On the correctness of monadic backward induction

Downloaded from: https://research.chalmers.se, 2026-11-19 02:24 UTC

Citation for the original published paper (version of record):
Brede, N., Botta, N. (2021). On the correctness of monadic backward induction. Journal of Functional Programming, 31. http://dx.doi.org/10.1017/S0956796821000228

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

JFP **31**, e26, 39 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

doi:10.1017/S0956796821000228

On the correctness of monadic backward induction

NURIA BREDE

University of Potsdam, Potsdam, Germany
Potsdam Institute for Climate Impact Research, Potsdam, Germany
(*e-mail:* brede@uni-potsdam.de)

NICOLA BOTTA

Potsdam Institute for Climate Impact Research, Potsdam, Germany
Chalmers University of Technology, Göteborg, Sweden
(*e-mail:* botta@pik-potsdam.de)

Abstract

In control theory, to solve a finite-horizon sequential decision problem (SDP) commonly means to find a list of decision rules that result in an optimal expected total reward (or cost) when taking a given number of decision steps. SDPs are routinely solved using Bellman's backward induction. Textbook authors (e.g. Bertsekas or Puterman) typically give more or less formal proofs to show that the backward induction algorithm is correct as solution method for deterministic and stochastic SDPs. Botta, Jansson and Ionescu propose a generic framework for finite horizon, *monadic* SDPs together with a monadic version of backward induction for solving such SDPs. In monadic SDPs, the monad captures a generic notion of uncertainty, while a generic measure function aggregates rewards. In the present paper, we define a notion of correctness for monadic SDPs and identify three conditions that allow us to prove a correctness result for monadic backward induction that is comparable to textbook correctness proofs for ordinary backward induction. The conditions that we impose are fairly general and can be cast in category-theoretical terms using the notion of Eilenberg–Moore algebra. They hold in familiar settings like those of deterministic or stochastic SDPs, but we also give examples in which they fail. Our results show that backward induction can safely be employed for a broader class of SDPs than usually treated in textbooks. However, they also rule out certain instances that were considered admissible in the context of Botta *et al.* 's generic framework. Our development is formalised in Idris as an extension of the Botta *et al.* framework and the sources are available as supplementary material.

1 Introduction

Backward induction is a method introduced by Bellman (1957) that is routinely used to solve *finite-horizon sequential decision problems* (SDPs). Such problems lie at the core of many applications in economics, logistics and computer science (Finus *et al.*, 2003; Helm, 2003; Heitzig, 2012; Gintis, 2007; Botta *et al.*, 2013; De Moor, 1995, 1999). Examples include inventory, scheduling and shortest path problems, but also the search for optimal strategies in games (Bertsekas, 1995; Diederich, 2001).

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Botta, Jansson & Ionescu (2017a) propose a generic framework for *monadic* finite-horizon SDPs as generalisation of the deterministic, non-deterministic and stochastic SDPs treated in control theory textbooks (Bertsekas, 1995; Puterman, 2014). This framework allows to specify such problems and to solve them with a generic version of backward induction that we will refer to as *monadic backward induction*.

The Botta–Jansson–Ionescu-framework, subsequently referred to as *BJI-framework*, *BJI-theory* or simply *framework*, already includes a verification of monadic backward induction with respect to a certain underlying *value* function (see Section 3.2). However, in the literature on stochastic SDPs, this formulation of the function is itself part of the backward induction algorithm and needs to be verified against an optimisation criterion, the *expected total reward* (Puterman, 2014, Ch. 4.2). For stochastic SDPs semi-formal proofs can be found in textbooks – but monadic SDPs are substantially more general than the stochastic SDPs for which these results are established. This observation raises a number of questions:

- What exactly should "correctness" mean for a solution of monadic SDPs?
- Does monadic backward induction provide correct solutions in this sense for monadic SDPs in their full generality?
- And if not, is there a class of monadic SDPs for which monadic backward induction does provide provably correct solutions?

In the present paper, we address these questions and make the following contributions to answering them:

- We put forward a formal specification that monadic backward induction should meet in order to be considered "correct" as solution method for monadic SDPs. This specification uses an optimisation criterion that is a generic version of the *expected total reward* of standard control theory textbooks.[1] In analogy, we call this criterion *measured total reward*.
- We consider the value function underlying monadic backward induction as "correct" if it computes the *measured total reward*.
- If the value function is correct, monadic backward induction can be proven to be correct in our sense by extending the result of Botta *et al.* (2017*a*). However, we show that this is not necessarily the case, i.e. the value function does not compute the *measured total reward* for arbitrary monadic SDPs.
- We therefore formulate conditions that identify a class of monadic SDPs for which the value function and thus monadic backward induction can be shown to be correct. The conditions are fairly simple and allow for a neat description in category-theoretical terms using the notion of Eilenberg-Moore-algebra.
- We give a formalised proof that monadic backward induction fulfils the correctness criterion if the conditions hold. This correctness result can be seen as a generic version of correctness results for standard backward induction like Bertsekas (1995, Prop. 1.3.1) and Puterman (2014, Th. 4.5.1.c).

[1] Note that in control theory backward induction is often referred to as *the dynamic programming algorithm* where the term *dynamic programming* is used in the original sense of Bellman (1957).

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Our results rule out the application of backward induction to certain monadic SDPs that were previously considered as admissible in the BJI-framework. Thus, they complement the verification result of Botta *et al.* and provide a necessary clarification. Still, the new conditions are simple enough to be checked for non-standard instantiations of the framework. This allows to broaden the applicability of backward induction to settings which are not commonly discussed in the literature and to obtain a formalised proof of correctness with considerably less effort. It is worth stressing that our conditions can be useful for anyone interested in applying monadic backward induction in non-standard situations – completely independent of the BJI-framework.

Finally, the value function underlying monadic backward induction is also interesting in itself. Given the conditions hold, it can be used to compute the measured total reward efficiently, using a method reminiscent of a *thinning* algorithm (Bird & Gibbons, 2020, Ch. 10).

For the reader unfamiliar with SDPs, we provide a brief informal overview and two simple examples in the next section. We recap the BJI-framework and its (partial) verification result for monadic backward induction in Section 3. In Section 4, we specify correctness for monadic backward induction and the BJI-value function. We also show that in the general monadic case the value function does not necessarily meet the specification. To resolve this problem, we identify conditions under which the value function does meet the specification. These conditions are stated and analysed in Section 5. In Section 6, we prove that, given the conditions hold, the BJI-value function and monadic backward induction are correct in the sense defined in Section 4. We discuss the conditions from a more abstract perspective in Section 7 and conclude in Section 8.

Throughout the paper, we use Idris as our host language (Brady, 2013, 2017). We assume some familiarity with Haskell-like syntax and notions like *functor* and *monad* as used in functional programming. We tacitly consider types as logical statements and programs as proofs, justified by the propositions-as-types correspondence (for an accessible introduction see Wadler, 2015).

Source code. Our development is formalised in Idris as an extension of a lightweight version of the BJI-framework. The proofs are machine-checked and the source code is available as supplementary material attached to this paper. The sources of this document have been written in literal Idris and are available at Brede & Botta (2021), together with some example code. All source files can be type checked with Idris 1.3.2.

2 Finite-horizon sequential decision problems

In deterministic, non-deterministic and stochastic finite-horizon SDPs, a decision maker seeks to control the evolution of a *(dynamical) system* at a finite number of *decision steps* by selecting certain *controls* in sequence, one after the other. The controls available to the decision maker at a given decision step typically depend on the *state* of the system at that step.

In *deterministic* problems, selecting a control in a state at decision step $t : \mathbb{N}$, determines a unique next state at decision step $t + 1$ through a given *transition function*. In

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

non-deterministic problems, the transition function yields a whole set of *possible* states at the next decision step. In *stochastic* problems, the transition function yields a *probability distribution* on states at the next decision step.

The notion of *monadic* problem generalises that of deterministic, non-deterministic and stochastic problem through a transition function that yields an M-structure of next states where M is a monad. For example, the identity monad can be applied to model deterministic systems. Non-deterministic systems can be represented in terms of transition functions that return lists (or some other representations of sets) of next states. Stochastic systems can be represented in terms of probability distribution monads (Giry, 1981; Erwig & Kollmansberger, 2006; Audebaud & Paulin-Mohring, 2009; Jacobs, 2011). The uncertainty monad, the states, the controls and the next function define what is often called a *decision process*.

The idea of sequential decision problems is that each single decision yields a *reward* and these rewards add up to a *total reward* over all decision steps. Rewards are often represented by values of a numeric type and added up using the canonical addition. If the transition function and thus the evolution of the system is not deterministic, then the resulting possible total rewards need to be aggregated to yield a single outcome value. In stochastic SDPs, evolving the underlying stochastic system leads to a probability distribution on total rewards which is usually aggregated using the familiar *expected value* measure. The value thus obtained is called the *expected total reward* (Puterman, 2014, ch. 4.1.2) and its role is central: It is the quantity that is to be optimised in an SDP.

In monadic SDPs, the measure is generic, i.e. it is not fixed in advance but has to be given as part of the specification of a concrete problem. Therefore, we will generalise the notion of *expected total reward* to a corresponding notion for monadic SDPs that we call *measured total reward* in analogy (see Section 4).

Solving a stochastic SDP consists in *finding a list of rules for selecting controls that maximises the expected total reward for n decision steps when starting at decision step t*. Similarly, we define that *solving a monadic SDP* consists in *finding a list of rules for selecting controls that maximises the measured total reward*. This means that when starting from any initial state at decision step t, following the computed list of rules for selecting controls will result in a value that is maximal as measure of the sum of rewards along all possible trajectories rooted in that initial state.

Equivalently, rewards can instead be considered as *costs* that need to be *minimised*. This dual perspective is taken e.g. in Bertsekas (1995). In the subsequent sections, we will follow the terminology of the BJI-framework and Puterman (2014) and speak of "rewards", but our second example below will illustrate the "cost" perspective.

In mathematical theories of optimal control, the rules for selecting controls are called *policies*. A *policy* for a decision step is simply a function that maps each possible state to a control. As mentioned above, the controls available in a given state typically depend on that state, thus policies are dependently typed functions. A list of such policies is called a *policy sequence*.

The central idea underlying backward induction is to compute a globally optimal solution of a multi-step SDP incrementally by solving local optimisation problems at each decision step. This is captured by *Bellman's principle*: *Extending an optimal policy sequence with an optimal policy yields again an optimal policy sequence*. However, as

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

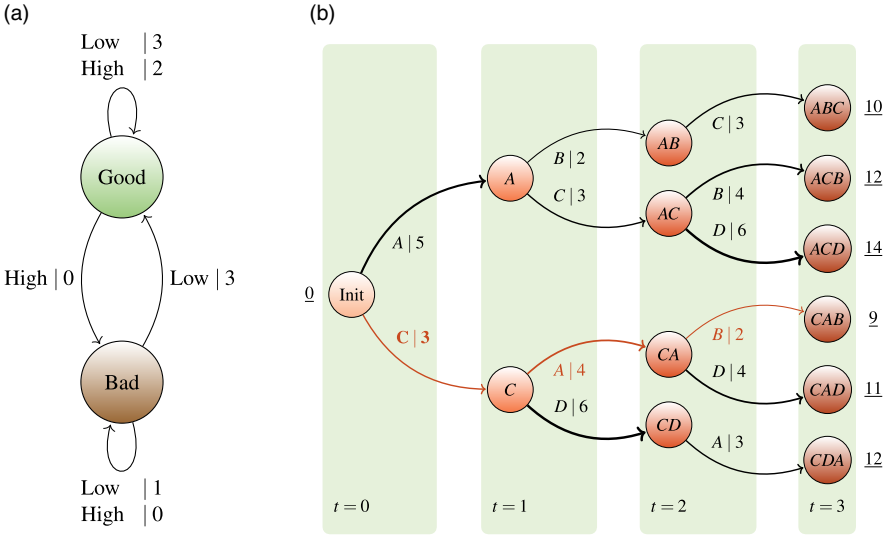


Fig. 1: Transition graphs for the example SDPs described in Section 2. The edge labels denote pairs *control* | *reward* for the associated transitions. In the first example, state and control spaces are constant over time therefore we have omitted the temporal dimension.

we will see in Section 4.2, one has to carefully check whether for a given SDP backward induction is indeed applicable.

Two features are crucial for finite-horizon, monadic SDPs to be solvable with the BJI-framework that we study in this paper: (1) the number of decision steps has to be given explicitly as input to the backward induction and (2) at each decision step, the number of possible next states has to be *finite*. While (2) is a necessary condition for backward induction to be computable, (1) is a genuine limitation of the BJI-framework: in many SDPs, for example in a game of tic-tac-toe, the number of decision steps is bounded but not known a priori.

Before we discuss the BJI-framework in the next section, we illustrate the notion of sequential decision problem with two simple examples, one in which the purpose is to maximise rewards and one in which the purpose is to minimise costs. Rewards and costs in these examples are just natural numbers and are summed up with ordinary addition. The first example is a non-deterministic SDP. Although it is somewhat oversimplified, it has the advantage of being tractable for computations by hand while still being sufficient as basis for illustrations in Sections 3–5. The second example is a deterministic SDP that stands for the important class of scheduling SDPs. This problem highlights why dependent types are necessary to model state and control spaces accurately. As in these simple examples state and control spaces are finite, the transition functions can be described by directed graphs. These are given in Fig. 1.

Example 1 (*A toy climate problem*)**.** Our first example is a variant of a stochastic climate science SDP studied in Botta *et al.* (2018), stripped down to a simple non-deterministic SDP. At every decision step, the world may be in one of two *states*, namely *Good* or *Bad*, and the *controls* determine whether a *Low* or a *High* amount of green house gases is

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

emitted into the atmosphere. If the world is in the *Good* state, choosing *Low* emissions will definitely keep the world in the *Good* state, but the result of choosing high emissions is non-deterministic: either the world may stay in the *Good* or tip to the *Bad* state. Similarly, in the *Bad* state, *High* emissions will definitely keep the world in *Bad*, while with *Low* emissions it might either stay in *Bad* or recover and return to the *Good* state. The transitions just described define a non-deterministic *transition function*. The *rewards* associated with each transition are determined by the respective control and reached state. Now we can formulate an SDP: *"Which policy sequence will maximise the worst-case sum of rewards along all possible trajectories when taking n decisions starting at decision step t?"*. In this simple example, the question is not hard to answer: always choose *Low* emissions, independent of decision step and state. The *optimal policy sequence* for any n and t would thus consist of n constant *Low* functions. But in a more realistic example, the situation will be more involved: every option will have its benefits and drawbacks encoded in a more complicated reward function, uncertainties might come with different probabilities, there might be intermediate states, different combinations of control options, etc. For more along these lines, we refer the interested reader to Botta *et al.* (2018).

Example 2 (*Scheduling*)**.** Scheduling problems serve as canonical examples in control theory textbooks. The one we present here is a slightly modified version of Bertsekas (1995, Example 1.1.2).

Think of some machine in a factory that can perform different operations, say A, B, C and D. Each of these operations is supposed to be performed once. The machine can only perform one operation at a time, thus an order has to be fixed in which to perform the operations. Setting the machine up for each operation incurs a specific cost that might vary according to which operation has been performed before. Moreover, operation B can only be performed after operation A has already been completed, and operation D only after operation C. It suffices to fix the order in which the first three operations are to be performed as this uniquely determines which will be the fourth task. The aim is now to choose an order that minimises the total cost of performing the four operations.

This situation can be modelled as follows as a problem with three decision steps: The *states at each step* are the sequences of operations already performed, with the empty sequence at step 0. The *controls at a decision step and in a state* are the operations which have not already been performed at previous steps and which are permitted in that state. For example, at decision step 0, only controls A and C are available because of the above constraint on performing B and D. The *transition* and *cost* functions of the problem are depicted by the graph in Figure 1b. As the problem is deterministic, picking a control will result in a unique next state and each sequence of policies will result in a unique trajectory. In this setting, solving the SDP reduces to finding a control sequence that *minimises the sum of costs along the single resulting trajectory*. In Figure 1b, this is the sequence $CAB(D)$.

