Specifications in small and large contexts

Nicola Botta [2] [1]

[2]Potsdam Institute for Climate Impact Research

[1]Thanks to C. Ionescu, P. Jansson and to the Cartesian Seminar people.

Outline

- Small context: vector indexing and lookup
- Large context: dynamic programming
- Specifications in large contexts
- Preliminary conclusions, guidelines
- Dynamic programming continued

Small context: vector indexing and lookup

Small context: vector indexing and lookup

- The challenge is implementing vector *index* and *lookup*:

Small context: vector indexing and lookup

- The challenge is implementing vector *index* and *lookup*:

$$\mathit{index} \;:\; \mathit{Fin}\ n \;\rightarrow\; \mathit{Vect}\ n\ X \;\rightarrow\; X$$

Small context: vector indexing and lookup

- The challenge is implementing vector *index* and *lookup*:

$$index : Fin\ n \rightarrow Vect\ n\ X \rightarrow X$$

$$lookup : (x : X) \rightarrow (xs : Vect\ n\ X) \rightarrow Elem\ x\ xs \rightarrow Fin\ n$$

Small context: vector indexing and lookup

- The challenge is implementing vector *index* and *lookup*:

  $$index \;:\; Fin\ n \;\rightarrow\; Vect\ n\ X \;\rightarrow\; X$$

  $$lookup \;:\; (x \;:\; X) \;\rightarrow\; (xs \;:\; Vect\ n\ X) \;\rightarrow\; Elem\ x\ xs \;\rightarrow\; Fin\ n$$

- The idea is that *index* shall be an "inverse" of *lookup*:

Small context: vector indexing and lookup

- The challenge is implementing vector *index* and *lookup*:

$$\mathit{index} \;:\; \mathit{Fin}\; n \;\rightarrow\; \mathit{Vect}\; n\; X \;\rightarrow\; X$$

$$\mathit{lookup} \;:\; (x \;:\; X) \;\rightarrow\; (\mathit{xs} \;:\; \mathit{Vect}\; n\; X) \;\rightarrow\; \mathit{Elem}\; x\; \mathit{xs} \;\rightarrow\; \mathit{Fin}\; n$$

- The idea is that *index* shall be an “inverse” of *lookup*:

$$\begin{aligned}\mathit{ilSpec} \;:\; & (x \;:\; X) \;\rightarrow\; (\mathit{xs} \;:\; \mathit{Vect}\; n\; X) \;\rightarrow\; (p \;:\; \mathit{Elem}\; x\; \mathit{xs}) \;\rightarrow \\ & \mathit{index}\; (\mathit{lookup}\; x\; \mathit{xs}\; p)\; \mathit{xs} = x\end{aligned}$$

Small context: vector indexing and lookup

- The challenge is implementing vector *index* and *lookup*:

$$index \ : \ Fin\ n \ \to \ Vect\ n\ X \ \to \ X$$

$$lookup \ : \ (x \ : \ X) \ \to \ (xs \ : \ Vect\ n\ X) \ \to \ Elem\ x\ xs \ \to \ Fin\ n$$

- The idea is that *index* shall be an “inverse” of *lookup*:

$$\begin{aligned} ilSpec \ : \ & (x \ : \ X) \ \to \ (xs \ : \ Vect\ n\ X) \ \to \ (p \ : \ Elem\ x\ xs) \ \to \\ & index\ (lookup\ x\ xs\ p)\ xs = x \end{aligned}$$

$$\begin{aligned} liSpec \ : \ & (k \ : \ Fin\ n) \ \to \ (xs \ : \ Vect\ n\ X) \ \to \\ & (p \ : \ Injective2\ xs) \ \to \ (q \ : \ Elem\ (index\ k\ xs)\ xs) \ \to \\ & lookup\ (index\ k\ xs)\ xs\ q = k \end{aligned}$$

Small context: vector indexing and lookup

- All functions are required to be total.

Small context: vector indexing and lookup

- All functions are required to be total.
- The context of the specification is *Vect*, *Elem* and *Injective2*.

Small context: vector indexing and lookup

- All functions are required to be total.
- The context of the specification is *Vect*, *Elem* and *Injective2*.
- *Injective2 xs* means that *xs* has no duplicates:

Small context: vector indexing and lookup

- All functions are required to be total.
- The context of the specification is *Vect*, *Elem* and *Injective2*.
- *Injective2 xs* means that *xs* has no duplicates:

$$\begin{array}{l} \mathit{Injective2} \;:\; \mathit{Vect}\; n\; X \;\rightarrow\; \mathit{Type} \\ \mathit{Injective2}\; \mathit{xs} = \mathit{Not}\; (i = j) \;\rightarrow\; \mathit{Not}\; (\mathit{index}\; i\; \mathit{xs} = \mathit{index}\; j\; \mathit{xs}) \end{array}$$

Small context: vector indexing and lookup

- Could we simplify the specification, e.g., by declaring *lookup* to return a list of *Fin n*?

  $$lookup : (x : X) \rightarrow (xs : Vect\ n\ X) \rightarrow List\ (Fin\ n)$$

Small context: vector indexing and lookup

- Could we simplify the specification, e.g., by declaring *lookup* to return a list of *Fin n*?

  $$lookup : (x : X) \to (xs : Vect\ n\ X) \to List\ (Fin\ n)$$

- Could we get rid of q in *liSpec*?

  $$\begin{aligned} liSpec \; : \; & (k : Fin\ n) \to (xs : Vect\ n\ X) \to \\ & (p : Injective2\ xs) \to (q : Elem\ (index\ k\ xs)\ xs) \to \\ & lookup\ (index\ k\ xs)\ xs\ q = k \end{aligned}$$

Small context: vector indexing and lookup

- Could we simplify the specification, e.g., by declaring *lookup* to return a list of *Fin n*?

  $$lookup : (x : X) \to (xs : Vect\ n\ X) \to List\ (Fin\ n)$$

- Could we get rid of q in *liSpec*?

  $$\begin{aligned} liSpec : {} & (k : Fin\ n) \to (xs : Vect\ n\ X) \to \\ & (p : Injective2\ xs) \to (q : Elem\ (index\ k\ xs)\ xs) \to \\ & lookup\ (index\ k\ xs)\ xs\ q = k \end{aligned}$$

- When is a specification "enough"?

