Seminar 7 - Sequential Decision Problems, Bellman's equation, backward induction

Adrian Perez Keilty

Chalmers University of Technology, Göteborg, Sweden

Note: Check out the lower sections from the attached file

`fpclimate.agda`

for type-checking.

Exercise 7.1:

What are the types of head and measure in the definitions of sumR and val? Define head. How could the type of measure be generalized?

- `head`: From the use of `(head xys)` as the fourth argument in `reward` (`reward t x y (head xys)` $\oplus$ `sumR xys`), we can infer that the type of `head` is `X t` for a strictly positive `t`. Also, since `head` tipically refers to the first element of a list or vector, we can define `head` as the state belonging to the first pair of state-control pairs of a sequence `XYSeq`. We use an equivalent definition in Agda to the one defined in Idris presented in "*On the Correctness of Monadic Backward Induction*":

```
-- head: state of the first pair of XYseq
head : {t n : Nat} → XYSeq t (suc n) → X t
head (Last x) = x
head ((x , y) || xys) = x
```

- `measure`: From the expression
  `val ps = measure ∘ (fmapM sumR ∘ trj ps)`
  `measure` takes in as input an element of type `M Val` and should return an element of type `Val`:

```
measure : M Val → Val
```

  In the general definition in vulnerability theory discussed earlier, we had

```
measure : F V → W
```

  where `V` was the type for harm values and `W` the type for vulnerability values, both equipped with preorders $\leq_V$ and $\leq_W$. The 'reward' in the current scenario replaces 'harm', and maximized instead of minimized. Similarly, the 'total reward' computed as the measure of the $\oplus$-sum of the rewards along possible trajectories mirrors the 'vulnerability' which does the same for 'harm'. It may make sense to generalize `Val` into two different types in settings were a single-step reward and a total aggregated reward fall into different types and preorders as well.

Exercise 7.2:

What is the type of $\leq_l$ in the definition of OptPolicySeq? Define $\leq_l$ in terms of $\leq$.

The type for $\leq_l$ is the pointwise inequality between functions `(x : X t) →  Val`. Here's its definition in terms of $\leq$:

```
_≤l_ : {t : Nat} → (X t → Val) → (X t → Val) → Set
f ≤l g = ∀ x → (f x ≤ g x)
```

Exercise 7.3:

On the fly: How many trajectories are in trj $[p_0, p_1]\, x_0$ *?*

The number of possible `XYSeq` trajectories can be determined by the number of state trajectories, which are exactly 3:

$[x_0, x_1^0, x_2^{0,0}]$
$[x_0, x_1^0, x_2^{0,1}]$
$[x_0, x_1^1, x_2^{1,0}]$

Exercise 7.4:

Define ηSP *,* fmap_{SP} *and* >>=*SP such that* trj $[p_0, p_1]\, x_0$ *yields the result of* $step_2$ *.*

We define fmap_{SP} the same way as for `fmapList`, but we also append the probability coordinate. Similarly η`SP` is defined as the singleton but appending the probability `p` = 1. For >>=`SP` we postulate μSP by replacing M by SP in the previous declaration of μM:

```
open import Data.Float using (Float) renaming (_+_ to _+Float_; _*_ to _*Float_)

-- Val = R, but we use Float instead
ValSP   : Set
ValSP = Float

-- Val = 0
0ValSP : ValSP
0ValSP = 0.0

-- usual addition
_⊕SP_ : ValSP → ValSP → ValSP
a ⊕SP b = a +Float b

SP  : Set → Set
SP X = List (X × Float)

fmapSP : {A B : Set} → (A → B) → SP A → SP B
fmapSP f [] = []
fmapSP f ((x , p) :: xps) = (f x , p) :: (fmapSP f xps)

-- Singleton equivalent, p = 1
ηSP : {X : Set} → X → SP X
ηSP x = (x , 1.0) :: []
```

```
postulate μSP : {X : Set} → SP (SP X) → SP X

_>>=SP_ : {A B : Set} → SP A → (A → SP B) → SP B
ma >>=SP f = μSP (fmapSP f ma)
```

Exercise 7.5:

In $step_4$ we have applied a definition of the exp. value measure ev *. Define* ev *consistently with $step_4$.*

We define the expected value as the sum of the products of the state-probability pair:

```
-- measure = expected value
ev : SP ValSP → ValSP
ev [] = 0ValSP
ev ((x , p) :: xps) = (x *Float p) +Float (ev xps)
```

which is now consistent with $step_4$.

