(\lambda d.$ *GS-inv* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
assumes $s \leq k$
shows *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
⟨*proof*⟩

lemma *is-arg-max-GS-imp-splitting*:
assumes $\bigwedge s.$ *is-arg-max* $(\lambda d.$ *GS-rec-step* $d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
shows *is-arg-max* $(\lambda d.$ *GS-inv* $d\ v\ \$\ k)\ (\lambda d.\ d \in D_D)\ d$
⟨*proof*⟩

lemma *is-arg-max-gs-iff*:
assumes $d \in D_D$

shows $(\forall s \le k.\ \textit{is-arg-max}\ (\lambda d.\ \textit{GS-inv}\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d)$
$\longleftrightarrow$
$(\forall s \le k.\ \textit{is-arg-max}\ (\lambda d.\ \textit{GS-rec-step}\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d)$
⟨*proof*⟩

lemma *GS-opt-indep-high*:
assumes $(\bigwedge s'.\ s' < s \Longrightarrow \textit{is-arg-max}\ (\lambda d.\ \textit{GS-rec-step}\ d\ v\ \$\ s')\ \textit{is-dec-det}\ d)\ s' < s\ a \in A\ s$
shows $\textit{is-arg-max}\ (\lambda d.\ \textit{GS-rec-step}\ d\ v\ \$\ s')\ \textit{is-dec-det}\ (d(s := a))$
⟨*proof*⟩

lemma *mult-mat-vec-nth*: $(X *v\ x)\ \$\ i = \textit{scalar-product}\ (\textit{row}\ i\ X)\ x$
⟨*proof*⟩

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

lemma *ex-GS-opt-le*:
shows $\exists d.\ (\forall s' \le s.\ \textit{is-arg-max}\ (\lambda d.\ \textit{GS-rec-step}\ d\ v\ \$\ s')\ (\lambda d.\ d \in D_D)\ d)$
⟨*proof*⟩

lemma *ex-GS-opt*:
shows $\exists d.\ \forall s.\ \textit{is-arg-max}\ (\lambda d.\ \textit{GS-rec-step}\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d$
⟨*proof*⟩

lemma *GS-rec-eq-GS-inv′*: $\textit{GS-rec}\ v\ \$\ s = (\bigsqcup d{\in}D_D.\ \textit{GS-inv}\ d\ v\ \$\ s)$
⟨*proof*⟩

lemma *GS-rec-fun-eq-GS-inv*: $\textit{GS-rec-fun}\ v\ s = (\bigsqcup d{\in}D_D.\ \textit{GS-inv}\ d\ (\textit{vec-lambda}\ v)\ \$\ s)$
⟨*proof*⟩

lemma *invertible-Q-GS*: $\textit{invertible}_L\ (\textit{Q-GS}\ d)$ **for** d
⟨*proof*⟩

lemma *ex-opt-blinfun*: $\exists d.\ \forall s.\ \textit{is-arg-max}\ (\lambda d.\ ((\textit{inv}_L\ (\textit{Q-GS}\ d))\ (\textit{r-det}_b\ d + (\textit{R-GS}\ d)\ v))\ s)\ \textit{is-dec-det}\ d$
⟨*proof*⟩

lemma *GS-inv-blinfun-to-matrix*: $((inv_L\ (Q\text{-}GS\ d))\ (r\text{-}det_b\ d + R\text{-}GS\ d\ v)) = Bfun\ (vec\text{-}nth\ (GS\text{-}inv\ d\ (vec\text{-}lambda\ v)))$
⟨*proof*⟩

lemma *norm-GS-QR-le-disc*: $norm\ (inv_L\ (Q\text{-}GS\ d)\ o_L\ R\text{-}GS\ d) \leq l$
⟨*proof*⟩

sublocale *GS*: *MDP-QR A K r l Q-GS R-GS*
rewrites $GS.\mathcal{L}_b\text{-}split = GS\text{-}rec\text{-}fun_b$
⟨*proof*⟩

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

lemma *dist-$\mathcal{L}_b$-split-lt-dist-opt*: $dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) \leq 2 * dist\ v\ \nu_b\text{-}opt$
⟨*proof*⟩

lemma *GS-QR-disc-le-disc*: $GS.QR\text{-}disc \leq l$
⟨*proof*⟩

lemma *gs-rel-dec*:
assumes $l \neq 0\ GS\text{-}rec\text{-}fun_b\ v \neq \nu_b\text{-}opt$
shows $\lceil log\ (1\ /\ l)\ (dist\ (GS\text{-}rec\text{-}fun_b\ v)\ \nu_b\text{-}opt) - c\rceil < \lceil log\ (1\ /\ l)\ (dist\ v\ \nu_b\text{-}opt) - c\rceil$
⟨*proof*⟩

function $gs\text{-}iteration :: real \Rightarrow ('s \Rightarrow_b real) \Rightarrow ('s \Rightarrow_b real)$ **where**
$gs\text{-}iteration\ eps\ v =$
$(if\ 2 * l * dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) < eps * (1-l) \lor eps \leq 0\ then\ GS\text{-}rec\text{-}fun_b\ v\ else\ gs\text{-}iteration\ eps\ (GS\text{-}rec\text{-}fun_b\ v))$
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *THE-fix-GS-rec-fun$_b$*: $(THE\ v.\ GS\text{-}rec\text{-}fun_b\ v = v) = \nu_b\text{-}opt$
⟨*proof*⟩

