Sequential decision problems, dependently typed solutions

Nicola Botta[1], Cezar Ionescu[1], and Edwin Brady[2]

[1] Potsdam Institute for Climate Impact Research, Telegrafenberg A31, 14473 Potsdam, Germany
{botta,ionescu}@pik-potsdam.de
[2] University of St Andrews, KY16 9SX, UK
ecb10@st-andrews.ac.uk

Abstract. We propose a dependently typed formalization for a simple class of sequential decision problems. For this class of problems, we implement a generic version of Bellman's backwards induction algorithm [2] and a machine checkable proof that the proposed implementation is correct. The formalization is generic. It is presented in Idris, but it can be easily translated to other dependently-typed programming languages. We conclude with an informal discussion of the problems we have faced in extending the formalization to generic monadic sequential decision problems.

1 Introduction

In this paper we formalize a simple class of sequential decision problems. For this class, we implement a generic version of Bellman's backwards induction algorithm [2] and derive a machine checkable proof that the proposed implementation is correct.

The approach is similar in spirit to that proposed by de Moor [7], with the focus on *generic* programming. Both the formalization of the problem and its solution are based on an abstract context. In order to solve specific sequential decision problems, clients must provide problem-specific instances of the context. There are important differences between our approach and de Moor's, however. In [7], Bellman's principle of optimality is rephrased in terms of three requirements: a *promotability* requirement for a feasibility predicate p and two monotonicity requirements for a list of *decision functions* fs (in [7], both p and fs are part of the abstract context). These requirements are expressed as predicates in first-order logic. In our approach, Bellman's principle of optimality is stated in the same language used for implementing backwards induction. This allows us to derive a proof of correctness (of the implementation) which can be machine checked.

An obvious consequence of our approach is that it requires a programming language which is expressive enough to implement generic algorithms and to encode (and prove) properties of such algorithms. Dependently typed languages such as Idris [6] and Agda [10] or Coq [3] support this by allowing types to be

predicated on *values*. A program's type encodes its meaning; allowing types to be predicated on values means a programmer may give a precise meaning to a program. Furthermore, generic programming is supported by allowing functions to compute types directly. We have implemented our formalization in Idris but a translation to Agda or other fully dependently-typed programming languages would be straightforward.

Another difference between our approach and the one described in [7] is that we explicitly introduce a state space in our context and that the set of controls (decisions) which are available at a given step depends on the actual state at that step. In contrast, the set of controls in [7] is constant. As it turns out, a state-dependent notion of feasible controls is essential to generalize the formalization presented here to the case of time-dependent state spaces and non-deterministic transition functions. We do not present such a generalization here, but we discuss the problem of extending our formalization in section 7.1.

1.1 Programming with Dependent Types

We will use the Idris programming language[3]. Idris is a general purpose language with dependent types. Its syntax, and many of its features such as type classes [12], are influenced by Haskell. Idris supports algebraic data types, such as natural numbers and polymorphic lists, declared as follows

```
data Nat = O | S Nat
data List a = Nil | (::) a (List a)
```

The canonical example of a dependent type is the type of lists indexed by length, usually called a vector. It is declared as follows, giving explicit types for the type constructor *Vect* as well as the constructors *Nil* and (::)

```
data Vect : Type → Nat → Type where
  Nil : Vect a O
  (::) : a → Vect a k → Vect a (S k)
```

Note that the constructors *Nil* and (::) have been overloaded for both *Vect* and *List* — overloading is resolved by type. A function over a dependent type encodes the invariant properties of its indices in its type, for example

```
(++) : Vect a n → Vect a m → Vect a (n + m)
(++) Nil ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

A program with a dependent type can be understood as a proof of the logical interpretation of that type. In particular, types can be used to represent relations, such as equality