Exercise 7.6:

Is the computation correct? Check it and report eventual errors!

We elaborate and specify the intermediate steps:

$$
\begin{aligned}
& r_0^0 * \alpha + r_1^{0,0} * \beta * \alpha + r_1^{0,1} * (1-\beta) * \alpha + r_0^1 * (1-\alpha) + r_1^{1,0} * (1-\alpha) \\
& = \left\{ \mathit{step}_6 : \left(r_0^0 + r_1^{0,0} * \beta + r_1^{0,1} * (1-\beta) \right) * \alpha + \left(r_0^1 + r_1^{1,0} \right) * (1-\alpha) \right\} = \\
& \mathsf{ev}\ \left[\left(r_0^0 + r_1^{0,0} * \beta + r_1^{0,1} * (1-\beta), \alpha \right), \left(r_0^1 + r_1^{1,0}, (1-\alpha) \right) \right] \\
& = \{ \mathit{step}_7 \text{ same as } \mathit{step}_6 \} = \\
& \mathsf{ev}\ [(r_0^0 + \mathsf{ev}\ [(r_1^{0,0}, \beta), (r_1^{0,1}, 1-\beta)], \alpha), (r_0^1 + \mathsf{ev}\ [(r_1^{1,0}, 1)], 1-\alpha)] \\
& = \left\{ \text{definitions of } r_1^{0,0}, r_1^{0,1} \text{ and } r_1^{1,0} \right\} \\
& \mathsf{ev}\ \left[\left(r_0^0 + \mathsf{ev}\ \left[\left(\mathsf{reward}\ 1\ x_1^0\ y_1^0\ (\mathsf{head}\ (\mathit{Last}\ x_2^{0,0})), \beta \right), \left(\mathsf{reward}\ 1\ x_1^0\ y_1^0\ (\mathsf{head}\ (\mathit{Last}\ x_2^{0,1})), 1-\beta \right) \right], \alpha \right), \right. \\
& \left. \left(r_0^1 + \mathsf{ev}\ \left[(\mathsf{reward}\ 1\ x_1^1\ y_1^1\ (\mathsf{head}\ (\mathit{Last}\ x_2^{1,0})), 1) \right], 1-\alpha \right) \right] \\
& = \{ \text{definition of } \mathsf{sumR} \} = \\
& \mathsf{ev}\ \left[\left(r_0^0 + \mathsf{ev}\ \left[\left(\mathsf{sumR}\ \left((x_1^0, y_1^0)\ ||\ \mathit{Last}\ x_2^{0,0} \right), \beta \right), \left(\mathsf{sumR}\ \left((x_1^0, y_1^0)\ ||\ \mathit{Last}\ x_2^{0,1} \right), 1-\beta \right) \right], \alpha \right), \right. \\
& \left. \left(r_0^1 + \mathsf{ev}\ \left[\left(\mathsf{sumR}\ \left((x_1^1, y_1^1)\ ||\ \mathit{Last}\ x_2^{1,0} \right), 1 \right) \right], 1-\alpha \right) \right] \\
& = \{ \text{definition of } \mathsf{fmap}_{SP} \} = \\
& \mathsf{ev}\ \left[\left(r_0^0 + \mathsf{ev}\ \left(\mathsf{fmap_{SP}}\ \mathsf{sumR}\ \left[\left(((x_1^0, y_1^0)\ ||\ \mathit{Last}\ x_2^{0,0}), \beta \right), \left(((x_1^0, y_1^0)\ ||\ \mathit{Last}\ x_2^{0,1}), 1-\beta \right) \right] \right), \alpha \right), \right. \\
& \left. \left(r_0^1 + \mathsf{ev}\ \left(\mathsf{fmap_{SP}}\ \mathsf{sumR}\ \left[\left(\left((x_1^1, y_1^1)\ ||\ \mathit{Last}\ x_2^{1,0} \right), 1 \right) \right] \right), 1-\alpha \right) \right] \\
& = \{ \mathit{step}_9 \text{ is the same as } \mathit{step}_2 \} = \\
& \mathsf{ev}\ [(r_0^0 + \mathsf{ev}\ (\mathsf{fmap_{SP}}\ \mathsf{sumR}\ (\mathsf{trj}\ [p_1]\ x_1^0)), \alpha), (r_0^1 + \mathsf{ev}\ (\mathsf{fmap_{SP}}\ \mathsf{sumR}\ (\mathsf{trj}\ [p_1]\ x_1^1)), 1-\alpha)] \\
& = \left\{ \mathit{step}_{10} : \text{definitions of } r_0^0 \text{ and } r_0^1 \text{ and definition of } \mathsf{val} \right\} = \\
& \mathsf{ev}\ [(\mathsf{reward}\ 0\ x_0\ y_0\ x_1^0 + \mathsf{val}\ [p_1]\ x_1^0, \alpha), (\mathsf{reward}\ 0\ x_0\ y_0\ x_1^1 + \mathsf{val}\ [p_1]\ x_1^1, 1-\alpha)] \\
& = \{ \mathit{step}_{12} : (\oplus_l) = (+) \text{ and definition of } \mathsf{fmap}_{SP} \} = \\
& \mathsf{ev}\ (\mathsf{fmap_{SP}}\ (\mathsf{reward}\ 0\ x_0\ y_0 \oplus_l \mathsf{val}\ [p_1])\ [(x_1^0, \alpha), (x_1^1, 1-\alpha)]) \\
& = \left\{ \mathit{step}_{13} : \mathsf{next}\ 0\ x_0\ y_0 = [(x_1^0, \alpha), (x_1^1, 1-\alpha)] \right\} = \\
& \mathsf{ev}\ (\mathsf{fmap_{SP}}\ (\mathsf{reward}\ 0\ x_0\ y_0 \oplus_l \mathsf{val}\ [p_1])\ (\mathsf{next}\ 0\ x_0\ y_0))
\end{aligned}
$$