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

lemma *contraction-$\mathcal{L}$-split-dist*: $(1 - l) * dist\ v\ \nu_b\text{-}opt \leq dist\ v\ (GS\text{-}rec\text{-}fun_b\ v)$
⟨*proof*⟩

lemma *dist-$\mathcal{L}_b$-split-opt-eps*:
assumes $eps > 0$ $2 * l * dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) < eps * (1-l)$
shows $dist\ (GS\text{-}rec\text{-}fun_b\ v)\ \nu_b\text{-}opt < eps\ /\ 2$
⟨*proof*⟩
end

context *MDP-ord*
begin

lemma *is-am-GS-inv-extend'*:
assumes $(\bigwedge s.\ s < x \Longrightarrow is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}inv\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ d)$
assumes $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}rec\text{-}step\ d\ v\ \$\ x)\ (\lambda d.\ d \in D_D)\ (d(x := a))$
assumes $s \leq x$ $d \in D_D$
shows $is\text{-}arg\text{-}max\ (\lambda d.\ GS\text{-}inv\ d\ v\ \$\ s)\ (\lambda d.\ d \in D_D)\ (d(x := a))$
⟨*proof*⟩

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

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

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

lemma *GS-rec-eq-iter*:
assumes $\bigwedge s.\ s < k \Longrightarrow v'\ \$\ s = GS\text{-}rec\ v\ \$\ s$ $\bigwedge s.\ k \leq s \Longrightarrow v'\ \$\ s = v\ \$\ s$
shows $GS\text{-}rec\text{-}step\ (d(k := a))\ v\ \$\ k = GS\text{-}iter\ a\ v'\ k$
⟨*proof*⟩

lemma *GS-rec-eq-iter-max*:
assumes $\bigwedge s.\ s < k \Longrightarrow v'\ \$\ s = GS\text{-}rec\ v\ \$\ s$ $\bigwedge s.\ k \leq s \Longrightarrow v'\ \$\ s = v\ \$\ s$
shows $GS\text{-}rec\ v\ \$\ k = GS\text{-}iter\text{-}max\ v'\ k$
⟨*proof*⟩

definition $GS\text{-}iter\text{-}arg\text{-}max\ v\ s = (LEAST\ a.\ is\text{-}arg\text{-}max\ (\lambda a.\ GS\text{-}iter\ a\ v\ s)\ (\lambda a.\ a \in A\ s)\ a)$

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

lemma *GS-rec-am-code'*: *GS-rec-am-code* = *GS-rec-am-code'*
⟨*proof*⟩

lemma *opt-policy-gs'-eq-GS-iter*:
assumes $\bigwedge s.\ s < k \Longrightarrow v'\ \$\ s = GS\text{-}rec\ v\ \$\ s$ $\bigwedge s.\ k \leq s \Longrightarrow v'\ \$\ s = v\ \$\ s$
shows *opt-policy-gs' d v k* = *GS-iter-arg-max v' k*
⟨*proof*⟩

lemma *opt-policy-gs'-eq-GS-iter'*:
opt-policy-gs' d v k = *GS-iter-arg-max* (χ *s. if s* < *k then GS-rec v* \$ *s else v* \$ *s*) *k*
⟨*proof*⟩

lemma *opt-policy-gs'-is-dec-det*: *opt-policy-gs' d v* $\in D_D$
⟨*proof*⟩

lemma *opt-policy-gs'-is-arg-max*: *is-arg-max* (λd. *GS-inv d v* \$ *s*) (λd. $d \in D_D$) (*opt-policy-gs' d v*)
⟨*proof*⟩

lemma *GS-rec-am-code v d s* = (*GS-rec v* \$ *s*, *opt-policy-gs' d v s*)
⟨*proof*⟩

lemma *GS-rec-am-code-eq*: *GS-rec-am-code v d s* = (*GS-rec v* \$ *s*, *opt-policy-gs' d v s*)
⟨*proof*⟩

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

lemma *opt-policy-gs-eq'*: *opt-policy-gs v* = *opt-policy-gs' d* (*vec-lambda v*)
⟨*proof*⟩

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

lemma *gs-iteration-error*:
assumes *eps* > *0*
shows *dist* (*gs-iteration eps v*) ν_b*-opt* < *eps* / *2*
⟨*proof*⟩

lemma *GS-rec-fun-inner-vec*: *GS-rec-fun-inner v s a* = *GS-rec-step* (*d*(*s* := *a*)) (*vec-lambda v*) \$ *s*
⟨*proof*⟩

lemma *find-policy-error-bound-gs*:
assumes $eps > 0$ $2 * l * dist\ v\ (GS\text{-}rec\text{-}fun_b\ v) < eps * (1-l)$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (opt\text{-}policy\text{-}gs\ (GS\text{-}rec\text{-}fun_b\ v))))$
$\nu_b\text{-}opt < eps$
⟨*proof*⟩

