SEQUENTIAL DECISION PROBLEMS, DEPENDENT TYPES AND GENERIC SOLUTIONS

NICOLA BOTTA, PATRIK JANSSON, CEZAR IONESCU, DAVID R. CHRISTIANSEN, AND EDWIN BRADY

Abstract. We present a computer-checked generic implementation for solving finite-horizon sequential decision problems. This is a wide class of problems, including inter-temporal optimizations, knapsack, optimal bracketing, scheduling, etc. The implementation can handle time-step dependent control and state spaces, and monadic representations of uncertainty (such as stochastic, non-deterministic, fuzzy, or combinations thereof). This level of genericity is achievable in a programming language with dependent types (we have used both Idris and Agda). Dependent types are also the means that allow us to obtain a formalization and computer-checked proof of the central component of our implementation: Bellman's principle of optimality and the associated backwards induction algorithm. The formalization clarifies certain aspects of backwards induction and, by making explicit notions such as viability and reachability, can serve as a starting point for a theory of controllability of monadic dynamical systems, commonly encountered in, e.g., climate impact research.

1. Introduction

In this paper we extend a previous formalization of *time-independent*, *deterministic* sequential decision problems [BIB13] to general sequential decision problems.

These are decision problems in which the state space and the control space can depend on the current step number 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, more generally, a monadic structure of states, as presented by Ionescu [Ion09].

Sequential decision problems for a finite number of steps, often called finite horizon sequential decision problems, are in principle well understood. They can be solved effectively by Bellman's backwards induction algorithm [Bel57].

There is, however, a difficulty. Sequential decision problems are typically introduced by examples [CSRL01, Ber95, SG98]. Single examples are analyzed and dissected and ad-hoc implementations of backwards induction are derived for the specific examples. Important properties of sequential decision problems and solution algorithms, such as overlapping sub-problems and optimal substructures, are discussed but not formalized.

In front of a particular instance of a sequential decision problem, be that a variant of knapsack, optimal bracketing, inter-temporal optimization of social welfare functions or more specific applications, scientists face two problems. First, they have to (re-)implement a

Received by the editors draft, August 2014.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

suitable solution algorithm — backwards induction or non-linear optimization, for example — for their particular problem. Second they have to show that their implementation actually delivers optimal solutions for their problem. For most practitioners, at least the second task is often insurmountable.

If generic, machine-checkable implementations of these algorithms were available, both steps could be done by simple instantiation. This would not only save time but also yield solutions which are provably correct. Expert implementors might still want to re-implement problem-specific solution algorithms, e.g., to exploit some known properties of the particular problem at hand. But they would at least be able to test their solutions against provably correct ones.

In this work, we propose a formalization of general sequential decision problems. For these problems, we implement a generic version of Bellman's backwards induction and derive a machine-checkable proof that the proposed implementation is correct.

Our approach is similar in spirit to that proposed by de Moor [dM95]: the focus is on generic programming over an abstract context, but we take this further by implementing the algorithm, the specification and the proofs in one common framework (Idris). For a discussion of the differences and of the similarities we refer the reader to the previous paper on time-independent deterministic problems [BIB13].

We rely on dependent types — types that are allowed to "depend" on values [Bra13] — to implement generic algorithms and to encode (and prove) properties of such algorithms. As in the previous paper [BIB13], we present our formalization in Idris but we have derived an equivalent formulation in Agda.

We present our extension in two steps. First, we generalize time-independent, deterministic decision problems to the case in which the state and the control spaces can depend on time but the transition function is still deterministic. Then, we extend this case to the general one of monadic transition functions. As it turns out, neither extension is trivial: the requirement of machine-checkable correctness brings to the light notions and assumptions which, in informal and semi-formal derivations are often swept under the rug.

In particular, the extension to the time-dependent case has lead us to formalize the notions of *viability* and *reachability* of states. For the deterministic case these notions are more or less straightforward. But they become more interesting when non-deterministic and stochastic transition functions are considered (as outlined in section 4).

We believe that these notions would be a good starting point for building a theory of controllability for the kind of dynamical systems commonly encountered in climate impact research. These were the systems originally studied in Ionescu's dissertation [Ion09] and the monadic case is an extension of the theory presented there for dynamical systems.

In section 2 we summarize the results for the time-independent, deterministic case and use this as the starting point for the two extensions discussed in sections 3 and 4, respectively.

2. Time-independent, deterministic problems

In a previous paper [BIB13], we presented a formalization of time-independent, deterministic sequential decision problems. For this class of problems, we introduced an abstract context and derived a generic, machine-checkable implementation of backwards induction.

In this section we recall the context and the main results from that paper [BIB13]. There, we illustrated time-independent, deterministic sequential decision problems using

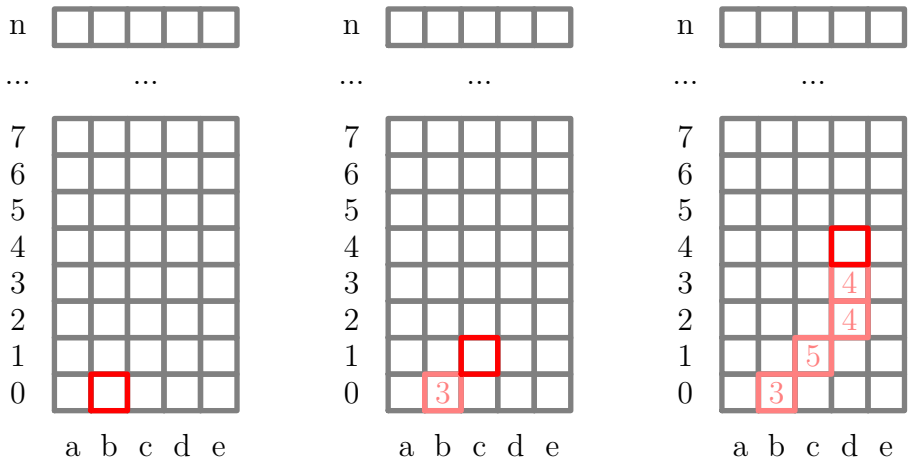

Figure 1: Possible evolutions for the "cylinder" problem. Initial state (b, left), state and reward after one step (c and 3, middle) and four steps trajectory and rewards (right).