Exercise 7.7:

Redo the computation for the non-deterministic case with the canonical monadic operations for List and with measure = sum. Do you obtain the same computational pattern?

Intuitively, both should behave in the same manner since in the stochastic case, the measure = ev is obtained by first converting the simple distribution to a simple list by multiplying the state-probability pairs and then adding them in the same way that the measure = sum would do when there's no probabilities involved in the first place. Hence, the cases (M = SP, measure = ev) and (M = List, measure = sum) should yield exactly the same pattern.

Exercise 7.8:

Prove Bellman's equation for the "plain" deterministic case using

```
postulate Lemma7 : (t n : Nat) → (p : Policy t) →
              (ps : PolicySeq (suc t) n) → (x : X t) →
              sumRId (trjId (p :: ps) x ) ≡
              rewardId t x (p x ) (nextId t x (p x )) ⊕Id
              (valId ps (nextId t x (p x)))
```

Error solved, ⊕lId (⊕Id extended to functions) had to be defined:

```
_⊕lId_ : {t : Nat} → (X t → ValId) → (X t → ValId) → (X t → ValId)
f ⊕lId g = (λ x → f x ⊕Id g x)
```

The proof is as follows:

```
BellmanEq : (t n : Nat) → (p : Policy t) → (ps : PolicySeq (suc t) n)
          → (x : X t) →
          valId (p :: ps) x ≡
          measureId (fmapId (rewardId t x (p x) ⊕lId valId ps) (nextId t x (p x)))
BellmanEq t n p Nil x =
  begin
    valId (p :: Nil) x
  =⟨⟩
    sumRId (trjId (p :: Nil) x)
  =⟨ Lemma7 t n p Nil x ⟩
    rewardId t x (p x ) (nextId t x (p x )) ⊕Id (valId Nil (nextId t x (p x)))
  =⟨⟩ -- def of ⊕lId
    (rewardId t x (p x) ⊕lId valId Nil) (nextId t x (p x))
  =⟨⟩
    measureId (fmapId (rewardId t x (p x) ⊕lId valId Nil) (nextId t x (p x)))
  end
BellmanEq t n p1 (p0 :: ps) x =
  begin
    valId (p1 :: (p0 :: ps)) x
  =⟨⟩
    sumRId (trjId (p1 :: (p0 :: ps)) x)
  =⟨ Lemma7 t n p1 (p0 :: ps) x ⟩
    rewardId t x (p1 x) (nextId t x (p1 x )) ⊕Id
    (valId (p0 :: ps) (nextId t x (p1 x)))
  =⟨⟩
    (rewardId t x (p1 x) ⊕lId valId (p0 :: ps)) (nextId t x (p1 x))
  =⟨⟩
    measureId (fmapId (rewardId t x (p1 x) ⊕lId valId (p0 :: ps))
    (nextId t x (p1 x)))
end
```