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

lemma *vi-gs-policy-opt*:
assumes $0 < eps$
shows $dist\ (\nu_b\ (mk\text{-}stationary\text{-}det\ (vi\text{-}gs\text{-}policy\ eps\ v)))\ \nu_b\text{-}opt < eps$
⟨*proof*⟩

lemma *GS-rec-iter-eq-iter-max*: *GS-rec-iter v = GS-iter-max (vec-lambda v)*
⟨*proof*⟩
end

end

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

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

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

7 Code Generation for MDP Algorithms

7.1 Least Argmax

lemma *least-list*:
assumes *sorted xs* $\exists x \in set\ xs.\ P\ x$

shows (*LEAST* $x \in$ *set xs*. *P x*) = *the* (*find P xs*)
⟨*proof*⟩

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

lemma *least-enum-eq*: $\exists x.\ P\ x \Longrightarrow$ *least-enum P* = (*LEAST x*. *P x*)
⟨*proof*⟩

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

lemma *snd-least-max-arg-max-list*:
snd (*least-max-arg-max-list f* (*n*, *f n*) *xs*) = (*MAX* $x \in$ *insert n* (*set xs*). *f x*)
⟨*proof*⟩

lemma *least-max-arg-max-list-snd-fst*: *snd* (*least-max-arg-max-list f* (*x*, *f x*) *xs*) = *f* (*fst* (*least-max-arg-max-list f* (*x*, *f x*) *xs*))
⟨*proof*⟩

lemma *fst-least-max-arg-max-list*:
fixes *f* :: - ⇒ - :: *linorder*
assumes *sorted* (*n#xs*)
shows *fst* (*least-max-arg-max-list f* (*n*, *f n*) *xs*)
= (*LEAST x*. *is-arg-max f* (λx. $x \in$ *insert n* (*set xs*)) *x*)
⟨*proof*⟩

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

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

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

lemma *least-max-arg-max-enum-correct1*:
assumes $X \neq \{\}$
shows *fst* (*least-max-arg-max-enum* (*f* :: - ⇒ (- :: *linorder*)) *X*) = (*LEAST x*. *is-arg-max f* (λx. $x \in X$) *x*)
⟨*proof*⟩

lemma *least-max-arg-max-enum-correct2*:
assumes $X \neq \{\}$
shows *snd* (*least-max-arg-max-enum* (*f* :: - ⇒ (- :: *linorder*)) *X*) = (*MAX* *x* ∈ *X*. *f* *x*)
⟨*proof*⟩

7.2 Functions as Vectors

typedef ('*a*, '*b*) *Fun* = *UNIV* :: ('*a* ⇒ '*b*) *set*
⟨*proof*⟩

setup-lifting *type-definition-Fun*

lift-definition *to-Fun* :: ('*a* ⇒ '*b*) ⇒ ('*a*, '*b*) *Fun* **is** *id*⟨*proof*⟩

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

lift-definition *vec-to-fun* :: '*b*^'*a* ⇒ ('*a*, '*b*) *Fun* **is** *vec-nth*⟨*proof*⟩

lemma *Fun-inverse*[*simp*]: *Rep-Fun* (*Abs-Fun* *f*) = *f*
⟨*proof*⟩

lift-definition *zero-Fun* :: ('*a*, '*b*::*zero*) *Fun* **is** *0*⟨*proof*⟩

code-datatype *vec-to-fun*

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

instantiation *Fun* :: (*enum*, *equal*) *equal*
begin
definition *equal-Fun* (*f* :: ('*a*::*enum*, '*b*::*equal*) *Fun*) *g* = (*Rep-Fun* *f* = *Rep-Fun* *g*)
instance
⟨*proof*⟩
end

7.3 Bounded Functions as Vectors

lemma *Bfun-inverse-fin*[*simp*]: *apply-bfun* (*Bfun* (*f* :: '*c* :: *finite* ⇒ -)) = *f*
⟨*proof*⟩

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

code-datatype *vec-to-bfun*

lemma *apply-bfun-vec-to-bfun*[*code*]: *apply-bfun* (*vec-to-bfun f*) *x* = *f* \$ *x*
⟨*proof*⟩

lemma [*code*]: *0* = *vec-to-bfun 0*
⟨*proof*⟩

7.4 IArrays with Lengths in the Type

typedef (′*s* :: *mod-type*, ′*a*) *iarray-type* = {*arr* :: ′*a iarray*. *IArray.length arr* = *CARD*(′*s*)}
⟨*proof*⟩