a simplified version of the "cylinder" example originally proposed by Reingold, Nievergelt and Deo [RND77] and extensively studied by Bird and de Moor [BdM97]. We use the same example here:

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 controls sometimes called "options" or "actions"): move ahead (control A) or move to the right (control 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 a control, the decision maker enters a new state. For instance, selecting R in b brings him from b to c, see Figure 1. 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 control R.

The challenge for the decision maker is to make a finite number of steps, say n, by selecting controls 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. In this example, the decision maker has so far collected a total reward of 16 by selecting controls according to the sequence $[R, R, A, A]$: R in the first and in the second steps, A in the third and in the fourth steps.

In this problem, the set of possible states $X = \{a, b, c, d, e\}$ is constant for all steps and the controls available in a state only depend on that state. The problem is an instance of a particular class of problems called time-independent, deterministic sequential decision problems. In our previous paper [BIB13] we characterized this class in terms of four assumptions:

(1) The state space does not depend on the current number of steps.
(2) The control space in a given state only depends on that state.
(3) At each step, the new state depends on the current state and on the selected control via a known deterministic function.
(4) At each step, the reward is a known function the current state, of the selected control and of the new state.

The results obtained [BIB13] for this class of sequential decision problems can be summarized as follows. The problems can be formalized in terms of a context containing states X and controls Y from each state:

```
X      : Type
Y      : X → Type
step   : (x : X) → Y x → X
reward : (x : X) → Y x → X → Float
```

and of the notions of control sequence *CtrlSeq x n* (from a starting state $x : X$ and for $n : Nat$ steps), value of control sequences and optimality of control sequences:

```
data CtrlSeq : X → Nat → Type where
  Nil : CtrlSeq x 0
  (::) : (y : Y x) → CtrlSeq (step x y) n → CtrlSeq x (S n)

val : CtrlSeq x n → Float
val      {n = Z}   _        = 0
val {x} {n = S m} (y :: ys) = reward x y (step x y) + val ys

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

In the above, the arguments x and n to *CtrlSeq* in the types of *val* and *OptCtrlSeq* occur free. In Idris (as in Haskell), this means that they will be automatically inserted as implicit arguments. In the definitions of *val* and *OptCtrlSeq*, these implicit arguments are brought into the local scope by adding them to the pattern match surrounded by curly braces. We also use the function *so* : *Bool* → *Type* for translating between Booleans and types (the only constructor is *oh* : *so True*).

We have shown that one can compute optimal control sequences from optimal policy sequences. These are policy vectors that maximize the value function *Val* for every state:

```
Policy : Type
Policy = (x : X) → Y x

PolicySeq : Nat → Type
PolicySeq n = Vect n Policy

Val : (x : X) → PolicySeq n → Float
Val {n = Z}   _ _        = 0
Val {n = S m} x (p :: ps) = reward x (p x) x' + Val x' ps where
  x' : X
  x' = step x (p x)