Small context: vector indexing and lookup

- Could we simplify the specification, e.g., by declaring *lookup* to return a list of *Fin n*?

  $$lookup : (x : X) \rightarrow (xs : Vect\ n\ X) \rightarrow List\ (Fin\ n)$$

- Could we get rid of q in *liSpec*?

  $$\begin{aligned} liSpec : {} & (k : Fin\ n) \rightarrow (xs : Vect\ n\ X) \rightarrow \\ & (p : Injective2\ xs) \rightarrow (q : Elem\ (index\ k\ xs)\ xs) \rightarrow \\ & lookup\ (index\ k\ xs)\ xs\ q = k \end{aligned}$$

- When is a specification "enough"?

- Can we put forward guidelines for specifications?

Large context: dynamic programming

Large context: dynamic programming

- The challenge is implementing total functions

$$bi : (t : \mathbb{N}) \rightarrow (n : \mathbb{N}) \rightarrow PolicySeq\ t\ n$$

Large context: dynamic programming

- The challenge is implementing total functions

$$bi : (t : \mathbb{N}) \to (n : \mathbb{N}) \to PolicySeq\ t\ n$$

and

$$biLemma : (t : \mathbb{N}) \to (n : \mathbb{N}) \to OptPolicySeq\ (bi\ t\ n)$$

Large context: dynamic programming

- The challenge is implementing total functions

$$bi : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{PolicySeq}\ t\ n$$

and

$$\mathit{biLemma} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)$$

- The idea is that *bi* shall be a generic implementation of dynamic programming (Bellman 1957).

Large context: dynamic programming

- The challenge is implementing total functions

$$bi \; : (t \; : \; \mathbb{N}) \; \rightarrow \; (n \; : \; \mathbb{N}) \; \rightarrow \; \mathit{PolicySeq} \; t \; n$$

and

$$\mathit{biLemma} \; : (t \; : \; \mathbb{N}) \; \rightarrow \; (n \; : \; \mathbb{N}) \; \rightarrow \; \mathit{OptPolicySeq} \; (bi \; t \; n)$$

- The idea is that *bi* shall be a generic implementation of dynamic programming (Bellman 1957).
- It shall return a sequence of policies for n decision steps starting from decision step t for arbitrary n and t.

Large context: dynamic programming

- The challenge is implementing total functions

$$bi : (t : \mathbb{N}) \rightarrow (n : \mathbb{N}) \rightarrow PolicySeq\ t\ n$$

  and

$$biLemma : (t : \mathbb{N}) \rightarrow (n : \mathbb{N}) \rightarrow OptPolicySeq\ (bi\ t\ n)$$

- The idea is that *bi* shall be a generic implementation of dynamic programming (Bellman 1957).
- It shall return a sequence of policies for n decision steps starting from decision step t for arbitrary n and t.
- *biLemma* states that $bi\ t\ n$ shall be an optimal policy sequence.

Large context: dynamic programming

- We can get an intuition of the specification by looking at *PolicySeq* and *OptPolicySeq*:

Large context: dynamic programming

- We can get an intuition of the specification by looking at *PolicySeq* and *OptPolicySeq*:

  data *PolicySeq* : $(t : \mathbb{N}) \to (n : \mathbb{N}) \to$ *Type* **where**
  Nil : *PolicySeq* t Z
  $(::)$: *Policy* $t \to$ *PolicySeq* $(t+1)$ $n \to$ *PolicySeq* t $(n+1)$

Large context: dynamic programming

- We can get an intuition of the specification by looking at *PolicySeq* and *OptPolicySeq*:

$$
\begin{array}{l}
\textbf{data}\ \mathit{PolicySeq} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{Type}\ \textbf{where} \\
\quad \mathit{Nil} : \mathit{PolicySeq}\ t\ Z \\
\quad (::) : \mathit{Policy}\ t \to \mathit{PolicySeq}\ (t+1)\ n \to \mathit{PolicySeq}\ t\ (n+1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{OptPolicySeq} : \mathit{PolicySeq}\ t\ n \to \mathit{Type} \\
\mathit{OptPolicySeq}\ ps = (ps' : \mathit{PolicySeq}\ t\ n) \to (x : \mathit{State}\ t) \to \\
\qquad\qquad\qquad\qquad \mathit{val}\ x\ ps' \sqsubseteq \mathit{val}\ x\ ps
\end{array}
$$

Large context: dynamic programming

- We can get an intuition of the specification by looking at *PolicySeq* and *OptPolicySeq*:

$$\begin{aligned}
&\textbf{data}\ \mathit{PolicySeq}\ : (t\ :\ \mathbb{N})\ \to\ (n\ :\ \mathbb{N})\ \to\ \mathit{Type}\ \textbf{where}\\
&\quad \mathit{Nil}\ :\ \mathit{PolicySeq}\ t\ Z\\
&\quad (::)\ :\ \mathit{Policy}\ t\ \to\ \mathit{PolicySeq}\ (t+1)\ n\ \to\ \mathit{PolicySeq}\ t\ (n+1)
\end{aligned}$$

$$\begin{aligned}
&\mathit{OptPolicySeq}\ :\ \mathit{PolicySeq}\ t\ n\ \to\ \mathit{Type}\\
&\mathit{OptPolicySeq}\ ps = (ps'\ :\ \mathit{PolicySeq}\ t\ n)\ \to\ (x\ :\ \mathit{State}\ t)\ \to\\
&\qquad\qquad\qquad \mathit{val}\ x\ ps' \sqsubseteq \mathit{val}\ x\ ps
\end{aligned}$$

- This brings into the context *Policy*, *State*, *val*, $\sqsubseteq$.