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

lift-definition *fun-to-iarray-t* :: (′*s*::{*mod-type*} ⇒ ′*a*) ⇒ (′*s*, ′*a*) *iarray-type* **is** λ*f*. *IArray.of-fun* (λ*s*. *f* (*from-nat s*)) (*CARD*(′*s*))
⟨*proof*⟩

lift-definition *iarray-t-sub* :: (′*s*::*mod-type*, ′*a*) *iarray-type* ⇒ ′*s* ⇒ ′*a* **is** λ*v x*. *IArray.sub v* (*to-nat x*)⟨*proof*⟩

lift-definition *iarray-to-vec* :: (′*s*, ′*a*) *iarray-type* ⇒ ′*a*^′*s*::{*mod-type*, *finite*}
is λ*v*. (χ *s*. *IArray.sub v* (*to-nat s*))⟨*proof*⟩

lift-definition *vec-to-iarray* :: ′*a*^′*s*::{*mod-type*, *finite*} ⇒ (′*s*, ′*a*) *iarray-type*
is λ*v*. *IArray.of-fun* (λ*s*. *v* \$ ((*from-nat s*) :: ′*s*)) (*CARD*(′*s*))
⟨*proof*⟩

lemma *length-iarray-type* [*simp*]: *length* (*IArray.list-of* (*Rep-iarray-type* (*v*:: (′*s*::{*mod-type*}, ′*a*) *iarray-type*))) = *CARD*(′*s*)
⟨*proof*⟩

lemma *iarray-t-eq-iff*: (*v* = *w*) = (∀ *x*. *iarray-t-sub v x* = *iarray-t-sub w x*)
⟨*proof*⟩

lemma *iarray-to-vec-inv*: *iarray-to-vec* (*vec-to-iarray v*) = *v*
⟨*proof*⟩

lemma *vec-to-iarray-inv*: *vec-to-iarray* (*iarray-to-vec v*) = *v*
⟨*proof*⟩

code-datatype *iarray-to-vec*

lemma *vec-nth-iarray-to-vec*[*code*]: *vec-nth* (*iarray-to-vec v*) *x* = *iarray-t-sub v x*

⟨*proof*⟩

lemma *vec-lambda-iarray-t*[*code*]: *vec-lambda v* = *iarray-to-vec* (*fun-to-iarray-t v*)
⟨*proof*⟩

lemma *zero-iarray*[*code*]: *0* = *iarray-to-vec* (*fun-to-iarray-t 0*)
⟨*proof*⟩

7.5 Value Iteration

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

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

lemma *vi-eq-partial*: *eps* > *0* ⟹ *value-iteration-partial eps v* = *value-iteration eps v*
⟨*proof*⟩

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

lemma *code-L-det* [*code*]: *L-det d* (*vec-to-bfun v*) = *vec-to-bfun* (χ *s*. L_a (*d s*) (*vec-nth v*) *s*)
⟨*proof*⟩

lemma *code-*$\mathcal{L}_b$ [*code*]: $\mathcal{L}_b$ (*vec-to-bfun v*) = *vec-to-bfun* (χ *s*. (*MAX a* ∈ *A s*. *r* (*s*, *a*) + *l* ∗ *measure-pmf.expectation* (*K* (*s*, *a*)) (*vec-nth v*)))
⟨*proof*⟩

lemma *code-value-iteration*[*code*]: *value-iteration eps* (*vec-to-bfun v*) =
(*if eps* ≤ *0 then* $\mathcal{L}_b$ (*vec-to-bfun v*) *else value-iteration-partial eps* (*vec-to-bfun v*))
⟨*proof*⟩

lift-definition *find-policy-impl* :: (*'s* $\Rightarrow_b$ *real*) ⇒ (*'s*, *'a*) *Fun* **is** λ*v*. *find-policy' v*⟨*proof*⟩
lemma *code-find-policy-impl*: *find-policy-impl v* = *vec-to-fun* (χ *s*. (*LEAST x*. *x* ∈ *opt-acts v s*))

⟨*proof*⟩

lemma *code-find-policy-impl-opt*[*code*]: *find-policy-impl v* = *vec-to-fun* (χ *s. least-arg-max-enum* (λa. L_a *a v s*) (*A s*))
⟨*proof*⟩

lemma *code-vi-policy′*[*code*]: *vi-policy′ eps v* = *Rep-Fun* (*find-policy-impl* (*value-iteration eps v*))
⟨*proof*⟩

7.6 Policy Iteration

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

lemma *pi-eq-partial*: $d \in D_D \Longrightarrow$ *policy-iteration-partial d* = *policy-iteration d*
⟨*proof*⟩

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

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

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

lemma *mat-eq-blinfun*: *mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*)) = *blinfun-to-matrix* (*id-blinfun* − *l* $*_R$ $\mathcal{P}_1$ (*mk-dec-det* (*vec-nth d*)))
⟨*proof*⟩

lemma ν_b*-vec*: *policy-eval′* (*vec-to-fun d*) = *vec-to-bfun* (*matrix-inv* (*mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*))) *∗v* (*r-vec′* (*vec-to-fun d*)))
⟨*proof*⟩

lemma ν_b*-vec-opt*[*code*]: *policy-eval′* (*vec-to-fun d*) = *vec-to-bfun* (*Matrix-To-IArray.iarray-to-vec* (*Matrix-To-IArray.vec-to-iarray* ((*fst* (*Gauss-Jordan-PA* ((*mat 1* − *l* $*_R$ (*P-mat* (*vec-to-fun d*)))))) *∗v* (*r-vec′* (*vec-to-fun d*)))))
⟨*proof*⟩

lift-definition *policy-improvement′* :: (*′s*, *′a*) *Fun* ⇒ (*′s* $\Rightarrow_b$ *real*) ⇒ (*′s*, *′a*) *Fun*
is *policy-improvement*⟨*proof*⟩