OptPolicySeq : (n : Nat) → PolicySeq n → Type
OptPolicySeq n ps = (x : X) → (ps' : PolicySeq n) → so (Val x ps' ⩽ Val x ps)
```

We have expressed Bellman's principle of optimality [Bel57] in terms of the notion of optimal extensions of policy sequences

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

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

and implemented a machine-checkable proof of *Bellman*. Another machine-checkable proof guarantees that, if one can implement a function *optExt* that computes an optimal extension of arbitrary policy sequences

$$\mathit{OptExtLemma} \;:\; (\mathit{ps} \;:\; \mathit{PolicySeq}\ n) \;\to\; \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})$$

then

$$\begin{aligned}
&\mathit{backwardsInduction} \;:\; (n \;:\; \mathit{Nat}) \;\to\; \mathit{PolicySeq}\ n\\
&\mathit{backwardsInduction}\ Z \quad\;\; = \mathit{Nil}\\
&\mathit{backwardsInduction}\ (S\ n) = \mathit{optExt}\ \mathit{ps} :: \mathit{backwardsInduction}\ n
\end{aligned}$$

yields optimal policy sequences (and, thus, optimal control sequences) of arbitrary length:

$$\mathit{BackwardsInductionLemma} \;:\; (n \;:\; \mathit{Nat}) \;\to\; \mathit{OptPolicySeq}\ n\ (\mathit{backwardsInduction}\ n)$$

In our previous paper [BIB13], we have shown that it is easy to implement a function that computes the optimal extension of an arbitrary policy sequence if one can implement

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

which fulfill the specifications

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

When $Y\ x$ is finite, such *max* and *argmax* can always be implemented in a finite number of comparisons.

3. Time-dependent state spaces

The results summarized in the previous section are valid under one implicit assumption: that one can construct control sequences of arbitrary length from arbitrary initial states. A sufficient (but not necessary) condition for this is that, for all $x \;:\; X$, the control space $Y\ x$ is not empty. As we shall see in a moment, this assumption is too strong and needs to be refined.

Consider, again, the cylinder problem. Assume that at a given time, only certain columns are *valid*. For instance, for $t \neq 3$ and $t \neq 5$ all states a through e are valid but at time 3 only e is valid and at time 6 only a, b and c are valid, see figure 2. Similarly, allow the controls available in a given state to depend both on that state and on time. For instance, from state b at step 0 one might be able to move ahead or to the right. But at step 6 and from the same state, one might be only able to move only to the left.

We can easily formalize the context for the time-dependent case by adding an extra *Nat* argument to the declarations of X and Y and extending the transition and the reward functions accordingly:

$$
\begin{array}{ll}
X & : (t : Nat) \to Type \\
Y & : (t : Nat) \to X\ t \to Type \\
step & : (t : Nat) \to (x : X\ t) \to Y\ t\ x \to X\ (S\ t) \\
reward & : (t : Nat) \to (x : X\ t) \to Y\ t\ x \to X\ (S\ t) \to Float
\end{array}
$$

In general we will be able to construct control sequences of a given length from a given initial states only if X, Y and *step* satisfy certain compatibility conditions. For example, assuming that the decision maker can move to the left, go ahead or move to the right as described in the previous section, there will be no sequence of more than two controls starting from a, see figure 2 left. At step t, there might be states which are valid but from which only $m < n - t$ steps can be done, see figure 2 middle. Conversely, there might be states which are valid but which cannot be reached from any initial state, see figure 2 right.

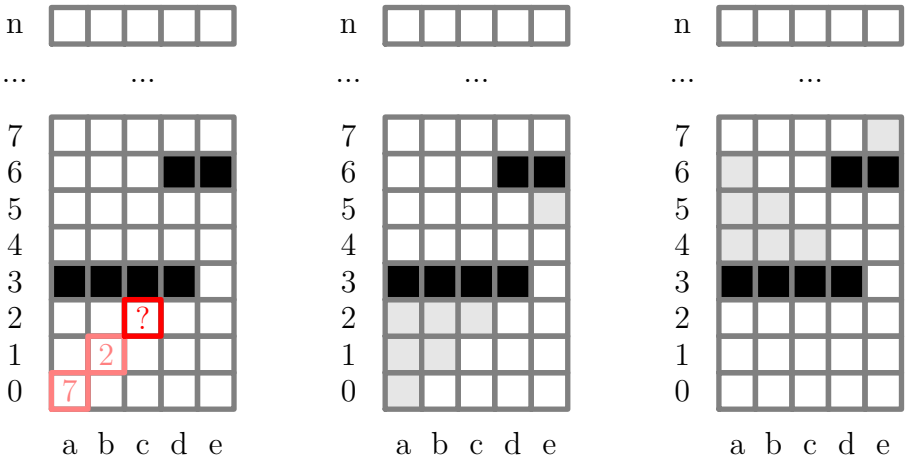

Figure 2: Possible evolutions for the "cylinder" problem. Initial state (b, left), state and reward after one step (c and 3, middle) and four steps trajectory and rewards (right).

3.1. **Viability.** The time dependent case makes it clear that, in general, we cannot assume to be able to construct control sequences of arbitrary length from arbitrary initial states. For a given number of steps n, we must, at the very least, be able to distinguish between initial states from which n steps can follow and initial states from which only $m < n$ steps can follow. Moreover, in building control sequences from initial states from which n steps can actually be made, we may only select controls that bring us to states from which $n-1$ steps can be made. In the example of figure 2 and with b as initial state, for instance, the only control that can be put on the top of a control sequence of length greater than 2 is R. Moves ahead or to the left would lead to dead ends.

We use the term *viability* to refer to the conditions that X, Y and *step* have to satisfy for a sequence of controls of length n starting in $x : X\ t$ to exist. More formally, we say that every state is viable for zero steps (*viableSpec0*) and that a state $x : X\ t$ is viable for $S\ n$ steps if and only if there exists a command in $Y\ t\ x$ which, via *step*, brings x into a state which is viable n steps (*viableSpec1* and *viableSpec2*):

$$
\begin{array}{ll}
viable & : (n : Nat) \to X\ t \to Bool \\
viableSpec0 & : (x : X\ t) \to Viable\ Z\ x
\end{array}
$$

```
viableSpec1 : (x : X t) → Viable (S n) x → YV t n x
viableSpec2 : (x : X t) → YV t n x → Viable (S n) x
```

In the above specifications we have introduced *Viable n x* as a shorthand for *so* (*viable n x*). In *viableSpec1* and *viableSpec2* we use subsets implemented as dependent pairs:

```
YV : (t : Nat) → (n : Nat) → X t → Type
YV t n x = (y : Y t x ** Viable n (step t x y))
```

These are pairs in which the type of the second element can depend on the value of the first one. The notation $p : (a : A ** P\ a)$ represents a pair in which the type $(P\ a)$ of the second element can refer to the value (a) of the first element, giving a kind of existential quantification [Bra13]. The projection functions are

```
outl : (a : A ** P a) → A
outr : (p : (a : A ** P a)) → P (outl p)
```

Thus, in general $(a : A ** P\ a)$ effectively represents the subset of A whose elements fulfill P and our case $YV\ t\ n\ x$ is the subset of controls available in x at time t which lead to next states which are viable n steps.

The declarations of *viableSpec0*, *viableSpec1* and *viableSpec2* are added to the context. In implementing an instance of a specific sequential decision problem, clients are required to define X, Y, *step*, *reward* and the *viable* predicate for that problem. In doing so, they have to prove (or postulate) that their definitions satisfy the the above specifications.

3.2. **Control sequences.** With the notion of viability in place, we can readily extend the notion of control sequences of section 2 to the time-dependent case:

```
data CtrlSeq : (x : X t) → (n : Nat) → Type where
  Nil : CtrlSeq {t} x Z
  (::) : {x : X t} → (yv : YV t n x) →
         CtrlSeq (step t x (outl yv)) n → CtrlSeq x (S n)
```

Notice that now the constructor :: (for constructing control sequences of length $S\ n$) can only be applied to those (implicit !) $x : X\ t$ for which there exists a control $y = outl\ yv : Y\ t\ x$ such that the new state *step t x y* is viable n steps. The specification *viableSpec2* ensures us that, in this case, x is viable $S\ n$ steps.

The extension of *val*, *OptCtrlSeq* and the proof of optimality of empty sequences of controls are, as one would expect, straightforward:

```
val : (x : X t) → (n : Nat) → CtrlSeq x n → Float
val _ Z _ = 0
val {t} x (S n) (yv :: yvs) = reward t x y x' + val x' n yvs where
  y : Y t x; y = outl yv;
  x' : X (S t); x' = step t x y

OptCtrlSeq : (x : X t) → (n : Nat) → CtrlSeq x n → Type
OptCtrlSeq x n ys = (ys' : CtrlSeq x n) → so (val x n ys' ⩽ val x n ys)

nilIsOptCtrlSeq : (x : X t) → OptCtrlSeq x Z Nil
nilIsOptCtrlSeq x Nil = reflexive_Float_lte 0
```

3.3. **Reachability, policy sequences.** In the time-independent case, policies are functions of type $(x : X) \to Y\ x$ and policy sequences are vectors of elements of that type. Given a policy sequence ps and an initial state x, one can construct its corresponding sequence of controls by *ctrls x ps*:

```
ctrls : PolicySeq n → (x : X) → CtrlSeq x n
ctrls Nil       x = Nil
ctrls (p :: ps) x = p x :: ctrl ps (step x (p x))
```

Thus $p\ x$ is of type $Y\ x$ which is, in turn, the type of the first (explicit) argument of the "cons" constructor of *CtrlSeq x n*, see section 2.

A seen above, in the time-dependent case the "cons" constructor of *CtrlSeq x n* takes as first argument dependent pairs of type $YV\ t\ n\ x$. For sequences of controls to be construable from policy sequences, policies have to return, in the time-dependent case, values of this type. Thus, what we want to formalize in the time-dependent case is the notion of a correspondence between states and sets of controls that, at a given time t, allows us to make a given number of steps n. Because of *viableSpec1* we know that such controls exist for a given $x : X\ t$ if and only if it is viable at least n steps. We use such a requirement to restrict the domain of policies

```
Policy : Nat → Nat → Type
Policy t Z     = ()
Policy t (S n) = (x : X t) → Viable (S n) x → YV t n x
```

Let us go back to the right-hand side of figure 2. At a given time there might be states which are valid but which cannot be reached. It could be a waste of computational resources to consider such states, e.g., when constructing optimal extensions inside a backwards induction step.

We can compute optimal policy sequences more efficiently if we restrict the domain of our policies to those states which can actually be reached from the initial states. We can do this by introducing the notion of *reachability*. We say that every initial state is reachable (*reachableSpec0*) and that if a state $x : X\ t$ is reachable, then every control $y : Y\ t\ x$ leads, via *step*, to a reachable state in $X\ (S\ t)$ (see *reachableSpec1*). Conversely, if a state $x' : X\ (S\ t)$ is reachable then there exist a state $x : X\ t$ and a control $y : Y\ t\ x$ such that x is reachable and x' is equal to *step t x y* (see *reachableSpec2*):

```
reachable      : X t → Bool
reachableSpec0 : (x : X Z)      → Reachable x
reachableSpec1 : (x : X t)      → Reachable x →
                 (y : Y t x)    → Reachable (step t x y)
reachableSpec2 : (x' : X (S t)) → Reachable x' →
                 (x : X t ** (Reachable x,
                              (y : Y t x ** x' = step t x y)))
```

As for viability, we have introduced *Reachable x* as a shorthand for *so* (*reachable x*) in the specification of *reachable*. We can now apply reachability to refine the notion of policy

```
Policy : Nat → Nat → Type
Policy t Z     = ()
Policy t (S n) = (x : X t) → Reachable x → Viable (S n) x → YV t n x
```

and policy sequences:

```
data PolicySeq : Nat → Nat → Type where
  Nil : PolicySeq t Z
  (::) : Policy t (S n) → PolicySeq (S t) n → PolicySeq t (S n)
```

In contrast to the time-independent case, *PolicySeq* now takes an additional *Nat* argument. This represents a discrete "time" coordinate or step number. The previous one-index idiom $(S\ n)\ \cdots\ n\ \cdots\ (S\ n)$ becomes a two-index idiom: $t\ (S\ n)\ \cdots\ (S\ t)\ n\ \cdots\ t\ (S\ n)$. The value function *Val* maximized by optimal policy sequences is:

```
Val : (t : Nat) → (n : Nat) →
      (x : X t) → (r : Reachable x) → (v : Viable n x) →
      PolicySeq t n → Float
Val _ Z _ _ _ _ = 0
Val t (S n) x r v (p :: ps) = reward t x y x' + Val (S t) n x' r' v' ps where
  y : Y t x;                        y  = outl (p x r v)
  x' : X (S t);                     x' = step t x y
  r' : Reachable {t = S t} x';      r' = reachableSpec1 x r y
  v' : Viable      {t = S t} n x'; v' = outr (p x r v)
```

3.4. **The full framework.** With these notions of viability, control sequence, reachability, policy and policy sequence, the previous [BIB13] formal framework for time-independent sequential decision problems can be easily extended to the time-dependent case.

The notions of optimality of policy sequences, optimal extension of policy sequences, Bellman's principle of optimality, the generic implementation of backwards induction and its machine-checkable correctness can all be derived almost automatically from the time-independent case.

We do not present the complete framework here. To give an idea of the differences between the time-dependent and the time-independent cases, we compare the proofs of Bellman's principle of optimality.

Consider, first, the time-independent case. As explained in section 2, Bellman's principle of optimality can be expressed by

```
Bellman : (ps : PolicySeq n) → OptPolicySeq n ps →
          (p : Policy) → OptExt ps p →
          OptPolicySeq (S n) (p :: ps)
```

This says that if ps is an optimal policy sequence of length n and p is an optimal extension of ps then $p :: ps$ is an optimal policy sequence of length $S\ n$.

Let's now turn to Bellman's principle for the time-dependent case. We need as additional argument the "time step" t and we make the number of steps n explicit. The other arguments are, as in the time-independent case, a policy sequence ps, a proof of optimality of ps, a policy p and a proof that p is an optimal extension of ps:

```
Bellman : (t : Nat) → (n : Nat) →
          (ps : PolicySeq (S t) n) → OptPolicySeq (S t) n ps →
          (p : Policy t (S n)) → OptExtension t n ps p →
          OptPolicySeq t (S n) (p :: ps)
```

The result is a proof of optimality of $p :: ps$. Notice that the types of the last 4 arguments of *Bellman* and the type of its result now depend on t.

As discussed in [BIB13], a proof of *Bellman* can be derived easily. According to the notion of optimality for policy sequences of section 2, one has to show that

$$Val\ t\ (S\ n)\ x\ r\ v\ (p' :: ps') \leqslant Val\ t\ (S\ n)\ x\ r\ v\ (p :: ps)$$

for arbitrary $x : X$ and $(p' :: ps') : PolicySeq\ t\ (S\ n)$. This is straightforward. Let

$$\begin{array}{ll} y = outl\ (p'\ x\ r\ v); & x' = step\ t\ x\ y; \\ r' = reachableSpec1\ x\ r\ y; & v' = outr\ (p'\ x\ r\ v); \end{array}$$

then

$$\begin{array}{l} Val\ t\ (S\ n)\ x\ r\ v\ (p' :: ps') \\ \quad = \quad \{\ \text{def. of } Val\ \} \\ reward\ t\ x\ y\ x' + Val\ (S\ t)\ n\ x'\ r'\ v'\ ps' \\ \quad \leqslant \quad \{\ \text{optimality of } ps\text{, monotonicity of } +\ \} \\ reward\ t\ x\ y\ x' + Val\ (S\ t)\ n\ x'\ r'\ v'\ ps \\ \quad = \quad \{\ \text{def. of } Val\ \} \\ Val\ t\ (S\ n)\ x\ r\ v\ (p' :: ps) \\ \quad \leqslant \quad \{\ p \text{ is an optimal extension of } ps\ \} \\ Val\ t\ (S\ n)\ x\ r\ v\ (p :: ps') \end{array}$$

We can turn the equational proof into an Idris proof:

```
Bellman t n ps ops p oep =
  opps where
    opps : OptPolicySeq t (S n) (p :: ps)
    opps Nil x r v impossible
    opps (p' :: ps') x r v =
      transitive_Float_lte step2 step3 where
        y  : Y t x;          y  = outl (p' x r v)
        x' : X (S t);        x' = step t x y
        r' : Reachable x';   r' = reachableSpec1 x r y
        v' : Viable n x';    v' = outr (p' x r v)
        step1 : so (Val (S t) n x' r' v' ps' <= Val (S t) n x' r' v' ps)
        step1 = ops ps' x' r' v'
        step2 : so (Val t (S n) x r v (p' :: ps') <= Val t (S n) x r v (p' :: ps))
        step2 = monotone_Float_plus_lte (reward t x y x') step1
        step3 : so (Val t (S n) x r v (p' :: ps) <= Val t (S n) x r v (p :: ps))
        step3 = oep p' x r v
```

Both the informal and the formal proof require only minor changes from the proofs for the time-independent case presented in the previous paper [BIB13].

4. Monadic transition functions

As explained in our previous paper [BIB13], many sequential decision problems cannot be described in terms of a deterministic transition function.

Even for physical systems which are believed to be governed by deterministic laws, uncertainties might have to be taken into account. They can arise because of different modelling options, imperfectly known initial and boundary conditions and phenomenological closures or through the choice of different approximate solution methods.

In decision problems in climate impact research, financial markets, and sports, for instance, uncertainties are the rule rather than the exception. It would be blatantly unrealistic to assume that we can predict the impact of, e.g., emission policies over a relevant time horizon in a perfectly deterministic manner. Even under the strongest rationality assumptions – each player perfectly knows how its cost and benefits depend on its options and on the options of the other players and has very strong reasons to assume that the other players enjoy the same kind of knowledge – errors, for instance "fat-finger" mistakes, can be made.

In systems which are not deterministic, the *kind* of knowledge which is available to a decision maker can be different in different cases. Sometimes one is able to assess not only which states can be obtained by selecting a given control in a given state but also their probabilities. These systems are called *stochastic*. In other cases, the decision maker might know the possible outcomes of a single decision but nothing more. The corresponding systems are called *non-deterministic*.

The notion of *monadic* systems, originally introduced by Ionescu [Ion09], is a simple, yet powerful, way of treating deterministic, non-deterministic, stochastic and other systems in a uniform fashion. It has been developed in the context of climate vulnerability research, but can be applied to other systems as well. In a nutshell, the idea is to generalize a generic transition function of type $\alpha \ \rightarrow \ \alpha$ to $\alpha \ \rightarrow \ M \ \alpha$ where M is a monad.

For $M = Id$, $M = List$ and $M = SimpleProb$, one recovers the deterministic, the non-deterministic and the stochastic cases. As in [Ion09], we use $SimpleProb \ \alpha$ to formalize the notion of finite probability distributions on α.

We write $M \ : \ Type \ \rightarrow \ Type$ for a monad and *fmap*, *ret* and $\gg\!=$ for its *fmap*, *return* and *bind* operators:

$$
\begin{aligned}
&fmap \ : \ (\alpha \ \rightarrow \ \beta) \ \rightarrow \ M \ \alpha \ \rightarrow \ M \ \beta \\
&ret \quad : \ \alpha \ \rightarrow \ M \ \alpha \\
&(\gg\!=) \ : \ M \ \alpha \ \rightarrow \ (\alpha \ \rightarrow \ M \ \beta) \ \rightarrow \ M \ \beta
\end{aligned}
$$

We can apply the approach developed for monadic dynamical systems to sequential decision problems to extend the transition function to the time-dependent monadic case

$$step \ : (t \ : \ Nat) \ \rightarrow \ (x \ : \ X \ t) \ \rightarrow \ Y \ t \ x \ \rightarrow \ M \ (X \ (S \ t))$$

As it turns out, extending the time-dependent formulation to the monadic case is almost straightforward and we do not present the full details here. There are, however, a few important aspects that needs to be taken into account. We discuss four aspects of the monadic extension in the next four sections.

4.1. Monadic containers.

For our application not all monads make sense. We generalise from the deterministic case where there is just one possible state to some form of container

of possible next states. A monadic container has, in addition to the monadic interface, a membership test:

$$(\in) : \alpha \to M\ \alpha \to \mathit{Bool}$$

For the generalisation of *viable* we also require the predicate *areAllTrue* defined on M-structures of Booleans

$$\mathit{areAllTrue} : M\ \mathit{Bool} \to \mathit{Bool}$$

The idea is that *areAllTrue mb* is true if and only if all Boolean values contained in mb are true. We express this by requiring the following specification

$$\begin{aligned}
&\mathit{areAllTrueSpec} &&: (b : \mathit{Bool}) \to \mathit{so}\ (\mathit{areAllTrue}\ (\mathit{ret}\ b) == b)\\
&\mathit{isInAreAllTrueSpec} &&: (ma : M\ \alpha) \to (p : \alpha \to \mathit{Bool}) \to\\
&&&\quad \mathit{so}\ (\mathit{areAllTrue}\ (\mathit{fmap}\ p\ ma)) \to\\
&&&\quad (a : \alpha) \to \mathit{so}\ (a \in ma) \to \mathit{so}\ (p\ a)
\end{aligned}$$