Large context: dynamic programming

- We can get an intuition of the specification by looking at *PolicySeq* and *OptPolicySeq*:

  $$\begin{array}{l} \textbf{data}\ \mathit{PolicySeq} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{Type}\ \textbf{where} \\ \quad \mathit{Nil} : \mathit{PolicySeq}\ t\ Z \\ \quad (::) : \mathit{Policy}\ t \to \mathit{PolicySeq}\ (t+1)\ n \to \mathit{PolicySeq}\ t\ (n+1) \end{array}$$

  $$\begin{array}{l} \mathit{OptPolicySeq} : \mathit{PolicySeq}\ t\ n \to \mathit{Type} \\ \mathit{OptPolicySeq}\ ps = (ps' : \mathit{PolicySeq}\ t\ n) \to (x : \mathit{State}\ t) \to \\ \qquad \mathit{val}\ x\ ps' \sqsubseteq \mathit{val}\ x\ ps \end{array}$$

- This brings into the context *Policy*, *State*, *val*, $\sqsubseteq$.
- Giving the full context of *bi*, *biLemma* means formalizing the theory of dynamic programming.

Dynamic programming: an informal write up

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.
- The set of states observable at a given decision step can depend on that step.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.
- The set of states observable at a given decision step can depend on that step.
- The set of controls available to the decision maker in a given state can depend on that state.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.
- The set of states observable at a given decision step can depend on that step.
- The set of controls available to the decision maker in a given state can depend on that state.
- Selecting a control in a state entails a set of possible next states.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.
- The set of states observable at a given decision step can depend on that step.
- The set of controls available to the decision maker in a given state can depend on that state.
- Selecting a control in a state entails a set of possible next states.
- A triple (current state, current control, next state) entails a reward.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.
- The set of states observable at a given decision step can depend on that step.
- The set of controls available to the decision maker in a given state can depend on that state.
- Selecting a control in a state entails a set of possible next states.
- A triple (current state, current control, next state) entails a reward.
- The decision maker aims at maximising a sum of possible rewards over a fixed number of decision steps.

Dynamic programming: an informal write up

- DP is a method for solving sequential decision problems.
- SDPs are decision problems in which a decision maker picks up a sequence of controls.
- At each decision step, the decision maker observes a state and picks up a control.
- The set of states observable at a given decision step can depend on that step.
- The set of controls available to the decision maker in a given state can depend on that state.
- Selecting a control in a state entails a set of possible next states.
- A triple (current state, current control, next state) entails a reward.
- The decision maker aims at maximising a sum of possible rewards over a fixed number of decision steps.

Dynamic programming: SDPs

Dynamic programming: SDPs

- A SDP can be specified in terms of four functions:

Dynamic programming: SDPs

- A SDP can be specified in terms of four functions:

$$State \;:\; (t \;:\; \mathbb{N}) \;\rightarrow\; Type$$

Dynamic programming: SDPs

- A SDP can be specified in terms of four functions:

$$State \ : \ (t \ : \ \mathbb{N}) \ \to \ Type$$

$$Ctrl \ : \ (t \ : \ \mathbb{N}) \ \to \ (x \ : \ State\ t) \ \to \ Type$$

Dynamic programming: SDPs

- A SDP can be specified in terms of four functions:

$$State \;:\; (t \;:\; \mathbb{N}) \;\rightarrow\; Type$$

$$Ctrl \;:\; (t \;:\; \mathbb{N}) \;\rightarrow\; (x \;:\; State\ t) \;\rightarrow\; Type$$

$$next \;:\; (t \;:\; \mathbb{N}) \;\rightarrow\; (x \;:\; State\ t) \;\rightarrow\; (y \;:\; Ctrl\ t\ x) \;\rightarrow\; M\ (State\,(t+1))$$

Dynamic programming: SDPs

- M is a functor representing the problem's uncertainties:

Dynamic programming: SDPs

- M is a functor representing the problem's uncertainties:
  - $M \ : \ Type \ \rightarrow \ Type$

Dynamic programming: SDPs

- M is a functor representing the problem's uncertainties:
  - $M : Type \rightarrow Type$
  - $M = Id$ (deterministic uncertainty)

Dynamic programming: SDPs

- M is a functor representing the problem's uncertainties:
  - $M \; : \; Type \; \rightarrow \; Type$
  - $M = Id$ (deterministic uncertainty)
  - $M = List$ (non-deterministic uncertainty)

Dynamic programming: SDPs

- M is a functor representing the problem's uncertainties:
  - $M \;:\; Type \;\to\; Type$
  - $M = Id$ (deterministic uncertainty)
  - $M = List$ (non-deterministic uncertainty)
  - $M = Prob$ (stochastic uncertainty)
- In many problems $M = Prob$ or $M = List$ (Monadic dynamical systems, Ionescu 2009).

