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*)) ⊆ *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*⟩

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*: ∀ *s* < *states*. *set-pmf* (*K* (*s*,*a*)) ⊆ {*0*..<*states*}
  assumes *K-closed-compl*: ∀ *s*≥ *states*. *set-pmf* (*K* (*s*,*a*)) ⊆ {*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*: ∀ *s*≥*states*. *r* (*s*,*a*) = *0*
begin
lemma $\mathcal{L}_b$-*eq*-L_a-*max′*: $\mathcal{L}_b\ v\ s = (MAX\ a \in A\ s.\ L_a\ a\ v\ s)$
  ⟨*proof*⟩

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

lemma *set-pmf-Xn′*: *s* ∉ *state-space* ⟹ *set-pmf* (*Xn′ p s i*) ⊆ {*states*..}
  ⟨*proof*⟩

lemma *set-pmf-Pn'*: $s \notin$ *state-space* $\Longrightarrow$ ($\forall$ *sa* $\in$ *set-pmf* (*Pn' p s i*). *fst sa*$\notin$ *state-space*)
⟨*proof*⟩

lemma *reward-Pn'-notin*: $s \notin$ *state-space* $\Longrightarrow$ ($\forall$ *sa* $\in$ *set-pmf* (*Pn' p s i*). *r sa* = *0*)
⟨*proof*⟩

lemma ν*-zero-notin*:
assumes $s \notin$ *state-space*
shows ν *p s* = *0*
⟨*proof*⟩

lemma ν*-opt-zero-notin*:
assumes $s \notin$ *state-space*
shows ν*-opt s* = *0*
⟨*proof*⟩

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* $\equiv$ (λ(*eps*, *v*). *LEAST n*. (*2* $*$ *l* $*$ *dist* (($\mathcal{L}_b$ ^^ (*Suc n*)) *v*) (($\mathcal{L}_b$ ^^ *n*) *v*) < *eps* $*$ (*1*−*l*)))

lemma *Least-Suc-less*:
  assumes $\exists\, n.\ P\ n\ \neg P\ 0$
  shows $Least\ (\lambda n.\ P\ (Suc\ n)) < Least\ P$
  ⟨*proof*⟩

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

lemma *dist-$\mathcal{L}_b$-lt-dist-opt*: $dist\ v\ (\mathcal{L}_b\ v) \leq 2 * dist\ v\ \nu_b\text{-}opt$
⟨*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 $2 * dist\ (value\text{-}iteration\ eps\ v)\ \nu_b\text{-}opt < eps$
  ⟨*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-policy* $(v :: 's \Rightarrow_b real)\ s = arg\text{-}max\text{-}on\ (\lambda a.\ L_a\ a\ v\ s)\ (A\ s)$

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

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

lemma *$\mathcal{L}_b$-iter-mono*:

assumes $u \leq v$ **shows** $vi\ u\ n \leq vi\ v\ n$
⟨*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 $(\lambda n.\ dist\ (vi\ v\ (Suc\ n))\ (vi\ v\ n)) \longrightarrow 0$
⟨*proof*⟩

end

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

lemma *is-arg-max-find-policy*: $is\text{-}arg\text{-}max\ (\lambda d.\ L_a\ d\ (apply\text{-}bfun\ v)\ s)\ (\lambda d.\ d \in A\ s)\ (find\text{-}policy\ v\ s)$
⟨*proof*⟩

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*⟩

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 $eps > 0\ 2 * l * dist\ (vi\ v\ (Suc\ n))\ (vi\ v\ n) < eps * (1-l)$

shows $2 * dist\ (vi\ v\ (Suc\ n))\ \nu_b\text{-}opt < eps$
⟨*proof*⟩
end

context *MDP-act-disc* **begin**

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

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

lemma *is-arg-max-find-policy′*: $is\text{-}arg\text{-}max\ (\lambda d.\ L_a\ d\ (apply\text{-}bfun\ v)\ s)$
$(\lambda d.\ d \in A\ s)\ (find\text{-}policy'\ v\ s)$
⟨*proof*⟩

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*⟩

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 *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* = [] ⟨*proof*⟩

lemma *tabulate-Suc*: *tabulate* (*Suc n*) *f* = *tabulate n f* @ [*f n*] ⟨*proof*⟩

lemma *tabulate-Suc'*: *tabulate* (*Suc n*) *f* = *f 0* # *tabulate n* (*f o Suc*)
⟨*proof*⟩

lemma *tabulate-const*[*simp*]: *tabulate n* (λ-. *c*) = *replicate n c* ⟨*proof*⟩

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

lemma *upd-tabulate*: (*tabulate n f*)[*i*:=*x*] = *tabulate n* (*f*(*i*:=*x*))
⟨*proof*⟩

lemma *take-tabulate*: *take n* (*tabulate m f*) = *tabulate* (*min n m*) *f*
⟨*proof*⟩

lemma *hd-tabulate*[*simp*]: *n*≠*0* ⟹ *hd* (*tabulate n f*) = *f 0*
⟨*proof*⟩

lemma *tl-tabulate*: *tl* (*tabulate n f*) = *tabulate* (*n*−*1*) (*f o Suc*)
⟨*proof*⟩

lemma *tabulate-cong*[*fundef-cong*]: *n*=*n'* ⟹ (⋀*i*. *i*<*n* ⟹ *f i* = *f' i*)
⟹ *tabulate n f* = *tabulate n' f'*
⟨*proof*⟩

lemma *tabulate-nth-take*: *n* ≤ *length xs* ⟹ *tabulate n* ((!) *xs*) = *take*
n xs
⟨*proof*⟩

end

lemma *drop-tabulate*: *drop n* (*tabulate m f*) = *tabulate* (*m*−*n*) (*f o*
(+)*n*)
⟨*proof*⟩

1.2 Primitive Operations

typedef *'a array* = *UNIV* :: *'a list set*
morphisms *array-α Array*
⟨*proof*⟩
setup-lifting *type-definition-array*

lift-definition *array-new* :: *nat* ⇒ *'a* ⇒ *'a array* **is** λ*n a*. *replicate n*
a ⟨*proof*⟩

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

lift-definition *array-length* :: *'a array* ⇒ *nat* **is** *length* ⟨*proof*⟩

lift-definition *array-get* :: *'a array* ⇒ *nat* ⇒ *'a* **is** *nth* ⟨*proof*⟩

lift-definition *array-set* :: *'a array* ⇒ *nat* ⇒ *'a* ⇒ *'a array* **is** *list-update* ⟨*proof*⟩

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

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* ⟨*proof*⟩

  lemma *array-eq-iff*: *a*=*b* ⟷ *array-α a* = *array-α b* ⟨*proof*⟩

  lemma *array-new-refine*[*simp*,*array-refine*]: *array-α* (*array-new n a*) = *replicate n a* ⟨*proof*⟩

  lemma *array-tabulate-refine*[*simp*,*array-refine*]: *array-α* (*array-tabulate n f*) = *tabulate n f* ⟨*proof*⟩

  lemma *array-length-refine*[*simp*,*array-refine*]: *array-length a* = *length* (*array-α a*) ⟨*proof*⟩

  lemma *array-get-refine*[*simp*,*array-refine*]: *array-get a i* = *array-α a* ! *i* ⟨*proof*⟩

  lemma *array-set-refine*[*simp*,*array-refine*]: *array-α* (*array-set a i x*) = (*array-α a*)[*i* := *x*] ⟨*proof*⟩

  lemma *array-of-list-refine*[*simp*,*array-refine*]: *array-α* (*array-of-list xs*) = *xs* ⟨*proof*⟩

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 (λ*i*. *if i*<*la then array-get a i else dflt*)
)

lemma *tabulate-grow*: *tabulate n* (λ*i*. *if i* < *length xs then xs*!*i else d*)
= *take n xs* @ (*replicate* (*n*−*length xs*) *d*)
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *array-take-refine*[*simp*,*array-refine*]: *array-α* (*array-take a n*)
= *take n* (*array-α a*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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* ()))
  ⟨*proof*⟩

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*)
⟨*proof*⟩

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′*
⟨*proof*⟩

context
fixes *f* :: *′a* ⇒ *′s* ⇒ *′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*)

⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma *foldl-idx-eq*: *foldl-idx i s* = *fold f* (*drop i xs*) *s*
⟨*proof*⟩

lemma *fold-by-idx*: *fold f xs s* = *foldl-idx 0 s* ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *foldr-by-idx*: *foldr f xs s* = *foldr-idx f xs* (*length xs*) *s* ⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*])
⟨*proof*⟩

lemma *array-new′-fallback*[*code*]: *array-new′ n x* = *array-of-list* (*replicate* (*nat-of-integer n*) *x*)
⟨*proof*⟩

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 DiffAr-
ray-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
$\forall$ *i*$\in$\{*0..*<*6*\}.
array-get a i = *i*+*1*
$\wedge$ *array-get b i* = *i*+*1*
$\wedge$ *array-get a' i* = (*if i=3 then 42 else i*+*1*)
$\wedge$ *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*
⟨*proof*⟩

lemma *test1*
⟨*proof*⟩

⟨*ML*⟩

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

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

experiment
begin

fun *allTrue* :: *bool list* $\Rightarrow$ *nat* $\Rightarrow$ *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*
⟨*proof*⟩

lemma *n* $\leq$ *length a* $\Longrightarrow$ $\forall$ *i* < *n*. (*allTrue a n*) ! *i*
⟨*proof*⟩

fun *allTrue'* :: *bool array* $\Rightarrow$ *nat* $\Rightarrow$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩
setup-lifting *type-definition-starray*

lift-definition *starray-new* :: *nat* ⇒ *′a* ⇒ *′a starray* **is** *array-new* ⟨*proof*⟩

lift-definition *starray-tabulate* :: *nat* ⇒ (*nat* ⇒ *′a*) ⇒ *′a starray* **is** *array-tabulate* ⟨*proof*⟩

lift-definition *starray-length* :: *′a starray* ⇒ *nat* **is** *array-length* ⟨*proof*⟩

lift-definition *starray-get* :: *′a starray* ⇒ *nat* ⇒ *′a* **is** *array-get* ⟨*proof*⟩

lift-definition *starray-set* :: *′a starray* ⇒ *nat* ⇒ *′a* ⇒ *′a starray* **is** *array-set* ⟨*proof*⟩

lift-definition *starray-of-list* :: *′a list* ⇒ *′a starray* **is** ‹*array-of-list*› ⟨*proof*⟩

lift-definition *starray-grow* :: *′a starray* ⇒ *nat* ⇒ *′a* ⇒ *′a starray* **is** *array-grow* ⟨*proof*⟩

lift-definition *starray-take* :: *′a starray* ⇒ *nat* ⇒ *′a starray* **is** *array-take* ⟨*proof*⟩

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

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

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

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

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

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* ⟨*proof*⟩

lemma *starray-eq-iff*: *a*=*b* $\longleftrightarrow$ *starray-α a* = *starray-α b* ⟨*proof*⟩

lemma *starray-new-refine*[*simp*,*array-refine*]: *starray-α* (*starray-new n a*) = *replicate n a* ⟨*proof*⟩

lemma *starray-tabulate-refine*[*simp*,*array-refine*]: *starray-α* (*starray-tabulate n f*) = *tabulate n f* ⟨*proof*⟩

lemma *starray-length-refine*[*simp*,*array-refine*]: *starray-length a* = *length* (*starray-α a*) ⟨*proof*⟩

lemma *starray-get-refine*[*simp*,*array-refine*]: *starray-get a i* = *starray-α a* ! *i* ⟨*proof*⟩

lemma *starray-set-refine*[*simp*,*array-refine*]: *starray-α* (*starray-set a i x*) = (*starray-α a*)[*i* := *x*] ⟨*proof*⟩

lemma *starray-of-list-refine*[*simp*,*array-refine*]: *starray-α* (*starray-of-list xs*) = *xs* ⟨*proof*⟩

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*

⟨*proof*⟩

lemma *starray-take-refine*[*simp*,*array-refine*]: *starray-α* (*starray-take a n*) = *take n* (*starray-α a*)
⟨*proof*⟩

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*) ⟨*proof*⟩

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* ()))
⟨*proof*⟩

lemma *starray-map-refine*[*simp*,*array-refine*]: *starray-α* (*starray-map f a*) = *map f* (*starray-α a*)
⟨*proof*⟩

lemma *starray-fold-refine*[*simp*, *array-refine*]: *starray-fold f a s* = *fold f* (*starray-α a*) *s*
⟨*proof*⟩

lemma *starray-foldr-refine*[*simp*, *array-refine*]: *starray-foldr f a s* = *foldr f* (*starray-α a*) *s*
⟨*proof*⟩

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*
⟨*proof*⟩

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*])
⟨*proof*⟩

lemma *starray-new'-fallback*[*code*]: *starray-new'* *n* *x* = *starray-of-list* (*replicate* (*nat-of-integer* *n*) *x*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *test1*
  ⟨*proof*⟩