```
data (=) : a → b → Type where
  refl : x = x
```

[3] `idris-lang.org`

Types are *first class*, meaning that they can be computed and manipulated like any other term. For example this allows us to define type synonyms

$$IntVect : Int \to Type$$
$$IntVect\ n = Vect\ Int\ n$$

We will make extensive use of the following type, *so*, which requires at type checking time that a boolean value is provably true

$$\textbf{data}\ so : Bool \to Type\ \textbf{where}$$
$$oh : so\ True$$

The only canonical instance of $so\ x$ is oh, and this can only be constructed if x is $True$. Dependent types allow the value of an input to a function to be mentioned in the type of the output — that is, a function type is a *binder*. In this way, we can use *so* to guarantee that a required dynamic check has been made, for example

$$safe\_divide : (x : Int) \to (y : Int) \to so\ (y \not\equiv 0) \to Int$$
$$safe\_divide\ x\ y\ oh = x\ `div`\ y$$

A full tutorial for Idris is available elsewhere [6].

1.2 Sequential decision problems

In textbooks on sequential decision problems and dynamic programming, the notions of state and control spaces are usually not formalized.

In Chapter 1 of [4], for instance, the notions of control sequence, policy, policy sequence, optimal policy sequence, Bellman's principle of optimality and backwards induction are discussed in the context of a discrete-time dynamical system of the form

$$x_{k+1} = f_k(x_k, u_k, w_k)$$

where x_k, u_k and w_k represent the state of the system, the selected control and a random parameter at time k. The types of x_k, u_k and, a fortiori, the type of f_k, are not explicitly stated. In many examples, the type of x_k turns out to be $\mathbb{R}^n$ or $\mathbb{N}^n$. But one can easily imagine sequential decision problems in which the state of the system is represented by pairs or perhaps by lists of pairs, as for example in the knapsack problem [7].

What about controls? Is it safe to assume that the type of u_k can be fixed, in specific sequential decision problems, independently of x_k? In problems of resource allocation, one can often take controls to be real numbers on the unit interval. The interpretation is that u_k represents a fraction of the available resource that are used for a certain purpose, for instance investment or saving.

In general, however, the controls available in a given state do depend on that specific state. Of course, one can always embed the set of controls which

are available in a given state in a larger set, thereby recovering the case of a constant control space. But then f_k becomes a partial function and may fail to produce a new state for some specific state-control pairs.

In the rest of this section we discuss a simple example of a sequential decision problem. It is an instance of the well known "cylinder" problem originally proposed by [11] and extensively studied in [5]. We will use this problem to characterize SDPs and to exemplify and discuss our formalization.

1.3 The "cylinder" problem

Consider the following problem: a decision-maker can be in one of five states: a, b, c, d or e. In a, the decision maker can choose between two options: move ahead (option A) or move to the right (option R). In b, c and d he can move to the left (L), ahead or to the right. In e he can move to the left or go ahead. Upon selecting an option, the decision maker enters a new state. For instance, selecting R in b brings him from b to c, see Figure 1, left. Thus, each step is characterized by a current state, a selected control and a new state. A step also yields a reward, for instance 3 for the transition from b to c and for option R, see Figure 1, middle. The challenge for the decision maker is to make a finite number of steps, say n, by selecting options that maximize the sum of the rewards collected. An example of a possible trajectory and corresponding rewards for the first four steps is shown on the right of Figure 1.

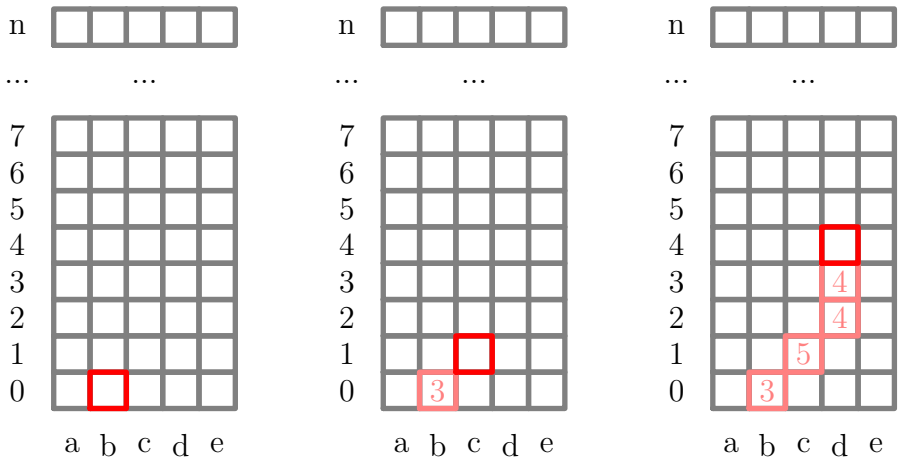

Fig. 1. The state of the agent is indicated by the bright red square, previous states are marked with faded color, the numbers in the faded squares represent the rewards the agent has collected. On the left we have the initial state, b. In the middle, the trajectory after one step: the agent has gone from b to c, collecting a reward of 3. On the right, we see the trajectory after 4 steps, the agent is now in state d, the last decision was to move ahead (A).

The cylinder problem is a very special example of a sequential decision problem. In this particular problem, the set of possible states — the *state space* – is finite and independent of the number of steps. The set of options available to the decision maker in a given state — the *control space* — is finite and only depends on that state and not, for instance, on the number of steps done.

In most SDPs, these conditions are not met. In assembly line scheduling problems [8], for instance, certain items have to be moved, at each of a fixed number of steps, to one of a finite number of assembly lines. Each step implies a transfer cost (if the item is to change assembly line) and a production cost. The latter depends on the current step and assembly line. The idea is to select moves that minimize the overall costs of assembling a given item. While in assembly line problems the state and the control space are finite, the number of assembly lines available at a given step might depend on the number of steps already done (assembly lines might go down for maintenance every so many steps).

In optimal investment problems, the state space, the control space or both might not be finite. The control space space, for instance, might be a real number on the unit interval representing, in a time-discrete setting, the fraction of the profit to be consumed at that step. Examples of SDPs are discussed, among others, in [1, 4, 8].

1.4 Basic assumptions

In this paper, we do not require the state and the control spaces to be finite[4], but we do assume that the state space is independent of the number of steps and that the options available to the decision maker in a given state only depend on that state. Moreover, we assume that, at each step, the new state depends on the current state and on the selected control via a transition function and that the transition function is known to the decision maker. Similarly, we assume that the reward collected in a given step depends on the current state, on the selected control and on the new state through a reward function.

These assumptions can be summarized in the notion of *time-independent*, *deterministic* sequential decision problems. This is the "simple class of sequential decision problems" considered in this paper.

In contrast, *general* sequential decision problems are sequential decision problems in which the state and control spaces might be time-dependent and the outcome of a step can be a set of new states (non-deterministic sequential decision problems) a probability distribution of new states (stochastic sequential decision problems) or some other monadic structure of states, see [9].

Time-independent, deterministic sequential decision problems are related to but different from *learning* problems: reinforcement learning, supervised learning, etc. In learning problems, the transition and the reward functions (or their monadic counterparts) are not known or only partially known to the decision maker and have to be "learned".

[4] But see section 6.

In section 7.1 we sketch the problems we have encountered in extending our formalization to time-dependent, monadic sequential decision problems. Such extension will be discussed in detail in a follow-up paper.

2 Context

We start by formalizing the context for time-independent, deterministic sequential decision problems. As anticipated in the introduction, we have very little constraints on the state space: we just require it to be a valid type

$$X : \mathit{Type}$$

To formalize the idea that the controls available in a specific state depend only on that state we introduce a function Y

$$Y : X \to \mathit{Type}$$

The type $Y\ x$ represents those controls (options, actions, etc.) which are available (feasible, admissible, etc.) in x. Of course, there are alternative ways to express the notion of state-specific admissible controls. For instance, one could explicitly introduce the set of all possible controls and represent admissibility in terms of a relation from X to that set.

Next, we formalize the notion of a deterministic transition function: selecting a control in a state yields a well defined new state

$$\mathit{step} : (x : X) \to Y\ x \to X$$

The fourth and last element of our generic context formalizes the notion of a reward function. As discussed in the introduction, each state transition yields a reward depending on the state in which the transition occurs, on the control selected in that state (e.g., via control costs) and on the new state:

$$\mathit{reward} : (x : X) \to Y\ x \to X \to \mathit{Float}$$

Throughout this paper we take rewards to be *Float*s but this assumption can be easily relaxed. Before turning to the next section a remark is in order: consider the type of *step*. The type of the first argument, X, is fixed, but the type of the second argument depends on the *value* of the first one. Without support for dependent types we would not be able to express such a type constraint precisely. In Haskell, for instance we could have written

$$\mathit{step} :: X \to Y' \to X$$

But then, *step* would not have been defined for all $y : Y'$ and it would have been difficult to express (let alone prove) properties of methods that rely on *step*.

3 Control sequences

We want to formalize the notion of a sequence of controls. For the generic context introduced in the previous section, the first element of a sequence of controls for $x : X$ has to be a value of type $Y\ x$, say y. The second element has to be a value of type $Y\ (step\ x\ y)$, say y'. The third element has to be a value of type $Y\ (step\ (step\ x\ y)\ y')$ and so on.

It is clear that we cannot express the type of a control sequence in terms of lists or vectors of elements of a *fixed* type: we need dependently typed sequences of controls. These can be indroduced by the type declaration:

```
data CtrlSeq : X → Nat → Type where
  Nil : CtrlSeq x O
  (::) : (y : Y x) → CtrlSeq (step x y) n → CtrlSeq x (S n)
```

As we explained in the introduction, the goal of the decision-maker is to maximize the sum of the rewards collected in a given number of steps. Consequently, the value of (selecting controls according to) a control sequence for an initial state $x : X$ and for a number of steps n is:

```
val : CtrlSeq x n → Float
val Nil = 0
val {x} (y :: ys) = reward x y (step x y) + val ys
```

The notation $\{x\}$ in the second clause allows us to bring the implicit argument x into scope, so that it can be used in the computation of the right-hand side.

In general, different control sequences for the same initial value yield different sequences of rewards and have different values. We want to formalize the notion of *optimal* control sequences. The idea is that a control sequence ys for initial x and n steps is optimal, if for every control sequence for the same x and n ys', the value of ys' is at most equal to the value of ys:

```
OptCtrlSeq : CtrlSeq x n → Type
OptCtrlSeq {x} {n} ys =
  (ys' : CtrlSeq x n) → so (val ys' ⩽ val ys)
```

Thus, proving that a control sequence $ys : CtrlSeq\ x\ n$ is optimal means implementing a function that computes a value of type $so\ (val\ ys' \leqslant val\ ys)$ for every $ys' : CtrlSeq\ x\ n$.

It is easy to prove that the empty control sequence is optimal for every initial state. All we have to do is to show that, for every initial state x, the value of an arbitrary control sequence of length 0 is smaller or equal to the value of Nil. This is certainly the case because, by definition of $CtrlSeq$, there are no control sequences of length 0 except for Nil:

```
nilIsOptCtrlSeq : (x : X) → OptCtrlSeq {x} Nil
nilIsOptCtrlSeq x ys' = reflexive_Float_lte 0
```

In proving the property *nilIsOptCtrlSeq* we have applied *reflexive_Float_lte* to 0. *reflexive_Float_lte* is a function of type $(x : Float) \to so\ (x \leqslant x)$. For all $(x : Float)$ it computes a value of type $so\ (x \leqslant x)$. For understanding *nilIsOptCtrlSeq* it is not important to know how *reflexive_Float_lte* has been implemented[5]. But it is important to notice that, when applied to 0, *reflexive_Float_lte* computes a value of type $so\ (0 \leqslant 0)$. The Idris type checker knows that for all $(x : X)$ and $ys' : CtrlSeq\ x\ O$, both $val\ ys'$ and $val\ Nil$ are equal to 0. Thus, the (type) value of $so\ (val\ ys' \leqslant val\ Nil)$ can be reduced to $so\ (0 \leqslant 0)$ and the type checker is fully satisfied by *reflexive_Float_lte* which, applied to 0, returns a value of exactly this type.

4 Policy sequences

It is easy to compute sequences of controls if one has a rule for which control to select in every state. Such rules are called *policies*:

$$
\begin{aligned}
&Policy : Type\\
&Policy = (x : X) \to Y\ x
\end{aligned}
$$

Sequences of policies can be represented by plain Idris vectors, as described in Section 1.1.

$$
\begin{aligned}
&PolicySeq : Nat \to Type\\
&PolicySeq\ n = Vect\ Policy\ n
\end{aligned}
$$

Given an initial state and a policy sequence, the computation of the corresponding sequence of controls is straightforward:

$$
\begin{aligned}
&ctrls : (x : X) \to PolicySeq\ n \to CtrlSeq\ x\ n\\
&ctrls\ x\ Nil = Nil\\
&ctrls\ x\ (p :: ps) = p\ x :: ctrls\ (step\ x\ (p\ x))\ ps
\end{aligned}
$$

The value of a policy sequence in terms of cumulated rewards is computed analogously to the value of a control sequence defined in section 3:

$$
\begin{aligned}
&Val : (x : X) \to PolicySeq\ n \to Float\\
&Val\ \{n = O\}\ x\ Nil = 0\\
&Val\ \{n = S\ m\}\ x\ (p :: ps) = reward\ x\ (p\ x)\ x' + Val\ x'\ ps\ \textbf{where}\\
&\quad x' : X\\
&\quad x' = step\ x\ (p\ x)
\end{aligned}
$$

But the notion of optimality for policy sequences is somewhat stronger: we say that a policy sequence *ps* is optimal if, for every initial state x, its value is at least as good as the value of every other policy sequence (of the same length of *ps*) for that x. Formally:

[5] In fact, for the sake of *nilIsOptCtrlSeq*, *reflexive_Float_lte* might just be a postulate.

$$
\begin{aligned}
&\mathit{OptPolicySeq} : (n : \mathit{Nat}) \to \mathit{PolicySeq}\ n \to \mathit{Type} \\
&\mathit{OptPolicySeq}\ n\ ps = (x : X) \to \\
&\qquad\qquad\qquad (ps' : \mathit{PolicySeq}\ n) \to \\
&\qquad\qquad\qquad \mathit{so}\ (\mathit{Val}\ x\ ps' \leqslant \mathit{Val}\ x\ ps)
\end{aligned}
$$

This notion of optimality is relevant because of the following lemma:

$$
\begin{aligned}
&\mathit{OptLemma} : (\mathit{ReflDecEq}\ X) \Rightarrow \\
&\qquad\qquad (n : \mathit{Nat}) \to \\
&\qquad\qquad (ps : \mathit{PolicySeq}\ n) \to \\
&\qquad\qquad (\mathit{ops} : \mathit{OptPolicySeq}\ n\ ps) \to \\
&\qquad\qquad (x : X) \to \\
&\qquad\qquad \mathit{OptCtrlSeq}\ (\mathit{ctrls}\ x\ ps)
\end{aligned}
$$

Let's consider *OptLemma* more closely. The type constraint $\mathit{ReflDecEq}\ X$ requires equality on X to be decidable and reflexive. This is the first example — in our formalization — of a type constraint expressed in terms of a type class.

OptLemma states that, for all X for which equality is decidable and reflexive, for all $n : \mathit{Nat}$ and $ps : \mathit{PolicySeq}\ n$, if ps is an optimal policy sequence, then, for every initial value x, $\mathit{ctrls}\ x\ ps$ is an optimal sequence of controls.

The quantification on x in the notion of optimality for policy sequences is essential: without a notion of optimality "for a given x", we would not be able to prove *OptLemma*.

Proving *OptLemma* means computing a value of type $\mathit{so}\ (\mathit{val}\ ys' \leqslant \mathit{val}\ (\mathit{ctrls}\ x\ ps))$ given a policy sequence ps, a proof that ps is optimal ops, an arbitrary state x and an arbitrary control sequence ys'. The proof is based on the the following lemma, using propositional equality:

$$
\begin{aligned}
&\mathit{valValLemma} : (x : X) \to \\