3 The BJI-framework

The BJI-framework is a dependently typed formalisation of optimal control theory for finite-horizon, discrete-time SDPs. It extends mathematical formulations for stochastic

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

SDPs (Bertsekas, 1995; Bertsekas & Shreve, 1996; Puterman, 2014) to the general problem of optimal decision making under *monadic* uncertainty.

For monadic SDPs, the framework provides a generic implementation of backward induction. It has been applied to study the impact of uncertainties on optimal emission policies (Botta *et al.*, 2018) and is currently used to investigate solar radiation management problems under tipping point uncertainty (TiPES, 2019–2023).

In a nutshell, the framework consists of two sets of components: one for the *specification* of an SDP and one for its *solution* with monadic backward induction.

3.1 Problem specification components

In the following, we discuss the components necessary to specify a monadic SDP.
The first one is the monad M:

$$M : Type \to Type$$

As discussed in the previous section, M accounts for the uncertainties that affect the decision problem. For our first example, we could for instance define M to be *List*. For the second example it suffices to use $M = Id$ as the problem is deterministic.

Further, the BJI-framework supports the specification of the *states*, the *controls* and the *transition function* of an SDP through

$$\begin{aligned} X &: (t : \mathbb{N}) \to Type \\ Y &: (t : \mathbb{N}) \to X\ t \to Type \\ next &: (t : \mathbb{N}) \to (x : X\ t) \to Y\ t\ x \to M\ (X\ (S\ t)) \end{aligned}$$

The interpretation is that $X\ t$ represents the states at decision step t.[2] In the first example of Section 2, there are just two states (*Good* and *Bad*) such that X is a constant family:

$$\begin{aligned} &\textbf{data}\ State = Good \mid Bad \\ &X\ _t \quad\quad = State \end{aligned}$$

But in the second example the possible states depend on the decision step t. Taking for example step $t = 2$, we could simply define

$$\begin{aligned} &\textbf{data}\ State2 = AB \mid AC \mid CA \mid CD \\ &X\ 2 \quad\quad\quad = State2 \end{aligned}$$

Alternatively, we could employ type dependency in a more systematic way to express that in Ex. 2 states are admissible sequences of actions

$$\textbf{data}\ Act = A \mid B \mid C \mid D$$

Recall that actions could require that another action was performed before, no action was to be carried out twice and the problem was limited to 3 steps. These conditions might be captured by a type-valued predicate

$$AdmissibleState : \{t : \mathbb{N}\} \to Vect\ t\ Act \to Type$$

and the type of states might then be expressed as dependent pair of a vector of actions and a proof that it is admissible.

[2] Note that in Idris, S and Z are the familiar constructors of the data type $\mathbb{N}$.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$X\ t = (as : Vect\ t\ Act \circledast AdmissibleState\ as)$$

Similarly, $Y\ t\ x$ represents the controls available at decision step t and in state x and $next\ t\ x\ y$ represents the states that can be obtained by selecting control y in state x at decision step t. In our first example, the available controls remain constant over time (*High* or *Low*) like the states, but for the second example, the type dependency is relevant: e.g. we might define (again at step $t = 2$)

$$\begin{aligned} &\textbf{data}\ CtrlAC = B \mid D \\ &Y\ 2\ AC \quad\ = CtrlAC \end{aligned}$$

or more elegantly use dependent pairs to define the type of controls, using the observation that an action is an admissible control for some state represented by a vector of actions, if adding the action to the vector results again in an admissible state :

$$\begin{aligned} &AdmissibleControl : \{t : \mathbb{N}\} \to Vect\ t\ Act \to Act \to Type \\ &AdmissibleControl\ as\ a = AdmissibleState\ (a :: as) \end{aligned}$$

$$Y\ t\ x = (a : Act \circledast AdmissibleControl\ (fst\ x)\ a)$$

Recall from Section 2 that the monad, the states, the controls and the next function together define a decision process. In order to fully specify a decision problem, one also has to define the rewards obtained at each decision step and the operation that is used to add up rewards. In the BJI-framework, this is done in terms of

$$\begin{aligned} &Val \quad\ : Type \\ &reward : (t : \mathbb{N}) \to (x : X\ t) \to Y\ t\ x \to X\ (S\ t) \to Val \\ &(\oplus) \quad\ : Val \to Val \to Val \end{aligned}$$

Here, Val is the type of rewards and $reward\ t\ x\ y\ x'$ is the reward obtained by selecting control y in state x if the next state is x', an element of the state space at step $t + 1$. Note that for deterministic problems it is unnecessary to parameterise the reward function over the next state as it is unique and can thus be obtained from the current state and control. But for non-deterministic problems, it is useful to be able to assign rewards depending on the (uncertain) outcome of a transition.

A few remarks are at place here.

- In many applications, Val is a numerical type and the controls of the SDP represent resources (fuel, water, etc.) that come at a cost. In these cases, the reward function encodes the costs and perhaps also the benefits associated with a decision step. Often, the latter also depends both on the current state x and on the next state x'. The BJI-framework nicely copes with all these situations.
- The operation $\oplus$ determines how rewards are added up. It could be a simple arithmetic operation, but it could also be defined in terms of problem-specific parameters, e.g. discount factors to give more weight to current rewards as compared to future rewards.
- Mapping $reward\ t\ x\ y$ onto $next\ t\ x\ y$ (remember that M is a monad and thus a functor) yields a value of type $M\ Val$. These are the *possible* rewards obtained by selecting control y in state x at decision step t. In mathematical theories of optimal control, the implicit assumption often is that Val is equal to $\mathbb{R}$ and that the M-structure is a

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Formalisation of Ex. 1:

We use the monoid and preorder structure on $\mathbb{N}$, i.e. $Val = \mathbb{N}$, $(\oplus) = (+)$, $zero = 0$, $(\leqslant) = (\sqsubseteq)$.

$$M = List$$

Measure:

$$\begin{aligned} &minList : List\ \mathbb{N} \to \mathbb{N} \\ &minList\ [\,] &&= 0 \\ &minList\ (x :: [\,]) &&= x \\ &minList\ (x :: xs) &&= x\ `minimum`\ xs \end{aligned}$$

$$meas = minList$$

States and Controls:

$$\begin{aligned} &\textbf{data}\ States &&= Good \mid Bad \\ &\textbf{data}\ Controls &&= High \mid Low \end{aligned}$$

$$\begin{aligned} X\ _t &= States \\ Y\ _t\ _x &= Controls \end{aligned}$$

Transition function:

$$\begin{aligned} next\ _t\ Good\ Low &= [Good] \\ next\ _t\ Bad\ \ High &= [Bad] \\ next\ _t\ _x\ \ _y &= [Good, Bad] \end{aligned}$$

Rewards:

$$\begin{aligned} reward\ _t\ _x\ Low\ \ Good &= 3 \\ reward\ _t\ _x\ High\ Good &= 2 \\ reward\ _t\ _x\ Low\ \ Bad &= 1 \\ reward\ _t\ _x\ High\ Bad &= 0 \end{aligned}$$

Fig. 2: A formalisation of Ex. 1 from Section 2.

probability distribution on real numbers which can be evaluated with the *expected value* measure. However, in many practical applications, measuring uncertainty of rewards in terms of the expected value is inadequate (Mercure *et al.*, 2020). The BJI-framework therefore takes a generic approach and allows the specification of SDPs in terms of problem-specific measures.

As just discussed, in SDPs with uncertainty a measure is required to aggregate multiple possible rewards. The BJI-framework supports the specification of the measure by:

$$meas : M\ Val \to Val$$

In our first example we could use the minimum of a list as worst-case measure, while in the second example the measure would just be the identity (as the problem is deterministic).

Before we get to the solution components of the BJI-framework, one more ingredient needs to be specified. In the next section, we will formalise a notion of optimality for which it is necessary to be able to compare elements of Val. The framework allows users to compare Val-values in terms of a problem specific comparison operator:

$$(\sqsubseteq) : Val \to Val \to Type$$

The operator $(\sqsubseteq)$ is required to define a total preorder on Val. In our two examples, we simply have:

$$\begin{aligned} Val &= \mathbb{N} \\ (\oplus) &= (+) \\ (\sqsubseteq) &= (\leqslant) \end{aligned}$$

Three more ingredients are necessary to fully specify a monadic SDP, but we defer discussing them to when they come up in the next subsection. For illustration, a formalisation of Ex. 1 can be found in Fig. 2. A formalisation of Ex. 2 is included in the supplementary material.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

3.2 Problem solution components

The second set of components of the BJI-framework is an extension of the mathematical theory of optimal control for stochastic sequential decision problems to monadic problems. Here, we provide a summary of the theory. Motivation and full details can be found in Botta *et al.* (2017*a*,b, 2018).

The theory formalises the notions of policy (decision rule) from Section 2 as

$$\begin{aligned} &\mathit{Policy} \;\; : (t : \mathbb{N}) \to \mathit{Type} \\ &\mathit{Policy}\; t = (x : X\; t) \to Y\; t\; x \end{aligned}$$

Policy sequences are then essentially vectors of policies[3].

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

Notice the role of the step (time) index t and of the length index n in the constructors of policy sequences: For a policy sequence to make sense, policies for taking decisions at step t can only be prepended to policy sequences for taking *first* decisions at step $t+1$ and the operation yields policy sequences for taking *first* decisions at step t. Thus, the time index allows to ensure a consistency property of policy sequences by construction. As for plain vectors and lists, prepending a policy to a policy sequence of length n yields a policy sequence of length $n+1$. Both the time and the length index will be useful below: they allow to express that the backward induction algorithm computes policy sequences starting at a specific time and having a specific length depending on its inputs.

The perhaps most important ingredient of backward induction is a *value function* that incrementally measures and adds up rewards. For a given decision problem, the value function takes two arguments: a policy sequence ps for making n decision steps starting from decision step t and an initial state $x : X\; t$. It computes the value of taking n decision steps according to the policies ps when starting in x. In the BJI-framework, the value function is defined as

$$\begin{aligned} &\mathit{val} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\; t\; n \to X\; t \to \mathit{Val} \\ &\mathit{val}\; \{t\}\; \mathit{Nil}\; x \qquad = \mathit{zero} \\ &\mathit{val}\; \{t\}\; (p :: ps)\; x = \textbf{let}\; y \quad = p\; x\; \textbf{in} \\ &\qquad\qquad\qquad\qquad\; \textbf{let}\; mx' = \mathit{next}\; t\; x\; y\; \textbf{in} \\ &\qquad\qquad\qquad\qquad\; \mathit{meas}\; (\mathit{map}\; (\mathit{reward}\; t\; x\; y \oplus \mathit{val}\; ps)\; mx') \end{aligned}$$

Notice that, independently of the initial state x, the value of the empty policy sequence is *zero*. This is a problem-specific reference value

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

that has to be provided as part of a problem specification.[4] It is one of the specification components that we have not discussed in Section 3.1. The value of a policy sequence consisting of a first policy p and of a tail policy sequence ps is defined inductively as

[3] The curly brackets in the types of *Nil* and (::) indicate that t and n are implicit arguments.

[4] The name might suggest that *zero* is supposed to be a neutral element relative to $\oplus$. However, this is not required by the framework.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

the measure of an M-structure of Val-values. These values are obtained by first computing the control y dictated by the policy p in state x and the M-structure of possible next states mx' dictated by the transition function *next*. Then, for all x' in mx' the current reward $reward\ t\ x\ y\ x'$ is added to the aggregated outcome for the tail policy sequence $val\ ps\ x'$. Finally, the result of this functorial mapping is aggregated with the problem-specific measure *meas* to obtain a result of type Val as outcome for the policy sequence $(p :: ps)$. The function which is mapped onto mx' is just a lifted version of $\oplus$:

$$(\bigoplus) : \{A : Type\} \to (f, g : A \to Val) \to A \to Val$$
$$f \bigoplus g = \lambda a \Rightarrow f\ a \oplus g\ a$$

We will come back to the value function of the BJI-theory in Section 4 where we will contrast it with a function val' that, for a policy sequence ps and an initial state x, computes the measure of the sum of the rewards along all possible trajectories starting at x under ps (the *measured total reward* that we anticipated in Section 2). For the time being, though, we accept the notion of value of a policy sequence as put forward in the BJI-theory and we show how the definition of *val* can be employed to compute policy sequences that are provably optimal in the sense of

$$OptPolicySeq : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to Type$$
$$OptPolicySeq\ \{t\}\ \{n\}\ ps = (ps' : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps'\ x \sqsubseteq val\ ps\ x$$

Notice the universal quantification in this definition: A policy sequence ps is defined to be optimal iff $val\ ps'\ x \sqsubseteq val\ ps\ x$ for any policy sequence ps' and for any state x.

The generic implementation of backward induction in the BJI-framework is an application of *Bellman's principle of optimality* mentioned in Section 2. In control theory textbooks, this principle is often referred to as *Bellman's equation*. It can be suitably formulated in terms of the notion of *optimal extension*. We say that a policy $p : Policy\ t$ is an optimal extension of a policy sequence $ps : Policy\ (S\ t)\ n$ if it is the case that the value of $p :: ps$ is at least as good as the value of $p' :: ps$ for any policy p' and for any state $x : X\ t$:

$$OptExt : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t \to Type$$
$$OptExt\ \{t\}\ ps\ p = (p' : Policy\ t) \to (x : X\ t) \to val\ (p' :: ps)\ x \sqsubseteq val\ (p :: ps)\ x$$

With the notion of optimal extension in place, Bellman's principle can be formulated as

$$\begin{aligned} Bellman : \{t, n : \mathbb{N}\} \to\ & \\ & (ps : PolicySeq\ (S\ t)\ n) \to OptPolicySeq\ ps \to \\ & (p\ \ : Policy\ t) \to OptExt\ ps\ p \to \\ & OptPolicySeq\ (p :: ps) \end{aligned}$$

In words: *extending an optimal policy sequence with an optimal extension (of that policy sequence) yields an optimal policy sequence* or shorter *prefixing with optimal extensions preserves optimality*. Proving Bellman's optimality principle is almost straightforward and relies on $\sqsubseteq$ being reflexive and transitive and two *monotonicity* properties:

$$plusMonSpec\ \ : \{v1, v2, v3, v4 : Val\} \to v1 \sqsubseteq v2 \to v3 \sqsubseteq v4 \to (v1 \oplus v3) \sqsubseteq (v2 \oplus v4)$$
$$\begin{aligned} measMonSpec : \{A : Type\} \to (f, g : A \to Val) \to ((a : A) \to f\ a \sqsubseteq g\ a) \to & \\ (ma : M\ A) \to meas\ (map\ f\ ma) \sqsubseteq meas\ (map\ g\ ma) & \end{aligned}$$

The second condition is a special case of the measure monotonicity requirement originally formulated by Ionescu (2009) in the context of a theory of vulnerability and monadic

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

dynamical systems. It is a natural property and the expected value measure, the worst-(best) case measure and any sound statistical measure fulfil it. Like the reference value *zero* discussed above, *plusMonSpec* and *measMonSpec* are specification components of the BJI-framework that we have not discussed in Section 3.1. We provide a proof of *Bellman* in Appendix 5. As one would expect, the proof makes essential use of the recursive definition of the function *val* discussed above. As a consequence, this precise definition of *val* plays a crucial role for the verification of backward induction in Botta *et al.* (2017*a*).