⟨*ML*⟩

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*
⟨*proof*⟩

lemma $n \leq$ *length a* $\Longrightarrow \forall i < n.$ (*allTrue a n*) ! *i*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $\bigwedge s.\ s \geq$ *states* $\Longrightarrow$ *apply-bfun v s* = *0 s* $\geq$ *states*
shows *L d v s* = *0*
⟨*proof*⟩

lemma $\mathcal{L}_b$*-zero*:
assumes $\bigwedge s.\ s \geq$ *states* $\Longrightarrow$ *apply-bfun v s* = *0 s* $\geq$ *states*
shows $\mathcal{L}_b$ *v s* = *0*
⟨*proof*⟩
end

lemma *max-geI*: *finite A* $\Longrightarrow A \neq \{\} \Longrightarrow (\exists\, a \in A.\ x \leq a) \Longrightarrow (x \leq$

Max A) **for** *x A*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *Max-insert-absorb*: *finite X* ⟹ (∃ *y* ∈ *X*. *x* ≤ *y*) ⟹ *Max* (*Set.insert x X*) = (*if X* = {} *then x else Max X*)
⟨*proof*⟩

lemma *Max-insert-absorb'*: *finite X* ⟹ *y*∈*X* ⟹ *x* ≤ *y* ⟹ *Max* (*Set.insert x X*) = (*if X* = {} *then x else Max X*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *least-arg-max-ne-correct*:
assumes *sorted* (*x*#*xs*)

shows *least-arg-max-ne* (f :: - ⇒ $'b$ ::*linorder*) ($x\#xs$) = *least-arg-max* f (*List.member* ($x\#xs$))
⟨*proof*⟩

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*⟩

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 ($x\#xs$))
⟨*proof*⟩

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)
⟨*proof*⟩

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))
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

typedef *Valid-MDP* = {*mdp*. *is-MDP mdp*}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *map-of-None-notin-set*: *AList-Upd-Del.map-of xs k* = *None* ⟹ *k* ∉ *fst* ‘ *set xs*
⟨*proof*⟩

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 $(x, y) \in$ *entries m*
⟨*proof*⟩

lemma *lookup-None-set-inorder*:
assumes *invar m lookup m x = None*
shows $x \notin$ *keys m*
⟨*proof*⟩

lemma *entries-imp-keys*[*intro*]: $(x,y) \in$ *entries m* $\Longrightarrow x \in$ *keys m*
⟨*proof*⟩

lemma *lookup-some-set-key*: *invar m* $\Longrightarrow$ *lookup m x = Some y* $\Longrightarrow$ $x \in$ *keys m*
⟨*proof*⟩

lemma *lookup-in-keys*: *invar m* $\Longrightarrow x \in$ *keys m* $\Longrightarrow \exists\, y.$ *lookup m x = Some y*
⟨*proof*⟩

lemma *lookup-notin-keys*: *invar m* $\Longrightarrow x \notin$ *keys m* $\Longrightarrow$ *lookup m x = None*
⟨*proof*⟩

lemma *inorder-delete*: *invar m* $\Longrightarrow$ *inorder m = kv#xs* $\Longrightarrow$ *inorder* ((*delete* (*fst kv*) *m*)) = *xs*
⟨*proof*⟩

lemma *inorder-lookup-Some*: *invar m* $\Longrightarrow$ $(k, v) \in$ *entries m* $\Longrightarrow$ *lookup m k = Some v*
⟨*proof*⟩

lemma *keys-eq-lookup-Some*: *invar m* $\Longrightarrow$ *keys m* = $\{k.\ \exists\, v.$ *lookup m k = Some v*$\}$
⟨*proof*⟩

lemma *keys-eq-fst-entries*: *invar m* $\Longrightarrow$ *keys m = fst ‘ entries m*
⟨*proof*⟩

lemma *keys-update*[*simp*]: *invar m* $\Longrightarrow$ *keys* (*update k v m*) = *Set.insert k* (*keys m*)
⟨*proof*⟩

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

lemma *is-empty-iff-entries-empty*: *is-empty t* $\longleftrightarrow$ *entries t* = {}
⟨*proof*⟩

lemma *is-empty-iff-keys-empty*: *is-empty t* $\longleftrightarrow$ *keys t* = {}

⟨*proof*⟩

lemma *finite-keys*: *finite* (*keys t*)
⟨*proof*⟩

lemma *finite-entries*: *finite* (*entries t*)
⟨*proof*⟩

lemma *keys-empty*[*simp*]: *keys empty* = {}
⟨*proof*⟩

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

6 Converting Lists to Maps

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

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

lemma *from-list-invar*[*simp*]: *invar* (*from-list′ f xs*)
⟨*proof*⟩

lemma *from-list-snoc*[*simp*]: (*from-list′ f* (*xs* @[*y*])) = *update y* (*f y*) (*from-list′ f xs*)
⟨*proof*⟩

lemma *from-list-empty*[*simp*]: *from-list′ f* [] = *empty*
⟨*proof*⟩

lemma *from-list-keys*[*simp*]: *keys* (*from-list′ f xs*) = *set xs*
⟨*proof*⟩

lemma *from-list-lookup*[*simp*]: *x* ∈ *set xs* ⟹ *lookup* (*from-list′ f xs*) *x* = *Some* (*f x*)
⟨*proof*⟩

lemma *from-list-lookup′*[*simp*]: *x* ∈ *set xs* ⟹ *lookup′* (*from-list′ f xs*) *x* = *f x*
⟨*proof*⟩

lemma *from-list-snoc′*[*simp*]: (*from-list* (*xs* @[(*k*,*v*)])) = *update k v* (*from-list xs*)
⟨*proof*⟩

lemma *from-list-invar′*[*simp*]: *invar* (*from-list xs*)
⟨*proof*⟩

lemma *lookup-from-list-distinct*: $(x,y) \in$ *set xs* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$ *lookup* (*from-list xs*) $x =$ *Some y*
⟨*proof*⟩

lemma *lookup′-from-list-distinct*: $(x,y) \in$ *set xs* $\Longrightarrow$ *distinct* (*map fst xs*) $\Longrightarrow$ *lookup′* (*from-list xs*) $x = y$
⟨*proof*⟩

lemma *distinct-inorder*: *invar m* $\Longrightarrow$ *distinct* (*map fst* (*inorder m*))
⟨*proof*⟩

lemmas *map-empty*[*simp*]

lemma *from-list-lookup-notin*[*simp*]: $x \notin$ *set xs* $\Longrightarrow$ *lookup* (*from-list′ f xs*) $x =$ *None*
⟨*proof*⟩
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*)
⟨*proof*⟩
end

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

lift-definition *map-to-bfun* :: ′*t* ⇒ *nat* $\Rightarrow_b$ *real* **is**
λ*m n. if invar m then case lookup m n of None* ⇒ *0* | *Some r* ⇒ *r else 0*
⟨*proof*⟩

lemma *map-to-bfun-update*: *invar m* $\Longrightarrow$ *apply-bfun* (*map-to-bfun* (*update k v m*)) = (*map-to-bfun m*)(*k* := *v*)
⟨*proof*⟩

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**
$\lambda m\ n.\ if\ invar\ m \wedge n < len\ m\ then\ lookup\ m\ n\ else\ 0$
⟨*proof*⟩

lemma *map-to-bfun-update*:
assumes *invar m* $k < len\ m$
shows $apply\text{-}bfun\ (map\text{-}to\text{-}bfun\ (update\ k\ v\ m)) = (map\text{-}to\text{-}bfun\ m)(k := v)$
⟨*proof*⟩
end

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

definition *map-to-fun* :: $'t \Rightarrow nat \Rightarrow 'a$ **where**
$map\text{-}to\text{-}fun\ m\ n = (if\ invar\ m \wedge n < len\ m\ then\ lookup\ m\ n\ else\ 0)$

lemma *map-to-fun-update*: $invar\ m \Longrightarrow k < len\ m \Longrightarrow (map\text{-}to\text{-}fun\ (update\ k\ v\ m)) = (map\text{-}to\text{-}fun\ m)(k := v)$
⟨*proof*⟩

end

context *Array′* **begin**
lemma *lookup-in-list*: $invar\ m \Longrightarrow x < len\ m \Longrightarrow lookup\ m\ x \in set\ (list\ m)$
⟨*proof*⟩

definition $arr\text{-}tabulate\ f\ n = array\ (map\ f\ [0..<n])$

lemma *invar-tabulate*[*simp*]: $invar\ (arr\text{-}tabulate\ f\ n)$
⟨*proof*⟩

lemma *len-tabulate*[*simp*]: $len\ (arr\text{-}tabulate\ f\ n) = n$
⟨*proof*⟩

lemma *lookup-tabulate*[*simp*]: $i < n \Longrightarrow lookup\ (arr\text{-}tabulate\ f\ n)\ i = f\ i$
⟨*proof*⟩

lemmas *invar-update*[*intro*]

end

lemma *foldr-Cons*[*simp*]: *foldr* (#) *xs ys* = *xs*@*ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *a-lookup-some-in-A*: *s < states* ⟹ *a-lookup* (*s-lookup mdp s*) *a* = *Some* (*aa*, *b*) ⟹ *a ∈ MDP-A s*
⟨*proof*⟩

lemma *a-lookup-None-notin-A*: *s < states* ⟹ *a-lookup* (*s-lookup mdp s*) *a* = *None* ⟹ *a ∉ MDP-A s*
⟨*proof*⟩

lemma *MDP-r-zero-notin-A*: *s < states* ⟹ *a ∉ MDP-A s* ⟹ *MDP-r* (*s*, *a*) = *0* **for** *s a*
⟨*proof*⟩

lemma *MDP-r-in-A-eq*: *s < states* ⟹ *a ∈ MDP-A s* ⟹ *MDP-r* (*s*, *a*) = *fst* ((*a-lookup'* (*s-lookup mdp s*) *a*))
⟨*proof*⟩

lemma *range-MDP-r-subs*: *range* (*MDP-r*) ⊆ {*0*} ∪ {*fst* ((*a-lookup'* (*s-lookup mdp s*) *a*)) | *s a*. *s < states* ∧ *a ∈ MDP-A s*}
⟨*proof*⟩

lemma *finite-MDP-A*[*simp*]: *finite* (*MDP-A s*)
⟨*proof*⟩

lemma *finite-sa*: *finite* {(*s*,*a*). *s* < *states* ∧ *a* ∈ *MDP-A s*}
⟨*proof*⟩

lemma *finite-r-lookup*: *finite* {*fst* ((*a-lookup′* (*s-lookup mdp s*) *a*)) | *s a*. *s* < *states* ∧ *a* ∈ *MDP-A s*}
⟨*proof*⟩

lemma *bounded-MDP-r*: *bounded* (*range MDP-r*)
⟨*proof*⟩

lemma *MDP-A-ne*[*simp*]: (*MDP-A s*) ≠ {}
⟨*proof*⟩

lemma *K-closed-locale′*:
am ∈ *set* (*s-list mdp*) ⟹ (*x*, *y*, *p*) ∈ *A-Map.entries am* ⟹ (*s′*, *prob*) ∈ *set p* ⟹ *s′* <*states*
⟨*proof*⟩

lemma *MDP-K-closed*:
assumes *s*<*states*
shows *set-pmf* (*MDP-K* (*s*, *a*)) ⊆ {*0*..<*states*}
⟨*proof*⟩

lemma *MDP-K-comp-closed*: *s* ≥ *states* ⟹ *set-pmf* (*MDP-K* (*s*, *a*)) ⊆ {*states*..}
⟨*proof*⟩

lemma *MDP-A-outside*: *states* ≤ *s* ⟹ *MDP-A s* = {*0*}
⟨*proof*⟩

lemma *invar-s-lookup*: *s* < *states* ⟹ *A-Map.invar* (*s-lookup mdp s*)
⟨*proof*⟩

lemma *ne-s-lookup*: *s* < *states* ⟹ ¬ *A-Map.is-empty* (*s-lookup mdp s*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *entries-A-eq-r*: *s* < *states* ⟹ (*a*, *r*, *succs*) ∈ *A-Map.entries* (*s-lookup mdp s*) ⟹ *r* = *MDP-r* (*s*, *a*)
⟨*proof*⟩

lemma *entries-A-eq-K*: *s* < *states* ⟹ (*a*, *r*, *succs*) ∈ *A-Map.entries* (*s-lookup mdp s*) ⟹ *pmf-of-list succs* = *MDP-K* (*s*, *a*)
⟨*proof*⟩

lemma *a-inorderD*:
assumes *s* < *states* (*a*, *r*, *succs*) ∈ *A-Map.entries* (*s-lookup mdp s*)
shows *a* ∈ *MDP-A s r* = *MDP-r* (*s*, *a*) *pmf-of-list succs* = *MDP-K* (*s*, *a*)
⟨*proof*⟩

lemma *a-map-entries-lookup*: *s* < *states* ⟹ *a* ∈ *MDP-A s* ⟹ (*a*, *a-lookup′* (*s-lookup mdp s*) *a*) ∈ *A-Map.entries* (*s-lookup mdp s*)
⟨*proof*⟩

lemma *lists-are-pmfs′*: *am*∈*set* (*s-list mdp*) ⟹ (*a*,*r*,*p*)∈*A-Map.entries am* ⟹ *pmf-of-list-wf p*
⟨*proof*⟩

lemma *lists-are-pmfs″*: *am*∈*set* (*s-list mdp*) ⟹ (*a*,*rp*)∈*A-Map.entries am* ⟹ *pmf-of-list-wf* (*snd rp*)
⟨*proof*⟩

lemma *lists-are-pmfs‴*: *s* < *states* ⟹ (*a*,*rp*)∈*A-Map.entries* (*s-lookup mdp s*) ⟹ *pmf-of-list-wf* (*snd rp*)
⟨*proof*⟩

lemma *pmf-of-list-wf-mdp*:
assumes *s* < *states a* ∈ *MDP-A s*
shows *pmf-of-list-wf* (*snd* (*a-lookup′* (*s-lookup mdp s*) *a*))
⟨*proof*⟩

lemma *set-list-pmf-in-states*:
assumes *s* < *states a* ∈ *MDP-A s* (*aa*, *b*) ∈ *set* (*snd* (*a-lookup′* (*s-lookup mdp s*) *a*))
shows

aa < *states*
⟨*proof*⟩
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*⟩

lemma *pmf-of-list-expectation*:
assumes *pmf-of-list-wf ps*
shows $measure\text{-}pmf.expectation\ (pmf\text{-}of\text{-}list\ ps)\ f = (\sum (s', p) \leftarrow ps.\ p * f\ s')$
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *V-Map*: *Array-zero v-lookup v-update v-len v-array v-list v-invar*
⟨*proof*⟩

sublocale *D-Map*: *Map-by-Ordered-nat-zero d-empty d-update d-delete d-lookup d-inorder d-inv*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *MDP*: *MDP-reward* (*MDP-A*) (*MDP-K*) (*MDP-r*) *l*