&\qquad\qquad (ps : \mathit{PolicySeq}\ n) \to \\
&\qquad\qquad \mathit{Val}\ x\ ps = \mathit{val}\ (\mathit{ctrls}\ x\ ps)
\end{aligned}
$$

valValLemma says that the value of a policy ps (as given by *Val*) is equal to the value of the corresponding sequence of controls as given by *val*. This is not really surprising: a semi-formal proof of the *val*- *Val* equivalence can be easily derived by induction on n. We have to show

$$\mathit{Val}\ x\ ps = \mathit{val}\ (\mathit{ctrls}\ x\ ps)$$

The base case ($n = O$ or, equivalently, $ps = \mathit{Nil}$) is trivial, since

$$\mathit{Val}\ x\ \mathit{Nil} = \mathit{val}\ (\mathit{ctrls}\ x\ \mathit{Nil})$$

follows from

$$
\begin{aligned}
&\mathit{Val}\ x\ \mathit{Nil} \\
&\quad = \quad \{\ \text{def. of}\ \mathit{Val}\ \} \\
&0
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{ \text{ def. of } val \ \} \\
&val\ Nil \\
&= \quad \{ \text{ def. of } ctrls \ \} \\
&val\ (ctrls\ x\ Nil)
\end{aligned}
$$

In the induction step, we have to show that

$$
\begin{aligned}
&Val\ x\ ps = val\ (ctrls\ x\ ps) \\
&\quad \Rightarrow \\
&Val\ x\ (p :: ps) = val\ (ctrls\ x\ (p :: ps))
\end{aligned}
$$

This can be readily done by applying the same definitions as above and the induction hypothesis:

$$
\begin{aligned}
&Val\ x\ (p :: ps) \\
&\quad = \quad \{ \text{ def. of } Val \text{ with } y = p\ x,\ x' = step\ x\ y \ \} \\
&reward\ x\ y\ x' + Val\ x'\ ps \\
&\quad = \quad \{ \text{ induction hypothesis } \} \\
&reward\ x\ y\ x' + val\ (ctrls\ x'\ ps) \\
&\quad = \quad \{ \text{ def. of } val \ \} \\
&val\ x\ (y :: (ctrls\ x'\ ps)) \\
&\quad = \quad \{ \text{ def. of } y,\ x',\ ctrls \ \} \\
&val\ x\ (ctrls\ x\ (p :: ps))
\end{aligned}
$$

The Idris-proof for *OptLemma* (not shown here) is a little bit more technical but the idea is very simple:

1. construct $ps' : PolicySeq\ n$ such that $ctrls\ x\ ps' = ys'$
2. apply optimality of ps to deduce $Val\ x\ ps' \leqslant Val\ x\ ps$
3. apply *valValLemma* to deduce that $val\ ys' \leqslant val\ (ctrls\ x\ ps)$

OptLemma ensures that optimal control sequences can be obtained from optimal sequences of policies. This is particularly useful because optimal policy sequences can be easily computed using Bellman's backwards induction algorithm. Before turning to the formalization of Bellman's principle and to a generic implementation of backwards induction, let us recall a trivial but important result: just as in the case of empty sequences of controls, empty policy sequences are optimal:

$$
\begin{aligned}
&nilIsOptPolicySeq : OptPolicySeq\ O\ Nil \\
&nilIsOptPolicySeq\ x\ ps' = reflexive\_Float\_lte\ 0
\end{aligned}
$$

5 Bellman's optimality principle and backwards induction

Bellman's principle of optimality [2] is based on the notion of optimal extensions of sequences of policies. Given a policy sequence ps, a policy p is an optimal extension of ps if the value of $p :: ps$ is at least as good as the value of $p' :: ps$ for arbitrary p'. Formally:

$$
\begin{aligned}
&\mathit{OptExt} : \mathit{PolicySeq}\ n \to \mathit{Policy} \to \mathit{Type} \\
&\mathit{OptExt}\ \mathit{ps}\ p = (p' : \mathit{Policy}) \to \\
&\qquad\qquad (x : X) \to \\
&\qquad\qquad \mathit{so}\ (\mathit{Val}\ x\ (p' :: \mathit{ps}) \leqslant \mathit{Val}\ x\ (p :: \mathit{ps}))
\end{aligned}
$$

Bellman's principle of optimality says that if we are given a policy sequence ps of length n, a proof of optimality for ps and an optimal extension p of ps then we can construct an optimal policy sequence of length $n + 1$ simply by consing p with ps:

$$
\begin{aligned}
&\mathit{Bellman} : (\mathit{ps} : \mathit{PolicySeq}\ n) \to \\
&\qquad\qquad \mathit{OptPolicySeq}\ n\ \mathit{ps} \to \\
&\qquad\qquad (p : \mathit{Policy}) \to \\
&\qquad\qquad \mathit{OptExt}\ \mathit{ps}\ p \to \\
&\qquad\qquad \mathit{OptPolicySeq}\ (S\ n)\ (p :: \mathit{ps})
\end{aligned}
$$

To prove *Bellman*, one has to show that, if $ps : \mathit{PolicySeq}\ n$ is optimal and p is an optimal extension of ps then $p :: ps$ is optimal, that is

$$
\mathit{Val}\ x\ (p' :: \mathit{ps}') \leqslant \mathit{Val}\ x\ (p :: \mathit{ps})
$$

for arbitrary $x : X$ and $(p' :: ps') : \mathit{PolicySeq}\ (S\ \ n)$. A semi-formal proof is straightforward:

$$
\begin{aligned}
&\mathit{Val}\ x\ (p' :: \mathit{ps}') \\