Apart from the increased level of generality, the definition of *val* and *Bellman* are in fact just an Idris formalisation of Bellman's equation as formulated in control theory textbooks. With *Bellman* and provided that we can compute optimal extensions of arbitrary policy sequences

$$\begin{aligned} &\mathit{optExt} &&: \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{Policy}\ t \\ &\mathit{optExtSpec} &&: \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps}) \end{aligned}$$

it is easy to derive an implementation of monadic backward induction that computes provably optimal policy sequences with respect to *val*: first, notice that the empty policy sequence is optimal:

$$\begin{aligned} &\mathit{nilOptPolicySeq} : \mathit{OptPolicySeq}\ \mathit{Nil} \\ &\mathit{nilOptPolicySeq}\ \mathit{Nil}\ x = \mathit{reflexive}\ \mathit{lteTP}\ \mathit{zero} \end{aligned}$$

This is the base case for constructing optimal policy sequences by backward induction, starting from the empty policy sequence:

$$\begin{aligned} &\mathit{bi} : (t, n : \mathbb{N}) \to \mathit{PolicySeq}\ t\ n \\ &\mathit{bi}\ t\ Z \quad\ = \mathit{Nil} \\ &\mathit{bi}\ t\ (S\ n) = \mathbf{let}\ \mathit{ps} = \mathit{bi}\ (S\ t)\ n\ \mathbf{in}\ \mathit{optExt}\ \mathit{ps} :: \mathit{ps} \end{aligned}$$

That *bi* computes optimal policy sequences with respect to *val* is then proved by induction on *n*, the input that determines the length of the resulting policy sequence:

$$\begin{aligned} &\mathit{biOptVal} : (t, n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n) \\ &\mathit{biOptVal}\ t\ Z \quad\ = \mathit{nilOptPolicySeq} \\ &\mathit{biOptVal}\ t\ (S\ n) = \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}\ \mathbf{where} \\ &\quad \mathit{ps} \ : \ \mathit{PolicySeq}\ (S\ t)\ n \ ; \ \mathit{ps} \ = \mathit{bi}\ (S\ t)\ n \\ &\quad \mathit{ops} : \ \mathit{OptPolicySeq}\ \mathit{ps} \ ; \ \mathit{ops} = \mathit{biOptVal}\ (S\ t)\ n \\ &\quad p \ \ : \ \mathit{Policy}\ t \ ; \ p \ \ = \mathit{optExt}\ \mathit{ps} \\ &\quad \mathit{oep} : \ \mathit{OptExt}\ \mathit{ps}\ p \ ; \ \mathit{oep} = \mathit{optExtSpec}\ \mathit{ps} \end{aligned}$$

This is the verification result for *bi* of Botta *et al.* (2017*a*).[5]

3.3 BJI-framework wrap-up

The specification and solution components discussed in the last two sections are all we need to formulate precisely the problem of correctness for monadic backward induction in the BJI-framework. This is done in the next section. Before turning to it, two further remarks are necessary:

[5] Note that *biOptVal* is called *biLemma* in Botta *et al.* (2017*a*). We chose the new name to emphasise that *bi* computes optimal policy sequences with respect to *val*.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

- The theory proposed in Botta *et al.* (2017*a*) is slightly more general than the one summarised above. Here, policies are just functions from states to controls:

  $$Policy : (t : \mathbb{N}) \to Type$$
  $$Policy\ t = (x : X\ t) \to Y\ t\ x$$

  By contrast, in Botta *et al.* (2017*a*), policies are indexed over a number of decision steps n

  $$Policy : (t, n : \mathbb{N}) \to Type$$
  $$Policy\ t\ Z \quad = Unit$$
  $$Policy\ t\ (S\ m) = (x : X\ t) \to Reachable\ x \to Viable\ (S\ m)\ x \to GoodCtrl\ t\ x\ m$$

  and their domain for $n > 0$ is restricted to states that are *reachable* and *viable* for n steps. This allows to cope with states whose control set is empty and with transition functions that return empty M-structures of next states. (For a discussion of reachability and viability see Botta *et al.* 2017*a*, Sections 3.7 and 3.8.)
  This generality, however, comes at a cost: Compare, e.g. the proof of Bellman's principle from the last subsection with the corresponding proof in Botta *et al.* (2017a, Appendix B). The impact of the reachability and viability constraints on other parts of the theory is even more severe.
  Here, we have decided to trade some generality for better readability and opted for a simplified version of the original theory. Still, for the generic backward induction algorithm we need to make sure that it is possible to define policy sequences of the length required for a specific SDP. This can, e.g. be done by postulating controls to be non-empty:

  $$notEmptyY : (t : \mathbb{N}) \to (x : X\ t) \to Y\ t\ x$$

  We also impose a non-emptiness requirement on the transition function *next* that will be discussed in Section 7.

  $$nextNotEmpty : \{t : \mathbb{N}\} \to (x : X\ t) \to (y : Y\ t\ x) \to NotEmpty\ (next\ t\ x\ y)$$

- In Section 3.2, we have not discussed under which conditions one can implement optimal extensions of arbitrary policy sequences. This is an interesting topic that is however orthogonal to the purpose of the current paper. For the same reason, we have not addressed the question of how to make *bi* more efficient by tabulation. We briefly discuss the specification and implementation of optimal extensions in the BJI-framework in Appendix 7. We refer the reader interested in tabulation of *bi* to SequentialDecisionProblems.TabBackwardsInduction of Botta (2016–2021).

4 Correctness for monadic backward induction

In this section, we formally specify the notions of correctness for monadic backward induction *bi* and the value function *val* of the BJI-framework that we will study in the remainder of this paper. We develop these notions as generic variants of the corresponding notions for stochastic SDPs.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

4.1 Extension of the BJI-framework

In the previous section, we have seen that a monadic SDP can be specified in terms of nine components: M, X, Y, *next*, *Val*, *zero*, $\oplus$, $\sqsubseteq$ and *reward*.

Given a policy sequence (optimal or not) and an initial state for an SDP, we can compute the M-structure of possible trajectories starting at that state:

```
data StateCtrlSeq : (t, n : ℕ) → Type where
  Last : {t : ℕ} → X t → StateCtrlSeq t (S Z)
  (##) : {t, n : ℕ} → (x : X t ⊛⊛ Y t x) → StateCtrlSeq (S t) (S n) → StateCtrlSeq t (S (S n))

trj : {t, n : ℕ} → PolicySeq t n → X t → M (StateCtrlSeq t (S n))
trj {t} Nil x       = pure (Last x)
trj {t} (p :: ps) x = let y = p x in
                      let mx' = next t x y in
                      map ((x ⊛⊛ y)##) (mx' >>= trj ps)
```

where we use *StateCtrlSeq* as type of trajectories. Essentially, it is a non-empty list of (dependent) state/control pairs, with the exception of the base case which is a singleton just containing the last state reached.

Furthermore, we can compute the *total reward* for a single trajectory, i.e. its sum of rewards:

```
sumR : {t, n : ℕ} → StateCtrlSeq t n → Val
sumR {t} (Last x)          = zero
sumR {t} ((x ⊛⊛ y) ## xys) = reward t x y (head xys) ⊕ sumR xys
```

where *head* is the helper function

```
head : {t, n : ℕ} → StateCtrlSeq t (S n) → X t
head (Last x)          = x
head ((x ⊛⊛ y) ## xys) = x
```

By mapping *sumR* onto an M-structure of trajectories, we obtain an M-structure containing the individual sums of rewards of the trajectories. Now, using the measure function, we can compute the generic analogue of the expected total reward for a policy sequence *ps* and an initial state x:

```
val' : {t, n : ℕ} → (ps : PolicySeq t n) → (x : X t) → Val
val' ps = meas ∘ map sumR ∘ trj ps
```

As anticipated in Section 2, we call the value computed by *val′* the *measured total reward*. Recall that solving a stochastic SDP commonly means finding a policy sequence that maximises the *expected total reward*. By analogy, we define that solving a monadic SDP means to find a policy sequence that maximises the *measured total reward*. That is, given t and n, the solution of a monadic SDP is a sequence of n policies that maximises the measure of the sum of rewards along all possible trajectories of length n that are rooted in an initial state at step t.

Again by analogy to the stochastic case, we define monadic backward induction to be correct if, for a given SDP, the policy sequence computed by *bi* is the solution to the SDP. That is, we consider *bi* to be correct if it meets the specification

```
biOptMeasTotalReward : (t, n : ℕ) → GenOptPolicySeq val' (bi t n)
```

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

where *GenOptPolicySeq* is a generalised version of the optimality predicate *OptPolicySeq* from Section 3.2. It now takes as an additional parameter the function with respect to which the policy sequence is to be optimal:

$$GenOptPolicySeq : \{t, n : \mathbb{N}\} \to (PolicySeq\ t\ n \to X\ t \to Val) \to PolicySeq\ t\ n \to Type$$
$$GenOptPolicySeq\ \{t\}\ \{n\}\ f\ ps = (ps' : PolicySeq\ t\ n) \to (x : X\ t) \to f\ ps'\ x \sqsubseteq f\ ps\ x$$

As recapitulated in Section 3.2, Botta *et al.* have already shown that if M is a monad, $\sqsubseteq$ a total preorder and $\oplus$ and *meas* fulfil two monotonicity conditions, then $bi\ t\ n$ yields an optimal policy sequence with respect to the value function *val* in the sense that $val\ ps'\ x \sqsubseteq val\ (bi\ t\ n)\ x$ for any policy sequence ps' and initial state x, for arbitrary $t, n : \mathbb{N}$. Or, expressed using the generalised optimality predicate, that the type

$$GenOptPolicySeq\ \{t\}\ \{n\}\ val\ (bi\ t\ n)$$

is inhabited. As seen in Section 3.2, the function *val* measures and adds rewards incrementally. But does it always compute the measured total reward like *val*′? Modulo differences in the presentation Puterman (2014, Theorem 4.2.1) suggests that for standard stochastic SDPs, *val* and *val*′ are extensionally equal, which in turn allows the use of backward induction for solving these SDPs. Generalising, we therefore consider *val* as correct if it fulfils the specification

$$valMeasTotalReward : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps\ x = val'\ ps\ x$$

If this equality held for the general monadic SDPs of the BJI-theory, we could prove the correctness of *bi* as immediate corollary of *valMeasTotalReward* and Botta *et al.* 's result *biOptVal*. The statement *biOptMeasTotalReward* can be seen as a generic version of textbook correctness statements for backward induction as solution method for stochastic SDPs like (Bertsekas 1995, prop.1.3.1) or Puterman (2014, Theorem 4.5.1.c). By proving *valMeasTotalReward*, we could therefore extend the verification of Botta *et al.* (2017*a*) and obtain a stronger correctness result for monadic backward induction.

Our main objective in the remainder of the paper is therefore to prove that *valMeasTotalReward* holds. But there is a problem.

4.2 The problem with the BJI-value function

A closer look at *val* and *val*′ reveals two quite different computational patterns: applied to a policy sequence *ps* of length $n + 1$ and a state x, the function *val* directly evaluates *meas* on the M-structure of rewards corresponding to the possible next states after one step. This entails further evaluations of *meas* for each possible next state. By contrast, $val'\ ps\ x$ entails only one evaluation of *meas*, independently of the length of *ps*. The computation, however, builds up an intermediate M-structure of state-control sequences. The elements of this M-structure, the state-control sequences, are then consumed by *sumR* and finally the M-structure of rewards is reduced by *meas*.

For illustration, let us revisit Ex. 1 from Section 2 as formalised in Figure 2. To do an example calculation with *val* and *val*′ we first need a concrete policy sequence as input. The simplest two policies are the two constant policies:

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$constH : (t : \mathbb{N}) \to Policy\ t$$
$$constH\ _t = const\ High$$
$$constL : (t : \mathbb{N}) \to Policy\ t$$
$$constL\ _t = const\ Low$$

From these, we can define a policy sequence

$$ps : PolicySeq\ 0\ 3$$
$$ps = constH\ 0 :: (constL\ 1 :: (constH\ 2 :: Nil))$$

It is instructive to compute $val\ ps\ Good$ and $val'\ ps\ Good$ by hand. Recall that in this example, we have $M = List$ with $(\gg\!=) = concatMap$ and $Val = \mathbb{N}$ with $\oplus = +$. The measure $meas$ thus needs to have the type $List\ \mathbb{N} \to \mathbb{N}$. Without instantiating $meas$ for the moment, the computations roughly exhibit the structure

$$val\ ps\ Good = meas\ [2 + meas\ [3 + meas\ [2, 0]], 0 + meas\ [3 + meas\ [2, 0], 1 + meas\ [0]]]$$
$$val'\ ps\ Good = meas\ [7, 5, 5, 3, 1]$$

and it is not "obviously clear" that val and val' are extensionally equal without further knowledge about $meas$.

In the deterministic case, i.e. for $M = Id$ and $meas = id$, $val\ ps\ x$ and $val'\ ps\ x$ are indeed equal for all ps and x, without imposing any further conditions (as we will see in Section 6). For the stochastic case, (Puterman, 2014, Theorem 4.2.1) suggests that the equality should hold. But for the monadic case, no such result has been established. And as it turns out, in general the functions val and val' are not unconditionally equal – consider the following counter-example: We continue in the setting of Ex. 1 from above, but now instantiate the measure to the plain arithmetic sum

$$meas = foldr\ (+)\ 0$$

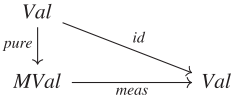
This measure fulfils the monotonicity condition (*measMonSpec*, Section 3.2) imposed by the BJI-framework. But if we instantiate the above computations with it, then we get $val\ ps\ Good = 13$ and $val'\ ps\ Good = 21$! We thus see that the equality between val and val' cannot hold unconditionally in the generic setting of the BJI-framework. In the next section, we therefore present conditions under which the equality *does* hold.

5 Correctness conditions

We now formulate three conditions on combinations of the monad, the measure function and the binary operation $\oplus$ that imply the extensional equality of val and val':

Condition 1. The measure needs to be left-inverse to *pure*: [6]

$$measPureSpec : meas \circ pure \doteq id$$

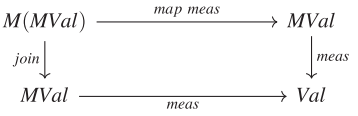


[6] The symbol $\doteq$ denotes *extensional* equality, see Appendix 1.2

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

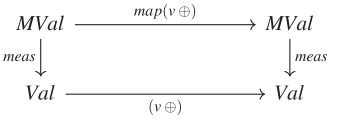
Condition 2. Applying the measure after *join* needs to be extensionally equal to applying it after *map meas*:

$$measJoinSpec : meas \circ join \doteq meas \circ map\ meas$$



Condition 3. For arbitrary $v : Val$ and non-empty $mv : M\ Val$ applying the measure after mapping $(v\oplus)$ onto mv needs to be equal to applying $(v\oplus)$ after the measure:

$$measPlusSpec : (v : Val) \to (mv : M\ Val) \to (NotEmpty\ mv) \to \\ (meas \circ map\ (v\oplus))\ mv = ((v\oplus) \circ meas)\ mv$$



Essentially, these conditions assure that the measure is well-behaved relative to the monad structure and the $\oplus$-operation. They arise by, again, generalising from the standard example of stochastic SDPs with a probability monad, the *expected value* as measure and ordinary addition as $\oplus$. The first two conditions are lifting properties that allow to do computations either in the monad or the underlying structure with the same result. The third condition is a distributivity law. For the computation of the measured total reward, it means that instead of adding the current reward to the outcome of each trajectory and then measuring, one may as well first measure the outcomes and then add the current reward.