⟨*proof*⟩

sublocale *MDP*: *MDP-nat-disc* (*MDP-A*) (*MDP-K*) (*MDP-r*) *l* λX. *LEAST y*. $y \in X$ *states*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *v-lookup-map-to-bfun*: *v-invar m* ⟹ *k* < *v-len m* ⟹ *v-lookup m k* = *V-Map.map-to-bfun m k*
⟨*proof*⟩

lemma *map-to-bfun-eq-fun*: *v-invar m* ⟹ *apply-bfun* (*V-Map.map-to-bfun v*) = *V-Map.map-to-fun v*
⟨*proof*⟩

lemma *map-to-fun-notin*: *D-Map.invar d* ⟹ *k* ∉ *D-Map.keys d* ⟹ *D-Map.map-to-fun d k* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *V-invar-fold*: *v-invar m* ⟹ *set xs* ⊆ {*0*..<*v-len m*} ⟹ *v-invar* (*fold* (λ*s v*. *v-update s* (*f s v*) *v*) *xs m*)
⟨*proof*⟩

lemma *V-len-fold*: *v-invar m* $\Longrightarrow$ *set xs* $\subseteq$ $\{$*0..<v-len m*$\}$ $\Longrightarrow$ *v-len* (*fold* (λ*s v. v-update s* (*f s v*) *v*) *xs m*) = *v-len m*
⟨*proof*⟩

lemma *v-len-update*: *v-invar m* $\Longrightarrow$ *j* < *v-len m* $\Longrightarrow$ *v-len* (*v-update j x m*) = *v-len m*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *invar-fold-update*: *D-Map.invar m* $\Longrightarrow$ *D-Map.invar* (*fold* (λ*s. d-update s* (*f s*)) *xs m*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *keys-*$\mathcal{L}$*-code*[*simp*]: *v-invar v* $\Longrightarrow$ *v-len v* = *states* $\Longrightarrow$ *v-len* ($\mathcal{L}$*-code v*) = *v-len v*

⟨*proof*⟩

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
⟨*proof*⟩

lemma *invar-*$\mathcal{L}$*-code*: *v-invar* $v \Longrightarrow$ *v-invar* ($\mathcal{L}$*-code* v)
⟨*proof*⟩

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)
⟨*proof*⟩

10 Code to check condition

definition *check-dist* $v\ v'\ eps = ($
let $m = eps * (1 - l) / (2 * l)$ *in*
$(\forall s < states.\ abs$ (*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* $v' = states$ $eps > 0$ $l \neq 0$
shows *check-dist* $v\ v'\ eps \longleftrightarrow$ *dist* (*V-Map.map-to-bfun* v) (*V-Map.map-to-bfun* v') $< eps * (1 - l) / (2 * l)$
⟨*proof*⟩

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)))))
⟨*proof*⟩

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*.L_a *a* (*V-Map.map-to-bfun v*) *s*) (λ*a*. *a* ∈ *MDP-A s*), *Max* ((λ*a*. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) ‘ (*MDP-A s*)))
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *d-keys-vi-find-policy-code*: *D-Map.keys* (*vi-find-policy-code v*) = {*0*..<*states*}
  ⟨*proof*⟩

lemma *vi-find-policy-code-notin*:
  assumes *s* ≥ *states* **shows** *d-lookup* (*vi-find-policy-code v*) *s* = *None*
  ⟨*proof*⟩

lemma *vi-find-policy-code-in*:
  assumes *s* < *states* **shows** ∃ *x*. *d-lookup* (*vi-find-policy-code v*) *s* = *Some x*
  ⟨*proof*⟩

lemma *vi-find-policy-code-ge*: *s* ≥ *states* ⟹ *D-Map.map-to-fun* (*vi-find-policy-code v*) *s* = *0*
  ⟨*proof*⟩

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* (λ*a*. *MDP*.L_a *a* (*V-Map.map-to-bfun v*) *s*) (λ*a*. *a* ∈ *MDP-A s*)
  ⟨*proof*⟩

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*⟩

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

lemma *v0-correct*: *v-invar v0 v-len v0* = *states*

⟨*proof*⟩

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*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

$\langle proof \rangle$

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)*
$\langle proof \rangle$

end

context *MDP-nat-disc*
begin

lemma *dist-opt-bound-*$\mathcal{L}_b$: *dist v* ν_b*-opt* $\leq$ *dist v* $(\mathcal{L}_b\ v)$ / $(1 - l)$
$\langle proof \rangle$

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* $\leq \varepsilon$
$\langle proof \rangle$

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* $(\nu_b$ *(mk-stationary-det d))* ν_b*-opt* $\leq$ *err*
$\langle proof \rangle$

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 $\in$ *{..< 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))*

⟨*proof*⟩

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

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*⟩

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 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 *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 *ℒ-GS-code* = *VI-Code.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*
⟨*proof*⟩

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*)
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *tabulate-Suc*: *j* ≤ *n′* ⟹ *update n′* (*f n′*) (*tabulate f m n′ j*) = *tabulate f m* (*Suc n′*) *j*
⟨*proof*⟩

lemma *from-list′-upt* [*code-unfold*]: *from-list′ f* [*0..<n*] = *tabulate f empty n 0*
⟨*proof*⟩

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 $d = d' \lor \neg$*is-dec-det* d *then* d *else policy-iteration* d')
⟨*proof*⟩

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* $d\ v$)
⟨*proof*⟩

lemma *policy-improvement-is-dec-det*: $d \in D_D \Longrightarrow$ *policy-improvement* $d\ v \in D_D$
⟨*proof*⟩

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*:
is-dec-det $d \Longrightarrow L$ (*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. $d = \textit{policy-step } d$, 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*
⟨*proof*⟩

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

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

lemma *finite-*D_D[*simp*]: *finite* D_D
⟨*proof*⟩

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

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* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*'s* $\Rightarrow_b$ *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

locale *MDP-ord* = *MDP-finite-type A K r l*
for *A* **and**
K :: *'s* :: {*finite*, *wellorder*} × *'a* :: {*finite*, *wellorder*} ⇒ *'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 *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-blinfun* $(inv_L\ Q) \wedge$ *nonneg-blinfun R*

lemma *is-splitting-blinD*[*dest*]:
assumes *is-splitting-blin X Q R*
shows $X = Q - R$ $invertible_L\ Q$ *nonneg-blinfun* $(inv_L\ Q)$ *nonneg-blinfun R*
⟨*proof*⟩

lemma *is-splitting-blinI*[*intro*]:
assumes $X = Q - R$ $invertible_L\ Q$ *nonneg-blinfun* $(inv_L\ Q)$ *nonneg-blinfun R*
shows *is-splitting-blin X Q R*
⟨*proof*⟩

13.2 Splitting Methods for MDPs

locale *MDP-QR* = *MDP-att-*$\mathcal{L}$ *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-splitting-blin* (*id-blinfun* $-$ $l *_R \mathcal{P}_1$ (*mk-dec-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-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-arg-max* $(\lambda d.\ inv_L\ (Q\ d)\ (r\text{-}dec_b$ (*mk-dec-det* d) $+\ R\ d\ v)\ s)\ (\lambda d.\ d \in D_D)\ d$
begin

lemma *inv-Q-mono*: $d \in D_D \Longrightarrow u \le v \Longrightarrow (inv_L\ (Q\ d))\ u \le (inv_L\ (Q\ d))\ v$
⟨*proof*⟩

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

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

lemma *Q-invertible*: $d \in D_D \Longrightarrow invertible_L\ (Q\ d)$
⟨*proof*⟩

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

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

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

lemma *QR-mono*: $d \in D_D \Longrightarrow u \le v \Longrightarrow (inv_L\ (Q\ d)\ o_L\ R\ d)\ u \le (inv_L\ (Q\ d)\ o_L\ R\ d)\ v$
⟨*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-blinfun* $-\ l *_R \mathcal{P}_1$ (*mk-dec-det* d) $= Q\ d - R\ d$
⟨*proof*⟩

13.3 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*⟩

13.4 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*⟩

lemma *L-split-le*: $d \in D_D \Longrightarrow L\text{-}split\ d\ v\ s \leq (\bigsqcup d \in D_D.\ norm\ (inv_L\ (Q\ d))) * r_M + norm\ v$
⟨*proof*⟩

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

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

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

lemma *$\mathcal{L}_b$-split-fix*[*simp*]: $\mathcal{L}_b\text{-}split\ \nu_b\text{-}opt = \nu_b\text{-}opt$
⟨*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* ∈ 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* ∈ D_D
shows *dist* (*L-split* *d* *v*) (*L-split* *d* *u*) ≤ *QR-disc* ∗ *dist* *v* *u*
⟨*proof*⟩

lemma *argmax-policy-error-bound*:
assumes *am*: ⋀*s*. *is-arg-max* (λ*d*. *L* (*mk-dec-det* *d*) ($\mathcal{L}_b$ *v*) *s*) (λ*d*. *d* ∈ D_D) *d*
shows (*1* − *l*) ∗ *dist* (ν_b (*mk-stationary-det* *d*)) ($\mathcal{L}_b$ *v*) ≤ *l* ∗ *dist* ($\mathcal{L}_b$ *v*) *v*
⟨*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*: ⋀*s*. *is-arg-max* (λ*d*. *L-split* *d* ($\mathcal{L}_b$*-split* *v*) *s*) (λ*d*. *d* ∈ D_D) *d*
shows *dist* (ν_b (*mk-stationary-det* *d*)) ν_b*-opt* < *eps*
⟨*proof*⟩

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* = (∑ *i*. (*id-blinfun* − *f*)^^*i*)
⟨*proof*⟩

lemma *blinfun-le-trans*: *blinfun-le* *X* *Y* ⟹ *blinfun-le* *Y* *Z* ⟹ *blinfun-le* *X* *Z*
⟨*proof*⟩

lemma *blinfun-leI*[*intro*]: (⋀*v*. *v* ≥ *0* ⟹ *blinfun-apply* *C* *v* ≤ *blin-*

fun-apply D v) $\Longrightarrow$ *blinfun-le C D*
⟨*proof*⟩

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 ^^ n) (D ^^ n)
⟨*proof*⟩

lemma *blinfun-le-iff*: *blinfun-le X Y* $\longleftrightarrow$ ($\forall v \geq 0.\ X\ v \leq Y\ v$)
⟨*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-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*⟩

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-bfun v j* = *apply-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-bfun v j* = *apply-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 $X\ v\ i = X\ u\ i$
⟨*proof*⟩

lemma *upper-triangularI*[*intro*]:
fixes X :: ($'a$::*linorder* $\Rightarrow_b$ *real*) $\Rightarrow_L$ ($'a \Rightarrow_b$ *real*)
assumes $\bigwedge i\ u\ v.\ (\bigwedge j.\ i \le j \Longrightarrow$ *apply-bfun* $v\ j =$ *apply-bfun* $u\ j)$ $\Longrightarrow X\ v\ i = X\ u\ i$
shows *upper-triangular-blin* X
⟨*proof*⟩