Dynamic programming: SDPs

- The fourth function defines the problem's rewards

Dynamic programming: SDPs

- The fourth function defines the problem's rewards

$$
\begin{aligned}
\mathit{reward} \;:\; & (t \;:\; \mathbb{N}) \;\rightarrow\; (x \;:\; \mathit{State}\ t) \;\rightarrow\; (y \;:\; \mathit{Ctrl}\ t\ x) \;\rightarrow \\
& (x' \;:\; \mathit{State}\,(t+1)) \;\rightarrow\; \mathit{Val}
\end{aligned}
$$

Dynamic programming: SDPs

- The fourth function defines the problem's rewards

$$\mathit{reward} : (t : \mathbb{N}) \to (x : \mathit{State}\ t) \to (y : \mathit{Ctrl}\ t\ x) \to (x' : \mathit{State}\ (t+1)) \to \mathit{Val}$$

- Thus $\mathit{map}\ (\mathit{reward}\ t\ x\ y)\ (\mathit{next}\ t\ x\ y) : M\ \mathit{Val}$

Dynamic programming: SDPs

- The fourth function defines the problem's rewards

$$\mathit{reward} \;:\; (t \;:\; \mathbb{N}) \;\to\; (x \;:\; \mathit{State}\; t) \;\to\; (y \;:\; \mathit{Ctrl}\; t\; x) \;\to$$
$$(x' \;:\; \mathit{State}\,(t+1)) \;\to\; \mathit{Val}$$

- Thus $\mathit{map}\,(\mathit{reward}\;t\;x\;y)\,(\mathit{next}\;t\;x\;y) \;:\; M\;\mathit{Val}$

- In many problems $\mathit{Val} = \mathbb{R}$ and sums are discounted sums of real numbers!

Dynamic programming: policies

- Policies are functions from states to controls

Dynamic programming: policies

- Policies are functions from states to controls

$$\begin{array}{l} \mathit{Policy} \quad : (t : \mathbb{N}) \rightarrow \mathit{Type} \\ \mathit{Policy}\ t = (x : \mathit{State}\ t) \rightarrow \mathit{Ctrl}\ t\ x \end{array}$$

Dynamic programming: policies

- Policies are functions from states to controls

$$
\begin{aligned}
&\mathit{Policy} \quad : (t : \mathbb{N}) \rightarrow \mathit{Type} \\
&\mathit{Policy}\ t = (x : \mathit{State}\ t) \rightarrow \mathit{Ctrl}\ t\ x
\end{aligned}
$$

- Policy sequences are sequences of policies:

Dynamic programming: policies

- Policies are functions from states to controls

$$\begin{aligned}&\mathit{Policy} \quad : (t : \mathbb{N}) \rightarrow \mathit{Type}\\&\mathit{Policy}\ t = (x : \mathit{State}\ t) \rightarrow \mathit{Ctrl}\ t\ x\end{aligned}$$

- Policy sequences are sequences of policies:

$$\begin{aligned}&\mathbf{data}\ \mathit{PolicySeq} : (t : \mathbb{N}) \rightarrow (n : \mathbb{N}) \rightarrow \mathit{Type}\ \mathbf{where}\\&\quad \mathit{Nil} : \mathit{PolicySeq}\ t\ Z\\&\quad (::) : \mathit{Policy}\ t \rightarrow \mathit{PolicySeq}\ (t+1)\ n \rightarrow \mathit{PolicySeq}\ t\ (n+1)\end{aligned}$$

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \; : \; \mathit{Val} \; \rightarrow \; \mathit{Val} \; \rightarrow \; \mathit{Type}$

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \; : \; \mathit{Val} \; \rightarrow \; \mathit{Val} \; \rightarrow \; \mathit{Type}$

- ... on how it adds them

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \;:\; \mathit{Val} \;\rightarrow\; \mathit{Val} \;\rightarrow\; \mathit{Type}$

- ... on how it adds them

  $\oplus \;:\; \mathit{Val} \;\rightarrow\; \mathit{Val} \;\rightarrow\; \mathit{Val}$

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \; : \; Val \; \rightarrow \; Val \; \rightarrow \; Type$

- ... on how it adds them

  $\oplus \; : \; Val \; \rightarrow \; Val \; \rightarrow \; Val$

- ... on a default "zero" value of type *Val*

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \; : \; \mathit{Val} \; \rightarrow \; \mathit{Val} \; \rightarrow \; \mathit{Type}$

- ... on how it adds them

  $\oplus \; : \; \mathit{Val} \; \rightarrow \; \mathit{Val} \; \rightarrow \; \mathit{Val}$

- ... on a default "zero" value of type *Val*

  $\mathit{zero} \; : \; \mathit{Val}$

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards
  $\sqsubseteq \; : \; Val \; \rightarrow \; Val \; \rightarrow \; Type$

- ... on how it adds them
  $\oplus \; : \; Val \; \rightarrow \; Val \; \rightarrow \; Val$

- ... on a default "zero" value of type Val
  $zero \; : \; Val$

- and on how it measures uncertain outcomes

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \ : \ Val \ \rightarrow \ Val \ \rightarrow \ Type$

- ... on how it adds them

  $\oplus \ : \ Val \ \rightarrow \ Val \ \rightarrow \ Val$

- ... on a default "zero" value of type *Val*

  $zero \ : \ Val$

- and on how it measures uncertain outcomes

  $meas \ : \ M \ Val \ \rightarrow \ Val$

Dynamic programming: optimality

- The notion of optimality for policy sequences depends on how the decision maker compares rewards

  $\sqsubseteq \ : \ Val \ \rightarrow \ Val \ \rightarrow \ Type$

- ... on how it adds them

  $\oplus \ : \ Val \ \rightarrow \ Val \ \rightarrow \ Val$

- ... on a default "zero" value of type *Val*

  $zero \ : \ Val$

- and on how it measures uncertain outcomes

  $meas \ : \ M \ Val \ \rightarrow \ Val$

Often $Val = \mathbb{R}$, $\sqsubseteq = lift \leqslant$, $\oplus = +$, $zero = 0$ and *meas* is the expected value function.