To illustrate the conditions, let us consider a simple representation of discrete probability distributions like in Erwig & Kollmansberger (2006).

$$Dist : Type \to Type$$
$$Dist\ Omega = List\ (Omega, Double)$$

with

$$distMap : \{A, B : Type\} \to (f : A \to B) \to Dist\ A \to Dist\ B$$
$$distMap\ f\ aps = [(f\ (fst\ ap), snd\ ap)\ |\ ap \leftarrow aps]$$

$$distPure \quad : \{A : Type\} \to A \to Dist\ A$$
$$distPure\ a = [(a, 1.0)]$$

$$distJoin : \{A : Type\} \to (Dist\ (Dist\ A)) \to Dist\ A$$
$$distJoin\ apsps = concat\ [[(fst\ ap, snd\ ap * snd\ aps)\ |\ ap \leftarrow fst\ aps]\ |\ aps \leftarrow apsps]$$

$Val = Double$ and as measure the expected value

$$expected : Dist\ Double \to Double$$
$$expected\ dps = sum\ [fst\ dp * snd\ dp\ |\ dp \leftarrow dps]$$

With $meas = expected$ and $\oplus = +$, we can now consider the three conditions from above.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Condition 1. The first condition *measPureSpec* holds since 1.0 is neutral for $*$.

$$expected \circ distPure\ a = a * 1.0 = a$$

The second and the third condition require some arithmetic reasoning, so let us just consider them for two examples. Let a, b, c, d be variables of type *Double* and say we have distributions

$$dps1 : Dist\ Double$$
$$dps1 = [(a, 0.5), (b, 0.3), (c, 0.2)]$$

$$dps2 : Dist\ Double$$
$$dps2 = [(a, 0.4), (d, 0.6)]$$

$$dpdps : Dist\ (Dist\ Double)$$
$$dpdps = [(dps1, 0.1), (dps2, 0.9)]$$

Condition 2. Then the second condition *measJoinSpec* instantiates to

$$(expected \circ distJoin)\ dpdps = (expected \circ distMap\ expected)\ dpdps$$

This equality holds because of the standard properties of addition and multiplication:

$$\begin{aligned}
&(expected \circ distJoin)\ dpdps && = \\
&expected\ [(a, 0.5 * 0.1), (b, 0.3 * 0.1), (c, 0.2 * 0.1), (a, 0.4 * 0.9), (d, 0.6 * 0.9)] && = \\
&(a * 0.5 * 0.1) + (b * 0.3 * 0.1) + (c * 0.2 * 0.1) + (a * 0.4 * 0.9) + (d * 0.6 * 0.9) && = \\
&((a * 0.5 + b * 0.3 + c * 0.2) * 0.1 + (a * 0.4 + d * 0.6) * 0.9 && = \\
&expected\ [(a * 0.5 + b * 0.3 + c * 0.2, 0.1), (a * 0.4 + d * 0.6, 0.9)] && = \\
&(expected \circ distMap\ expected)\ dpdps
\end{aligned}$$

Condition 3. For the third condition *measPlusSpec*, consider for some $v : Double$ the equation

$$(expected \circ distMap\ (v+))\ dps1 = ((v+) \circ expected)\ dps1$$

Again using the usual arithmetic laws for $+$ and $*$, we can calculate

$$\begin{aligned}
&expected\ (distMap\ (v+)\ [(a, 0.5), (b, 0.3), (c, 0.2)]) && = \\
&(v + a) * 0.5 + (v + b) * 0.3 + (v + c) * 0.2 && = \\
&(v * 0.5 + a * 0.5) + (v * 0.3 + b * 0.3) + (v * 0.2 + c * 0.2) && = \\
&(v * 0.5 + v * 0.3 + v * 0.2) + (a * 0.5 + b * 0.3 + c * 0.2) && = \\
&v + expected\ [(a, 0.5), (b, 0.3), (c, 0.2)]
\end{aligned}$$

As we can see, an essential ingredient for the equality to hold is that the mapped occurrences of $(v+)$ are weighted by the probabilities which add up to 1.

Note that in this example, we have glossed over problems that might arise from the use of *Dist* to represent probability distributions. [7] We will briefly address probability monads and the expected value from a more abstract perspective in Subsection 5.2.

[7] For the sake of simplicity, we do not address (important) conceptional questions concerning the representation of probability distributions or the problems caused by the use of floating point arithmetic in this example. Note,

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

5.1 Examples and counter-examples

Besides the motivating example above, let us now consider some more functions that have the correct type to serve as a measure, and that do or do not fulfil the three conditions.

Simple examples of admissible measures are the minimum (*minList* as defined in Figure 2) or maximum ($maxList = foldr\ `maximum`\ 0$) of a list for $M = List$ with $\mathbb{N}$ as type of values and ordinary addition as $\oplus$. It is straightforward to prove that the conditions hold for these two measures and the proofs for *maxList* are included in the supplementary material.

The function *length* is a very simple counter-example: It has the right type for a list measure but fails all three of the conditions. As to other counter-examples, let us revisit the conditions one by one. Throughout, we use $M = List$ with $map = listMap$, $join = concat$ and $\oplus = +$ (the canonical addition for the respective type of *Val*).

Condition 1. We remain in the setting of Ex. 1 with $Val = \mathbb{N}$ and just vary the measure. Using a somewhat contrived variation of *maxList*

$$maxListVar\ :\ List\,\mathbb{N} \to \mathbb{N}$$
$$maxListVar = foldr\,(\lambda x, v \Rightarrow (x + 1\ `maximum`\ v))\,0$$

with $meas = maxListVar$ it suffices to consider that for an arbitrary $n : \mathbb{N}$

$$(maxListVar \circ pure)\,n = maxListVar\,[n] = (n+1)\ `maximum`\ 0 = n + 1 \neq n = id\,n$$

to see that now the condition *measPureSpec* fails.

Condition 2. To exhibit a measure that fails the condition *measJoinSpec*, we switch to $Val = Double$ and use the arithmetic average

$$avg :\ List\,Double \to Double$$
$$avg\,[\,] = 0.0$$
$$avg\,ds = sum\,ds\ /\ cast\,(length\,ds)$$

as measure $meas = avg$. Taking a list of lists of different lengths like [[1], [2, 3]], we have

$$\begin{aligned} avg\,(concat\,[[1],[2,3]]) \quad &= avg\,[1,2,3] = 2 \\ &\neq \\ avg\,(listMap\,avg\,[[1],[2,3]]) &= avg\,[1,2.5]\ = 1.75 \end{aligned}$$

Condition 3. Let again $Val = \mathbb{N}$ to take another look at our counter-example from the last section with $meas = sum$, the arithmetic sum of a list. It does fulfil *measPureSpec* and *measJoinSpec*, the first by definition, the second by structural induction using the associativity of $+$. But it fails to fulfil *measPlusSpec*. If the list has the form $a :: as$, we would have to show the following equality for *measPlusSpec* to hold:

$$(sum \circ listMap\,(v+))\,(a :: as) = ((v+) \circ sum)\,(a :: as)$$

Clearly, if $v \neq 0$ and $as \neq [\,]$ this equality cannot hold. This is why in the last section the equality of *val* and *val′* failed for $meas = sum$.

however, that the chosen type *Prob* does, e.g. neither enforce that the probabilities lie in the interval $[0, 1]$ nor that they add up to 1. These properties would however be crucial for actual proofs.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

A similar failure would arise if we chose $meas = foldr\,(*)\,1$ instead, as $+$ does not distribute over $*$. But if we turned the situation around by setting $\oplus = *$ and $meas = sum$, the condition *measPlusSpec* would hold thanks to the usual arithmetic distributivity law for $*$ over $+$.

All of the measures considered in this subsection do fulfil the *measMonSpec* condition imposed by the BJI-theory. This raises the question how previously admissible measures are impacted by adding the three new conditions to the framework.

5.2 Impact on previously admissible measures

As we have seen in Section 3.2, the BJI-framework already requires measures to fulfil the monotonicity condition

$$measMonSpec : \{A : Type\} \to (f, g : A \to Val) \to ((a : A) \to (f\ a) \sqsubseteq (g\ a)) \to \\ (ma : M\ A) \to meas\ (map\ f\ ma) \sqsubseteq meas\ (map\ g\ ma)$$

Botta *et al.* show that the arithmetic average (for $M = List$), the worst-case measure (for $M = List$ and for a probability monad $M = Prob$) and the expected value measure (for $M = Prob$) all fulfil *measMonSpec*. Thus, a natural question is whether these measures also fulfil the three additional requirements.

Expected value for probability distributions. As already discussed, most applications of backward induction concern stochastic SDPs where possible rewards are aggregated using the expected value measure from probability theory, commonly denoted as E.

Essentially, for a numerical type Q, the expected value of a probability distribution on Q is

$$E : Num\ Q \Rightarrow Prob\ Q \to Q$$
$$E\ pq = sum\ [q * prob\ pq\ q \mid q \leftarrow supp\ pq]$$

where *prob* and *supp* are generic functions that encode the notions of *probability* and of *support* associated with a finite probability distribution:

$$prob : \{A : Type\} \to Prob\ A \to A \to Q$$
$$supp : \{A : Type\} \to Prob\ A \to List\ A$$

For *pa* and *a* of suitable types, *prob pa a* represents the probability of *a* according to *pa*. Similarly, *supp pa* returns a list of those values whose probability is not zero in *pa*. The probability function *prob* has to fulfil the axioms of probability theory. In particular,

$$sum\ [prob\ pa\ a \mid a \leftarrow supp\ pa] = 1$$

This condition implies that probability distributions cannot be empty, a precondition of *measPlusSpec*. Putting forward minimal specifications for *prob* and *supp* is not completely trivial but if the $+$-operation associated with Q is commutative and associative, if $*$ distributes over $+$ and if the *map* and *join* associated with *Prob* – for f, a, b, *pa* and *ppa* of suitable types – fulfil the conservation law

$$prob\ (map\ f\ pa)\ b = sum\ [prob\ pa\ a \mid a \leftarrow supp\ pa, f\ a == b]$$

and the total probability law

$$prob\ (join\ ppa)\ a = sum\ [prob\ pa\ a * prob\ ppa\ pa \mid pa \leftarrow supp\ ppa]$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

then the expected value fulfils *measPureSpec*, *measJoinSpec* and *measPlusSpec*. This is not surprising since – as stated above – this has been the guiding example for the generalisation to monadic SDPs and the formulation of the three conditions.

Average and arithmetic sum. As can already be concluded from the corresponding counter-examples in the previous subsection, neither the plain arithmetic average nor the arithmetic sum are suited as measure when using the standard monad structure on *List* to represent non-deterministic uncertainty. We think this is an important observation, as the average seems innocent enough to come to mind as a simple way to represent uniformly distributed outcomes: *"The probability of each element can simply be inferred from the length of the list – so why bother to explicitly deal with probabilities?"* Although our counter-example shows that this idea is flawed, the intuition behind it can be employed to define an alternative, but less general monad structure on lists by incorporating the averaging operation into the joining of lists (i.e. by choosing $join = map\ avg$). However, this only makes sense for types that are instances of the *Num* and *Fractional* type classes, and naturality only holds for a restricted class of functions (namely additive functions). As a consequence, this alternative structure does not seem particularly useful for our current purpose either.

Worst-case measures. In many important applications in climate impact research but also in portfolio management and sports, decisions are taken as to minimise the consequences of worst case outcomes. Depending on how "worse" is defined, the corresponding measures might pick the maximum or minimum from an M-structure of values. In the previous subsection, we considered an example in which the monad was *List*, the operation $\oplus$ plain addition together with either *maxList* or *minList* as measure. And indeed we can prove that for both measures the three requirements hold (the proofs for *maxList* can be found in the supplementary material). This gives us a useful notion of worst-case measure that is admissible for monadic backward induction.

We can thus conclude that the new requirements hold for certain familiar measures, but that they also rule out certain instances that were considered admissible in the BJI-framework. Given the three conditions *measPureSpec*, *measJoinSpec*, *measPlusSpec* hold, we can prove the extensional equality of the functions *val* and *val′* generically. This is what we will do in the next section.

6 Correctness proofs

In this section, we show that *val* (Section 3.2) and *val′* (Section 4) are extensionally equal

$$valMeasTotalReward : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val'\ ps\ x = val\ ps\ x$$

given the three conditions from the previous section hold. As a corollary, we then obtain our correctness result for monadic backward induction.

We can understand the proof of *valMeasTotalReward* as an optimising program transformation from the less efficient but "obviously correct" implementation *val′* to the more efficient implementation *val*. Therefore, the equational reasoning proofs in this section will

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

proceed from val' to val. In Section 5, we have stated sufficient conditions for this transformation to be possible: *measPureSpec*, *measJoinSpec*, *measPlusSpec*. We also have seen the different computational patterns that the two implementations exhibit: While val' first computes all possible trajectories for the given policy sequence and initial state, then computes their individual sum of rewards and finally applies the measure once, val computes its final result by adding the current reward to an intermediate outcome and applying the measure locally at each decision step. This suggests that a transformation from val' to val will essentially have to push the application of the measure into the recursive computation of the sum of rewards. The proof will be carried out by induction on the structure of policy sequences.

6.1 Deterministic case

To get a first intuition, let us have a look at what the induction step looks like in the deterministic case (i.e. if we fix monad and measure to be identities):

$$
\begin{aligned}
&valMeasTotalReward\ (p :: ps)\ x = \\
&\quad (val'\ (p :: ps)\ x) && =\{ \text{ by definition of } val' \}= \\
&\quad (sumR\ ((x \ast\!\!* y) \mathbin{\#\!\#} trj\ ps\ x')) && =\{ \text{ by definition of } sumR \}= \\
&\quad (r\ (head\ (trj\ ps\ x')) \oplus val'\ ps\ x') && =\{ \text{ by } headLemma \}= \\
&\quad (r\ x' \oplus val'\ ps\ x') && =\{ \text{ by induction hypothesis} \}= \\
&\quad (r\ x' \oplus val\ ps\ x') && =\{ \text{ by definition of } val \}= \\
&\quad (val\ (p :: ps)\ x) && \square
\end{aligned}
$$

where $y = p\ x$, $x' = next\ t\ x\ y$ and $r = reward\ t\ x\ y$. In the proof sketch, we have first applied the definitions of val' and *sumR*. Using the fact that in the deterministic case *trj* returns exactly one state-control sequence and that the *head* of any trajectory starting in x' is just x' (let us call the latter *headLemma*), the left-hand side of the sum simplifies to $r\ x'$. Its right-hand side amounts to $val'\ ps\ x'$ so that we can apply the induction hypothesis. The rest of the proof only relies on definitional equalities. Thus in the deterministic case val and val' are unconditionally extensionally equal – or rather, the conditions of Section 5 are trivially fulfilled.

6.2 Lemmas

To prove the general, monadic case, we proceed similarly. This time, however, the situation is complicated by the presence of the abstract monad M. Instead of being able to use the type structure of some concrete monad, we need to leverage on the properties of M, *meas* and $\oplus$ postulated in Section 5. To facilitate the main proof, we first prove three lemmas about the interaction of the measure with the monad structure and the $\oplus$-operator on *Val*. Machine-checked proofs are given in the Appendices 2, 3 and 4. The monad laws we use are stated in Appendix 1.2. In the remainder of this section, we discuss semi-formal versions of the proofs.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Monad algebras. The first lemma allows us to lift and eliminate an application of the monad's *join* operation:

$$\begin{aligned} &\mathit{measAlgLemma} : \{A, B : \mathit{Type}\} \to (f : B \to \mathit{Val}) \to (g : A \to M\ B) \to \\ &\qquad (\mathit{meas} \circ \mathit{map}\ (\mathit{meas} \circ \mathit{map}\ f \circ g)) \doteq (\mathit{meas} \circ \mathit{map}\ f \circ \mathit{join} \circ \mathit{map}\ g) \end{aligned}$$

The proof of this lemma hinges on the condition *measJoinSpec*. It allows to trade the application of *join* against an application of *map meas*. The rest is just standard reasoning with monad and functor laws, i.e. we use that the functorial map for M preserves composition and that *join* is a natural transformation:

$$\begin{aligned} &\mathit{measAlgLemma}\ f\ g\ \mathit{ma} = \\ &\quad ((\mathit{meas} \circ \mathit{map}\ (\mathit{meas} \circ \mathit{map}\ f \circ g))\ \mathit{ma}) && =\{\ \mathit{map} \text{ preserves composition}\ \}= \\ &\quad ((\mathit{meas} \circ \mathit{map}\ (\mathit{meas} \circ \mathit{map}\ f) \circ \mathit{map}\ g)\ \mathit{ma}) && =\{\ \mathit{map} \text{ preserves composition}\ \}= \\ &\quad ((\mathit{meas} \circ \mathit{map}\ \mathit{meas} \circ \mathit{map}\ (\mathit{map}\ f) \circ \mathit{map}\ g)\ \mathit{ma}) && =\{\ \text{by } \mathit{measJoinSpec}\ \}= \\ &\quad ((\mathit{meas} \circ \mathit{join} \circ \mathit{map}\ (\mathit{map}\ f) \circ \mathit{map}\ g)\ \mathit{ma}) && =\{\ \mathit{join} \text{ is a natural transformation}\ \}= \\ &\quad ((\mathit{meas} \circ \mathit{map}\ f \circ \mathit{join} \circ \mathit{map}\ g)\ \mathit{ma}) && \square \end{aligned}$$

This lemma is generic in the sense that it holds for arbitrary Eilenberg–Moore algebras of a monad. Here we prove it for the framework's measure *meas*, but note that in the appendix, we prove a generic version that is then appropriately instantiated.