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 $X\ v\ i = X\ u\ i$
⟨*proof*⟩

lemma *strict-imp-upper-triangular-blin*: *strict-upper-triangular-blin* X $\Longrightarrow$ *upper-triangular-blin* X
⟨*proof*⟩

definition *lower-triangular-blin* :: ($'a$::*linorder* $\Rightarrow_b$ *real*) $\Rightarrow_L$ ($'a \Rightarrow_b$ *real*) $\Rightarrow$ *bool* **where**
lower-triangular-blin $X \longleftrightarrow (\forall u\ v\ i.\ (\forall j \le i.$ *apply-bfun* $v\ j =$ *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 $X \longleftrightarrow (\forall u\ v\ i.\ (\forall j < i.$ *apply-bfun* $v\ j =$ *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** $\bigwedge j.\ i \ge j \Longrightarrow v\ j = u\ j$
shows $X\ v\ i = X\ u\ i$
⟨*proof*⟩

lemma *lower-triangularI*[*intro*]:
fixes X :: ($'a$::*linorder* $\Rightarrow_b$ *real*) $\Rightarrow_L$ ($'a \Rightarrow_b$ *real*)
assumes $\bigwedge i\ u\ v.\ (\bigwedge j.\ i \ge j \Longrightarrow$ *apply-bfun* $v\ j =$ *apply-bfun* $u\ j)$ $\Longrightarrow X\ v\ i = X\ u\ i$
shows *lower-triangular-blin* X
⟨*proof*⟩

lemma *strict-lower-triangularI*[*intro*]:
fixes X :: ($'a$::*linorder* $\Rightarrow_b$ *real*) $\Rightarrow_L$ ($'a \Rightarrow_b$ *real*)
assumes $\bigwedge i\ u\ v.\ (\bigwedge j.\ i > j \Longrightarrow$ *apply-bfun* $v\ j =$ *apply-bfun* $u\ j)$ $\Longrightarrow X\ v\ i = X\ u\ i$
shows *strict-lower-triangular-blin* X
⟨*proof*⟩

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 $X\ v\ i = X\ u\ i$
⟨*proof*⟩

lemma *strict-imp-lower-triangular-blin*: *strict-lower-triangular-blin* X $\Longrightarrow$ *lower-triangular-blin* X
⟨*proof*⟩

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*⟩

lemma *bounded-mult*:
assumes *bounded* $((f :: 'c \Rightarrow real)\ `\ X)$ *bounded* $(g\ `\ X)$
shows *bounded* $((\lambda x.\ f\ x * g\ x)\ `\ X)$
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lift-definition $\mathcal{P}_L$ :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow_b real) \Rightarrow_L nat \Rightarrow_b real$ **is** $\lambda d\ (v :: nat \Rightarrow_b real).$
Bfun ($\lambda s.$ ($\mathcal{P}_1$ (*mk-dec-det* d) (*bfun-if* $(\lambda s'.\ s' \geq s)$ *0* v) s))
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

lemma $\mathcal{P}_L$*-rep-eq'*: $\mathcal{P}_L$ *d v s* = $\mathcal{P}_1$ (*mk-dec-det d*) (*bfun-if* ((≤) *s*) *0* *v*) *s*
⟨*proof*⟩

lemma *apply-bfun-plus*: *apply-bfun f a* + *apply-bfun g a* = *apply-bfun* (*f* + *g*) *a*
⟨*proof*⟩

lemma $\mathcal{P}_1$*-sum-lower-upper*: $\mathcal{P}_1$ (*mk-dec-det d*) = $\mathcal{P}_L$ *d* + $\mathcal{P}_U$ *d*
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma $\mathcal{P}_L$*-le-*$\mathcal{P}_1$: *0* ≤ *v* ⟹ $\mathcal{P}_L$ *d v* ≤ $\mathcal{P}_1$ (*mk-dec-det d*) *v*
⟨*proof*⟩

lemma $\mathcal{P}_U$*-le-*$\mathcal{P}_1$: *0* ≤ *v* ⟹ $\mathcal{P}_U$ *d v* ≤ $\mathcal{P}_1$ (*mk-dec-det d*) *v*
⟨*proof*⟩

lemma $\mathcal{P}_U$*-indep*: *d s* = *d' s* ⟹ $\mathcal{P}_U$ *d v s* = $\mathcal{P}_U$ *d' v s*
⟨*proof*⟩
lemma $\mathcal{P}_L$*-indep*: *d s* = *d' s* ⟹ $\mathcal{P}_L$ *d v s* = $\mathcal{P}_L$ *d' v s*
⟨*proof*⟩

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 $\mathcal{P}_U\ \ d\ v\ s = \mathcal{P}_U\ \ d'\ v'\ s$
⟨*proof*⟩

lemma $\mathcal{P}_L$*-indep2*: $d\ s = d'\ s \Longrightarrow (\bigwedge s'.\ s' < s \Longrightarrow$ *apply-bfun* $v\ s' =$ *apply-bfun* $v'\ s') \Longrightarrow \mathcal{P}_L\ \ d\ v\ s = \mathcal{P}_L\ \ d'\ v'\ s$
⟨*proof*⟩

lemma $\mathcal{P}_1$*-indep*: $d\ s = d'\ s \Longrightarrow \mathcal{P}_1\ \ d\ v\ s = \mathcal{P}_1\ \ d'\ v\ s$
⟨*proof*⟩

lemma $\mathcal{P}_U$*-upper*: *upper-triangular-blin* ($\mathcal{P}_U\ \ d$)
⟨*proof*⟩

lemma $\mathcal{P}_L$*-strict-lower*: *strict-lower-triangular-blin* ($\mathcal{P}_L\ \ d$)
⟨*proof*⟩

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)
⟨*proof*⟩

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

abbreviation *r-det*$_b$ $d \equiv$ *r-dec*$_b$ (*mk-dec-det* d)

definition *GS-inv* $d\ v =$ *inv*$_L$ (*Q-GS* d) (*r-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) $= (\sum k.\ (l *_R (\mathcal{P}_L\ \ d))\ \widehat{\ \ }\ k)$
⟨*proof*⟩

This recursive definition mimics the computation of the GS iteration.

lemma *GS-inv-rec*: *GS-inv* $d\ v =$ *r-det*$_b$ $d + l *_R (\mathcal{P}_U\ \ d\ v + \mathcal{P}_L\ \ d$ (*GS-inv* $d\ v$))
⟨*proof*⟩

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

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

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 $s \leq k$
and $d \in D_D$
shows *is-arg-max* ($\lambda d.$ *GS-inv d v s*) ($\lambda d.\ d \in D_D$) (d ($k := a$))
⟨*proof*⟩

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 $s \leq k$
and $d \in D_D$
shows *is-arg-max* ($\lambda d.$ *GS-inv d v s*) ($\lambda d.\ d \in D_D$) d
⟨*proof*⟩

lemma *norm-*$\mathcal{P}_L$*-pow*: *norm* (($\sum k.$ ($l *_R \mathcal{P}_L\ d$) ^^ k)) $\leq 1 / (1-l)$
⟨*proof*⟩

lemma *summable-disc-*$\mathcal{P}_L$: *summable* ($\lambda i.$ (($l *_R \mathcal{P}_L\ d$)^^i))
⟨*proof*⟩

lemma *norm-*$\mathcal{P}_L$*-pow-elem*: *norm* (($\sum k.$ ($l *_R \mathcal{P}_L\ d$) ^^ k) v) $\leq$ *norm v* / ($1-l$)
⟨*proof*⟩

lemma *norm-Q-GS*: *norm* (inv_L (*Q-GS d*) v) $\leq$ *norm v* / ($1-l$)
⟨*proof*⟩

lemma *norm-GS-inv-le*: *norm* (*GS-inv d v*) $\leq$ (r_M + l * *norm v*) / ($1 - l$)
⟨*proof*⟩

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* ($\lambda s'.\ s \leq s'$) v (*GS-inv d v*)))) s
⟨*proof*⟩

13.7 Maximizing Decision Rule for GS

lemma *ex-GS-inv-arg-max*: $\exists$ a. *is-arg-max* ($\lambda a.$ *GS-inv* (d(s:= a)) v s) ($\lambda a.\ a \in A\ s$) a
⟨*proof*⟩

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* $\in D_D$
⟨*proof*⟩

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*) (λa. $a \in A\ n$) *a*)
⟨*proof*⟩

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

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*⟩

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

sublocale *GS*: *MDP-QR A K r l Q-GS R-GS*
⟨*proof*⟩

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*⟩

lemma *GS-QR-disc-le-disc*: *GS.QR-disc* $\leq l$
⟨*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 *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*⟩

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

function *gs-iteration* :: *real* ⇒ (*nat* $\Rightarrow_b$ *real*) ⇒ (*nat* $\Rightarrow_b$ *real*) **where**
gs-iteration eps v =
(*if 2* ∗ *l* ∗ *dist v* (*GS*.$\mathcal{L}_b$*-split v*) < *eps* ∗ (*1* − *l*) ∨ *eps* ≤ *0 then GS*.$\mathcal{L}_b$*-split v else gs-iteration eps* (*GS*.$\mathcal{L}_b$*-split v*))
⟨*proof*⟩
termination
⟨*proof*⟩

13.10 Optimality

lemma *THE-fix-GS*: (*THE v. GS*.$\mathcal{L}_b$*-split v* = *v*) = ν_b*-opt*
⟨*proof*⟩

lemma *contraction-$\mathcal{L}$-split-dist*: (*1* − *l*) ∗ *dist v* ν_b*-opt* ≤ *dist v* (*GS*.$\mathcal{L}_b$*-split v*)
⟨*proof*⟩

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*⟩

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

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*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma $\mathcal{L}_b$*-split-GS-iter*:
assumes $\bigwedge s'.\ s' < s \Longrightarrow v'\ s' =$ *GS*.$\mathcal{L}_b$*-split* $v\ s'$ $\bigwedge s'.\ s' \geq s \Longrightarrow v'\ s' = v\ s'$
shows *GS*.$\mathcal{L}_b$*-split* $v\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v'\ s)$
⟨*proof*⟩

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-pmf.expectation* $(K\ (s,\ a))\ v)))$ (*Suc* s))
⟨*proof*⟩
termination
⟨*proof*⟩

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

lemma *GS-rec-upto-ge*:
assumes $s' \geq n$
shows *GS-rec-upto* $n\ v\ s\ s' = v\ s'$
⟨*proof*⟩

lemma *GS-rec-upto-less*:
assumes $s > s'$
shows *GS-rec-upto* $n\ v\ s\ s' = v\ s'$
⟨*proof*⟩

lemma *GS-rec-upto-eq*:
assumes $s < n$
shows *GS-rec-upto* $n\ v\ s\ s = (\bigsqcup a \in A\ s.\ L_a\ a\ v\ s)$
⟨*proof*⟩

lemma *GS-rec-upto-Suc*:
assumes $s' < n$
shows *GS-rec-upto* (*Suc* n) $v\ s\ s'$ = *GS-rec-upto* $n\ v\ s\ s'$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *GS-rec-upto-correct*:
assumes $s < n$
shows *GS*.$\mathcal{L}_b$*-split v s* = *GS-rec-upto n v 0 s*
⟨*proof*⟩

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-bfun v s* = *0*
shows *GS*.$\mathcal{L}_b$*-split v s* = *GS-rec-upto states v 0 s*
⟨*proof*⟩
end

context *MDP-Code* **begin**

function *GS-iter-aux* :: *nat* ⇒ *'tv* ⇒ *real* ⇒ (*'tv* × *real*) **where**
GS-iter-aux s v md = (
if s ≥ *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*)))
⟨*proof*⟩
termination
⟨*proof*⟩

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* ∧ *v-invar* (*fst* (*GS-iter-aux n v md*))
⟨*proof*⟩

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* ((λ*s. abs* (*MDP.GS-rec-upto states* (*V-Map.map-to-bfun v*) *n s* − (*V-Map.map-to-bfun v*) *s*)) ‘ {*n..<states*}))
⟨*proof*⟩

lemma *invar-GS-iter-aux*: *v-len v* = *states* ⟹ *v-invar v* ⟹ *v-invar* (*fst* (*GS-iter-aux n v md*))
⟨*proof*⟩

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

lemma *len-GS-iter-aux*[*simp*]: *v-invar v* ⟹ *v-len v* = *states* ⟹ *v-len* (*fst* (*GS-iter-aux n v md*)) = *states*
⟨*proof*⟩

lemma *len-GS-iter*[*simp*]: *v-invar v* ⟹ *v-len v* = *states* ⟹ *v-len* (*fst* (*GS-iter v*)) = *v-len v*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *v-len-invar-find-policy-code-aux-upt*: $n \leq j \Longrightarrow$ *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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *find-policy-code-invar*: *D-Map.invar* (*find-policy-code v*)
⟨*proof*⟩

lemma *find-policy-code-notin*:
assumes *s* ≥ *states* **shows** *d-lookup* (*find-policy-code v*) *s* = *None*
⟨*proof*⟩

lemma *find-policy-code-in*:

assumes *s* < *states* **shows** ∃ *x*. *d-lookup* (*find-policy-code v*) *s* = *Some x*
⟨*proof*⟩

lemma *GS-iter-aux-fold*: *fst* (*GS-iter-aux s v md*) = *fold* (λ*s v*. *v-update s* (ℒ*-GS-code* (*s-lookup mdp s*) *v*) *v*) [*s*..<*states*] *v*
⟨*proof*⟩

lemma *find-policy-state-code-aux′-eq-*ℒ*-GS-code*:
assumes *v-len v* = *states v-invar v s* < *states*
shows *snd* (*find-policy-state-code-aux′ v s*) = ℒ*-GS-code* (*s-lookup mdp s*) *v*
⟨*proof*⟩