lemma [*code*]: *policy-improvement′* (*vec-to-fun d*) *v* = *vec-to-fun* (χ *s.* (*if is-arg-max* (λa. L_a *a v s*) (λa. $a \in A$ *s*) (*d* \$ *s*) *then d* \$ *s else LEAST x. is-arg-max* (λa. L_a *a v s*) (λa. $a \in A$ *s*) *x*))
⟨*proof*⟩

lift-definition *policy-step′* :: ('s, 'a) *Fun* ⇒ ('s, 'a) *Fun*
is *policy-step*⟨*proof*⟩

lemma [*code*]: *policy-step′ d* = *policy-improvement′ d* (*policy-eval′ d*)
⟨*proof*⟩

lift-definition *policy-iteration-partial′* :: ('s, 'a) *Fun* ⇒ ('s, 'a) *Fun*
is *policy-iteration-partial*⟨*proof*⟩

lemma [*code*]: *policy-iteration-partial′ d* = (*let d′* = *policy-step′ d in if d* = *d′ then d else policy-iteration-partial′ d′*)
⟨*proof*⟩

lift-definition *policy-iteration′* :: ('s, 'a) *Fun* ⇒ ('s, 'a) *Fun* **is** *policy-iteration*⟨*proof*⟩

lemma *code-policy-iteration′*[*code*]: *policy-iteration′ d* =
(*if Rep-Fun d* $\notin$ D_D *then d else* (*policy-iteration-partial′ d*))
⟨*proof*⟩

lemma *code-policy-iteration*[*code*]: *policy-iteration d* = *Rep-Fun* (*policy-iteration′* (*vec-to-fun* (*vec-lambda d*)))
⟨*proof*⟩

7.7 Gauss-Seidel Iteration

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

lemma *gs-iteration-partial-eq*: *eps* > *0* ⟹ *gs-iteration-partial eps v* = *gs-iteration eps v*
⟨*proof*⟩

lemma *gs-iteration-code-opt*[*code*]: *gs-iteration eps v* = (*if eps* ≤ *0 then GS-rec-fun*$_b$ *v else gs-iteration-partial eps v*)
⟨*proof*⟩

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

lemma *GS-rec-eq-fold*: *GS-rec v* = *foldl* (λ*v s*. (*vec-upd v s* (*GS-iter-max v s*))) *v* (*sorted-list-of-set UNIV*)
⟨*proof*⟩

lemma *GS-rec-fun-code′′′′*[*code*]: *GS-rec-fun*$_b$ (*vec-to-bfun v*) = *vec-to-bfun* (*foldl* (λ*v s*. (*vec-upd v s* (*GS-iter-max v s*))) *v* (*sorted-list-of-set*

UNIV))
⟨*proof*⟩

lemma *GS-iter-max-code* [*code*]: *GS-iter-max v s* = (*MAX a* ∈ *A s*. *GS-iter a v s*)
⟨*proof*⟩

lift-definition *opt-policy-gs''* :: (*'s* $\Rightarrow_b$ *real*) ⇒ (*'s*, *'a*) *Fun* **is** *opt-policy-gs*⟨*proof*⟩

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

lemma *GS-rec-am-code'-prod*: *GS-rec-am-code' v d* =
(λ*s'*. (
let (*v'*, *d'*) = *foldl* (λ(*v*,*d*) *s*. (*v*(*s* := (*GS-iter-max* (*vec-lambda v*) *s*)), *d*(*s* := *GS-iter-arg-max* (*vec-lambda v*) *s*))) (*vec-nth v*, *d*) (*sorted-list-of-set UNIV*)
in (*v' s'*, *d' s'*)))
⟨*proof*⟩

lemma *code-GS-rec-am-arr-opt*[*code*]: *opt-policy-gs''* (*vec-to-bfun v*) = *vec-to-fun* ((*snd* (*foldl* (λ(*v*, *d*) *s*.
let (*am*, *m*) = *least-max-arg-max-enum* (λ*a*. *r* (*s*, *a*) + *l* ∗ (∑ *s'* ∈ *UNIV*. *pmf* (*K* (*s*,*a*)) *s'* ∗ *v* $ *s'*)) (*A s*) *in*
(*vec-upd v s m*, *vec-upd d s am*))
(*v*, (χ *s*. (*least-enum* (λ*a*. *a* ∈ *A s*)))) (*sorted-list-of-set UNIV*))))
⟨*proof*⟩

7.8 Modified Policy Iteration

sublocale *MDP-MPI A K r l* λ*X*. *Least* (λ*x*. *x* ∈ *X*)
⟨*proof*⟩

definition *d0 s* = (*LEAST a*. *a* ∈ *A s*)
lift-definition *d0'* :: (*'s*, *'a*) *Fun* **is** *d0*⟨*proof*⟩

lemma *d0-dec-det*: *is-dec-det d0*
⟨*proof*⟩

lemma *v0-code*[*code*]: $v0\text{-}mpi_b$ = *vec-to-bfun* (χ *s*. *r-min* / (*1* − *l*))
⟨*proof*⟩

lemma *d0'-code*[*code*]: *d0'* = *vec-to-fun* (χ *s*. (*LEAST a*. *a* ∈ *A s*))
⟨*proof*⟩

lemma *step-value-code*[*code*]: *L-pow v d m* = (*L-det d* ^^ *Suc m*) *v*
⟨*proof*⟩