Head/trajectory interaction. The second lemma amounts to a lifted version of *headLemma* in the deterministic case. Mapping *head* onto an M-structure of trajectories computed with *trj* results in an M-structure filled with the initial states of these trajectories; similarly, mapping $(r \circ \mathit{head} \oplus s)$ onto $\mathit{trj}\ \mathit{ps}\ x$ for functions r and s of appropriate type is the same as mapping $(\mathit{const}\ (r\ x) \oplus s)$ onto $\mathit{trj}\ \mathit{ps}\ x$ (where *const* is the constant function). We can prove

$$\begin{aligned} &\mathit{headTrjLemma} : \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ t\ n) \to (r : X\ t \to \mathit{Val}) \to \\ &\qquad (s : \mathit{StateCtrlSeq}\ t\ (S\ n) \to \mathit{Val}) \to (x : X\ t) \to \\ &\qquad (\mathit{map}\ (r \circ \mathit{head} \oplus s) \circ \mathit{trj}\ \mathit{ps})\ x = \\ &\qquad (\mathit{map}\ (\mathit{const}\ (r\ x) \oplus s) \circ \mathit{trj}\ \mathit{ps})\ x \end{aligned}$$

by doing a case split on *ps*. In case $\mathit{ps} = \mathit{Nil}$, the equality holds because the monad's *pure* is a natural transformation and in case $\mathit{ps} = p :: \mathit{ps}'$ because M's functorial *map* preserves composition.

Measure/sum interaction. The third lemma allows us to both commute the measure into the right summand of an $\oplus$-sum and to perform the head/trajectory simplification. It lies at the core of the relationship between *val* and *val′*.

$$\begin{aligned} &\mathit{measSumLemma} : \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ t\ n) \to \\ &\qquad (r : X\ t \to \mathit{Val}) \to \\ &\qquad (s : \mathit{StateCtrlSeq}\ t\ (S\ n) \to \mathit{Val}) \to \\ &\qquad (\mathit{meas} \circ \mathit{map}\ (r \circ \mathit{head} \oplus s) \circ \mathit{trj}\ \mathit{ps}) \doteq \\ &\qquad (r \oplus \mathit{meas} \circ \mathit{map}\ s \circ \mathit{trj}\ \mathit{ps}) \end{aligned}$$

Recall that our third condition from Section 5, *measPlusSpec*, plays the role of a distributive law and allows us to "factor out" a partially applied sum $(v\oplus)$ for arbitrary $v : \mathit{Val}$. Given that *measPlusSpec* holds, the lemma is provable by simple equational reasoning using the above head-trajectory lemma and the fact that map preserves composition:

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$
\begin{aligned}
&\mathit{measSumLemma}\ \mathit{ps}\ r\ s\ x' = \\
&\quad ((\mathit{meas} \circ \mathit{map}\ (r \circ \mathit{head} \oplus s) \circ \mathit{trj}\ \mathit{ps})\ x') && =\{ \text{ by } \mathit{headTrjLemma} \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\ (\mathit{const}\ (r\ x') \oplus s) \circ \mathit{trj}\ \mathit{ps})\ x') && =\{ \text{ by definition of } \oplus, \circ \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\ ((\mathit{const}\ (r\ x') \oplus \mathit{id}) \circ s) \circ \mathit{trj}\ \mathit{ps})\ x') && =\{ \mathit{map} \text{ preserves composition} \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\ (\mathit{const}\ (r\ x') \oplus \mathit{id}) \circ \mathit{map}\ s \circ \mathit{trj}\ \mathit{ps})\ x') && =\{ \text{ by definition of } \oplus \}= \\
&\quad ((\mathit{meas} \circ \mathit{map}\ ((r\ x')\oplus) \circ \mathit{map}\ s \circ \mathit{trj}\ \mathit{ps})\ x') && =\{ \text{ by } \mathit{measPlusSpec} \}= \\
&\quad ((((r\ x')\oplus) \circ \mathit{meas} \circ \mathit{map}\ s \circ \mathit{trj}\ \mathit{ps})\ x') && =\{ \text{ by definition of } \oplus \}= \\
&\quad ((r \oplus \mathit{meas} \circ \mathit{map}\ s \circ \mathit{trj}\ \mathit{ps})\ x') && \square
\end{aligned}
$$

Notice how *measPlusSpec* is used to transform an application of $\mathit{meas} \circ \mathit{map}\ ((r\ x')\oplus)$ into an application of $((r\ x')\oplus) \circ \mathit{meas}$. This is essential to simplify the computation of the measured total reward: instead of first adding the current reward to the intermediate outcome of each individual trajectory and then measuring the outcomes, one can first measure the intermediate outcomes of the trajectories and then add the current reward.

6.3 Correctness of the BJI-value function

With the above lemmas in place, we now prove that *val* is extensionally equal to val'.

Let $t, n : \mathbb{N}$, $\mathit{ps} : \mathit{PolicySeq}\ t\ n$. We prove *valMeasTotalReward* by induction on *ps*.

Base case. We need to show that for all $x : X\ t$, $\mathit{val}'\ \mathit{Nil}\ x = \mathit{val}\ \mathit{Nil}\ x$. The right-hand side of this equation reduces to *zero* by definition. The left-hand side can be simplified to $\mathit{meas}\ (\mathit{pure}\ \mathit{zero})$ since *pure* is a natural transformation. At this point, our first condition, *measPureSpec*, comes into play: Using that *meas* is inverse to *pure* on the left, we can conclude that the equality holds.

In equational reasoning style: For all $x : X\ t$,

$$
\begin{aligned}
&\mathit{valMeasTotalReward}\ \mathit{Nil}\ x = \\
&\quad (\mathit{val}'\ \mathit{Nil}\ x) && =\{ \text{ by definition of } \mathit{val}' \}= \\
&\quad (\mathit{meas}\ (\mathit{map}\ \mathit{sumR}\ (\mathit{trj}\ \mathit{Nil}\ x))) && =\{ \text{ by definition of } \mathit{trj} \}= \\
&\quad (\mathit{meas}\ (\mathit{map}\ \mathit{sumR}\ (\mathit{pure}\ (\mathit{Last}\ x)))) && =\{ \mathit{pure} \text{ is a natural transformation} \}= \\
&\quad (\mathit{meas}\ (\mathit{pure}\ (\mathit{sumR}\ (\mathit{Last}\ x)))) && =\{ \text{ by definition of } \mathit{sumR} \}= \\
&\quad (\mathit{meas}\ (\mathit{pure}\ \mathit{zero})) && =\{ \text{ by } \mathit{measPureSpec} \}= \\
&\quad (\mathit{zero}) && =\{ \text{ by definition of } \mathit{val} \}= \\
&\quad (\mathit{val}\ \mathit{Nil}\ x)
\end{aligned}
$$

Step case. The induction hypothesis (*IH*) is: for all $x : X\ t$, $\mathit{val}'\ \mathit{ps}\ x = \mathit{val}\ \mathit{ps}\ x$. We have to show that *IH* implies that for all $p : \mathit{Policy}\ t$ and $x : X\ t$, the equality $\mathit{val}'\ (p :: \mathit{ps})\ x = \mathit{val}\ (p :: \mathit{ps})\ x$ holds.

For brevity (and to economise on brackets), let in the following $y = p\ x$, $mx' = \mathit{next}\ t\ x\ y$, $r = \mathit{reward}\ t\ x\ y$, $\mathit{trjps} = \mathit{trj}\ \mathit{ps}$, and $\mathit{consxy} = ((x \ast\!\ast y)\#\#)$.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

As in the base case, all that has to be done on the *val*-side of the equation only depends on definitional equality. However, it is more involved to bring the val'-side into a form in which the induction hypothesis can be applied. This is where we leverage on the lemmas proved above.

By definition and because *map* preserves composition, we know that $val'\ (p :: ps)\ x$ is equal to $(meas \circ map\ ((r \circ head) \bigoplus sumR))\ (mx' \gg\!= trjps)$. We use the relation between the monad's *bind* and *join* to eliminate the *bind*-operator from the term. Now we can apply the first lemma from above, *measAlgLemma*, to lift and eliminate the *join* operation.

To commute the measure under the $\bigoplus$ and get rid of the application of *head*, we use our third lemma, *measSumLemma*. At this point, we can apply the induction hypothesis and the resulting term is equal to *val ps x* by definition.
The more detailed equational reasoning proof: [8]

$$
\begin{aligned}
&valMeasTotalReward\ (p :: ps)\ x = \\
&\quad (val'\ (p :: ps)\ x) && =\{ \text{ by definition of } val' \}= \\
&\quad (meas\ (map\ sumR\ (trj\ (p :: ps)\ x))) && =\{ \text{ by definition of } trj \}= \\
&\quad (meas\ (map\ sumR\ (map\ consxy\ (mx' \gg\!= trjps)))) && =\{ map \text{ preserves composition} \}= \\
&\quad (meas\ (map\ (sumR \circ consxy)\ (mx' \gg\!= trjps))) && =\{ \text{ by definition of } sumR \}= \\
&\quad (meas\ (map\ ((r \circ head) \bigoplus sumR)\ (mx' \gg\!= trjps))) && =\{ \text{ relation } bind/join \}= \\
&\quad (meas\ (map\ ((r \circ head) \bigoplus sumR)\ (join\ (map\ trjps\ mx')))) && =\{ \text{ by } measAlgLemma \}= \\
&\quad (meas\ (map\ (meas \circ map\ (r \circ head \bigoplus sumR) \circ trjps)\ mx')) && =\{ \text{ by } measSumLemma \}= \\
&\quad (meas\ (map\ (r \bigoplus meas \circ map\ sumR \circ trjps)\ mx')) && =\{ \text{ by definition of } val' \}= \\
&\quad (meas\ (map\ (r \bigoplus val'\ ps)\ mx')) && =\{ \text{ by induction hypothesis} \}= \\
&\quad (meas\ (map\ (r \bigoplus val\ ps)\ mx')) && =\{ \text{ by definition of } val \}= \\
&\quad (val\ (p :: ps)\ x)
\end{aligned}
$$

□

Technical remarks. The above proof of *valMeasTotalReward* omits some technical details that may be uninteresting for a pen and paper proof, but turn out to be crucial in the setting of an intensional type theory – like Idris – where function extensionality does not hold in general. In particular, we have to postulate that the functorial *map* preserves extensional equality (see Appendix 1.2 and Botta *et al.* (*n.d.*)) for Idris to accept the proof. In fact, most of the reasoning proceeds by replacing functions that are mapped onto monadic values by other functions that are only extensionally equal. Using that *map* preserves extensional equality allows to carry out such proofs generically without knowledge of the concrete structure of the functor.

6.4 Correctness of monadic backward induction

As corollary, we can now prove the correctness of monadic backward induction, namely that the policy sequences computed by *bi* are optimal with respect to the measured total reward computed by val':

$$biOptMeasTotalReward : (t, n : \mathbb{N}) \to GenOptPolicySeq\ val'\ (bi\ t\ n)$$

[8] We are very grateful to the anonymous reviewer who suggested an alternative proof for the induction step. The proof presented here is based on his proof, and his suggestions have lead to significantly weaker conditions on the measure and thus a stronger result.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

```
biOptMeasTotalReward t n ps' x =
  let vvEqL = sym (valMeasTotalReward ps' x)      in
  let vvEqR = sym (valMeasTotalReward (bi t n) x) in
  let biOpt  = biOptVal t n ps' x                 in
  replace vvEqR (replace vvEqL biOpt)
```

7 Discussion

In the last two sections, we have seen what the three conditions mean for concrete examples and how they are used in the correctness proof. In this section, we take a step back and consider them from a more abstract point of view.

Category-theoretical perspective. Readers familiar with the theory of monads might have recognised that the first two conditions ensure that *meas* is the structure map of a monad algebra for M on Val and thus the pair $(Val, meas)$ is an object of the Eilenberg-Moore category associated with the monad M. The third condition requires the map $(v\oplus)$ to be an M-algebra homomorphism – a structure preserving map – for arbitrary values v.

This perspective allows us to use existing knowledge about monad algebras as a first criterion for choosing measures. For example, the Eilenberg–Moore algebras of the list monad are monoids – this implicitly played a role in the examples we considered above. Jacobs (2011) shows that the algebras of the distribution monad for probability distributions with finite support correspond to convex sets. Interestingly, convex sets play an important role in the theory of optimal control (Bertsekas *et al.*, 2003).

Measures for the list monad. The knowledge that monoids are $List$-algebras suggests a generic description of admissible measures for $M = List$: Given a monoid $(Val, \odot, b)$, we can prove that monoid homomorphisms of the form $foldr\ \odot\ b$ fulfil the three conditions, if $\oplus$ distributes over $\odot$ on the left. I.e. for $meas = foldr\ \odot\ b$ the three conditions can be proven from

$$
\begin{aligned}
&odotNeutrRight &&: (l : Val) &&\to l \odot neutr &&= l\\
&odotNeutrLeft &&: (r : Val) &&\to neutr \odot r &&= r\\
&odotAssociative &&: (l, v, r : Val) &&\to l \odot (v \odot r) = (l \odot v) \odot r\\
&oplusOdotDistrLeft &&: (n, l, r : Val) &&\to n \oplus (l \odot r) = (n \oplus l) \odot (n \oplus r)
\end{aligned}
$$

Neutrality of b on the right is needed for *measPureSpec*, while *measJoinSpec* follows from neutrality on the left and the associativity of $\odot$. The algebra morphism condition on $(v\oplus)$ is provable from the distributivity of $\oplus$ over $\odot$ and again neutrality of b on the right. If moreover $\odot$ is monotone with respect to $\sqsubseteq$

$$odotMon : \{a, b, c, d : Val\} \to a \sqsubseteq b \to c \sqsubseteq d \to (a \odot c) \sqsubseteq (b \odot d)$$

then we can also prove *measMonSpec* using the transitivity of $\sqsubseteq$. The proofs are simple and can be found in the supplementary material to this paper. This also illustrates how the three abstract conditions follow from more familiar algebraic properties.

Mutual independence. Although it does not seem surprising, it should be noted that the three conditions are mutually independent. This can be concluded from the

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

counter-examples in Section 5.1: The sum, the modified list maximum and the arithmetic average each fail exactly one of the three conditions.

Sufficient versus necessary. The three conditions are sufficient to prove the extensional equality of the functions *val* and *val′*. They are justified by their level of generality and the fact that they hold for standard *measures* used in control theory. However, we leave open the interesting question whether these conditions are also necessary for the correctness of monadic backward induction.