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*⟩

lemma *GS-rec-upto-Suc*: *MDP.GS-rec-upto* (*Suc n*) *v 0* = (*MDP.GS-rec-upto n v 0*)(*n* := (⨆ *a*∈*MDP-A n*. *MDP.*L_a *a* (*MDP.GS-rec-upto n v 0*) *n*))
⟨*proof*⟩

lemma *keys-fst-find-policy-code-aux-upt*: *s* ≤ *states* ⟹ *D-Map.keys* (*fst* (*find-policy-code-aux-upt v s*)) = {*0*..<*s*}
⟨*proof*⟩

lemma *keys-fst-find-policy-code-aux*: *D-Map.keys* (*fst* (*find-policy-code-aux v*)) = {*0*..<*states*}
⟨*proof*⟩

lemma *find-policy-code-ge*: *s* ≥ *states* ⟹ *D-Map.map-to-fun* (*find-policy-code v*) *s* = *0*
⟨*proof*⟩

lemma *find-policy-code-aux-upt-zero*[*simp*]: *find-policy-code-aux-upt v 0* = (*d-empty*, *v*)
⟨*proof*⟩

lemma *GS-rec-upto-zero*[*simp*]: *MDP.GS-rec-upto 0 v n* = *v*
⟨*proof*⟩

lemma *keys-find-policy-code-aux-upt*:*n* < *states* ⟹ *v-invar v* ⟹ *v-len v* = *states* ⟹ *v-len* (*snd* (*find-policy-code-aux-upt v n*)) = *states*
⟨*proof*⟩

lemma *split-eq-GS-rec-upto-Sup*:

$MDP.GS.\mathcal{L}_b$*-split v s* = ($\bigsqcup$ *a*∈*MDP-A s. MDP.*L_a *a* (*MDP.GS-rec-upto s* (*apply-bfun v*) *0*) *s*)
⟨*proof*⟩

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* ∈ *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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *snd-find-policy-code-aux-upt-correct*:
assumes *v-len v* = *states v-invar v n* ≤ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *GS-inv-GS-least-eq-split*: *MDP.GS-inv* (*MDP.d-GS-least v*) *v* = $MDP.GS.\mathcal{L}_b$*-split v*
⟨*proof*⟩

lemma *is-arg-max-*L_a*-GS-inv-d-GS-least*:

is-arg-max ($\lambda a.$ *MDP*.L_a *a* (*MDP.GS-rec-upto s* (*apply-bfun v*) *0*) *s*) ($\lambda a.$ $a \in$ *MDP-A s*) *a*
$\longleftrightarrow$ *is-arg-max* ($\lambda a.$ (*MDP.GS-inv* ((*MDP.d-GS-least v*)(*s* := *a*)) *v s*)) ($\lambda a.$ $a \in$ *MDP-A s*) *a*
⟨*proof*⟩

lemma *d-GS-least-ge*: *s* $\geq$ *states* $\Longrightarrow$ *MDP.d-GS-least* (*V-Map.map-to-bfun v*) *s* = *0*
⟨*proof*⟩

lemma *fst-find-policy-code-aux-upt-correct*:
assumes *v-len v* = *states v-invar v n* $\leq$ *states s* < *n*
shows *D-Map.map-to-fun* (*fst* (*find-policy-code-aux-upt v 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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*⟩

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*⟩

end

lemma *inorder-empty*: *Tree2.inorder am* = [] ⟹ *am* = ⟨⟩
  ⟨*proof*⟩

context *MDP-nat-disc*
begin

lemma *dist-opt-bound-*$\mathcal{L}_b$*-split*: *dist v* ν_b*-opt* ≤ *dist v* (*GS.*$\mathcal{L}_b$*-split v*) / (*1* − *l*)
  ⟨*proof*⟩

lemma *cert-*$\mathcal{L}_b$*-split*:
  assumes ε ≥ *0 dist v* (*GS.*$\mathcal{L}_b$*-split v*) / (*1* − *l*) ≤ ε
  shows *dist v* ν_b*-opt* ≤ ε
  ⟨*proof*⟩

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

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

$(err,\ d\text{-}GS\text{-}least\ v'))$

lemma $\mathcal{L}_b$-*split-eq-L-opt*: $GS.\mathcal{L}_b\text{-}split\ v = GS.L\text{-}split\ (d\text{-}GS\text{-}least\ v)\ v$
⟨*proof*⟩

lemma *L-split-fix-ν*:
assumes $d \in D_D$
assumes $GS.L\text{-}split\ d\ v = v$
shows $v = \nu_b\ (mk\text{-}stationary\text{-}det\ d)$
⟨*proof*⟩

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*⟩

end

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

lemma
assumes $v\text{-}len\ v = states\ v\text{-}invar\ v$
shows $D\text{-}Map.map\text{-}to\text{-}fun\ (snd\ (gs\text{-}policy\text{-}bound\text{-}error\text{-}code\ v)) = snd\ (MDP.gs\text{-}policy\text{-}bound\text{-}error\ (V\text{-}Map.map\text{-}to\text{-}bfun\ v))$
⟨*proof*⟩

lemma
assumes $v\text{-}len\ v = states\ v\text{-}invar\ v$
shows $(fst\ (gs\text{-}policy\text{-}bound\text{-}error\text{-}code\ v)) = fst\ (MDP.gs\text{-}policy\text{-}bound\text{-}error\ (V\text{-}Map.map\text{-}to\text{-}bfun\ v))$
⟨*proof*⟩

end

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

$IArray.sub\ \lambda n\ x\ arr.\ IArray\ ((IArray.list\text{-}of\ arr)[n{:=}\ x])\ IArray.length\ IArray\ IArray.list\text{-}of\ \lambda\text{-}.\ True$

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*
⟨*proof*⟩

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* × *'a* :: *countable* ⇒ *'s pmf* **and** *r l arb-act*
begin

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

lemma *cSUP-minus*:
assumes $X \neq \{\}$ *bdd-above* $(f`X)$
shows $(\bigsqcup x \in X.\ f\ x - c) = (\bigsqcup x \in X.\ f\ x) - (c{::}real)$
⟨*proof*⟩

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*⟩

15.2 Optimization of the Value Function over Multiple Steps

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

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*: $dist\ (U_b\ m\ v)\ (U_b\ m\ u) \leq l\ \widehat{}\ m * dist\ v\ u$
⟨*proof*⟩

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

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

lemma U_b-*mono*:
assumes $v \leq u$
shows $U_b\ m\ v \leq U_b\ m\ u$
⟨*proof*⟩

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

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

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*⟩

lemma U_b-*ge*: $d \in D_R \Longrightarrow U_b\ m\ u \geq \nu_b$-*fin* (*mk-stationary* d) m + (($l *_R \mathcal{P}_1\ d$) ^^ m) u
⟨*proof*⟩

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

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*⟩

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

15.4 Computing the Bellman Operator over Multiple Steps

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

lemma *L-pow-eq*:
fixes d **defines** $d' \equiv$ *mk-dec-det* d
assumes ν-*improving* $v\ d'$
shows *L-pow* $v\ d\ m = v + (\sum i < m.\ ((l *_R \mathcal{P}_1\ d')$^^$i))\ (B_b\ v)$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *find-policy'-is-dec-det*: *is-dec-det* (*find-policy'* v)
⟨*proof*⟩

lemma *find-policy'-improving*: ν-*improving* v (*mk-dec-det* (*find-policy'* v))
⟨*proof*⟩

lemma *L-pow-eq-W'*: *L-pow* v (*find-policy'* v) m = W (*mk-dec-det* (*find-policy'* v)) m v

⟨*proof*⟩

lemma $\mathcal{L}_b$-*W-ge*:
assumes $u \leq \mathcal{L}_b\ u$ ν-*improving* $u\ d$
shows $W\ d\ m\ u \leq \mathcal{L}_b\ (W\ d\ m\ u)$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

15.5 The Modified Policy Iteration Algorithm

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

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

fun *mpi* :: *nat* ⇒ ((*'s* ⇒ *'a*) × (*'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*
⟨*proof*⟩

lemma *mpi-pol-Suc*: *mpi-pol* (*Suc n*) = *find-policy'* (*mpi-val* (*Suc n*))
⟨*proof*⟩

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

lemma ν-*improving-mpi-pol*: ν-*improving* (*mpi-val n*) (*mk-dec-det* (*mpi-pol n*))
⟨*proof*⟩

lemma *mpi-val-zero*[*simp*]: *mpi-val 0 = v0*
⟨*proof*⟩

lemma *mpi-val-Suc*: *mpi-val* (*Suc n*) = *L-pow* (*mpi-val n*) (*mpi-pol n*) (*Suc* (*m n* (*mpi-val n*)))
⟨*proof*⟩

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

Value Iteration is a special case of MPI where ∀ *n v*. *m n v* = *0*.

lemma *mpi-includes-value-it*:
assumes ∀ *n v*. *m n v* = *0*
shows *mpi-val* (*Suc n*) = $\mathcal{L}_b$ (*mpi-val n*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dist-*ν_b*-opt-n*: *dist* (*w n*) ν_b*-opt* ≤ *l*^*n* ∗ *dist v0* ν_b*-opt*
⟨*proof*⟩

lemma *w-conv*: *w* ⟶ ν_b*-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-conv*:
assumes *v0* ≤ $\mathcal{L}_b$ *v0*
shows *mpi-val* ⟶ ν_b*-opt* **and** $\bigwedge$*n*. *mpi-val n* ≤ *mpi-val* (*Suc n*)
⟨*proof*⟩

15.7 ϵ-Optimality

This gives an upper bound on the error of MPI.

lemma *mpi-pol-eps-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-pol-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-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*⟩
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*)))
($\lambda n.\ m\ (Suc\ n)$))
  ⟨*proof*⟩

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-pow v* (*find-policy′ v*) (*Suc* (*m 0*
v))) ($\lambda a.\ m\ (Suc\ a)$) *n*
  ⟨*proof*⟩

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

$\langle proof \rangle$

lemma *termination-mpi-algo*:
assumes $eps > 0$ $d \in D_D$ $v \le \mathcal{L}_b\ v$
shows *mpi-algo-dom* (d, v, m)
$\langle proof \rangle$

abbreviation *mpi-alg-rec 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*)))

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*
$\langle proof \rangle$

lemma *mpi-algo-def′′*:
assumes $d \in D_D$ $v \le \mathcal{L}_b\ v$ $eps > 0$
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-pow v′ d′* ((*m 0 v*))) ($\lambda n.$ *m* (*Suc n*)))
$\langle proof \rangle$

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 v m* (*LEAST n.* $2 * l *$ *dist* (*mpi-val v m n*) ($\mathcal{L}_b$ (*mpi-val v m n*)) $< eps * (1 - l)$)
$\langle proof \rangle$

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

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

lift-definition *v0-mpi*$_b$ :: *'s* $\Rightarrow_b$ *real* **is** *v0-mpi*
⟨*proof*⟩

lemma *v0-mpi*$_b$*-le-*$\mathcal{L}_b$: *v0-mpi*$_b$ $\leq$ $\mathcal{L}_b$ *v0-mpi*$_b$
⟨*proof*⟩

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

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

lemma *mpi-user-eq*:
assumes *eps* > *0*
shows *mpi-user eps* = *mpi-alg-rec eps* (λx. *arb-act* (*A x*)) *v0-mpi*$_b$
⟨*proof*⟩

lemma *mpi-user-opt*:
assumes *eps* > *0*
shows *dist* (ν_b (*mk-stationary-det* (*fst* (*mpi-user eps n*)))) ν_b*-opt* < *eps*
⟨*proof*⟩
end

end
theory *MPI-Code*
imports
Code-Setup
../Modified-Policy-Iteration
HOL−Library.Code-Target-Numeral
begin

sublocale *MDP-nat-disc* $\subseteq$ *MDP-MPI*
⟨*proof*⟩

context *MDP-Code* **begin**

definition *d0* = *D-Map.from-list'* (λs. *fst* (*hd* (*a-inorder* (*s-lookup mdp s*)))) [*0..<states*]

definition *r-min-code* =
min 0 (*MIN s* $\in$ *set* [*0..<states*]. *MIN* (-, *r*, -) $\in$ *set* (*a-inorder* (*s-lookup mdp s*)). *r*)

definition *v0-code* = *V-Map.arr-tabulate* (λs. *r-min-code* / (*1* − *l*)) *states*

definition *d0-code* = *D-Map.from-list′* (*λs. fst* (*hd* (*a-inorder* (*s-lookup mdp s*)))) [*0..<states*]

definition *find-policy-L-code v* =
  fold (*λs* (*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 (*λs* (*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* (*λx* (*a1*, *a2*). (*f x a1*, *g x a2*)) *xs* (*z1*, *z2*) =
    (*fold f xs z1*, *fold g xs z2*)
  ⟨*proof*⟩