Exercise 7.9:

Implement

```
optExt : {t n : Nat} → PolicySeq (suc t) n → Policy t
```

applying

```
postulate Finite    : Set → Set
postulate toList    : {A : Set } → Finite A        → List A
postulate max       : {A : Set } → (f : A → Val) → List A → Val
postulate argmax    : {A : Set } → (f : A → Val) → List A → A
```

We add the extra postulate finiteY in order to create an element of type Finite (Y t x):

```
postulate finiteY : {t : Nat} (x : X t) → Finite (Y t x)
optExt : {t n : Nat} → PolicySeq (suc t) n → Policy t
optExt {t} ps x =
    argmax (λ y → measure (fmapM (reward t x y ⊕l vallBell ps) (next t x y)))
    (toList (finiteY x))
```

Exercise 7.10:

Formulate minimal requirements on toList, max and argmax for optExt to satisfy

```
optExtSpec : {t n : Nat} → (ps : PolicySeq (suc t) n) → OptExt ps (optExt ps)
```

Small type-checking obstacle: how to combine inequalities and equalities in equational reasoning? I've defined

```
begin≤_ : {A : Set} → {x y : A} → x ≤ y → x ≤ y
begin≤ p = p

_end≤ : {A : Set} → (x : A) → x ≤ x
x end≤ = refl≤

_≤⟨_⟩_ : {A : Set} → (x : A) → {y z : A} → x ≤ y → y ≤ z → x ≤ z
x ≤⟨ p ⟩ q = trans≤ p q

_≤⟨⟩_ : {A : Set} → (x : A) → {y : A} → x ≤ y → x ≤ y
x ≤⟨⟩ q = x ≤⟨ refl≤ ⟩ q

infix   1 begin≤_
infix   3 _end≤
infixr  2 _≤⟨_⟩_
infixr  2 _≤⟨⟩_
```

but there's a missing link (an `Either` probably?) to the ≡ counterparts in the proof. Here's the reasoning but it doesn't type check.

```
postulate maxSpec : {t : Nat} → {x : X t} → (y : Y t x)
                  → (f : Y t x → Val) → List (Y t x)
                  → f y ≤ max f (toList (finiteY x))
postulate argmaxSpec : {t : Nat} → {x : X t} → (f : Y t x → Val) → List (Y t x)
                  → f (argmax f (toList (finiteY x))) ≡ max f (toList (finiteY x))

optExtSpec : {t n : Nat} → (ps : PolicySeq (suc t) n) → OptExt ps (optExt ps)
optExtSpec {t} {n} ps p' x =
    begin≤
        vallBell (p' :: ps) x
    =⟨⟩
        measure (fmapM (reward t x (p' x) ⊕l vallBell ps) (next t x (p' x)))
    ≤⟨ maxSpec (p' x) (λ y → measure (fmapM (reward t x y ⊕l vallBell ps)
    (next t x y))) (toList (finiteY x)) ⟩
        max (λ y → measure (fmapM (reward t x y ⊕l vallBell ps) (next t x y)))
        (toList (finiteY x))
    =⟨ argmaxSpec (λ y → measure (fmapM (reward t x y ⊕l vallBell ps) (next t x y)))
    (toList (finiteY x)) ⟩
        measure (fmapM (reward t x (optExt {t} ps x) ⊕l vallBell ps)
        (next t x (optExt {t} ps x)))
    ≤⟨⟩
        vallBell (optExt ps :: ps) x
    end≤
```

Exercise 7.11:

Postulate measureMon, plusMon and implement Bellman.