Non-emptiness requirement. Note that *mv* in the premises of *measPlusSpec* is required to be non-empty. This condition arises from a pragmatic consideration. As an example, let us again use the list monad with $Val = \mathbb{N}$ and $\oplus = +$. It is not hard to see that for any natural number n greater than 0 the equality $meas\,(map\,(n+)\,[\,]) = n + meas\,[\,]$ must fail. So, if we wish to use the standard list data type instead of defining a custom type of non-empty lists, the only way to prove the base case of *measPlusSpec* is by contradiction with the non-emptiness premise.

However, omitting the premise $mv : NotEmpty$ would not prevent us from generically proving the correctness result of Section 6 – it would even simplify matters as it would spare us reasoning about preservation of non-emptiness. But it would implicitly restrict the class of monads that can be used to instantiate M. For example, we have seen above that *measPlusSpec* is not provable for the empty list without the non-emptiness premise and we would therefore need to resort to a custom type of non-empty lists instead.

The price to pay for including the non-emptiness premise is the additional condition *nextNotEmpty* on the transition function *next* that was already stated in Section 3.3. Moreover, we have to postulate non-emptiness preservation laws for the monad operations (Appendix 1.2) and to prove an additional lemma about the preservation of non-emptiness (Appendix 4). Conceptually, it might seem cleaner to omit the non-emptiness condition: In this case, the remaining conditions would only concern the interaction between the monad, the measure, the type of values and the binary operation $\oplus$. However, the non-emptiness preservation laws seem less restrictive with respect to the monad. In particular, for our above example of ordinary lists they hold (the relevant proofs can be found in the supplementary material). Thus, we have opted for explicitly restricting the *next* function instead of implicitly restricting the class of monads for which the result of Section 6 holds.

8 Conclusion

In this paper, we have proposed correctness criteria for monadic backward induction and its underlying value function in the framework for specifying and solving finite-horizon, monadic SDPs proposed in Botta *et al.* (2017*a*). After having shown that these criteria are not necessarily met for arbitrary monadic SDPs, we have formulated three general compatibility conditions. We have given a proof that monadic backward induction and its underlying value function are correct if these conditions are fulfilled.

The main theorem has been proved via the extensional equality of two functions: (1) the value function of Bellman's dynamic programming (Bellman, 1957) and optimal control

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

theory (Bertsekas, 1995; Puterman, 2014) that is also at the core of the generic backward induction algorithm of Botta *et al.* (2017*a*) and (2) the measured total reward function that specifies the objective of decision making in monadic SDPs: the maximisation of a measure of the sum of the rewards along the trajectories rooted at the state associated with the first decision.

Our contribution to verified optimal decision making is twofold: On the one hand, we have implemented a machine-checked generalisation of the semi-formal results for deterministic and stochastic SDPs discussed in Bertsekas (1995, Prop. 1.3.1) and Puterman (2014, Theorem 4.5.1.c). As a consequence, we now have a provably correct method for solving deterministic and stochastic sequential decision problems with their canonical measure functions. On the other hand, we have identified three general conditions that are sufficient for the equivalence between the two functions and thus the correctness result to hold. The first two conditions are natural compatibility conditions between the measure of uncertainty *meas* and the monadic operations associated with the uncertainty monad M. The third condition is a distributivity principle concerning the relationship between *meas*, the functorial map associated with M and the rule for adding rewards $\oplus$. All three conditions have a straightforward category-theoretical interpretation in terms of Eilenberg-Moore algebras (MacLane, 1978, ch. VI.2). As discussed in Section 7, the three conditions are independent and have non-trivial implications for the measure and the addition function that cannot be derived from the monotonicity condition on *meas* already imposed in Ionescu (2009); Botta *et al.* (2017*a*).

A consequence of this contribution is more flexibility: We can now compute verified solutions of stochastic sequential decision problems in which the measure of uncertainty is different from the expected value measure. This is important for applications in which the goal of decision making is, for example, of maximising the value of worst-case outcomes. To the best of our knowledge, the formulation of the compatibility condition and the proof of the equivalence between the two value functions are novel results.

The latter can be employed in a wider context than the one that has motivated our study: in many practical problems in science and engineering, the computation of optimal policies via backward induction (let apart brute-force or gradient methods) is simply not feasible. In these problems one often still needs to generate, evaluate and compare different policies and our result shows under which conditions such evaluation can safely be done via the "fast" value function *val* of standard control theory.

Finally, our contribution is an application of verified, literal programming to optimal decision making: the sources of this document have been written in literal Idris and are available at Brede & Botta (2021), where the reader can also find the bare code and some examples. Although the development has been carried out in Idris, it should be readily reproducible in other implementations of type theory like Agda or Coq.

Acknowledgements

The work presented in this paper was motivated by a remark of Marina Martínez Montero who raised the question of the equivalence between *val* and *val′* (and, thus, of the correctness of the Botta *et al.* framework) during an introduction to verified decision making that

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

the authors gave at UCL (Université catholique de Louvain) in 2019. We are especially thankful to Marina for that question!

We are grateful to Jeremy Gibbons, Christoph Kreitz, Patrik Jansson, Tim Richter and to the JFP editors and reviewers, whose comments and recommendations have lead to significant improvements of the original manuscript.

A very special thanks goes to the anonymous reviewer who has suggested both a more straightforward proof of the *val*-*val*′ equality and, crucially, weaker conditions on the measure function for the result to hold. This warrants the applicability of the Botta *et al.* framework for verified decision making to a wider class of problems than our original conditions.

The work presented in this paper heavily relies on free software, among others on Coq, Idris, Agda, GHC, git, vi, Emacs, LaTeX and on the FreeBSD and Debian GNU/Linux operating systems. It is our pleasure to thank all developers of these excellent products. This is TiPES contribution No 37. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 820970 (TiPES –Tipping Points in the Earth System, 2019–2023).

Conflicts of interest

None.

Supplementary material

For supplementary material for this article, please visit http://dx.doi.org/10.1017/S0956796821000228.

References

Audebaud, P. & Paulin-Mohring, C. (2009) Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* **74**(8), 568–589.

Bellman, R. (1957) *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P. & Shreve, S. E. (1996) *Stochastic Optimal Control: The Discrete Time Case*. Athena Scientific.

Bertsekas, D., Nedić, A. & Ozdaglar, A. (2003) *Convex Analysis and Optimization*. Athena Scientific Optimization and Computation Series. Athena Scientific.

Bertsekas, D., P. (1995) *Dynamic Programming and Optimal Control*. Athena Scientific.

Bird, R. (2014) *Thinking Functionally with Haskell*. Cambridge University Press.

Bird, R. & Gibbons, J. (2020) *Algorithm Design with Haskell*. Cambridge University Press.

Botta, N., Mandel, A., Hofmann, M., Schupp, S. & Ionescu, C. (2013) Mathematical specification of an agent-based model of exchange. In *Proceedings of the AISB Convention 2013, "Do-Form: Enabling Domain Experts to use Formalized Reasoning" Symposium*.

Botta, N., Jansson, P. & Ionescu, C. (2018) The impact of uncertainty on optimal emission policies. *Earth Syst. Dyn.* **9**(2), 525–542.

Botta, N. (2016–2021) *IdrisLibs*. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`.

Botta, N., Brede, N., Jansson, P. & Richter, T. (in press) Extensional equality preservation and verified generic programming. *J. Funct. Program.* (Accepted for publication August 2021). `https://arxiv.org/abs/2008.02123`.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Botta, N., Jansson, P. & Ionescu, C. (2017a) Contributions to a computational theory of policy advice and avoidability. *J. Funct. Program.* **27**, e23.
Botta, N., Jansson, P., Ionescu, C., Christiansen, D. R. & Brady, E. (2017b) Sequential decision problems, dependent types and generic solutions. *Log. Meth. Comput. Sci.* **13**(1).
Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* **23**(9), 552–593.
Brady, E. (2017) *Type-Driven Development in Idris*. Manning Publications Co.
Brede, N. & Botta, N. (2021) *On the Correctness of Monadic Backward Induction*. Git repository.
De Moor, O. (1995) A generic program for sequential decision processes. In *PLILPS '95 Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs,* pp. 1–23. Springer.
De Moor, O. (1999) Dynamic programming as a software component. In *Proc. 3rd WSEAS Int. Conf. Circuits, Systems, Communications and Computers (CSCC 1999),* pp. 4–8.
Diederich, A. (2001) Sequential decision making. In *International Encyclopedia of the Social & Behavioral Sciences*, Smelser, N. J. & Baltes, P. B. (eds), pp. 13917–13922. Pergamon.
Erwig, M. and Kollmansberger, S. (2006) Functional Pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.* **16**(1), 21–34.
Finus, M., van Ierland, E. & Dellink, R. (2003) *Stability of Climate Coalitions in a Cartel Formation Game*. FEEM Working Paper No. 61.2003.
Gintis, H. (2007) The dynamics of general equilibrium. *Econ. J.* **117**, 1280–1309.
Giry, M. (1981) A categorial approach to probability theory. In *Categorical Aspects of Topology and Analysis*, Banaschewski, B. (ed). Lecture Notes in Mathematics 915, pp. 68–85. Springer.
Heitzig, J. (2012) *Bottom-Up Strategic Linking of Carbon Markets: Which Climate Coalitions Would Farsighted Players Form?* SSRN Environmental Economics eJournal.
Helm, C. (2003) International emissions trading with endogenous allowance choices. *J. Public Econ.* **87**, 2737–2747.
Ionescu, C. (2009) *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin.
Jacobs, B. (2011) Probabilities, distribution monads, and convex categories. *Theor. Comput. Sci.* **412**(28), 3323–3336.
MacLane, S. (1978) *Categories for the Working Mathematician*. 2nd edn. Graduate Texts in Mathematics. Springer.
Mercure, J.-F., Sharpe, S., Vinuales, J., Ives, M., Grubb, M., Pollitt, H., Knobloch, F. & Nijsse, F. (2020) Risk-opportunity analysis for transformative policy design and appraisal. *C-EENRG Working Papers* **2020-4**, 1–40.
Puterman, M. L. (2014) *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.
TiPES. (2019–2023) *TiPES H2020 Project Website*. https://www.tipes.dk/.
Wadler, P. (2015) Propositions as types. *Commun. ACM* **58**(12), 75–84.

Appendices

1 Preliminaries

1.1 General remarks concerning the Idris formalisation

- Idris' type checker often struggles with dependencies in implicit arguments. We sometimes use abbreviations to avoid cluttering the proofs with implicit arguments.
- As a standard, we write $(f \circ g \circ h)\, x$ instead of $f\ (g\ (h\, x))$.
  When this is a problem for the type checker, we use the second notation.
- Functions that are not defined explicitly are from the Idris standard library.
  Examples are *cong*, *void* and *concat*.

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

- Proofs in Idris can be implemented by preorder reasoning.
  I.e. equational reasoning steps of the form

  ```
  (t1) ={ step }=
  (t2)          □
  ```

  as displayed in this appendix are actual type-checkable implementations of proofs.

1.2 Monad laws

In the BJI-framework, M is required to be a *container monad* but none of the standard monad laws (Bird, 2014) is required for the verification result to hold. By contrast, to prove our extended correctness result, we need M to be a full-fledged monad. More specifically, we require of the monad M that

- it is equipped with functor and monad operations:

  $$\begin{aligned}
  &\mathit{map} &&: \{A, B : \mathit{Type}\} \to (A \to B) \to M\,A \to M\,B\\
  &\mathit{pure} &&: \{A : \mathit{Type}\} \to A \to M\,A\\
  &(\gg\!=) &&: \{A, B : \mathit{Type}\} \to M\,A \to (A \to M\,B) \to M\,B\\
  &\mathit{join} &&: \{A : \mathit{Type}\} \to M\,(M\,A) \to M\,A
  \end{aligned}$$

- it preserves extensional equality (Botta *et al.*, n.d.), identity and composition of arrows:

  $$\begin{aligned}
  &\mathit{mapPresEE} &&: \{A, B : \mathit{Type}\} \to (f, g : A \to B) \to f \doteq g \to \mathit{map}\,f \doteq \mathit{map}\,g\\
  &\mathit{mapPresId} &&: \{A : \mathit{Type}\} \to \mathit{map}\,\mathit{id} \doteq \mathit{id}\\
  &\mathit{mapPresComp} &&: \{A, B, C : \mathit{Type}\} \to (f : A \to B) \to (g : B \to C) \to\\
  &&&\quad \mathit{map}\,(g \circ f) \doteq \mathit{map}\,g \circ \mathit{map}\,f
  \end{aligned}$$

- Its *pure* and *join* operations are natural transformations (see MacLane, 1978, I.4):

  $$\begin{aligned}
  &\mathit{pureNatTrans} &&: \{A, B : \mathit{Type}\} \to (f : A \to B) \to \mathit{map}\,f \circ \mathit{pure} \doteq \mathit{pure} \circ f\\
  &\mathit{joinNatTrans} &&: \{A, B : \mathit{Type}\} \to (f : A \to B) \to \mathit{map}\,f \circ \mathit{join} \doteq \mathit{join} \circ \mathit{map}\,(\mathit{map}\,f)
  \end{aligned}$$

  and fulfil the neutrality and associativity axioms:

  $$\begin{aligned}
  &\mathit{pureNeutralLeft} &&: \{A : \mathit{Type}\} \to \mathit{join} \circ \mathit{pure} \doteq \mathit{id}\\
  &\mathit{pureNeutralRight} &&: \{A : \mathit{Type}\} \to \mathit{join} \circ \mathit{map}\,\mathit{pure} \doteq \mathit{id}\\
  &\mathit{joinAssoc} &&: \{A : \mathit{Type}\} \to \mathit{join} \circ \mathit{map}\,\mathit{join} \doteq \mathit{join} \circ \mathit{join}
  \end{aligned}$$

Notice that the above specification of the monad operations is not minimal: $(\gg\!=)$ is not assumed to be implemented in terms of *join* and *map* (or *map* and *join* via $(\gg\!=)$ and *pure*). Therefore, $(\gg\!=)$ (pronounced "*bind*"), *join* and *map* have to fulfil:

$$\mathit{bindJoinSpec} : \{A, B : \mathit{Type}\} \to (ma : M\,A) \to (f : A \to M\,B) \to (ma \gg\!= f) = \mathit{join}\,(\mathit{map}\,f\,ma)$$

The equality in the axioms is extensional equality, not the type theory's definitional equality:

$$\begin{aligned}
&(\doteq) : \{A, B : \mathit{Type}\} \to (f, g : A \to B) \to \mathit{Type}\\
&(\doteq)\ \ \{A\}\,f\,g = (a : A) \to f\,a = g\,a
\end{aligned}$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

As Idris does not have function extensionality, not postulating definitional equalities makes the axioms strictly weaker.

The BJI-framework also requires M to be equipped with type-level membership, with a universal quantifier and with a type-valued predicate

$$NotEmpty : \{A : Type\} \to M\ A \to Type$$

For our purposes, the monad operations are moreover required to fulfil the following preservation laws:

- The M-structure obtained from lifting an element into the monad is not empty:

$$pureNotEmpty : \{A : Type\} \to (a : A) \to NotEmpty\ (pure\ a)$$

- The monad's *map* and *bind* preserve non-emptiness:

$$\begin{aligned} &mapPresNotEmpty : \{A, B : Type\} \to (f : A \to B) \to (ma : M\ A) \to \\ &\qquad NotEmpty\ ma \to NotEmpty\ (map\ f\ ma) \end{aligned}$$

$$\begin{aligned} &bindPresNotEmpty : \{A, B : Type\} \to (f : A \to M\ B) \to (ma : M\ A) \to \\ &\qquad NotEmpty\ ma \to ((a : A) \to NotEmpty\ (f\ a)) \to NotEmpty\ (ma \gg\!= f) \end{aligned}$$

As discussed in Section 7, we could have omitted these non-emptiness preservation laws, but instead would have implicitly restricted the class of monads for which the correctness result holds.

1.3 Preservation of extensional equality

We have stated above that for our correctness proof the functor M has to satisfy the monad laws and moreover its functorial *map* has to preserve extensional equality.