Dynamic programming: optimality

- With $\sqsubseteq$, $\oplus$ and *meas*, one can compute the value of taking n decisions starting from some initial state and according to a policy sequence *ps*:

$$val \; : \; (x \; : \; State \; t) \; \rightarrow \; PolicySeq \; t \; n \; \rightarrow \; Val$$

Dynamic programming: optimality

- With $\sqsubseteq$, $\oplus$ and *meas*, one can compute the value of taking n decisions starting from some initial state and according to a policy sequence *ps*:

$$val \; : \; (x \; : \; State \; t) \; \rightarrow \; PolicySeq \; t \; n \; \rightarrow \; Val$$

$$val \; \{t\} \; \{n = Z\} \; x \; Nil = zero$$

Dynamic programming: optimality

- With $\sqsubseteq$, $\oplus$ and *meas*, one can compute the value of taking n decisions starting from some initial state and according to a policy sequence *ps*:

$$val \; : \; (x \; : \; State \; t) \; \to \; PolicySeq \; t \; n \; \to \; Val$$

$$val \; \{t\} \; \{n = Z\} \; x \; Nil = zero$$

$$\begin{array}{l} val \; \{t\} \; \{n = m+1\} \; x \; (p :: ps) = meas \; (fmap \; f \; mx') \; \textbf{where} \\ \quad y \quad\;\; : \; Ctrl \; t \; x \\ \quad y \quad\;\; = p \; x \\ \quad mx' \; : \; M \; (State \; (t+1)) \\ \quad mx' = next \; t \; x \; y \\ \quad f \quad\;\; : \; State \; (t+1) \; \to \; Val \\ \quad f \; x' = reward \; t \; x \; y \; x' \oplus val \; x' \; ps \end{array}$$

Dynamic programming: optimality

- ... formalize the notion of optimality for policy sequences

Dynamic programming: optimality

- … formalize the notion of optimality for policy sequences

$$
\begin{aligned}
&\mathit{OptPolicySeq} \;:\; \mathit{PolicySeq}\ t\ n \;\rightarrow\; \mathit{Type}\\
&\mathit{OptPolicySeq}\ ps = (ps' \;:\; \mathit{PolicySeq}\ t\ n) \;\rightarrow\; (x \;:\; \mathit{State}\ t) \;\rightarrow\\
&\qquad\qquad\qquad\qquad \mathit{val}\ x\ ps' \sqsubseteq \mathit{val}\ x\ ps
\end{aligned}
$$

Dynamic programming: optimality

- ... formalize the notion of optimality for policy sequences

$$
\begin{aligned}
&\mathit{OptPolicySeq} \ : \ \mathit{PolicySeq}\ t\ n \ \to \ \mathit{Type} \\
&\mathit{OptPolicySeq}\ ps = (ps' \ : \ \mathit{PolicySeq}\ t\ n) \ \to \ (x \ : \ \mathit{State}\ t) \ \to \\
&\qquad\qquad\qquad\qquad\quad \mathit{val}\ x\ ps' \sqsubseteq \mathit{val}\ x\ ps
\end{aligned}
$$

- ... and derive

Dynamic programming: optimality

- ... formalize the notion of optimality for policy sequences

$$\begin{aligned}
&\mathit{OptPolicySeq} \;:\; \mathit{PolicySeq}\; t\; n \;\to\; \mathit{Type} \\
&\mathit{OptPolicySeq}\; \mathit{ps} = (\mathit{ps}' \;:\; \mathit{PolicySeq}\; t\; n) \;\to\; (x \;:\; \mathit{State}\; t) \;\to \\
&\qquad\qquad\qquad\qquad \mathit{val}\; x\; \mathit{ps}' \sqsubseteq \mathit{val}\; x\; \mathit{ps}
\end{aligned}$$

- ... and derive

$$\mathit{bi} \;:\; (t \;:\; \mathbb{N}) \;\to\; (n \;:\; \mathbb{N}) \;\to\; \mathit{PolicySeq}\; t\; n$$

$$\mathit{biLemma} \;:\; (t \;:\; \mathbb{N}) \;\to\; (n \;:\; \mathbb{N}) \;\to\; \mathit{OptPolicySeq}\; (\mathit{bi}\; t\; n)$$

Dynamic programming: optimality

- ... formalize the notion of optimality for policy sequences

$$
\begin{aligned}
&\mathit{OptPolicySeq} \;:\; \mathit{PolicySeq}\ t\ n \;\to\; \mathit{Type}\\
&\mathit{OptPolicySeq}\ ps = (ps' \;:\; \mathit{PolicySeq}\ t\ n) \;\to\; (x \;:\; \mathit{State}\ t) \;\to\\
&\qquad\qquad\qquad\qquad \mathit{val}\ x\ ps' \sqsubseteq \mathit{val}\ x\ ps
\end{aligned}
$$

- ... and derive

$$\mathit{bi} \;:\; (t \;:\; \mathbb{N}) \;\to\; (n \;:\; \mathbb{N}) \;\to\; \mathit{PolicySeq}\ t\ n$$

$$\mathit{biLemma} \;:\; (t \;:\; \mathbb{N}) \;\to\; (n \;:\; \mathbb{N}) \;\to\; \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)$$

- from Bellman's principle of optimality.

Specifications in large contexts

Specifications in large contexts

- *State*, *Ctrl*, *M*, *Policy*, $\sqsubseteq$ have been introduced as functions that return values of type *Type*.

Specifications in large contexts

- *State*, *Ctrl*, *M*, *Policy*, $\sqsubseteq$ have been introduced as functions that return values of type *Type*.
- Perhaps it would have been better to use data declarations instead?

Specifications in large contexts

- *State*, *Ctrl*, *M*, *Policy*, $\sqsubseteq$ have been introduced as functions that return values of type *Type*.

- Perhaps it would have been better to use data declarations instead? When do we use functions? When data declarations?