&\quad = \quad \{\ \text{def. of } \mathit{Val} \text{ with } x' = \mathit{step}\ x\ (p'\ x)\ \} \\
&\mathit{reward}\ x\ (p'\ x)\ x' + \mathit{Val}\ x'\ \mathit{ps}' \\
&\quad \leqslant \quad \{\ \text{optimality of } \mathit{ps}\text{, monotonicity of } +\ \} \\
&\mathit{reward}\ x\ (p'\ x)\ x' + \mathit{Val}\ x'\ \mathit{ps} \\
&\quad = \quad \{\ \text{def. of } \mathit{Val}\ \} \\
&\mathit{Val}\ x\ (p' :: \mathit{ps}) \\
&\quad \leqslant \quad \{\ p \text{ is an optimal extension of } \mathit{ps}\ \} \\
&\mathit{Val}\ x\ (p :: \mathit{ps})
\end{aligned}
$$

We can easily turn the semi-formal proof into a program and ask Idris to type check Bellman's principle:

$$
\begin{aligned}
&\mathit{Bellman}\ \{n\}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep} = \mathit{opps}\ \textbf{where} \\
&\quad \mathit{opps} : \mathit{OptPolicySeq}\ (S\ n)\ (p :: \mathit{ps}) \\
&\quad \mathit{opps}\ x\ (p' :: \mathit{ps}') = \mathit{transitive\_Float\_lte}\ \mathit{step2}\ \mathit{step3}\ \textbf{where} \\
&\qquad \mathit{step1}\ :\ \mathit{so}\ (\mathit{Val}\ (\mathit{step}\ x\ (p'\ x))\ \mathit{ps}' \leqslant \mathit{Val}\ (\mathit{step}\ x\ (p'\ x))\ \mathit{ps}) \\
&\qquad \mathit{step1} = \mathit{ops}\ (\mathit{step}\ x\ (p'\ x))\ \mathit{ps}' \\
&\qquad \mathit{step2}\ :\ \mathit{so}\ (\mathit{Val}\ x\ (p' :: \mathit{ps}') \leqslant \mathit{Val}\ x\ (p' :: \mathit{ps})) \\
&\qquad \mathit{step2} = \mathit{monotone\_Float\_plus\_lte} \\
&\qquad\qquad\qquad (\mathit{reward}\ x\ (p'\ x)\ (\mathit{step}\ x\ (p'\ x))) \\
&\qquad\qquad\qquad \mathit{step1} \\
&\qquad \mathit{step3}\ :\ \mathit{so}\ (\mathit{Val}\ x\ (p' :: \mathit{ps}) \leqslant \mathit{Val}\ x\ (p :: \mathit{ps})) \\
&\qquad \mathit{step3} = \mathit{oep}\ p'\ x
\end{aligned}
$$

Let's assume that we can implement a function $optExt : PolicySeq\ n \to Policy$ that computes optimal extensions, that is, *optExt* fulfills

$$OptExtLemma : (ps : PolicySeq\ n) \to OptExt\ ps\ (optExt\ ps)$$

The specification says that for all ps, $optExt\ ps$ is an optimal extension of ps. Recall that empty policy sequences are optimal: *nilIsOptPolicySeq*. Thus, one can use *optExt* to compute optimal policy sequences backwards (Bellman, 1957):

$$\begin{aligned}
&backwardsInduction : (n : Nat) \to PolicySeq\ n\\
&backwardsInduction\ O = Nil\\
&backwardsInduction\ (S\ n) = ((optExt\ ps) :: ps)\ \textbf{where}\\
&\quad ps : PolicySeq\ n\\
&\quad ps = backwardsInduction\ n
\end{aligned}$$

This implementation of backwards induction is not particularly efficient[6]. But it can be easily proved to be correct that is, to fulfill the specification:

$$\begin{aligned}
&BackwardsInductionLemma : (n : Nat) \to\\
&\quad OptPolicySeq\ n\ (backwardsInduction\ n)
\end{aligned}$$

For $n = O$, *BackwardsInductionLemma* certainly holds because of empty policy sequences are optimal:

$$BackwardsInductionLemma\ O = nilIsOptPolicySeq$$

The induction step can be proved using Bellman's principle of optimality. All one has to do is to construct the four arguments needed to apply *Bellman*:

$$\begin{aligned}
&BackwardsInductionLemma\ (S\ m) =\\
&\quad Bellman\ ps\ ops\ p\ oep\ \textbf{where}\\
&\qquad ps\ \ : PolicySeq\ m\\
&\qquad ps\ \ = backwardsInduction\ m\\
&\qquad ops\ : OptPolicySeq\ m\ ps\\
&\qquad ops = BackwardsInductionLemma\ m\\
&\qquad p\ \ \ : Policy\\
&\qquad p\ \ \ = optExt\ ps\\
&\qquad oep\ : OptExt\ ps\ p\\
&\qquad oep = OptExtLemma\ ps
\end{aligned}$$

Notice the usage of the induction hypothesis *BackwardsInductionLemma* m in the construction of *ops*, the proof that *ps* is optimal.

[6] As it turns out, *backwardsInduction* executes in exponential time in n. If the state space X is finite, it is easy to derive an implementation of backwards induction which executes in linear time in n.

6 Optimal extensions of policy sequences

It is easy to implement a function that computes the optimal extension of a policy sequence ps if one can find, for every $x : X$, an optimal control. This is a $y : Y\ x$ such that for all $y' : Y\ x$

$$\begin{aligned}&reward\ x\ y'\ (step\ x\ y') + Val\ (step\ x\ y')\ ps\\ &\leqslant\\ &reward\ x\ y\ (step\ x\ y) + Val\ (step\ x\ y)\ ps\end{aligned}$$