This e.g. is the case for $M = Identity$ (no uncertainty), $M = List$ (non-deterministic uncertainty) and for the finite probability distributions (stochastic uncertainty, $M = Prob$) discussed in Botta *et al.* (2017*a*). For $M = List$ the proof of *mapPresEE* amounts to:

$$\begin{aligned} &mapPresEE : \{A, B : Type\} \to (f, g : A \to B) \to f \doteq g \to map\ f \doteq map\ g \\ &mapPresEE\ f\ g\ p\ \ Nil \qquad = Refl \\ &mapPresEE\ f\ g\ p\ (a :: as) = \\ &\quad (map\ f\ (a :: as)) \ =\{\ Refl\ \}= \\ &\quad (f\ a :: map\ f\ as) \ =\{\ cong\ \{f = \lambda x \Rightarrow x :: map\ f\ as\}\ (p\ a)\ \}= \\ &\quad (g\ a :: map\ f\ as) \ =\{\ cong\ (mapPresEE\ f\ g\ p\ as)\ \}= \\ &\quad (g\ a :: map\ g\ as) \ =\{\ Refl\ \}= \\ &\quad (map\ g\ (a :: as)) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square \end{aligned}$$

The principle of extensional equality preservation is discussed in more detail in Botta *et al.* (*n.d.*).

2 Correctness of the value function

$$\begin{aligned} &valMeasTotalReward : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to \\ &\qquad val'\ ps\ x = val\ ps\ x \end{aligned}$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$
\begin{array}{l}
valMeasTotalReward\ Nil\ x = \\
\quad (val'\ Nil\ x) \qquad\qquad\qquad\qquad =\{\ Refl\ \}= \\
\quad (meas\ (map\ sumR\ (trj\ Nil\ x))) \qquad =\{\ Refl\ \}= \\
\quad (meas\ (map\ sumR\ (pure\ (Last\ x)))) =\{\ cong\ (pureNatTrans\ sumR\ (Last\ x))\ \}= \\
\quad (meas\ (pure\ (sumR\ (Last\ x)))) \qquad =\{\ Refl\ \}= \\
\quad (meas\ (pure\ zero)) \qquad\qquad\qquad =\{\ measPureSpec\ zero\ \}= \\
\quad (zero) \qquad\qquad\qquad\qquad\qquad =\{\ Refl\ \}= \\
\quad (val\ Nil\ x) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \square
\end{array}
$$

$$
\begin{array}{l}
valMeasTotalReward\ \{t\}\ \{n = S\ m\}\ (p :: ps)\ x = \\
\quad \text{-- type abbreviations} \\
\quad \mathbf{let}\ SCS \quad : Type \qquad\qquad = StateCtrlSeq\ (S\ t)\ (S\ m) \qquad \mathbf{in} \\
\quad \mathbf{let}\ SCS' \quad : Type \qquad\qquad = StateCtrlSeq\ t\ (S\ (S\ m)) \qquad \mathbf{in} \\
\quad \text{-- element and function abbreviations} \\
\quad \mathbf{let}\ y \quad : Y\ t\ x \qquad\qquad = p\ x \qquad \mathbf{in} \\
\quad \mathbf{let}\ mx' \quad : M\ (X\ (S\ t)) \qquad = next\ t\ x\ y \qquad \mathbf{in} \\
\quad \mathbf{let}\ r \quad :(X\ (S\ t) \to Val) \qquad = reward\ t\ x\ y \qquad \mathbf{in} \\
\quad \mathbf{let}\ trjps \quad :(X\ (S\ t) \to M\ SCS) = trj\ ps \qquad \mathbf{in} \\
\quad \mathbf{let}\ consxy \quad :(SCS \to SCS') \qquad = ((x \circledast y)\#\#) \qquad \mathbf{in} \\
\quad \mathbf{let}\ mx'trjps : M\ SCS \qquad = (mx' \gg\!= trjps) \qquad \mathbf{in} \\
\quad \mathbf{let}\ sR \quad :(SCS \to Val) \qquad = sumR\ \{t = S\ t\}\ \{n = S\ m\} \qquad \mathbf{in} \\
\quad \mathbf{let}\ hd \quad :(SCS \to X\ (S\ t)) \qquad = head\ \{t = S\ t\}\ \{n = m\} \qquad \mathbf{in} \\
\quad \text{-- proof steps:} \\
\quad \mathbf{let}\ useMapPresComp \quad = mapPresComp\ consxy\ sumR\ mx'trjps \qquad \mathbf{in} \\
\quad \mathbf{let}\ useBindJoinSpec \quad = bindJoinSpec\ \{B = SCS\}\ trjps\ mx' \qquad \mathbf{in} \\
\quad \mathbf{let}\ useAlgLemma \quad = algLemma\ \{B = SCS\} \\
\qquad\qquad\qquad\qquad\quad meas\ measJoinSpec \\
\qquad\qquad\qquad\qquad\quad ((r \circ hd) \bigoplus sumR)\ trjps\ mx' \qquad \mathbf{in} \\
\quad \mathbf{let}\ useMeasSumLemma = mapPresEE \\
\qquad\qquad\qquad\qquad\quad (meas \circ map\ (r \circ hd \bigoplus sR) \circ trjps) \\
\qquad\qquad\qquad\qquad\quad (r \bigoplus meas \circ map\ sR \circ trjps) \\
\qquad\qquad\qquad\qquad\quad (measSumLemma\ ps\ r\ sR) \\
\qquad\qquad\qquad\qquad\quad mx' \qquad \mathbf{in} \\
\quad \mathbf{let}\ useIH \quad = mapPresEE \\
\qquad\qquad\qquad\qquad\quad (r \bigoplus val'\ ps) \\
\qquad\qquad\qquad\qquad\quad (r \bigoplus val\ ps) \\
\qquad\qquad\qquad\qquad\quad (oplusLiftEERight\ (val'\ ps) \\
\qquad\qquad\qquad\qquad\qquad (val\ ps)\ r\ (valMeasTotalReward\ ps)) \\
\qquad\qquad\qquad\qquad\quad mx' \qquad \mathbf{in} \\
\quad \mathbf{let}\ ctx \quad = \lambda a \Rightarrow meas\ (map\ ((r \circ hd) \bigoplus sumR)\ a)\ \mathbf{in} \\
\quad (val'\ (p :: ps)\ x) \qquad\qquad\qquad\qquad\qquad =\{\ Refl\ \}= \\
\quad (meas\ (map\ sumR\ (trj\ (p :: ps)\ x))) \qquad\qquad =\{\ Refl\ \}= \\
\quad (meas\ (map\ sumR\ (map\ consxy\ mx'trjps))) \qquad\qquad =\{\ cong\ (sym\ useMapPresComp)\ \}= \\
\quad (meas\ (map\ (sumR \circ consxy)\ mx'trjps)) \qquad\qquad =\{\ Refl\ \}= \\
\quad (meas\ (map\ ((r \circ hd) \bigoplus sR)\ mx'trjps)) \qquad\qquad =\{\ cong\ \{f = ctx\}\ useBindJoinSpec\ \}= \\
\quad (meas\ (map\ ((r \circ hd) \bigoplus sR)\ (join\ (map\ trjps\ mx')))) \qquad =\{\ sym\ useAlgLemma\ \}= \\
\quad (meas\ (map\ (meas \circ map\ (r \circ hd \bigoplus sR) \circ trjps)\ mx')) =\{\ cong\ useMeasSumLemma\ \}= \\
\quad (meas\ (map\ (r \bigoplus meas \circ map\ sR \circ trjps)\ mx')) \qquad =\{\ Refl\ \}=
\end{array}
$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$
\begin{aligned}
&(meas\ (map\ (r \bigoplus val'\ ps)\ mx')) && =\{\ cong\ useIH\ \}= \\
&(meas\ (map\ (r \bigoplus val\ ps)\ mx')) && =\{\ Refl\ \}= \\
&(val\ (p :: ps)\ x) && \square
\end{aligned}
$$

3 Correctness of monadic backward induction

Together with the result of Botta *et al.* (*biOptVal*, see Appendix 6 below), we can prove the correctness of monadic backward induction as corollary, using a generalised optimality of policy sequences predicate:

$$
\begin{aligned}
&GenOptPolicySeq : \{t, n : \mathbb{N}\} \to (f : PolicySeq\ t\ n \to X\ t \to Val) \to PolicySeq\ t\ n \to Type \\
&GenOptPolicySeq\ \{t\}\ \{n\}\ f\ ps = (ps' : PolicySeq\ t\ n) \to (x : X\ t) \to f\ ps'\ x \sqsubseteq f\ ps\ x
\end{aligned}
$$

$$
\begin{aligned}
&biOptMeasTotalReward : (t : \mathbb{N}) \to (n : \mathbb{N}) \to GenOptPolicySeq\ val'\ (bi\ t\ n) \\
&biOptMeasTotalReward\ t\ n\ ps'\ x = \\
&\quad \textbf{let}\ vvEqL = sym\ (valMeasTotalReward\ ps'\ x) \quad \textbf{in} \quad \text{-- } val'\ ps'\ x = val\ ps'\ x \\
&\quad \textbf{let}\ vvEqR = sym\ (valMeasTotalReward\ (bi\ t\ n)\ x) \ \textbf{in} \quad \text{-- } val'\ (bi\ t\ n)\ x = val\ (bi\ t\ n)\ x \\
&\quad \textbf{let}\ biOpt\ = biOptVal\ t\ n\ ps'\ x \quad \textbf{in} \quad \text{-- } val\ ps'\ x \sqsubseteq val\ (bi\ t\ n)\ x \\
&\quad \textbf{let}\ lP\ \ = \lambda v \Rightarrow v \sqsubseteq val\ (bi\ t\ n)\ x \quad \textbf{in} \\
&\quad \textbf{let}\ rP\ \ = \lambda v \Rightarrow val'\ ps'\ x \sqsubseteq v \quad \textbf{in} \\
&\quad replace\ \{P = rP\}\ vvEqR\ (replace\ \{P = lP\}\ vvEqL\ biOpt)
\end{aligned}
$$

4 Lemmas

The proof of *valMeasTotalReward* relies on a few auxiliary results which we prove here.

Lemma about the interaction of *head* and *trj* interleaved with *map*:

$$
\begin{aligned}
&headTrjLemma : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to \\
&\qquad (r : X\ t \to Val) \to (s : StateCtrlSeq\ t\ (S\ n) \to Val) \to (x' : X\ t) \to \\
&\qquad (map\ (r \circ head \bigoplus s) \circ trj\ ps)\ x' = (map\ (const\ (r\ x') \bigoplus s) \circ trj\ ps)\ x'
\end{aligned}
$$

$$
\begin{aligned}
&headTrjLemma\ \{t\}\ \{n = Z\}\ Nil\ r\ s\ x' = \\
&\quad \textbf{let}\ hd\ \ = head\ \{t\} \quad \textbf{in} \\
&\quad \textbf{let}\ lastx' = Last\ \{t\}\ x' \quad \textbf{in} \\
&\quad \textbf{let}\ nil\ \ = Nil\ \{t\} \quad \textbf{in} \\
&\quad \textbf{let}\ usePureNatTrans\ \ = pureNatTrans\ (r \circ hd \bigoplus s)\ lastx' \quad \textbf{in} \\
&\quad \textbf{let}\ usePureNatTransSym = sym\ (pureNatTrans\ ((\bigoplus)\ (r\ x') \circ s)\ lastx') \ \textbf{in} \\
&\qquad ((map\ (r \circ hd \bigoplus s))\ (trj\ nil\ x')) \quad =\{\ Refl\ \}= \\
&\qquad ((map\ (r \circ hd \bigoplus s))\ (pure\ lastx')) \quad =\{\ usePureNatTrans\ \}= \\
&\qquad ((pure \circ (r \circ hd \bigoplus s))\ lastx') \quad =\{\ Refl\ \}= \\
&\qquad ((pure \circ (const\ (r\ x') \bigoplus s))\ lastx') \quad =\{\ usePureNatTransSym\ \}= \\
&\qquad (map\ (const\ (r\ x') \bigoplus s)\ (pure\ lastx')) =\{\ Refl\ \}= \\
&\qquad (map\ (const\ (r\ x') \bigoplus s)\ (trj\ nil\ x')) \quad \square \\
&headTrjLemma\ \{t = t\}\ \{n = S\ n\}\ (p :: ps)\ r\ s\ x' = \\
&\quad \textbf{let}\ y'\ \ = p\ x' \quad \textbf{in}
\end{aligned}
$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$
\begin{array}{ll}
\textbf{let } mx'' = next\ t\ x'\ y' & \textbf{in} \\
\textbf{let } SCS = (StateCtrlSeq\ (S\ t)\ (S\ n)) & \textbf{in} \\
\textbf{let } consx'y' = (\#\#)\ \{t\}\ \{n\}\ (x' \circledast y') & \textbf{in} \\
\textbf{let } mx''trjps = (\gg\!=)\ \{B = SCS\}\ mx''\ (trj\ ps) & \textbf{in} \\
\textbf{let } traj = trj\ \{t\} & \textbf{in} \\
\textbf{let } useMapPresComp = mapPresComp\ consx'y'\ (const\ (r\ x') \oplus s)\ mx''trjps & \textbf{in} \\
\textbf{let } useMapPresCompSym = sym\ (mapPresComp\ consx'y'\ (r \circ head \oplus s)\ mx''trjps) & \textbf{in} \\
((map\ (r \circ head \oplus s))\ (traj\ (p :: ps)\ x')) & =\{\ Refl\ \}= \\
((map\ (r \circ head \oplus s) \circ map\ consx'y')\ mx''trjps) & =\{\ useMapPresCompSym\ \}= \\
((map\ ((r \circ head \oplus s) \circ consx'y'))\ mx''trjps) & =\{\ Refl\ \}= \\
(map\ ((const\ (r\ x') \oplus s) \circ consx'y')\ mx''trjps) & =\{\ useMapPresComp\ \}= \\
(map\ (const\ (r\ x') \oplus s)\ (map\ consx'y'\ mx''trjps)) & =\{\ Refl\ \}= \\
(map\ (const\ (r\ x') \oplus s)\ (traj\ (p :: ps)\ x')) & \square
\end{array}
$$

Lemma about the commutation of $meas \circ map$ and $\bigoplus$:

$$
\begin{array}{l}
measSumLemma : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to \\
\qquad (r : X\ t \to Val) \to \\
\qquad (s : StateCtrlSeq\ t\ (S\ n) \to Val) \to \\
\qquad (meas \circ map\ (r \circ head \oplus s) \circ trj\ ps) \doteq \\
\qquad (r \oplus meas \circ map\ s \circ trj\ ps)
\end{array}
$$

$$
\begin{array}{ll}
measSumLemma\ \{t\}\ \{n\}\ ps\ r\ s\ x' = & \\
\quad \text{-- non-emptiness proofs} & \\
\textbf{let } trjNE = trjNotEmptyLemma\ ps\ x' & \textbf{in} \\
\textbf{let } sNE = mapPresNotEmpty\ s\ (trj\ ps\ x')\ trjNE & \textbf{in} \\
\quad \text{-- proof steps} & \\
\textbf{let } useMeasPlusSpec = measPlusSpec\ (r\ x')\ (map\ s\ (trj\ ps\ x'))\ sNE & \textbf{in} \\
\textbf{let } useMapPresComp = cong\ \{f = \backslash\ prf \Rightarrow meas\ prf\} & \\
\qquad (mapPresComp\ s\ ((r\ x') \oplus)\ (trj\ ps\ x')) & \textbf{in} \\
\textbf{let } useHdTrjLemma = cong\ (headTrjLemma\ ps\ r\ s\ x') & \textbf{in} \\
((meas \circ map\ (r \circ head \oplus s) \circ trj\ ps)\ x') & =\{\ useHdTrjLemma\ \}= \\
((meas \circ map\ (const\ (r\ x') \oplus s) \circ trj\ ps)\ x') & =\{\ Refl\ \}= \\
((meas \circ map\ ((const\ (r\ x') \oplus id) \circ s) \circ trj\ ps)\ x') & =\{\ useMapPresComp\ \}= \\
((meas \circ map\ (const\ (r\ x') \oplus id) \circ map\ s \circ trj\ ps)\ x') & =\{\ Refl\ \}= \\
((meas \circ map\ ((r\ x') \oplus) \circ map\ s \circ trj\ ps)\ x') & =\{\ useMeasPlusSpec\ \}= \\
((((r\ x') \oplus) \circ meas \circ map\ s \circ trj\ ps)\ x') & =\{\ Refl\ \}= \\
(((const\ (r\ x') \oplus id) \circ meas \circ map\ s \circ trj\ ps)\ x') & =\{\ Refl\ \}= \\
((r \oplus meas \circ map\ s \circ trj\ ps)\ x') & \square
\end{array}
$$