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

lemma *mpi-partial-eq-algo*:
  assumes *eps* > *0* $d \in D_D$ $v \le \mathcal{L}_b\ v$
  shows *mpi-partial eps d v m* = *mpi-algo eps d v m*
⟨*proof*⟩

lift-definition *mpi-partial'* :: *real* ⇒ ('*s*, '*a*) *Fun* ⇒ ('*s* $\Rightarrow_b$ *real*) ⇒ (*nat* ⇒ ('*s* $\Rightarrow_b$ *real*) ⇒ *nat*)
    ⇒ ('*s*, '*a*) *Fun* × ('*s* $\Rightarrow_b$ *real*) **is** *mpi-partial*⟨*proof*⟩

lemma *mpi-partial'-code*[*code*]: *mpi-partial' eps d v m* =
  (*let* d' = *policy-improvement' d v in* (
    if $2 * l * dist\ v\ (\mathcal{L}_b\ v) < eps * (1 - l)$
    then (d', v)
    else mpi-partial' eps d' (*L-pow v* (*Rep-Fun* d') (*m 0 v*)) ($\lambda n.$ *m* (*Suc n*))))
  ⟨*proof*⟩

lemma *r-min-code*[*code-unfold*]: *r-min* = (*MIN s. MIN a. r*(*s*,*a*))
  ⟨*proof*⟩

lemma *mpi-user-code*[*code*]: *mpi-user eps m* =
  (*if eps* ≤ *0 then undefined else*
    let (*d*, *v*) = *mpi-partial' eps d0' v0-mpi*$_b$ *m in* (*Rep-Fun d*, *v*))
  ⟨*proof*⟩
end

7.9 Auxiliary Equations

lemma [*code-unfold*]: *dist* (*f*::'*a*::*finite* $\Rightarrow_b$ '*b*::*metric-space*) *g* = (*MAX a. dist* (*apply-bfun f a*) (*g a*))
  ⟨*proof*⟩

lemma *member-code*[*code del*]: *x* ∈ *List.coset xs* ⟷ ¬ *List.member xs x*
  ⟨*proof*⟩

lemma [*code*]: *iarray-to-vec v* + *iarray-to-vec u* = (*Matrix-To-IArray.iarray-to-vec* (*Rep-iarray-type v* + *Rep-iarray-type u*))
  ⟨*proof*⟩

lemma [*code*]: *iarray-to-vec v* − *iarray-to-vec u* = (*Matrix-To-IArray.iarray-to-vec* (*Rep-iarray-type v* − *Rep-iarray-type u*))

⟨*proof*⟩

lemma *matrix-to-iarray-minus*[*code-unfold*]: *matrix-to-iarray* $(A - B)$ $=$ *matrix-to-iarray* $A -$ *matrix-to-iarray* B
⟨*proof*⟩

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

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

8 Code Generation for Concrete Finite MDPs

locale *mod-MDP* =
fixes *transition* :: $'s$::{*enum*, *mod-type*} $\times$ $'a$::{*enum*, *mod-type*} $\Rightarrow$ $'s$ *pmf*
and A :: $'s \Rightarrow$ $'a$ *set*
and *reward* :: $'s \times 'a \Rightarrow$ *real*
and *discount* :: *real*
begin

sublocale *mdp*: *vi-code*
$\lambda s.$ (*if Set.is-empty* $(A\ s)$ *then UNIV else* $A\ s$)
transition
reward
(*if* $1 \le$ *discount* $\vee$ *discount* < 0 *then* 0 *else discount*)
defines $\mathcal{L}_b$ = *mdp*.$\mathcal{L}_b$
and *L-det* = *mdp.L-det*
and *value-iteration* = *mdp.value-iteration*
and *vi-policy′* = *mdp.vi-policy′*
and *find-policy′* = *mdp.find-policy′*
and *find-policy-impl* = *mdp.find-policy-impl*
and *is-opt-act* = *mdp.is-opt-act*
and *value-iteration-partial* = *mdp.value-iteration-partial*
and *policy-iteration* = *mdp.policy-iteration*
and *is-dec-det* = *mdp.is-dec-det*
and *policy-step* = *mdp.policy-step*
and *policy-improvement* = *mdp.policy-improvement*
and *policy-eval* = *mdp.policy-eval*
and *mk-markovian* = *mdp.mk-markovian*
and *policy-eval′* = *mdp.policy-eval′*
and *policy-iteration-partial′* = *mdp.policy-iteration-partial′*
and *policy-iteration′* = *mdp.policy-iteration′*
and *policy-iteration-policy-step′* = *mdp.policy-step′*
and *policy-iteration-policy-eval′* = *mdp.policy-eval′*
and *policy-iteration-policy-improvement′* = *mdp.policy-improvement′*
and *gs-iteration* = *mdp.gs-iteration*