If $Y\ x$ is finite, it is possible to find an optimal control with a finite number of comparisons. In general, we can compute an optimal extension of a policy sequence if we have functions

$$\begin{aligned}&max : (x : X) \to (Y\ x \to Float) \to Float\\ &argmax : (x : X) \to (Y\ x \to Float) \to Y\ x\end{aligned}$$

which fulfill the specifications

$$\begin{aligned}&MaxSpec : Type\\ &MaxSpec = (x : X) \to\\ &\quad (f : Y\ x \to Float) \to\\ &\quad (y : Y\ x) \to\\ &\quad so\ (f\ y \leqslant max\ x\ f)\\ &ArgmaxSpec : Type\\ &ArgmaxSpec = (x : X) \to\\ &\quad (f : Y\ x \to Float) \to\\ &\quad so\ (f\ (argmax\ x\ f) \equiv max\ x\ f)\end{aligned}$$

In this case

$$\begin{aligned}&optExt : PolicySeq\ n \to Policy\\ &optExt\ ps\ x = argmax\ x\ f\ \textbf{where}\\ &\quad f : Y\ x \to Float\\ &\quad f\ y = reward\ x\ y\ x' + Val\ x'\ ps\ \textbf{where}\\ &\qquad x' : X\\ &\qquad x' = step\ x\ y\end{aligned}$$

fulfills *OptExtLemma*. This can be seen easily by equational reasoning. We have to show that

$$Val\ x\ (p' :: ps) \leqslant Val\ x\ ((optExt\ ps) :: ps)$$

for arbitrary $p' : Policy$. We have:

$$\begin{aligned}&Val\ x\ (p' :: ps)\\ &\quad = \quad \{\ \text{def. of } Val \text{ with } x' = step\ x\ (p'\ x)\ \}\\ &reward\ x\ (p'\ x)\ x' + Val\ x'\ ps\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ def. of } f \ \} \\
&f\ (p'\ x) \\
&\leqslant \quad \{ \ \mathit{MaxSpec} \ \} \\
&\mathit{max}\ x\ f \\
&= \quad \{ \ \mathit{ArgmaxSpec} \ \} \\
&f\ (\mathit{argmax}\ x\ f) \\
&= \quad \{ \text{ def. of } \mathit{optExt} \ \} \\
&f\ ((\mathit{optExt}\ ps)\ x) \\
&= \quad \{ \text{ def. of } f \text{ with } x' = \mathit{step}\ x\ (\mathit{optExt}\ ps\ x) \ \} \\
&\mathit{reward}\ x\ (\mathit{optExt}\ ps\ x)\ x' + \mathit{Val}\ x'\ ps \\
&= \quad \{ \text{ def. of } \mathit{Val} \ \} \\
&\mathit{Val}\ x\ ((\mathit{optExt}\ ps) :: ps)
\end{aligned}
$$

As for the case of *Bellman* and *BackwardsInductionLemma*, it is straightforward to rewrite this proof in Idris and machine check its correctness.

The existence of *max* and *argmax* is a condition which is simple to formulate and sufficient for computing optimal extensions, but it is also stronger than necessary: it requires computing the maximum of *any* function of type $Y\ x \to \mathit{Float}$, which is in many cases intractable. In the optimal extension algorithm we are however only interested in maximizing functions having the specific form determined by *reward*, *Val* and *step*. Weaker conditions can be formulated by taking this into account and restricting the class of functions for which *max* and *argmax* need to be implemented. For example, if *reward* and *step* are linear, so are *Val* and f, and we could replace the general optimization required by *max* and *argmax* with the simpler problem of linear optimization.

7 Summary

For time-independent, deterministic sequential decision problems, we have formalized the notions of state space, available controls, transition and reward functions (the context), optimal control and policy sequences and Bellman's principle of optimality. Provided we are given a function *optExt* which fulfills *OptExtLemma*, we have derived a generic implementation of backwards induction and proved that such implementation is correct, that is, that it computes policy sequences which are optimal. From an optimal policy sequence, an optimal control sequence can be recovered for arbitrary initial states through *ctrls*. In the last section we have specified sufficient conditions for *optExt* to be implementable, under which we have derived a generic implementation of *optExt*.

7.1 Outlook

Our final aim is to extend the formalization presented in this article to sequential decision problems which can be time-dependent and for which *step* can be deterministic, non-deterministic, stochastic or, more generally, monadic.

We present such extension in detail in a follow-up paper. Here, we just hint at the two difficulties that have to be faced in deriving such a generic extension. A

first difficulty is that one has to find a way of extending the formalization to the case in which the target of *step* is a generic M-structure on X, for a monad M. Obviously, a monadic *step* function naturally induces an M-structure on *Float* via the *reward* function. As it turns out, one can apply the ideas developed in [9] for monadic dynamical systems to derive a natural and fully generic approach for iterating monadic step functions and to account for M-structures of rewards.