We first define `OptPolicySeqBell` specific to `valBell` (`val` computed defined in order to perform backwards induction):

```
OptPolicySeqBell : {t n : Nat} → PolicySeq t n → Set
OptPolicySeqBell {t } {n} ps = ∀ (ps' : PolicySeq t n) → valBell ps' ≤l valBell ps
```

Now we postulate

```
postulate measureMon : {A : Set } → (f g : A → Val) → (f ≤l g) → (ma : M A)
                     → measure (fmapM f ma) ≤ measure (fmapM g ma)
postulate plusMon : {a b c d : Val } → a ≤ b → c ≤ d → (a ⊕ c) ≤ (b ⊕ d)

postulate Bellman : {t n : Nat} → (p : Policy t) → (ps : PolicySeq (suc t) n) →
                  OptExt ps p → OptPolicySeqBell ps → OptPolicySeqBell (p :: ps)
```

To implement Bellman we prove that for every policy `p'` and policy sequence `ps'` it holds that

$$\texttt{valBell (p' :: ps')} \leq_l \texttt{valBell (p :: ps)}.$$

The equational reasoning goes as follows (no type-checking has been attempted due to the previous issue of combining inequalities and equalities in the same proof):

```
{OptPolicySeqBell ps}
⇔
val ps' ≤l val ps
{def}
⇔
∀ x̄ → val ps' x̄ ≤ val ps x̄
{(refl≤ reward t x (p' x) x̄) + plusMon postulate}
 ⟹
∀ x̄ → reward t x (p' x) x̄ ⊕ val ps' x̄ ≤ reward t x (p' x) x̄ ⊕ val ps' x̄
{def}
⇔
reward t x (p' x)  ⊕l val ps' ≤l reward t x (p' x)  ⊕l val ps'
{measureMon postulate}
 ⟹
∀ mx' → measure (fmapM (reward t x (p' x) ⊕l vallBell ps') (mx'))
      ≤
      measure (fmapM (reward t x (p' x) ⊕l vallBell ps) (mx'))
 ⟹
∀ x → measure (fmapM (reward t x (p' x) ⊕l vallBell ps') (next t x (p' x)))
      ≤
      measure (fmapM (reward t x (p' x) ⊕l vallBell ps) (next t x (p' x)))
{def}
⇔
∀ x → valBell (p' :: ps') x ≤ valBell (p' :: ps) x
⇔
valBell (p' :: ps') ≤l valBell (p' :: ps)
{OptExt ps p = ∀ p' → valBell (p' :: ps) ≤l valBell (p :: ps)}
 ⟹
valBell (p' :: ps') ≤l valBell (p :: ps)
```

Exercise 7.12:

Implement biOptVal *by induction on n:*

The induction step type-checks but the base case even if obvious due to the reflexivity of $\leq$ doesn't, so we base it on a postulate:

```
-- With optExt one can solve SDPs by backward induction
bi : (t n : Nat) → PolicySeq t n
bi t zero = Nil
bi t (suc n) = let ps = bi (suc t) n in optExt ps :: ps

-- By reflexivity of ≤
postulate nilIsOptPolicySeq : {t : Nat} → OptPolicySeqBell {t} Nil
-- nilIsOptPolicySeq {t} = {!   !}

-- With Bellman and optExtSpec one can verify that bi yields optimal
-- policy sequences
```

```
biOptVal : (t n : Nat)  → OptPolicySeqBell (bi t n)
biOptVal t zero         = nilIsOptPolicySeq
biOptVal t (suc n)      = Bellman (optExt (bi (suc t) n)) (bi (suc t) n)
                           (optExtSpec (bi (suc t) n)) (biOptVal (suc t) n)
```

Exercise 7.13:

There are also more "practical" limitations which ones come up to your mind?

Even restricting the domain of policies to the "viable" and "reachable" states, this doesn't fix the computational intractability of backwards induction. Look up tables may be integrated at the cost of not being able to machine-check correctness proofs ("SEQUENTIAL DECISION PROBLEMS, DEPENDENT TYPES AND GENERIC SOLUTIONS").