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*⟩

lemma *a-lookup-entries*: *A-Map.invar m* ⟹ *kv* ∈ *A-Map.entries m* ⟹ *a-lookup′ m* (*fst kv*) = *snd kv*
  ⟨*proof*⟩

lemma *a-inorder-eq-MDP-A*: *x* < *states* ⟹ *fst ‘ set* (*a-inorder* (*s-lookup mdp x*)) = *MDP-A x*
  ⟨*proof*⟩

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*⟩

definition *L-code d v* =
 V-Map.arr-tabulate (*λs.* 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* (⋀*s*. *s* < *states*

$\Longrightarrow$ *d-lookup′ d s* ∈ *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*
⟨*proof*⟩

lemma *L-code-invar*: *v-invar* (*L-code d v*)
⟨*proof*⟩

lemma *L-code-keys*:
assumes *v-len v* = *states v-invar v*
D-Map.keys d = *MDP.state-space D-Map.invar d* (⋀*s*. *s* < *states* $\Longrightarrow$ *d-lookup′ d s* ∈ *MDP-A s*)
shows *v-len* (*L-code d v*) = *states*
⟨*proof*⟩

definition *L-pow-code v d m* = (*L-code d* ^^ *m*) *v*

lemma *L-pow-code-Suc*: *L-pow-code v d* (*Suc m*) = *L-code d* (*L-pow-code v d m*)
⟨*proof*⟩

lemma *L-code-to-bfun*:
assumes *v-len v* = *states v-invar v*
D-Map.keys d = *MDP.state-space D-Map.invar d* (⋀*s*. *s* < *states* $\Longrightarrow$ *d-lookup′ d s* ∈ *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*⟩

lemma *L-pow-code-correct*:
assumes *v-len v* = *states v-invar v*
D-Map.keys d = *MDP.state-space D-Map.invar d* (⋀*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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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* ∈ *MDP-A s*)
for *s v v' d*
⟨*proof*⟩

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* ∈ *MDP-A s*)
v-len v = *states v-len v'* = *states v-invar v v-invar v'*
⟨*proof*⟩

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* ∈ *MDP-A s*) *v-len v* = *states v-len v'* = *states v-invar v v-invar v'*
for *v v' d i m*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

definition *MPI-code eps m* =
(*if eps* ≤ *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*)
⟨*proof*⟩

lemma *Min-cong*: *finite X* ⟹ $X \neq \{\}$ ⟹ ($\bigwedge x.\ x \in X \Longrightarrow f\ x = g\ x$) ⟹ (*MIN* $x \in X.\ f\ x$) = (*MIN* $x \in X.\ g\ x$)
⟨*proof*⟩

lemma *r-min-code-correct*:
assumes *states* > *0*
shows *r-min-code* = *MDP.r-min*
⟨*proof*⟩

lemma *v0-code-correct*: *s* < *states* ⟹ *v-lookup v0-code s* = ($MDP.v0\text{-}mpi_b$ *s*)
⟨*proof*⟩

lemma *v0-invar*: *v-invar v0-code*
⟨*proof*⟩

lemma *v0-keys*: *v-len v0-code* = *states*
⟨*proof*⟩

lemma L_a*-indep-notin*:
assumes *s* < *states*
shows $MDP.L_a$ *d* (*apply-bfun v*) *s* = $MDP.L_a$ *d* (*bfun-if* (λ*s*. *s* < *states*) *v u*) *s*
⟨*proof*⟩

lemma $\mathcal{L}_b$*-indep-notin*: *s* < *states* ⟹ $MDP.\mathcal{L}_b$ *v s* = $MDP.\mathcal{L}_b$ (*bfun-if*

$(\lambda s.\ s < states)\ v\ u)\ s$
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩
end

global-interpretation *MPI-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 *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*
⟨*proof*⟩

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*⟩

lemma *vec-add*: *Matrix.vec n* (λ*i*. *f i* + *g i*) = *Matrix.vec n f* + *Matrix.vec n g*
⟨*proof*⟩

lemma *vec-scale*: *Matrix.vec n* (λ*i*. *r* ∗ *f i*) = *r* $\cdot_v$ (*Matrix.vec n f*)
⟨*proof*⟩

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*⟩

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*⟩

lift-definition *mat-to-blinfun* :: *real mat* ⇒ (*nat* $\Rightarrow_b$ *real*) $\Rightarrow_L$ (*nat* $\Rightarrow_b$ *real*) **is** *bfun-mat*
⟨*proof*⟩

lemma *mat-to-blinfun-mult*: *mat-to-blinfun m* (v :: *nat* $\Rightarrow_b$ *real*) i = *bfun-mat m v i*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mat-scale-one*[*simp*]: *1* $\cdot_m$ (*m*::*real mat*) = *m*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *blinfun-to-mat-zero*[*simp*]: *blinfun-to-mat n m 0* = 0_m *n m*
⟨*proof*⟩

lemma *blinfun-to-mat-scale*: (*blinfun-to-mat n m* (r $*_R$ v) :: *real mat*) = r $\cdot_m$ (*blinfun-to-mat n m v*)
⟨*proof*⟩

lemma *Bfun-if*[*simp*]: *apply-bfun* (*bfun.Bfun* (λk. *if b k then a else c*)) = (λk. *if b k then a else c*)
⟨*proof*⟩

lemma *blinfun-to-mat-correct*: *blinfun-to-mat* (*dim-row v*) (*dim-col v*) (*mat-to-blinfun v*) = *v*
⟨*proof*⟩

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

lemma *nonneg-mult-vec-mono*:
assumes 0_m (*dim-row X*) (*dim-col X*) ≤ *X v* ≤ *u dim-vec v* = *dim-col X*
shows *X* $*_v$ (*v* :: *real vec*) ≤ *X* $*_v$ *u*
⟨*proof*⟩

unbundle *no vec-syntax*

lemma *nonneg-blinfun-mat*: *nonneg-blinfun* (*mat-to-blinfun M*) ⟷ (0_m (*dim-row M*) (*dim-col M*) ≤ *M*)

⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inverse-mats-unique*:
assumes *inverse-mats C D inverse-mats C E* **shows** $D = E$
⟨*proof*⟩

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*⟩

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*⟩

lemma *invertible-inverse-mats*:
assumes *invertible-mat M*
shows *inverse-mats M* (*inverse-mat M*)
⟨*proof*⟩

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*⟩

lemma *Max-geI*:
assumes *finite X* (*y*::-:: *linorder*) $\in X$ $x \le y$ **shows** $x \le$ *Max X*
⟨*proof*⟩

lift-definition *vec-to-bfun* :: *real vec* ⇒ (*nat* $\Rightarrow_b$ *real*) **is**
λv i. if i < dim-vec v then v $ i else 0
⟨*proof*⟩

lemma *vec-to-bfun-to-vec*[*simp*]: *bfun-to-vec* (*dim-vec v*) (*vec-to-bfun v*) = *v*
⟨*proof*⟩

lemma *bfun-to-vec-to-bfun*[*simp*]: *vec-to-bfun* (*bfun-to-vec m v*) = *bfun-if* (*λi. i < m*) *v 0*
⟨*proof*⟩

lemma *bfun-if-vec-to-bfun*[*simp*]: (*bfun-if* (*λi. i < dim-vec v*) (*vec-to-bfun v*) *0*) = *vec-to-bfun v*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *matrix-eqI*:
fixes *A* :: *real mat*
assumes ⋀*v. v* ∈ *carrier-vec m* ⟹ *A* $*_v$ *v* = *B* $*_v$ *v A* ∈ *carrier-mat n m B* ∈ *carrier-mat n m*
shows *A* = *B*
⟨*proof*⟩

lemma *blinfun-to-mat-in-carrier*[*simp*]: *blinfun-to-mat m p A* ∈ *carrier-mat m p*
⟨*proof*⟩

lemma *blinfun-to-mat-dim-col*[*simp*]: *dim-col* (*blinfun-to-mat m p A*) = *p*
⟨*proof*⟩

lemma *blinfun-to-mat-dim-row*[*simp*]: *dim-row* (*blinfun-to-mat m p A*) = *m*
⟨*proof*⟩

lemma *bfun-to-vec-carrier*[*simp*]: *bfun-to-vec m v* ∈ *carrier-vec m*
⟨*proof*⟩

lemma *vec-cong*: ($\bigwedge i.\ i < n \Longrightarrow f\ i = g\ i$) $\Longrightarrow$ *vec n f* = *vec n g*
⟨*proof*⟩

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*⟩

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*⟩

lemma *invertible-mat-dims*: *invertible-mat A* $\Longrightarrow$ *dim-col A* = *dim-row A*
⟨*proof*⟩

lemma *invertible-mat-square*: *invertible-mat A* $\Longrightarrow$ *square-mat A*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *inverse-blinfun-to-mat*:
fixes A :: $(nat \Rightarrow_b real) \Rightarrow_L (nat \Rightarrow_b real)$
assumes $invertible_L\ A$
assumes $(\bigwedge v\ v'\ j.\ (\bigwedge i.\ i < m \Longrightarrow apply\text{-}bfun\ v\ i = apply\text{-}bfun\ v'\ i) \Longrightarrow j < m \Longrightarrow (A\ v)\ j = (A\ v')\ j)$
assumes $(\bigwedge v\ v'\ j.\ (\bigwedge i.\ i < m \Longrightarrow apply\text{-}bfun\ v\ i = apply\text{-}bfun\ v'\ i) \Longrightarrow j < m \Longrightarrow (inv_L\ A\ v)\ j = (inv_L\ A\ v')\ j)$
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*⟩

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*⟩

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

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*⟩

termination *policy-iteration*
⟨*proof*⟩

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*⟩

lemma ν_b*-zero-notin*: $s \geq states \Longrightarrow \nu_b\ p\ s = 0$

⟨proof⟩

lemma *r-dec$_b$-zero-notin*: $s \geq states \Longrightarrow r\text{-}dec_b\ d\ s = 0$
⟨proof⟩

lemma ν_b*-eq-inv*: ν_b (*mk-stationary d*) = inv_L (*id-blinfun* − $l *_R \mathcal{P}_1$ *d*) (*r-dec$_b$ d*)
⟨proof⟩

lemma ν_b*-eq-bfun-if*: ν_b (*mk-stationary d*) = *bfun-if* ($\lambda i.\ i < states$) (ν_b (*mk-stationary d*)) *0*
⟨proof⟩

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⟩

lemma *summable-geom-*$\mathcal{P}_1$: *summable* ($\lambda k.$ (($l *_R \mathcal{P}_1\ d$)^^*k*))
⟨proof⟩

lemma *summable-geom-*$\mathcal{P}_1{}'$: *summable* ($\lambda k.$ (($l *_R \mathcal{P}_1\ d$)^^*k*) *v*) **for** *v*
⟨proof⟩

lemma *summable-geom-*$\mathcal{P}_1{}''$: *summable* ($\lambda k.$ (($l *_R \mathcal{P}_1\ d$)^^*k*) *v s*) **for** *v s*
⟨proof⟩

lemma *K-closed′*: $s{<}states \Longrightarrow j \in set\text{-}pmf\ (K\ (s, a)) \Longrightarrow j < states$
⟨proof⟩

lemma $\mathcal{P}_1$*-indep*:
assumes ($\bigwedge i.\ i < states \Longrightarrow apply\text{-}bfun\ v\ i = apply\text{-}bfun\ v'\ i$) $j < states$
shows $(l *_R \mathcal{P}_1\ d)\ v\ j = (l *_R \mathcal{P}_1\ d)\ v'\ j$
⟨proof⟩

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-blinfun* − $l *_R \mathcal{P}_1\ d$)) *v*) *j* = ((inv_L (*id-blinfun* − $l *_R \mathcal{P}_1\ d$)) *v′*) *j*
⟨proof⟩

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⟩

lemma *invertible-ν_b-mat*: *invertible-mat* ((*1$_m$ states*) − *l* $\cdot_m$ (*blinfun-to-mat states states* ($\mathcal{P}_1$ *d*)))
⟨*proof*⟩

lemma *mat-cong*:
  assumes ($\bigwedge$*i j*. *i* < *n* ⟹ *j* < *m* ⟹ *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*)
  ⟨*proof*⟩

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*⟩

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*)))
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
⟨*proof*⟩

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*.ν_b (*MDP.mk-stationary-det* (*D-Map.map-to-fun d*)) *s*
⟨*proof*⟩