and *gs-iteration-partial* = *mdp.gs-iteration-partial*
and *vi-gs-policy* = *mdp.vi-gs-policy*
and *opt-policy-gs* = *mdp.opt-policy-gs*
and *opt-policy-gs''* = *mdp.opt-policy-gs''*
and *P-mat* = *mdp.P-mat*
and *r-vec'* = *mdp.r-vec'*
and $GS\text{-}rec\text{-}fun_b$ = $mdp.GS\text{-}rec\text{-}fun_b$
and *GS-iter-max* = *mdp.GS-iter-max*
and *GS-iter* = *mdp.GS-iter*
and *mpi-user* = *mdp.mpi-user*
and $v0\text{-}mpi_b$ = $mdp.v0\text{-}mpi_b$
and *mpi-partial'* = *mdp.mpi-partial'*
and *L-pow* = *mdp.L-pow*
and *v0-mpi* = *mdp.v0-mpi*
and *r-min* = *mdp.r-min*
and *d0* = *mdp.d0*
and *d0'* = *mdp.d0'*
and ν_b = $mdp.\nu_b$
and *vi-test* = *mdp.vi-test*
⟨*proof*⟩
end

global-interpretation *mod-MDP transition A reward discount*
for *transition A reward discount*
defines $mod\text{-}MDP\text{-}\mathcal{L}_b$ = $mdp.\mathcal{L}_b$
and $mod\text{-}MDP\text{-}\mathcal{L}_b\text{-}L\text{-}det$ = *mdp.L-det*
and *mod-MDP-value-iteration* = *mdp.value-iteration*
and *mod-MDP-vi-policy'* = *mdp.vi-policy'*
and *mod-MDP-find-policy'* = *mdp.find-policy'*
and *mod-MDP-find-policy-impl* = *mdp.find-policy-impl*
and *mod-MDP-is-opt-act* = *mdp.is-opt-act*
and *mod-MDP-value-iteration-partial* = *mdp.value-iteration-partial*
and *mod-MDP-policy-iteration* = *mdp.policy-iteration*
and *mod-MDP-is-dec-det* = *mdp.is-dec-det*
and *mod-MDP-policy-step* = *mdp.policy-step*
and *mod-MDP-policy-improvement* = *mdp.policy-improvement*
and *mod-MDP-policy-eval* = *mdp.policy-eval*
and *mod-MDP-mk-markovian* = *mdp.mk-markovian*
and *mod-MDP-policy-eval'* = *mdp.policy-eval'*
and *mod-MDP-policy-iteration-partial'* = *mdp.policy-iteration-partial'*
and *mod-MDP-policy-iteration'* = *mdp.policy-iteration'*
and *mod-MDP-policy-iteration-policy-step'* = *mdp.policy-step'*
and *mod-MDP-policy-iteration-policy-eval'* = *mdp.policy-eval'*
and *mod-MDP-policy-iteration-policy-improvement'* = *mdp.policy-improvement'*
and *mod-MDP-gs-iteration* = *mdp.gs-iteration*
and *mod-MDP-gs-iteration-partial* = *mdp.gs-iteration-partial*
and *mod-MDP-vi-gs-policy* = *mdp.vi-gs-policy*
and *mod-MDP-opt-policy-gs* = *mdp.opt-policy-gs*
and *mod-MDP-opt-policy-gs''* = *mdp.opt-policy-gs''*

and *mod-MDP-P-mat* = *mdp.P-mat*
and *mod-MDP-r-vec'* = *mdp.r-vec'*
and *mod-MDP-GS-rec-fun*$_b$ = *mdp.GS-rec-fun*$_b$
and *mod-MDP-GS-iter-max* = *mdp.GS-iter-max*
and *mod-MDP-GS-iter* = *mdp.GS-iter*
and *mod-MDP-mpi-user* = *mdp.mpi-user*
and *mod-MDP-v0-mpi*$_b$ = *mdp.v0-mpi*$_b$
and *mod-MDP-mpi-partial'* = *mdp.mpi-partial'*
and *mod-MDP-L-pow* = *mdp.L-pow*
and *mod-MDP-v0-mpi* = *mdp.v0-mpi*
and *mod-MDP-r-min* = *mdp.r-min*
and *mod-MDP-d0* = *mdp.d0*
and *mod-MDP-d0'* = *mdp.d0'*
and *mod-MDP-*ν_b = *mdp.*ν_b
and *mod-MDP-vi-test* = *mdp.vi-test*
⟨*proof*⟩

end
theory *Code-Real-Approx-By-Float-Fix*
imports
HOL−Library.Code-Real-Approx-By-Float
Gauss-Jordan.Code-Real-Approx-By-Float-Haskell
beginend

theory *Code-Inventory*
imports
Code-Mod

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

9 Inventory Management Example

lemma [*code abstype*]: *embed-pmf* (*pmf P*) = *P*
⟨*proof*⟩

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

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

lemma [*code*]: *pmf* (*return-pmf x*) = ($\lambda y.$ *indicat-real* {*y*} *x*)
⟨*proof*⟩

lemma [*code*]:
  pmf (*bind-pmf N f*) = (λ*i* :: *'a*. *measure-pmf*.*expectation N* (λ(*x* :: *'b* ::*enum*). *pmf* (*f x*) *i*))
  ⟨*proof*⟩

lemma *pmf-finite-le*: *finite* (*X* :: (*'a*::*finite*) *set*) $\Longrightarrow$ *sum* (*pmf p*) *X* $\leq$ *1*
  ⟨*proof*⟩

lemma *mod-less-diff*:
  assumes *0* < (*x*::*'s*::{*mod-type*}) *x* $\leq$ *y*
  shows *y* − *x* < *y*
⟨*proof*⟩