Another difficulty is that when X, and therefore Y, do depend on the number of steps, one cannot assume, in general, that admissible controls exist for every state. To see why this is the case, let us turn back to the cylinder problem and consider a case in which the state space X is not constant, but depends on the number of steps $t{:}Nat$. For concreteness, consider a case in which all five columns – a, b, c, d and e – are valid for $t < 3$ and $t > 3$ but, at $t = 3$, only column e is a valid state, see Figure 2, left.

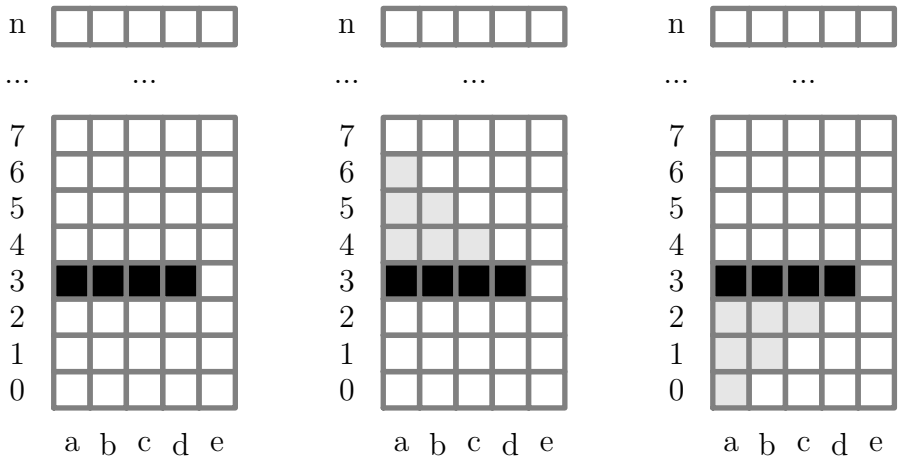

Fig. 2. Cylinder problem with time-dependent state space. Non-valid columns ($t = 3$, black), non-reachable states (middle, light grey) and non-viable states (right, light grey).

The consequences of a reduced state space at time 3 are twofold. First, certain states at $t > 3$ become *unreachable*. These are the states which are "shadowed" by the invalid columns at time 3. They are represented in light grey in the middle of Figure 2.

Second, certain states become *non-viable*: these are states at $t < 3$ from which we can take at most two steps. They are represented, also in light grey, on the right of Figure 2. Not accounting for unreachable states can be inefficient because optimal extensions are computed for states which effectively cannot be realized by any sequence of controls. This is particularly true if the set of options associated to such unreachable states is big.

Viability, however, has deeper implications. In Figure 2 right, the initial state a is essentialy different from b, c, d or e: there is simply no trajectory of length >2 starting in a. Similarly a and b cannot be initial states of trajectories of length >1 starting at $t = 1$ and so on.

Thus, extending the generic framework presented in this paper to the time-dependent case requires, on the one hand, formalizing the notion of viability generically. This becomes particularly interesting in the general case of monadic *step* functions. On the other hand, it requires accounting for viability constraints both in the domain of policy functions and in their range — the set of controls that can be selected at a given time. In computing optimal extensions for policy sequences of length n at a time t, only admissible controls leading to states from which (at least) $n - t$ further steps can be done are suitable candidates. Controls leading to "dead-ends" have to be carefully identified and rejected.

Acknowledgments

The work presented in this paper heavily relies on free software, among others on hugs, GHC, vi, the GCC compiler, Emacs, LaTeX and on the FreeBSD and Debian / GNU Linux operating systems. It is our pleasure to thank all developers of these excellent products. We also thank the anonymous referees for their helpful comments.

References

1. A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. Learning and sequential decision making. Technical report, COINS Technical Report No. 89-95, 1989.
2. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
3. Y. Bertot and P Castran. *Interactive theorem proving and program development, Coq'art: the calculus of inductive constructions*. Texts in Theoretical Computer Science. Springer, 2004.
4. P. Bertsekas, D. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Mass., 1995.
5. Richard Bird and Oege de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, 1997.
6. Edwin Brady. Programming in Idris : a tutorial, 2013.
7. O. de Moor. A generic program for sequential decision processes. In *PLILPS '95 Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 1–23. Springer, 1995.
8. Cormen T. H., Stein C., Rivest R. L., and Leiserson C. E. *Introduction to algorithms*. McGraw-Hill, second edition, 2001.
9. Cezar Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin, 2009.
10. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.
11. E. M. Reingold, Nievergelt J., and Deo N. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, 1977.
12. Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, pages 278—-293, 2008.