Specifications in large contexts

- *State*, *Ctrl*, *M*, *Policy*, $\sqsubseteq$ have been introduced as functions that return values of type *Type*.
- Perhaps it would have been better to use data declarations instead? When do we use functions? When data declarations?
- We need more than just *State*, *Ctrl*, *M*, *next*, *reward*, $\sqsubseteq$, $\oplus$ and *meas* to specify the context of a DP problem.

Specifications in large contexts

- *State*, *Ctrl*, *M*, *Policy*, $\sqsubseteq$ have been introduced as functions that return values of type *Type*.
- Perhaps it would have been better to use data declarations instead? When do we use functions? When data declarations?
- We need more than just *State*, *Ctrl*, *M*, *next*, *reward*, $\sqsubseteq$, $\oplus$ and *meas* to specify the context of a DP problem.
- For instance, we need to require *M* to be a container monad, $\sqsubseteq$ to be a total preorder, $\oplus$ to be monotone with respect to $\sqsubseteq$ ...

Specifications in large contexts

- Type classes are useful to inject a context in the scope of a functions but . . .

Specifications in large contexts

- Type classes are useful to inject a context in the scope of a functions but ...
- How to stipulate *Monad M* for a module?

Specifications in large contexts

- Type classes are useful to inject a context in the scope of a functions but ...
- How to stipulate *Monad M* for a module?
- We have not been able to use type classes effectively to structure the context of *bi*, *biLemma*!

Specifications in large contexts

- Type classes are useful to inject a context in the scope of a functions but ...
- How to stipulate *Monad M* for a module?
- We have not been able to use type classes effectively to structure the context of *bi*, *biLemma*!
- We need to formalize properties of context elements whose implementation is deferred to applications, for instance

  $$\mathit{finiteAllViable} \ : \ \mathit{FiniteAll} \ \rightarrow \ \mathit{FiniteViable} \ \rightarrow \ \mathit{FiniteAllViable}$$

Specifications in large contexts

- Type classes are useful to inject a context in the scope of a functions but ...
- How to stipulate *Monad M* for a module?
- We have not been able to use type classes effectively to structure the context of *bi*, *biLemma*!
- We need to formalize properties of context elements whose implementation is deferred to applications, for instance

  $$\mathit{finiteAllViable} \;:\; \mathit{FiniteAll} \;\rightarrow\; \mathit{FiniteViable} \;\rightarrow\; \mathit{FiniteAllViable}$$

- This has turned out to be problematic due to current language limitations: explicit filling in of scoped (not top level) implicits is not yet implemented.

Preliminary conclusions, guidelines

Preliminary conclusions, guidelines

- In DP we can implement a generic verified *bi* without relying on unimplementable postulates.

Preliminary conclusions, guidelines

- In DP we can implement a generic verified *bi* without relying on unimplementable postulates.

- In formalizations of probability theory, numerical analysis, machine learning, unimplementable postulated are unavoidable.

Preliminary conclusions, guidelines

- In DP we can implement a generic verified *bi* without relying on unimplementable postulates.
- In formalizations of probability theory, numerical analysis, machine learning, unimplementable postulated are unavoidable.
- This leads to notions of correctness that are conditional and incremental: one can type check a program to be correct but one cannot compute a correctness proof.

Preliminary conclusions, guidelines

- Conditional, incrementally correct implemetations require separating programs from specifications:

Preliminary conclusions, guidelines

- Conditional, incrementally correct implemetations require separating programs from specifications:

$$
\begin{aligned}
&\mathit{index} &&: \mathit{Fin}\ n \ \rightarrow\ \mathit{Vect}\ n\ X \ \rightarrow\ X \\
&\mathit{indexSpec} &&: (k\ :\ \mathit{Fin}\ n)\ \rightarrow\ (\mathit{xs}\ :\ \mathit{Vect}\ n\ X)\ \rightarrow \\
& &&\quad \mathit{Elem}\ (\mathit{index}\ k\ \mathit{xs})\ \mathit{xs}
\end{aligned}
$$

Preliminary conclusions, guidelines

- Conditional, incrementally correct implemetations require separating programs from specifications:

$$
\begin{aligned}
&\mathit{index} &&: \mathit{Fin}\ n \ \rightarrow\ \mathit{Vect}\ n\ X \ \rightarrow\ X \\
&\mathit{indexSpec} &&: (k\ :\ \mathit{Fin}\ n)\ \rightarrow\ (\mathit{xs}\ :\ \mathit{Vect}\ n\ X)\ \rightarrow \\
& && \quad \mathit{Elem}\ (\mathit{index}\ k\ \mathit{xs})\ \mathit{xs}
\end{aligned}
$$

not

$$
\begin{aligned}
&\mathit{index}\ : (k\ :\ \mathit{Fin}\ n)\ \rightarrow\ (\mathit{xs}\ :\ \mathit{Vect}\ n\ X)\ \rightarrow \\
&\qquad \Sigma\ X\ (\lambda x \Rightarrow \mathit{Elem}\ x\ \mathit{xs})
\end{aligned}
$$

Preliminary conclusions, guidelines

- Conditional, incrementally correct implemetations require separating programs from specifications:

$$\begin{aligned} &\mathit{index} &&: \mathit{Fin}\ n \ \to\ \mathit{Vect}\ n\ X \ \to\ X \\ &\mathit{indexSpec} &&: (k\ :\ \mathit{Fin}\ n)\ \to\ (\mathit{xs}\ :\ \mathit{Vect}\ n\ X)\ \to \\ & && \quad \mathit{Elem}\ (\mathit{index}\ k\ \mathit{xs})\ \mathit{xs} \end{aligned}$$

  not

$$\begin{aligned} \mathit{index}\ :\ & (k\ :\ \mathit{Fin}\ n)\ \to\ (\mathit{xs}\ :\ \mathit{Vect}\ n\ X)\ \to \\ & \Sigma\ X\ (\lambda x \Rightarrow \mathit{Elem}\ x\ \mathit{xs}) \end{aligned}$$

- For a program or data type, one would like a minimal set of specifications that allows the proving useful results about that program independently of its implementation.