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

⟨*proof*⟩

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

end

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

lift-definition *demand-unif* :: (*'a*::*finite*) *pmf* **is** λ-. *1* / *card* (*UNIV*::*'a set*)
  ⟨*proof*⟩

lift-definition *demand-three* :: *3 pmf* **is** $\lambda i.$ *if* $i = 1$ *then 1/4 else if* $i = 2$ *then 1/2 else 1/4*
⟨*proof*⟩

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

lemma *card-ge-2-imp-ne*: $CARD('a) \geq 2 \Longrightarrow \exists (x{::}'a{::}finite)\ y{::}'a.\ x \neq y$
⟨*proof*⟩

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

abbreviation K ≡ *inventory-ex.transition*
abbreviation A ≡ *inventory-ex.A-inv*
abbreviation r ≡ *inventory-ex.reward*
abbreviation l ≡ *0.95*
definition *eps* = *0.1*

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

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

definition *vs-n′ n* = *vs-n n 0*

definition *benchmark-vi-n n* = (*fun-to-list* (*vs-n n 0*))
definition *benchmark-vi-nopol* = (*fun-to-list* (*mod-MDP-value-iteration*

K A r l (*1/10*) *0*))

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

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

end

theory *Code-Gridworld*
imports
Code-Mod
begin

10 Gridworld Example

lemma [*code abstype*]: *embed-pmf* (*pmf P*) = *P*
⟨*proof*⟩

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

lemma [*code-unfold*]:
$\textit{measure-pmf.expectation}\ P\ (f :: {'a} :: \textit{enum} \Rightarrow \textit{real}) = (\sum x \in \textit{UNIV}.\ \textit{pmf}\ P\ x * f\ x)$
⟨*proof*⟩

lemma [*code*]: $\textit{pmf}\ (\textit{return-pmf}\ x) = (\lambda y.\ \textit{indicat-real}\ \{y\}\ x)$
⟨*proof*⟩

lemma [*code*]:
$\textit{pmf}\ (\textit{bind-pmf}\ N\ f) = (\lambda i :: {'a}.\ \textit{measure-pmf.expectation}\ N\ (\lambda(x :: {'b} :: \textit{enum}).\ \textit{pmf}\ (f\ x)\ i))$
⟨*proof*⟩

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

definition $\textit{from-state}\ x = (\textit{Rep}\ x\ \textit{div}\ 4,\ \textit{Rep}\ x\ \textit{mod}\ 4)$
definition $\textit{to-state}\ x = (\textit{Abs}\ (\textit{fst}\ x * 4 + \textit{snd}\ x) :: \textit{state-robot})$

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

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

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

lift-definition *add-noise* :: *action-robot* ⇒ *action-robot pmf* **is** λ*det rnd*. (
  if det = *rnd then 1* − *noise else if det* = *rnd* − *1* ∨ *det* = *rnd* + *1 then noise* / *2 else 0*)
  ⟨*proof*⟩

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

fun *K-robot* :: (*state-robot* × *action-robot*) ⇒ *state-robot pmf* **where**
  K-robot (*loc*, *a*) =
  do {
  a ← *add-noise a*;
  let (*y*, *x*) = *from-state loc*;
  let (*y*′, *x*′) =
    (*if a* = *0 then* (*y* + *1* , *x*)
      else if a = *1 then* (*y*, *x*+*1*)
      else if a = *2 then* (*y*−*1*, *x*)
      else if a = *3 then* (*y*, *x*−*1*)
      else undefined);
   return-pmf (
      if (*y*,*x*) = (*2*,*3*) ∨ (*y*,*x*) = (*1*,*3*) ∨ (*y*,*x*) = (*3*,*0*)
        then to-state (*3*,*0*)
      else if y′ < *0* ∨ *y*′ > *2* ∨ *x*′ < *0* ∨ *x*′ > *3* ∨ (*y*′,*x*′) = (*1*,*1*)
      then to-state (*y*, *x*)
        else to-state (*y*′, *x*′))
  }

definition *l-robot* = *0.9*

lemma *vi-code A-robot r-robot l-robot*
  ⟨*proof*⟩

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

abbreviation *gridworld-policy-eval*′ ≡ *to-gridworld mod-MDP-policy-eval*′
abbreviation *gridworld-policy-step*′ ≡ *to-gridworld*′ *mod-MDP-policy-iteration-policy-step*′
abbreviation *gridworld-mpi-user* ≡ *to-gridworld*′ *mod-MDP-mpi-user*

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

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

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

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

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

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

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

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

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

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

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

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

end

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

References

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