It is enough to to require this for the special case of α equal to $X\ (S\ t)$.

A key property of the monadic containers is that if we map a function f over a container ma, f will only be used on values in the subset of α which are in ma. We model the subset as $(a : \alpha\ **\ \mathit{so}\ (a \in ma))$ and we formalise the key property by requiring a function *tagElem* which takes any $a : \alpha$ in the container into the subset:

$$\begin{aligned}
&\mathit{tagElem} &&: (ma : M\ \alpha) \to M\ (a : \alpha\ **\ \mathit{so}\ (a \in ma))\\
&\mathit{tagElemSpec} &&: (ma : M\ \alpha) \to \mathit{fmap}\ \mathit{outl}\ (\mathit{tagElem}\ ma) = ma
\end{aligned}$$

The specification requires *tagElem* to be a tagged identity function. For the cases mentioned above ($M = \mathit{Id}$, *List* and *SimpleProb*) this is easily implemented.

4.2. **Viability, reachability.** In section 3 we said that a state $x : X\ t$ is viable for $S\ n$ steps if an only if there exists a control $y : Y\ t\ x$ such that $\mathit{step}\ t\ x\ y$ is viable n steps.

As explained above, monadic extensions are introduced to generalize the notion of deterministic transition function. For $M = \mathit{SimpleProb}$, for instance, $\mathit{step}\ t\ x\ y$ is a probability distribution on $X\ (S\ t)$. Its support represents the set of states that can be reached in one step from x by selecting the control y.

According to this interpretation, M is a monadic container and the states in $\mathit{step}\ t\ x\ y$ are *possible* states at time $S\ t$. For $x : X\ t$ to be viable for $S\ n$ steps, there must exist a control $y : Y\ t\ x$ such that all next states which are possible are viable for n steps. We call such a control a *feasible* control

$$\begin{aligned}
&\mathit{viable} &&: (n : \mathit{Nat}) \to X\ t \to \mathit{Bool}\\
&\mathit{feasible} &&: (n : \mathit{Nat}) \to (x : X\ t) \to Y\ t\ x \to \mathit{Bool}\\
&\mathit{feasible}\ \{t\}\ n\ x\ y = \mathit{areAllTrue}\ (\mathit{fmap}\ (\mathit{viable}\ n)\ (\mathit{step}\ t\ x\ y))
\end{aligned}$$

With the notion of feasibility in place, we can extend the specification of *viable* to the monadic case

$$\begin{aligned}
&\mathit{viableSpec0} &&: (x : X\ t) \to \mathit{Viable}\ Z\ x\\
&\mathit{viableSpec1} &&: (x : X\ t) \to \mathit{Viable}\ (S\ n)\ x \to YF\ t\ n\ x\\
&\mathit{viableSpec2} &&: (x : X\ t) \to YF\ t\ n\ x \to \mathit{Viable}\ (S\ n)\ x
\end{aligned}$$

As in the time-dependent case, $\mathit{Viable}\ n\ x$ as a shorthand for $\mathit{so}\ (\mathit{viable}\ n\ x)$ and

$$YF : (t : Nat) \to (n : Nat) \to X\ t \to Type$$
$$YF\ t\ n\ x = (y : Y\ t\ x \ **\ Feasible\ n\ x\ y)$$

Here, *Feasible* $n\ x\ y$ is a shorthand for *so* (*feasible* $n\ x\ y$).

The notion of reachability introduced in section 3 can be extended to the monadic case straightforwardly: every initial state is reachable. If a state $x : X\ t$ is reachable, every control $y : Y\ t\ x$ leads, via *step* to a M-structure of reachable states. Conversely, if a state $x' : X\ (S\ t)$ is reachable then there exist a state $x : X\ t$ and a control $y : Y\ t\ x$ such that x is reachable and x' is in the M-structure *step* $t\ x\ y$:

$$\begin{aligned}
&reachable : X\ t \to Bool\\
&reachableSpec0 : (x : X\ Z) \to Reachable\ x\\
&reachableSpec1 : (x : X\ t) \to Reachable\ x \to (y : Y\ t\ x) \to\\
&\qquad (x' : X\ (S\ t)) \to so\ (x' \in step\ t\ x\ y) \to Reachable\ x'\\
&reachableSpec2 : (x' : X\ (S\ t)) \to Reachable\ x' \to\\
&\qquad (x : X\ t\ **\ (Reachable\ x,\\
&\qquad\qquad (y : Y\ t\ x\ **\ so\ (x' \in step\ t\ x\ y))))
\end{aligned}$$

As in the time-dependent case, *Reachable* x is a shorthand for *so* (*reachable* x).

4.3. **Aggregation measure.** In the monadic case, the notions of policy and policy sequence are the same as in the deterministic case. The notion of value of a policy sequence, however, requires some attention.

In the deterministic case, the value of selecting controls according to the policy sequence $(p :: ps)$ of length $S\ n$ when in state x at time t is given by

$$reward\ t\ x\ y\ x' + Val\ (S\ t)\ n\ x'\ r'\ v'\ ps$$

where y is the control selected by p, $x' = step\ t\ x\ y$ and r' and v' are proofs that x' is reachable and viable for n steps. In the monadic case, *step* returns an M-structure of states. In general, for each possible state in *step* $t\ x\ y$ there will be a corresponding value of the above sum.

As shown by Ionescu [Ion09], one can easily extend the notion of the value of a policy sequence to the monadic case if one has a way of *measuring* (or aggregating) an M-structure of *Float* satisfying a monotonicity condition:

$$\begin{aligned}
&meas : M\ Float \to Float\\
&measMon : (f : X\ t \to Float) \to (g : X\ t \to Float) \to\\
&\qquad ((x : X\ t) \to so\ (f\ x \leqslant g\ x)) \to\\
&\qquad (mx : M\ (X\ t)) \to so\ (meas\ (fmap f\ mx) \leqslant meas\ (fmap g\ mx))
\end{aligned}$$

With *meas*, the value of a policy sequence in the monadic case can be easily computed

$$\begin{aligned}
&Val : (t : Nat) \to (n : Nat) \to\\
&\qquad (x : X\ t) \to (r : Reachable\ x) \to (v : Viable\ n\ x) \to\\
&\qquad PolicySeq\ t\ n \to Float\\
&Val\ _\ Z\ \ _\ _\ _\ _ = 0\\
&Val\ t\ (S\ n)\ x\ r\ v\ (p :: ps) = meas\ (fmap f\ (tagElem\ mx'))\ \textbf{where}\\
&\quad y : Y\ t\ x; \quad y = outl\ (p\ x\ r\ v)\\
&\quad mx' : M\ (X\ (S\ t)); \ mx' = step\ t\ x\ y