- Small context: vector indexing and lookup
- Large context: dynamic programming
- Specifications in large contexts
- Preliminary conclusions
- Dynamic programming continued

Dynamic programming: Bellman's principle of optimality

- Bellman's principle rests on the notion of optimal extension of a policy sequence:

Dynamic programming: Bellman's principle of optimality

- Bellman's principle rests on the notion of optimal extension of a policy sequence:

$$
\begin{aligned}
&\mathit{OptExt} &&: \mathit{PolicySeq}\ (t+1)\ m \ \to\ \mathit{Policy}\ t \ \to\ \mathit{Type} \\
&\mathit{OptExt}\ \mathit{ps}\ p &&= (x\ :\ \mathit{State}\ t) \ \to\ (p'\ :\ \mathit{Policy}\ t) \ \to \\
& && \quad \mathit{val}\ x\ (p' :: \mathit{ps}) \sqsubseteq \mathit{val}\ x\ (p :: \mathit{ps})
\end{aligned}
$$

Dynamic programming: Bellman's principle of optimality

- Bellman's principle rests on the notion of optimal extension of a policy sequence:

$$
\begin{aligned}
&\mathit{OptExt} \quad : \ \mathit{PolicySeq}\ (t+1)\ m \ \to\ \mathit{Policy}\ t \ \to\ \mathit{Type} \\
&\mathit{OptExt}\ \mathit{ps}\ p = (x\ :\ \mathit{State}\ t) \ \to\ (p'\ :\ \mathit{Policy}\ t) \ \to \\
&\qquad\qquad\qquad \mathit{val}\ x\ (p' :: \mathit{ps}) \sqsubseteq \mathit{val}\ x\ (p :: \mathit{ps})
\end{aligned}
$$

- With this notion, Bellman's principle can be formulated as

$$
\begin{aligned}
\mathit{Bellman}\ :\ &(\mathit{ps}\ :\ \mathit{PolicySeq}\ (t+1)\ m) \ \to\ \mathit{OptPolicySeq}\ \mathit{ps} \ \to \\
&(p\ \ :\ \mathit{Policy}\ t) \ \to\ \mathit{OptExt}\ \mathit{ps}\ p \ \to \\
&\mathit{OptPolicySeq}\ (p :: \mathit{ps})
\end{aligned}
$$

Dynamic programming: Bellman's principle of optimality

- Bellman's principle rests on the notion of optimal extension of a policy sequence:

$$\begin{aligned} & \mathit{OptExt} \quad\quad\; : \; \mathit{PolicySeq}\ (t+1)\ m \;\to\; \mathit{Policy}\ t \;\to\; \mathit{Type} \\ & \mathit{OptExt}\ ps\ p = (x\ :\ \mathit{State}\ t) \;\to\; (p'\ :\ \mathit{Policy}\ t) \;\to \\ & \qquad\qquad\qquad \mathit{val}\ x\ (p' :: ps) \sqsubseteq \mathit{val}\ x\ (p :: ps) \end{aligned}$$

- With this notion, Bellman's principle can be formulated as

$$\begin{aligned} \mathit{Bellman}\ :\ & (ps\ :\ \mathit{PolicySeq}\ (t+1)\ m) \;\to\; \mathit{OptPolicySeq}\ ps \;\to \\ & (p\ \ :\ \mathit{Policy}\ t) \;\to\; \mathit{OptExt}\ ps\ p \;\to \\ & \mathit{OptPolicySeq}\ (p :: ps) \end{aligned}$$

- We can implement *Bellman* if ...

Dynamic programming: minimal requirements

- ... $\sqsubseteq$ is reflexive and transitive.

Dynamic programming: minimal requirements

- ... $\sqsubseteq$ is reflexive and transitive.
- $\oplus$ is monotone w.r.t. $\sqsubseteq$:

  $$monotonePlusLTE \ : \ a \sqsubseteq b \ \rightarrow \ c \sqsubseteq d \ \rightarrow \ (a \oplus c) \sqsubseteq (b \oplus d)$$

Dynamic programming: minimal requirements

- ... $\sqsubseteq$ is reflexive and transitive.
- $\oplus$ is monotone w.r.t. $\sqsubseteq$:

  $$\mathit{monotonePlusLTE} : a \sqsubseteq b \rightarrow c \sqsubseteq d \rightarrow (a \oplus c) \sqsubseteq (b \oplus d)$$

- *meas* fulfills a monotonicity condition (Ionescu 2009):

  $$\begin{aligned}\mathit{measMon} : \; & \{A : \mathit{Type}\} \rightarrow \\ & (f : A \rightarrow \mathit{Val}) \rightarrow (g : A \rightarrow \mathit{Val}) \rightarrow \\ & ((a : A) \rightarrow (f\ a) \sqsubseteq (g\ a)) \rightarrow \\ & (ma : M\ A) \rightarrow \\ & \mathit{meas}\ (\mathit{fmap}\ f\ ma) \sqsubseteq \mathit{meas}\ (\mathit{fmap}\ g\ ma)\end{aligned}$$

Dynamic programming: minimal requirements

- ... $\sqsubseteq$ is reflexive and transitive.
- $\oplus$ is monotone w.r.t. $\sqsubseteq$:

  $$monotonePlusLTE \ : \ a \sqsubseteq b \ \rightarrow \ c \sqsubseteq d \ \rightarrow \ (a \oplus c) \sqsubseteq (b \oplus d)$$

- *meas* fulfills a monotonicity condition (Ionescu 2009):

  $$\begin{aligned} measMon \ : \ & \{A \ : \ Type\} \ \rightarrow \\ & (f \ : \ A \ \rightarrow \ Val) \ \rightarrow \ (g \ : \ A \ \rightarrow \ Val) \ \rightarrow \\ & ((a \ : \ A) \ \rightarrow \ (f \ a) \sqsubseteq (g \ a)) \ \rightarrow \\ & (ma \ : \ M \ A) \ \rightarrow \\ & meas \ (fmap \ f \ ma) \sqsubseteq meas \ (fmap \ g \ ma) \end{aligned}$$

Proof idea: $val \ x \ (p' :: ps') \sqsubseteq val \ x \ (p' :: ps) \sqsubseteq val \ x \ (p :: ps)$ and transitivity of $\sqsubseteq$.