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*⟩

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*)))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *policy-eval-code′*:
fixes *d*
defines $m \equiv (one\text{-}st - l \cdot_m Matrix.mat\ states\ states\ (\lambda(s,\ y).\ pmf\ (MDP\text{-}K\ (s,\ d\text{-}lookup'\ d\ s))\ y))$
shows $policy\text{-}eval\text{-}code\ d = snd\ (gauss\text{-}jordan\ m\ (1_m\ states)) *_v (vec\ states\ (\lambda i.\ MDP\text{-}r\ (i,\ d\text{-}lookup'\ d\ i)))$
⟨*proof*⟩

lemma *policy-eval-code″*[*code*]:
fixes *d*
defines $m \equiv (one\text{-}st - l \cdot_m ((k\text{-}mat\ d)))$
shows $policy\text{-}eval\text{-}code\ d = snd\ (gauss\text{-}jordan\ m\ one\text{-}st) *_v (vec\ states\ (\lambda i.\ MDP\text{-}r\ (i,\ d\text{-}lookup'\ d\ i)))$
⟨*proof*⟩

definition $policy\text{-}eval\text{-}code'\ d\ m = snd\ (gauss\text{-}jordan\ (one\text{-}st - l \cdot_m ((k\text{-}mat'\ d\ m)))\ one\text{-}st) *_v (vec\ states\ (\lambda i.\ MDP\text{-}r\ (i,\ d\text{-}lookup'\ d\ i)))$

lemma *dim-policy-eval-code*: *dim-vec* (*policy-eval-code d*) = *states*
⟨*proof*⟩

lemma *policy-eval-correct′*:
assumes *D-Map.keys d* = {*0..<states*} *D-Map.invar d*
shows *vec-to-bfun* (*policy-eval-code d*) = $MDP.\nu_b$ (*MDP.mk-stationary-det* (*D-Map.map-to-fun d*))
⟨*proof*⟩

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′ (λ*s. 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*)
⟨*proof*⟩

lemma *keys-vi-find-policy-code-aux-upt*: *D-Map.keys* (*pi-find-policy-code d v*) = {*0..<states*}
⟨*proof*⟩

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*) ∈ *MDP-A s*
⟨*proof*⟩

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* ∈ *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*⟩

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* ⋀*s. s* < *states* ⟹ *d-lookup′ d s* ∈ *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*
⟨*proof*⟩

definition *eq-policy d1 d2* = (∀ *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*)
⟨*proof*⟩

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*⟩

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*⟩

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 transition-vecs*
⟨*proof*⟩

lemma *policy-eval-code-eq′*: *is-dec-det-code d* ⟹ *policy-eval-code d* = *policy-eval-code′ d transition-vecs*
⟨*proof*⟩

lemma *policy-step-code-eq′*: *is-dec-det-code d* ⟹ *policy-step-code d* = *policy-step-code′ d transition-vecs*
⟨*proof*⟩

lemma *policy-step-code-correct*:
assumes *D-Map.keys d* = *MDP.state-space D-Map.invar d* (⋀*s. s* < *states* ⟹ *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*))
⟨*proof*⟩

lemma *PI-code-aux-correct-aux*:
assumes *D-Map.invar d D-Map.keys d* = {*0..<states*} (⋀*s. s* < *states* ⟹ *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* ⟶ *PI-code-aux d* = *PI-code-aux′ d transition-vecs*)
⟨*proof*⟩

lemma *PI-code-correct*:

assumes *D-Map.invar d D-Map.keys d* = *MDP.state-space* ($\bigwedge$*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*⟩

lemma [*code*]: *PI-code d0* = (*if* ¬ *is-dec-det-code d0 then d0 else PI-code-aux′ d0 transition-vecs*)
⟨*proof*⟩

definition *d0* = *D-Map.from-list′* (λ*s. fst* (*hd* (*a-inorder* (*s-lookup mdp s*)))) [*0..<states*]

end

lemma *inorder-empty*: *Tree2.inorder am* = [] ⟹ *am* = ⟨⟩
⟨*proof*⟩

global-interpretation *PI-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 *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′*

⟨*proof*⟩

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*))
⟨*proof*⟩

lift-definition *vec-to-vec-impl* :: *′a vec* ⇒ *′a vec-impl* **is**
λv. vec-of-fun (*dim-vec v*) ((\$) *v*)⟨*proof*⟩

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*
⟨*proof*⟩

lemma *vec-impl-exhaust*: (⋀*v*. *P* (*Abs-vec-impl* (*IArray.length* *v*, *v*)))
⟹ *P* *u*
⟨*proof*⟩

lemma *vec-to-vec-impl-code*[*code*]: *vec-to-vec-impl* (*vec-impl* *v*) = *v*
⟨*proof*⟩

lemma *dim-row-mat-of-row-fun-code*[*simp*]: *dim-row* (*mat-impl* (*mat-of-row-fun-code* *nr* *nc* *f*)) = *nr*
⟨*proof*⟩

lemma *dim-col-mat-of-row-fun-code*[*simp*]: *dim-col* (*mat-impl* (*mat-of-row-fun-code* *nr* *nc* *f*)) = *nc*
⟨*proof*⟩

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*⟩
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* ⟨*proof*⟩

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*)
⟨*proof*⟩

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 inverse-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* ⟨*proof*⟩

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*)
⟨*proof*⟩

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* × *'a*) ⇒ *real* **and**
r-fin :: *'s* ⇒ *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** $\bigwedge x.(1{::}real)\hat{}(x{::}nat)=1$
⟨*proof*⟩

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*]: (λ*t*. *r-fin* ((*t* !! *i*))) ∈ *borel-measurable S*
⟨*proof*⟩

lemma *integrable-r-fin-nth* [*simp*]: *integrable* (𝒯 *p s*) (λ*t*. *r-fin* (*fst*(*t* !! *i*)))
⟨*proof*⟩

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*⟩

function *νN-eval* **where**
νN-eval p h s = (
if length h = *N then r-fin s else*
if length h > *N then 0 else*
measure-pmf.expectation (*p h s*) (λ*a*. *r* (*s*,*a*) +
measure-pmf.expectation (*K* (*s*,*a*)) (λ*s*′. *νN-eval p* (*h*@[(*s*,*a*)]) *s*′)))
⟨*proof*⟩

termination
⟨*proof*⟩

lemmas *abs-disc-eq*[*simp del*]
lemmas *νN-eval.simps*[*simp del*]

lemma *pmf-bounded-integrable*: *bounded* (*range* (*f*::- ⇒ *real*)) ⟹ *integrable* (*measure-pmf p*) *f*
⟨*proof*⟩

lemma *abs-boundedD*[*dest*]: (⋀*x*. |*f x*| ≤ (*c*::*real*)) ⟹ *bounded* (*range f*)
⟨*proof*⟩

lemma *abs-integral-le*[*intro*]: (⋀*x*. |*f x*| ≤ (*c*::*real*)) ⟹ *abs* (*measure-pmf.expectation p f*) ≤ *c*
⟨*proof*⟩

lemma *abs-νN-eval-le*: $|\nu N\text{-}eval\ p\ h\ s| \le (N - length\ h) * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *abs-νN-eval-le'*: $|\nu N\text{-}eval\ p\ h\ s| \le N * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *measure-pmf-expectation-bind*:
assumes *bounded (range f)*
shows *measure-pmf.expectation* $(p \gg= q)$ *(f::- ⇒ real) = measure-pmf.expectation p (λx. measure-pmf.expectation (q x) f)*
⟨*proof*⟩

lemma *Pn'-shift*: *bounded (range (f :: - ⇒ real)) ⟹ measure-pmf.expectation (p h s)*
(λa. measure-pmf.expectation (K (s, a))
(λs'. measure-pmf.expectation (Pn' (λh'. p ((h @ (s, a)# h'))) s' n) f))
= measure-pmf.expectation (Pn' (λh'. p (h @ h')) s (Suc n)) f
⟨*proof*⟩

lemma *bounded-r-snd'*: *bounded ((λa. r (s, a)) ‘ X)*
⟨*proof*⟩

lemma *bounded-r-snd*: *bounded (range (λa. r (s, a)))*
⟨*proof*⟩

lemma *νN-eval-eq*: *length h ≤ N ⟹ νN-eval p h s =*
$(\sum i \in \{length\ h..< N\}.$
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*⟩

lemma *νN-eval-correct*: *νN-eval p [] s = νN p s*
⟨*proof*⟩

lift-definition νN_b *:: ('s, 'a) pol ⇒ 's* $\Rightarrow_b$ *real* **is** *νN*
⟨*proof*⟩

definition *νN-opt s =* $(\bigsqcup p \in \Pi_{HR}.\ \nu N\ p\ s)$
definition *νN-eval-opt h s =* $(\bigsqcup p \in \Pi_{HR}.\ \nu N\text{-}eval\ p\ h\ s)$

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)) (λs'. νN-opt-eqn (h@[(s,a)]) s')))

⟨*proof*⟩

termination
⟨*proof*⟩

lemmas *νN-opt-eqn.simps*[*simp del*]

lemma *abs-νN-opt-eqn-le*: $|\nu N\text{-}opt\text{-}eqn\ h\ s| \leq (N - length\ h) * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *abs-νN-opt-eqn-le′*: $|\nu N\text{-}opt\text{-}eqn\ h\ s| \leq N * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *abs-νN-eval-opt-le′*: $|\nu N\text{-}eval\text{-}opt\ h\ s| \leq N * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *exp-νN-eval-opt-le*: $|measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\text{-}opt\ h)| \leq N * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *bounded-exp-νN-eval-opt*: $(bounded\ ((\lambda a.\ measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\text{-}opt\ (h\ a)))\ `\ X))$
⟨*proof*⟩

lemma *bounded-r-exp-νN-eval-opt*: $(bounded\ ((\lambda a.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\text{-}opt\ (h\ a)))\ `\ X))$
⟨*proof*⟩

lemma *integrable-r-exp-νN-eval-opt*: $(integrable\ (measure\text{-}pmf\ q)\ ((\lambda a.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\text{-}opt\ (h\ a)))))$
⟨*proof*⟩

lemma *exp-νN-eval-le*: $|measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\ p\ h)| \leq N * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *bounded-exp-νN-eval*: $(bounded\ ((\lambda a.\ measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\ p\ (h\ a)))\ `\ X))$
⟨*proof*⟩

lemma *bounded-r-exp-νN-eval*: $(bounded\ ((\lambda a.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (\nu N\text{-}eval\ p\ (h\ a)))\ `\ X))$
⟨*proof*⟩

lemma *integrable-r-exp-νN-eval*: $(integrable\ (measure\text{-}pmf\ q)\ ((\lambda a.\ r$

(s, a) + *measure-pmf.expectation* $(K\ (s,\ a))$ $(\nu N\text{-}eval\ p\ (h\ a)))))$
⟨*proof*⟩

lemma *exp-νN-opt-eqn-le*: |*measure-pmf.expectation* $(K\ (s,\ a))$ $(\nu N\text{-}opt\text{-}eqn$ $h)| \leq N * r_M + (\bigsqcup s.\ |r\text{-}fin\ s|)$
⟨*proof*⟩

lemma *bounded-exp-νN-opt-eqn*: (*bounded* ((λ*a*. *measure-pmf.expectation* $(K\ (s,\ a))$ $(\nu N\text{-}opt\text{-}eqn\ (h\ a)))$ ‘ X))
⟨*proof*⟩

lemma *bounded-r-exp-νN-opt-eqn*: (*bounded* ((λ*a*. $r\ (s,\ a)$ + *measure-pmf.expectation* $(K\ (s,\ a))$ $(\nu N\text{-}opt\text{-}eqn\ (h\ a)))$ ‘ X))
⟨*proof*⟩

lemma *integrable-r-exp-νN-opt-eqn*: (*integrable* (*measure-pmf q*) ((λ*a*. $r\ (s,\ a)$ + *measure-pmf.expectation* $(K\ (s,\ a))$ $(\nu N\text{-}opt\text{-}eqn\ (h\ a)))))$
⟨*proof*⟩

lemma *νN-eval-le-opt-eqn*: $p \in \Pi_{HR} \Longrightarrow \nu N\text{-}eval\ p\ h\ s \leq \nu N\text{-}opt\text{-}eqn$ $h\ s$
⟨*proof*⟩

lemma *νN-eval-le-opt*: $p{\in}\Pi_{HR} \Longrightarrow \nu N\text{-}eval\text{-}opt\ h\ s \geq \nu N\text{-}eval\ p\ h\ s$
⟨*proof*⟩

lemma *νN-opt-eqn-bounded*[*simp*, *intro*]: *bounded* ((νN-opt-eqn h) ‘ X)
⟨*proof*⟩

lemma *νN-eval-opt-bounded*[*simp*, *intro*]: *bounded* ((νN-eval-opt h) ‘ X)
⟨*proof*⟩

lemma *νN-eval-bounded*[*simp*, *intro*]: *bounded* ((νN-eval p h) ‘ X)
⟨*proof*⟩

lemma *νN-opt-ge*: *length* $h \leq N \Longrightarrow \nu N\text{-}opt\text{-}eqn\ h\ s \geq \nu N\text{-}eval\text{-}opt\ h$ s
⟨*proof*⟩

lemma *Sup-wit-ex*:
assumes $(d :: real) > 0$
assumes $X \neq \{\}$
assumes *bdd-above* (f ‘ X)
shows $\exists x \in X.\ (\bigsqcup x \in X.\ f\ x) < f\ x + d$
⟨*proof*⟩

lemma *νN-opt-eqn-markov*: *length h* $\leq$ *N* $\Longrightarrow$ *length h* = *length h'* $\Longrightarrow$ *νN-opt-eqn h* = *νN-opt-eqn h'*
⟨*proof*⟩