The *trj* function never produces an empty M-structure of trajectories:

$$
trjNotEmptyLemma : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to NotEmpty\ (trj\ ps\ x)
$$

$$
\begin{array}{ll}
trjNotEmptyLemma\ (Nil) & x = pureNotEmpty\ (Last\ x) \\
trjNotEmptyLemma\ \{t = t\}\ \{n = S\ n\}\ (p :: ps)\ x = & \\
\quad \textbf{let } y = p\ x & \textbf{in} \\
\quad \textbf{let } trjps = trj\ ps & \textbf{in} \\
\quad \textbf{let } nxpx = next\ t\ x\ y & \textbf{in} \\
\quad \textbf{let } consxy = ((x \circledast y)\#\#) & \textbf{in}
\end{array}
$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

```
let nne    = nextNotEmpty x y                          in
let netrjps = trjNotEmptyLemma ps                      in
let bne    = bindPresNotEmpty trjps nxpx nne netrjps in
mapPresNotEmpty consxy (nxpx >>= trjps) bne
```

A technical lemma to lift equalities into the right component of $\oplus$:

$$oplusLiftEERight : \{A : Type\} \to (f, g, h : A \to Val) \to (g \doteq h) \to (f \oplus g) \doteq (f \oplus h)$$

$$oplusLiftEERight\ \{A\}\ f\ g\ h\ ee\ a = cong\ (ee\ a)$$

4.1 Properties of monad algebras

Condition for a function f to be an M-algebra homomorphism:

$$algMorSpec : \{A, B : Type\} \to (\alpha : M\ A \to A) \to (\beta : M\ B \to B) \to (f : A \to B) \to Type$$
$$algMorSpec\ \{A\}\ \{B\}\ \alpha\ \beta\ f = (\beta \circ map\ f) \doteq (f \circ \alpha)$$

Structure maps of M-algebras are left inverses of *pure*:

$$algPureSpec : \{A : Type\} \to (\alpha : M\ A \to A) \to Type$$
$$algPureSpec\ \{A\}\ \alpha = \alpha \circ pure \doteq id$$

Structure maps of M-algebras are themselves M-algebra homomorphisms:

$$algJoinSpec : \{A : Type\} \to (\alpha : M\ A \to A) \to Type$$
$$algJoinSpec\ \{A\}\ \alpha = algMorSpec\ join\ \alpha\ \alpha \quad \text{-- } (\alpha \circ map\ \alpha) \doteq (\alpha \circ join)$$

A lemma about computation with M-algebras:

$$algLemma : \{A, B, C : Type\} \to (\alpha : M\ C \to C) \to (ee : algJoinSpec\ \alpha) \to$$
$$(f : B \to C) \to (g : A \to M\ B) \to$$
$$(\alpha \circ map\ (\alpha \circ map\ f \circ g)) \doteq (\alpha \circ map\ f \circ join \circ map\ g)$$

$$algLemma\ \{A\}\ \{B\}\ \{C\}\ \alpha\ ee\ f\ g\ ma =$$
$$((\alpha \circ map\ (\alpha \circ map\ f \circ g))\ ma) \quad =\{\ cong\ (mapPresComp\ g\ (\alpha \circ map\ f)\ ma)\ \}=$$
$$((\alpha \circ map\ (\alpha \circ map\ f) \circ map\ g)\ ma) \quad =\{\ cong\ (mapPresComp\ (map\ f)\ \alpha\ (map\ g\ ma))\ \}=$$
$$((\alpha \circ map\ \alpha \circ map\ (map\ f) \circ map\ g)\ ma) =\{\ ee\ (map\ (map\ f)\ (map\ g\ ma))\ \}=$$
$$((\alpha \circ join \circ map\ (map\ f) \circ map\ g)\ ma) \quad =\{\ cong\ (sym\ (joinNatTrans\ f\ (map\ g\ ma)))\ \}=$$
$$((\alpha \circ map\ f \circ join \circ map\ g)\ ma) \qquad \square$$

4.2 Measure specifications

The measure needs to be the structure map of an M-algebra on *Val*. This means:

- It is a left inverse to *pure*:

  $$measPureSpec : algPureSpec\ meas \quad \text{-- } meas \circ pure \doteq id$$

- It is an M-algebra homomorphism from *join* to itself:

  $$measJoinSpec : algJoinSpec\ meas \quad \text{-- } meas \circ join \doteq meas \circ map\ meas$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

Moreover, for all $v : Val$, the function $(v\oplus)$ needs to be an M-algebra homomorphism:

$$\begin{aligned}&\mathit{measPlusSpec} : (v : \mathit{Val}) \to (mv : M\ \mathit{Val}) \to (\mathit{NotEmpty}\ mv) \to \\ &\qquad (\mathit{meas} \circ \mathit{map}\ (v\oplus))\ mv = ((v\oplus) \circ \mathit{meas})\ mv\end{aligned}$$

We can omit the non-emptiness condition but this means we implicitly restrict the class of monads that can be used for M. The condition could then be expressed as

$$\mathit{measPlusSpec}' : (v : \mathit{Val}) \to \mathit{algMorSpec}\ \mathit{meas}\ \mathit{meas}\ (\oplus v)$$

5 Bellman's principle of optimality

Basic requirements for monadic backward induction:

$$(\sqsubseteq) : \mathit{Val} \to \mathit{Val} \to \mathit{Type}$$

$$\begin{aligned}&\mathit{lteRefl} && : \{a : \mathit{Val}\} \to a \sqsubseteq a \\ &\mathit{lteTrans} && : \{a, b, c : \mathit{Val}\} \to a \sqsubseteq b \to b \sqsubseteq c \to a \sqsubseteq c \\ &\mathit{plusMonSpec} && : \{a, b, c, d : \mathit{Val}\} \to a \sqsubseteq b \to c \sqsubseteq d \to (a \oplus c) \sqsubseteq (b \oplus d)\end{aligned}$$

$$\begin{aligned}&\mathit{measMonSpec} : \{A : \mathit{Type}\} \to (f, g : A \to \mathit{Val}) \to ((a : A) \to (f\ a) \sqsubseteq (g\ a)) \to \\ &\qquad (ma : M\ A) \to \mathit{meas}\ (\mathit{map}\ f\ ma) \sqsubseteq \mathit{meas}\ (\mathit{map}\ g\ ma)\end{aligned}$$

Optimality of policy sequences:

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

Optimality of extensions of policy sequences:

$$\begin{aligned}&\mathit{OptExt} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{Policy}\ t \to \mathit{Type} \\ &\mathit{OptExt}\ \{t\}\ ps\ p = (p' : \mathit{Policy}\ t) \to (x : X\ t) \to \mathit{val}\ (p' :: ps)\ x \sqsubseteq \mathit{val}\ (p :: ps)\ x\end{aligned}$$

Bellman's principle of optimality:

$$\begin{aligned}&\mathit{Bellman} : \{t, n : \mathbb{N}\} \to (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{OptPolicySeq}\ ps \to \\ &\qquad\qquad (p\ \ : \mathit{Policy}\ t) \qquad\quad \to \mathit{OptExt}\ ps\ p \qquad\ \to \mathit{OptPolicySeq}\ (p :: ps) \\ &\mathit{Bellman}\ \{t\}\ ps\ ops\ p\ oep\ (p' :: ps')\ x = \\ &\quad \mathbf{let}\ y' \ \ = p'\ x \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in} \\ &\quad \mathbf{let}\ mx' = \mathit{next}\ t\ x\ y' \qquad\qquad\qquad\qquad\quad \mathbf{in} \\ &\quad \mathbf{let}\ f' \ \ = \mathit{reward}\ t\ x\ y' \bigoplus \mathit{val}\ ps' \qquad\quad \mathbf{in} \\ &\quad \mathbf{let}\ f \ \ = \mathit{reward}\ t\ x\ y' \bigoplus \mathit{val}\ ps \qquad\quad \mathbf{in} \\ &\quad \mathbf{let}\ s_0 \ \ = \lambda x' \Rightarrow \mathit{plusMonSpec}\ \mathit{lteRefl}\ (\mathit{ops}\ ps'\ x') \ \mathbf{in} \\ &\quad \mathbf{let}\ s1 \ \ = \mathit{measMonSpec}\ f'\ f\ s_0\ mx' \qquad\quad \mathbf{in} \quad \text{-- } \mathit{val}\ (p' :: ps')\ x \sqsubseteq \mathit{val}\ (p' :: ps)\ x \\ &\quad \mathbf{let}\ s2 \ \ = oep\ p'\ x \qquad\qquad\qquad\qquad\quad \mathbf{in} \quad \text{-- } \mathit{val}\ (p' :: ps)\ x \sqsubseteq \mathit{val}\ (p :: ps)\ x \\ &\quad \mathit{lteTrans}\ s1\ s2\end{aligned}$$

6 Verification with respect to *val*

The empty policy sequence is optimal:

$$\begin{aligned}&\mathit{nilOptPolicySeq} : \mathit{OptPolicySeq}\ \mathit{Nil} \\ &\mathit{nilOptPolicySeq}\ \mathit{Nil}\ x = \mathit{lteRefl}\end{aligned}$$

Now, provided that we can implement

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

$$
\begin{aligned}
&\mathit{optExt} &&: \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{Policy}\ t \\
&\mathit{optExtSpec} &&: \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})
\end{aligned}
$$

then

$$
\begin{aligned}
&\mathit{bi} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{PolicySeq}\ t\ n \\
&\mathit{bi}\ t\ Z \quad = \mathit{Nil} \\
&\mathit{bi}\ t\ (S\ n) = \mathbf{let}\ \mathit{ps} = \mathit{bi}\ (S\ t)\ n\ \mathbf{in}\ \mathit{optExt}\ \mathit{ps} :: \mathit{ps}
\end{aligned}
$$

is correct with respect to *val*:

$$
\begin{aligned}
&\mathit{biOptVal} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n) \\
&\\
&\mathit{biOptVal}\ t\ Z \quad = \mathit{nilOptPolicySeq} \\
&\mathit{biOptVal}\ t\ (S\ n) = \\
&\quad \mathbf{let}\ \mathit{ps} \ = \mathit{bi}\ (S\ t)\ n \quad\ \mathbf{in} \\
&\quad \mathbf{let}\ \mathit{ops} = \mathit{biOptVal}\ (S\ t)\ n\ \mathbf{in} \\
&\quad \mathbf{let}\ p \quad = \mathit{optExt}\ \mathit{ps} \quad\ \mathbf{in} \\
&\quad \mathbf{let}\ \mathit{oep} = \mathit{optExtSpec}\ \mathit{ps} \quad \mathbf{in} \\
&\quad \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}
\end{aligned}
$$

7 Optimal extension

The generic implementation of backward induction *bi* naturally raises the question under which conditions one can implement *optExt* such that *optExtSpec* holds.
To this end, consider the function

$$
\begin{aligned}
&\mathit{cval} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to (x : X\ t) \to Y\ t\ x \to \mathit{Val} \\
&\mathit{cval}\ \{t\}\ \mathit{ps}\ x\ y = \mathbf{let}\ \mathit{mx}' = \mathit{next}\ t\ x\ y\ \mathbf{in} \\
&\qquad\qquad\qquad \mathit{meas}\ (\mathit{map}\ (\mathit{reward}\ t\ x\ y \bigoplus \mathit{val}\ \mathit{ps})\ \mathit{mx}')
\end{aligned}
$$

By definition of *val* and *cval*, one has

$$
\begin{aligned}
&\mathit{val}\ (p :: \mathit{ps})\ x \\
&\quad = \\
&\mathit{meas}\ (\mathit{map}\ (\mathit{reward}\ t\ x\ (p\ x) \bigoplus \mathit{val}\ \mathit{ps})\ (\mathit{next}\ t\ x\ (p\ x))) \\
&\quad = \\
&\mathit{cval}\ \mathit{ps}\ x\ (p\ x)
\end{aligned}
$$

This suggests that, if we can maximise *cval*, i.e. implement

$$
\begin{aligned}
&\mathit{cvalmax} &&: \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to (x : X\ t) \to \mathit{Val} \\
&\mathit{cvalargmax} &&: \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to (x : X\ t) \to Y\ t\ x
\end{aligned}
$$

that fulfil

$$
\begin{aligned}
&\mathit{cvalmaxSpec} : \quad \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ (S\ t)\ n) \to (x : X\ t) \to \\
&\qquad\qquad\qquad\quad (y : Y\ t\ x) \to \mathit{cval}\ \mathit{ps}\ x\ y \sqsubseteq \mathit{cvalmax}\ \mathit{ps}\ x \\
&\mathit{cvalargmaxSpec} : \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ (S\ t)\ n) \to (x : X\ t) \to \\
&\qquad\qquad\qquad\quad \mathit{cvalmax}\ \mathit{ps}\ x = \mathit{cval}\ \mathit{ps}\ x\ (\mathit{cvalargmax}\ \mathit{ps}\ x)
\end{aligned}
$$

then we can implement optimal extensions of arbitrary policy sequences. As it turns out, this intuition is correct. With

$$
\mathit{optExt} = \mathit{cvalargmax}
$$

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228

one has

$$
\begin{array}{l}
\mathit{optExtSpec}\,\{t\}\,\{n\}\,\mathit{ps}\,p'\,x = \\
\quad \textbf{let}\, p \;= \mathit{optExt}\,\mathit{ps} \quad \textbf{in} \\
\quad \textbf{let}\, y \;= p\,x \quad \textbf{in} \\
\quad \textbf{let}\, y' = p'\,x \quad \textbf{in} \\
\quad \textbf{let}\, s1 = \mathit{cvalmaxSpec}\,\mathit{ps}\,x\,y' \;\textbf{in} \\
\quad \textbf{let}\, s2 = \mathit{replace}\,\{P = \lambda z \Rightarrow (\mathit{cval}\,\mathit{ps}\,x\,y' \sqsubseteq z)\}\,(\mathit{cvalargmaxSpec}\,\mathit{ps}\,x)\,s1 \;\textbf{in} \\
\quad s2
\end{array}
$$

The observation that functions *cvalmax* and *cvalargmax* that fulfil *cvalmaxSpec* and *cvalargmaxSpec* are sufficient to implement an optimal extension *optExt* that fulfils *optExtSpec* naturally raises the question of what are necessary and sufficient conditions for *cvalmax* and *cvalargmax*. Answering this question necessarily requires discussing properties of *cval* and goes well beyond the scope of formulating a theory of SDPs. Here, we limit ourselves to remark that if $Y\,t\,x$ is finite and non-empty one can implement the functions *cvalmax* and *cvalargmax* by linear search. A generic implementation of *cvalmax* and *cvalargmax* can be found under Brede & Botta (2021).

For the original BJI-theory, tabulated backward induction and several example applications can be found in the *SequentialDecisionProblems* folder of Botta (2016–2021).

Downloaded from https://www.cambridge.org/core. IP address: 98.128.166.114, on 17 Nov 2021 at 10:17:38, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796821000228