Dynamic programming: *bi*

Dynamic programming: *bi*

- How can we take advantage of Bellman's principle?

Dynamic programming: *bi*

- How can we take advantage of Bellman's principle?
- Assume that we can compute optimal extensions of arbitrary policy sequences:

$$
\begin{aligned}
\mathit{optExt} &: \mathit{PolicySeq}\ (t+1)\ n \ \to\ \mathit{Policy}\ t \\
\mathit{optExtLemma} &: (\mathit{ps} \ :\ \mathit{PolicySeq}\ (t+1)\ n) \ \to \\
&\ \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})
\end{aligned}
$$

Dynamic programming: *bi*

- How can we take advantage of Bellman's principle?
- Assume that we can compute optimal extensions of arbitrary policy sequences:

$$
\begin{aligned}
&\mathit{optExt} &&: \mathit{PolicySeq}\ (t+1)\ n \ \rightarrow\ \mathit{Policy}\ t \\
&\mathit{optExtLemma} &&: (\mathit{ps}\ :\ \mathit{PolicySeq}\ (t+1)\ n)\ \rightarrow \\
& && \quad \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})
\end{aligned}
$$

- Then

$$
\begin{aligned}
&\mathit{bi}\ t\ Z &&= \mathit{Nil} \\
&\mathit{bi}\ t\ (n+1) &&= \mathit{optExt}\ \mathit{ps} :: \mathit{ps}\ \textbf{where}\ \mathit{ps} = \mathit{bi}\ (t+1)\ n
\end{aligned}
$$

is correct.

Dynamic programming: *bi* is correct

- The task is to implement

  $$biLemma : (t : \mathbb{N}) \rightarrow (n : \mathbb{N}) \rightarrow OptPolicySeq\ (bi\ t\ n)$$

  for

  $$\begin{aligned} bi\ t\ Z &= Nil \\ bi\ t\ (m+1) &= optExt\ ps :: ps\ \textbf{where}\ ps = bi\ (t+1)\ m \end{aligned}$$

Dynamic programming: *bi* is correct

- The task is to implement

$$biLemma : (t : \mathbb{N}) \rightarrow (n : \mathbb{N}) \rightarrow OptPolicySeq\,(bi\ t\ n)$$

  for

$$\begin{aligned} &bi\ t\ Z &&= Nil \\ &bi\ t\ (m+1) &&= optExt\ ps :: ps\ \textbf{where}\ ps = bi\ (t+1)\ m \end{aligned}$$

- Case $n = 0$: reflexivity of $\sqsubseteq$.

Dynamic programming: *bi* is correct

- The task is to implement

$$\mathit{biLemma} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)$$

  for

$$\begin{aligned}
&\mathit{bi}\ t\ Z && = \mathit{Nil}\\
&\mathit{bi}\ t\ (m+1) && = \mathit{optExt}\ \mathit{ps} :: \mathit{ps}\ \textbf{where}\ \mathit{ps} = \mathit{bi}\ (t+1)\ m
\end{aligned}$$

- Case $n = 0$: reflexivity of $\sqsubseteq$.
- Case $n = m + 1$: induction on *biLemma*:

$$\begin{aligned}
&\mathit{biLemma}\ t\ (m+1) = \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}\ \textbf{where}\\
&\quad \begin{array}{lll} \mathit{ps} & : \mathit{PolicySeq}\ (t+1)\ m; & \mathit{ps} \;\; = \mathit{bi}\ (t+1)\ m\\
\mathit{ops} & : \mathit{OptPolicySeq}\ \mathit{ps}; & \mathit{ops} = \mathit{biLemma}\ (t+1)\ m\\
p & : \mathit{Policy}\ t; & p \;\;\; = \mathit{optExt}\ \mathit{ps}\\
\mathit{oep} & : \mathit{OptExt}\ \mathit{ps}\ p; & \mathit{oep} = \mathit{optExtLemma}\ \mathit{ps}
\end{array}
\end{aligned}$$

Dynamic programming: optimal extensions

- Under which conditions can one compute optimal extensions

$$\begin{aligned} \mathit{optExt} &: \mathit{PolicySeq}\ (t+1)\ n \ \rightarrow\ \mathit{Policy}\ t \\ \mathit{optExtLemma} &: (\mathit{ps}\ :\ \mathit{PolicySeq}\ (t+1)\ n)\ \rightarrow \\ &\quad \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps}) \end{aligned}$$

  of arbitrary policy sequences?

Dynamic programming: optimal extensions

- Under which conditions can one compute optimal extensions

$$
\begin{aligned}
&\mathit{optExt} &&: \mathit{PolicySeq}\ (t+1)\ n\ \rightarrow\ \mathit{Policy}\ t\\
&\mathit{optExtLemma} &&: (\mathit{ps}\ :\ \mathit{PolicySeq}\ (t+1)\ n)\ \rightarrow\\
& &&\quad \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})
\end{aligned}
$$

  of arbitrary policy sequences?

- This is for another talk but ...