lemma *νN-opt-le*:
fixes *eps* :: *real*
assumes *eps* > *0*
shows $\exists\, p \in \Pi_{MD}$. $\forall$ *h s*. *length h* $\leq$ *N* $\longrightarrow$ *νN-eval* (*mk-markovian-det p*) *h s* + *real* (*N* − *length h*) ∗ *eps* $\geq$ *νN-opt-eqn h s*
⟨*proof*⟩

lemma *νN-opt-le'*:
fixes *eps* :: *real*
assumes *eps* > *0*
shows $\exists\, p \in \Pi_{MD}$. $\forall$ *h s*. *length h* $\leq$ *N* $\longrightarrow$ *νN-eval* (*mk-markovian-det p*) *h s* + *eps* $\geq$ *νN-opt-eqn h s*
⟨*proof*⟩

lemma *mk-det-preserves*: *p* $\in \Pi_{HD}$ $\Longrightarrow$ (*mk-det p*) $\in \Pi_{HR}$
⟨*proof*⟩

lemma *mk-markovian-det-preserves*: *p* $\in \Pi_{MD}$ $\Longrightarrow$ (*mk-markovian-det p*) $\in \Pi_{HR}$
⟨*proof*⟩

lemma *νN-opt-eq*:
assumes *length h* $\leq$ *N*
shows *νN-opt-eqn h s* = *νN-eval-opt h s*
⟨*proof*⟩

lemma *νN-opt-eqn-correct*: *νN-opt s* = *νN-opt-eqn* [] *s*
⟨*proof*⟩

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*)) (*νN-opt-eqn* (*h*@[(*s*, *p* (*length h*) *s*)])) + *eps*
$\geq$ ($\bigsqcup$ *a* $\in$ *A s*. *r* (*s*, *a*) + *measure-pmf.expectation* (*K* (*s*,*a*)) (*νN-opt-eqn* (*h*@[(*s*, *a*)])))
shows $\bigwedge$*h s*. *length h* $\leq$ *N* $\Longrightarrow$ *νN-eval* (*mk-markovian-det p*) *h s* + (*N* − *length h*) ∗ *eps* $\geq$ *νN-opt-eqn h s*
$\bigwedge$*s*. *νN* (*mk-markovian-det p*) *s* + *N* ∗ *eps* $\geq$ *νN-opt s*
⟨*proof*⟩

lemma *νN-has-eps-opt-pol*:
assumes *eps* > *0*
shows $\exists\, p \in \Pi_{MD}$. $\forall$ *s*. *νN* (*mk-markovian-det p*) *s* + *eps* $\geq$ *νN-opt s*

⟨*proof*⟩

lemma $\nu N\text{-}le\text{-}opt$: $p \in \Pi_{HR} \Longrightarrow \nu N\ p\ s \leq \nu N\text{-}opt\ s$
⟨*proof*⟩

lemma $\nu N\text{-}has\text{-}opt\text{-}pol$:
assumes $\bigwedge h\ s.$
$length\ h < N$
$\Longrightarrow \exists\, a \in A\ s.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,a))\ (\nu N\text{-}opt\text{-}eqn\ (h@[(s,a)]))$
$= (\bigsqcup a \in A\ s.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,a))\ (\nu N\text{-}opt\text{-}eqn\ (h@[(s,a)])))$
shows $\exists\, p \in \Pi_{MD}.\ \forall\, s.\ \nu N\ (mk\text{-}markovian\text{-}det\ p)\ s = \nu N\text{-}opt\ s$
⟨*proof*⟩

lemma $ex\text{-}Max$: $finite\ X \Longrightarrow X \neq \{\} \Longrightarrow \exists\, x \in X.\ \ f\ x = Max\ (f\ `\ X)$
⟨*proof*⟩

lemma $fin\text{-}A\text{-}imp\text{-}opt\text{-}pol$:
assumes $\bigwedge s.\ finite\ (A\ s)$
shows $\exists\, p{\in}\Pi_{MD}.\ \forall\, s.\ \nu N\ (mk\text{-}markovian\text{-}det\ p)\ s = \nu N\text{-}opt\ s$
⟨*proof*⟩

16 Backward Induction

function *bw-ind-aux* **where**
$bw\text{-}ind\text{-}aux\ n\ s = ($
$if\ n = N\ then\ r\text{-}fin\ s\ else$
$if\ n > N\ then\ 0\ else$
$\bigsqcup a \in A\ s.\ (r\ (s,a) +$
$measure\text{-}pmf.expectation\ (K\ (s,a))\ (\lambda s'.\ bw\text{-}ind\text{-}aux\ (Suc\ n)\ s')))$
⟨*proof*⟩

termination
⟨*proof*⟩

lemmas *bw-ind-aux.simps*[*simp del*]

lemma $bw\text{-}ind\text{-}aux\text{-}eq$: $bw\text{-}ind\text{-}aux\ (length\ h)\ s = \nu N\text{-}opt\text{-}eqn\ h\ s$
⟨*proof*⟩

fun *bw-ind-aux′* **where**
$bw\text{-}ind\text{-}aux'\ (Suc\ n)\ m = ($
$let\ m' = (\lambda i\ s.$
$if\ i = n$
$then\ (\bigsqcup a \in A\ s.\ (r\ (s,a) + measure\text{-}pmf.expectation\ (K\ (s,a))$

$(m\ (Suc\ n))))$
$\quad else\ m\ i\ s)\ in$
$\quad bw\text{-}ind\text{-}aux'\ n\ m')\ |$
$bw\text{-}ind\text{-}aux'\ 0\ m = m$

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 $bw\text{-}ind\text{-}aux'\ n\ m\ i = m\ i$
$\langle proof \rangle$

lemma $bw\text{-}ind\text{-}aux'\text{-}indep$:
assumes $i < n$ **and**
$\bigwedge j.\ j > i \Longrightarrow m\ j = m'\ j$
shows $bw\text{-}ind\text{-}aux'\ n\ m\ i\ s = bw\text{-}ind\text{-}aux'\ n\ m'\ i\ s$
$\langle proof \rangle$

lemma $bw\text{-}ind\text{-}aux'\text{-}simps'$: $i < n \Longrightarrow bw\text{-}ind\text{-}aux'\ n\ m\ i\ s = (\bigsqcup a \in A\ s.\ (r\ (s,a) + measure\text{-}pmf.expectation\ (K\ (s,a))\ (bw\text{-}ind\text{-}aux'\ \ n\ m\ (Suc\ i))))$
$\langle proof \rangle$

lemma $bw\text{-}ind\text{-}correct$: $n \leq N \Longrightarrow bw\text{-}ind\ n = bw\text{-}ind\text{-}aux\ n$
$\langle proof \rangle$

definition $bw\text{-}ind\text{-}pol\text{-}gen\ (d :: 'a\ set \Rightarrow 'a)\ n\ s = ($
$if\ n \geq N\ then\ d\ (A\ s)$
$else$
$d\ (\{a\ .\ is\text{-}arg\text{-}max\ (\lambda a.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (bw\text{-}ind\text{-}aux\ (Suc\ n)))\ (\lambda a.\ a \in A\ s)\ a\}))$

lemma $bw\text{-}ind\text{-}pol\text{-}is\text{-}arg\text{-}max$:
assumes $\bigwedge X.\ X \neq \{\} \Longrightarrow d\ X \in X\ \bigwedge s.\ finite\ (A\ s)$
shows $is\text{-}arg\text{-}max\ (\lambda a.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (bw\text{-}ind\text{-}aux\ (Suc\ n)))\ (\lambda a.\ a \in A\ s)\ (d\ (\{a\ .\ is\text{-}arg\text{-}max\ (\lambda a.\ r\ (s,\ a) + measure\text{-}pmf.expectation\ (K\ (s,\ a))\ (bw\text{-}ind\text{-}aux\ (Suc\ n)))\ (\lambda a.\ a \in A\ s)\ a\}))$
$\langle proof \rangle$

lemma $bw\text{-}ind\text{-}pol\text{-}gen$:
assumes $\bigwedge X.\ X \neq \{\} \Longrightarrow d\ X \in X\ \bigwedge s.\ finite\ (A\ s)$
shows $bw\text{-}ind\text{-}pol\text{-}gen\ d \in \Pi_{MD}$
$\langle proof \rangle$

lemma
assumes $\bigwedge X.\ X \neq \{\} \Longrightarrow d\ X \in X\ \ \bigwedge s.\ finite\ (A\ s)\ length\ h \leq N$
shows $\nu N\text{-}eval\ (mk\text{-}markovian\text{-}det\ (bw\text{-}ind\text{-}pol\text{-}gen\ d))\ h\ s = \nu N\text{-}eval\text{-}opt$

h s
⟨*proof*⟩

lemma *bw-ind-aux′-eq*: $n \leq N \Longrightarrow$ *bw-ind-aux′ N* (λi *s. if i* = *N then r-fin s else 0*) *n* = *bw-ind-aux n*
⟨*proof*⟩
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*)
⟨*proof*⟩

sublocale *MDP*: *MDP-reward-disc* (*MDP-A*) (*MDP-K*) (*MDP-r*) *0*
⟨*proof*⟩

sublocale *MDP*: *MDP-act* (*MDP-A*) (*MDP-K*) λX. *LEAST* x. $x \in$
X
⟨*proof*⟩

sublocale *MDP*: *MDP-nat-fin* λX. *LEAST* x. $x \in X$ (*MDP-A*) (*MDP-K*)
states (*MDP-r*) *MDP-r-fin N-code*
⟨*proof*⟩

sublocale *V-Map*: *Array-real v-lookup v-update v-len v-array v-list*
v-invar
⟨*proof*⟩

sublocale *V-Map*: *Array-zero v-lookup v-update v-len v-array v-list*
v-invar
⟨*proof*⟩

sublocale *D-Map*: *Array-zero d-lookup d-update d-len d-array d-list*
d-invar
⟨*proof*⟩

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*⟩

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*)))))

⟨*proof*⟩

lemma *L-GS-code-correct'*:
assumes $s <$ *states v-len v* = *states v-invar v a* $\in$ *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*)
⟨*proof*⟩

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* $\in$ *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*⟩

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*
(λ*a*. *MDP-r*(*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*,*a*)) (*V-Map.map-to-bfun v*))
(λ*a*. *a* $\in$ *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*⟩

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* (λ*s*. *snd* (*vd-lookup vd s*)) *states*;
d = *D-Map.arr-tabulate* (λ*s*. *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* $\Longrightarrow$ *fst* (*bw-ind-aux-code n vl v0 d0*) ! ($i + n$) =
(*vl*#*v0*) ! *i*
⟨*proof*⟩

lemma *bw-ind-aux-code-fst-index'*: $n \leq i \Longrightarrow$ *fst* (*bw-ind-aux-code n vl v0 d0*) ! *i* =
(*vl*#*v0*) ! ($i - n$)

⟨*proof*⟩

lemma *bw-ind-aux-code-snd-index'*: $n \leq i \implies$ *snd* (*bw-ind-aux-code* *n vl v0 d0*) ! i =
(*d0*) ! $(i - n)$
⟨*proof*⟩

lemma *bw-ind-code-aux-correct*:
fixes *n vl v0 d0*
defines $d \equiv$ *snd* (*bw-ind-aux-code n vl v0 d0*)
defines $v \equiv$ *fst* (*bw-ind-aux-code n vl v0 d0*)
assumes *v-len vl* = *states*
assumes *v-invar vl*
assumes $\bigwedge s.\ s <$ *states* $\implies$ *m n s* = *v-lookup vl s*
assumes $s <$ *states*
shows ($i \leq n \longrightarrow$ *v-lookup* (*v* ! *i*) *s* = *MDP.bw-ind-aux'* *n m i s*) $\wedge$
($i < n \longrightarrow$ *d-lookup* (*d* ! *i*) *s* = (*least-arg-max*
($\lambda a.$ *MDP-r* (*s*, *a*) + *measure-pmf.expectation* (*MDP-K* (*s*, *a*)) (*MDP.bw-ind-aux'* *n m* (*Suc i*)))
($\lambda a.\ a \in$ *MDP-A s*)))
⟨*proof*⟩

lemma *bw-ind-code-correct*:
defines $d \equiv$ *snd bw-ind-code*
defines $v \equiv$ *fst bw-ind-code*
shows $\bigwedge n\ s.\ n \leq$ *N-code* $\implies s <$ *states* $\implies$ *v-lookup* (*v* ! *n*) *s* = *MDP.bw-ind n s*
and $\bigwedge n.\ n <$ *N-code* $\implies s <$ *states* $\implies$ *d-lookup* (*d* ! *n*) *s* = *MDP.bw-ind-pol-gen* ($\lambda X.$ *LEAST a.* $a \in X$) *n s*
⟨*proof*⟩
end

global-interpretation *Fin-Code*:
MDP-Code-Fin

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* 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*

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*

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*

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*

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*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩
termination
  ⟨*proof*⟩

lemma *tabulate-Suc*: $j \le n' \Longrightarrow$ *update n'* (*f n'*) (*tabulate f m n' j*) = *tabulate f m* (*Suc n'*) *j*
⟨*proof*⟩

lemma *from-list'-upt* [*code-unfold*]: *from-list' f* [*0..*<*n*] = *tabulate f empty n 0*

⟨*proof*⟩

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.