\end{aligned}$$

$$f\ :\ (x'\ :\ X\ (S\ t)\ **\ so\ (x' \in mx'))\ \to\ Float$$
$$f\ (x'\ **\ x'ins) = reward\ t\ x\ y\ x' + Val\ (S\ t)\ n\ x'\ r'\ v'\ ps\ \textbf{where}$$
$$r'\ :\ Reachable\ x';\ r' = reachableSpec1\ x\ r\ y\ x'\ x'ins$$
$$v'\ :\ Viable\ n\ x';\quad v' = isInAreAllTrueSpec\ mx'\ (viable\ n)\ (outr\ (p\ x\ r\ v))\ x'\ x'ins$$

Notice that, in the implementation of f, we (can) call *Val* only for those values of x' which are provably reachable (r') and viable for n steps (v'). Using r, v and $outr\ (p\ x\ r\ v)$, it is easy to compute r' and v' for x' in mx'.

The monotonicity condition for *meas* plays a crucial role in proving Bellman's principle of optimality. The principle itself is formulated as in the deterministic case, see page 12. But now, proving

$$Val\ t\ (S\ n)\ x\ r\ v\ (p' :: ps') \leqslant Val\ t\ (S\ n)\ x\ r\ v\ (p' :: ps)$$

requires proving that

$$meas\ (fmap f\ (step\ t\ x\ y)) \leqslant meas\ (fmap g\ (step\ t\ x\ y))$$

where f and g are the functions that map x' in mx' to

$$reward\ t\ x\ y\ x' + Val\ (S\ t)\ n\ x'\ r'\ v'\ ps'$$

and

$$reward\ t\ x\ y\ x' + Val\ (S\ t)\ n\ x'\ r'\ v'\ ps$$

respectively (and r', v' are reachability and viability proofs for x', as above). We can use optimality of ps and monotonicity of $+$ as in the deterministic case to infer that the first sum is not bigger than the second one for arbitrary x'. The monotonicity condition guarantees that inequality of measured values follows.

A final remark on *meas*: in standard textbooks, stochastic sequential decision problems are often tackled by assuming *meas* to be the function that computes (at time t, state x, for a policy sequence $p :: ps$, etc.) the expected value of $fmap\ g\ (step\ t\ x\ y)$ where y is the control selected by p at time t and g is defined as above. Our framework allows clients to apply whatever aggregation best suits their specific application domain as long as it fulfills the monotonicity requirement. This holds for arbitrary M, not just for the stochastic case.

4.4. **Trajectories.** Dropping the assumption of determinism has an important implication on sequential decision problems: the notion of control sequences (and, therefore, of optimal control sequences) becomes, in a certain sense, void. What in the non-deterministic and stochastic cases do matter are just policies and policy sequences.

Intuitively, this is easy to understand. If the evolution of a system is not deterministic, it makes very little sense to ask for the best actions for the future. The best action at time $t + n$ will depend on the state that will be reached after n steps. Such state is not known in advance. Thus – for non deterministic systems – only policies are relevant: if we have an optimal policy for time $t + n$, we know all we need to optimally select controls at that time.

On a more formal level, the implication of dropping the assumption of determinism is that the notions of control sequence and policy sequence become roughly equivalent. Consider, for example, a non-deterministic case ($M = List$) and an initial state $x_0\ :\ X\ 0$. Assume we have a rule

$$p_0\ :(x\ :\ X\ 0)\ \to\ Y\ 0\ x$$

which allows us to select a control $y_0 = p_0\ x_0$ at time 0. We can then compute the singleton list consisting of the dependent pair $(x_0\ **\ y_0)$:

$$
\begin{aligned}
&mxy_0\ :\ M\ (x\ :\ X\ 0\ **\ Y\ 0\ x)\\
&mxy_0 = ret\ (x_0\ **\ y_0)
\end{aligned}
$$

Via the transition function, mxy_0 yields a list of possible future states mx_1:

$$
\begin{aligned}
&mx_1\ :\ M\ (X\ 1)\\
&mx_1 = step\ 0\ x_0\ y_0
\end{aligned}
$$

Thus, after one step and in contrast to the deterministic case, we do not have just one set of controls to choose from. Instead, we have as many sets as there are elements in mx_1. Because the controls available in a given state depend, in general, on that state, we do not have a uniformly valid rule for joining all such control spaces into a single one to select a new control from. But again, if we have a rule for selecting controls at time 1

$$p_1\ :\ (x\ :\ X\ 1)\ \to\ Y\ 1\ x$$

we can pair each state in mx_1 with its corresponding control and compute a list of state-control dependent pairs

$$
\begin{aligned}
&mxy_1\ :\ M\ (x\ :\ X\ 1\ **\ Y\ 1\ x)\\
&mxy_1 = fmap f\ mx_1\ \textbf{where}\\
&\quad f\ \ \ :\ (x\ :\ X\ 1)\ \to\ (x\ :\ X\ 1\ **\ Y\ 1\ x)\\
&\quad f\ x_1 = (x_1\ **\ p_1\ x_1)
\end{aligned}
$$

In general, if we have a rule p

$$p\ :(t\ :\ Nat)\ \to\ (x\ :\ X\ t)\ \to\ Y\ t\ x$$

and an initial state x_0, we can compute lists of state-control pairs $mxy\ t$ for arbitrary t

$$
\begin{aligned}
&mxy\ :(t\ :\ Nat)\ \to\ M\ (x\ :\ X\ t\ **\ Y\ t\ x)\\
&mxy\ Z\ \ \ \ = ret\ (x_0\ **\ p\ Z\ x_0)\\
&mxy\ (S\ t) = (mxy\ t)\ \gg\!=\ g\ \textbf{where}\\
&\quad g\ :(x\ :\ X\ t\ **\ Y\ t\ x)\ \to\ M\ (x\ :\ X\ (S\ t)\ **\ Y\ (S\ t)\ x)\\
&\quad g\ (xt\ **\ yt) = fmap f\ (step\ t\ xt\ yt)\ \textbf{where}\\
&\quad\quad f\ \ \ :\ (x\ :\ X\ (S\ t))\ \to\ (x\ :\ X\ (S\ t)\ **\ Y\ (S\ t)\ x)\\
&\quad\quad f\ xt = (xt\ **\ p\ (S\ t)\ xt)
\end{aligned}
$$

For a given t, $mxy\ t$ is a list of state-control pairs. It contains all states and controls which can be reached in t steps from x_0 by selecting controls according to $p\ 0\ \ldots\ p\ t$. We can see $mxy\ t$ as a list-based, possibly redundant representation of a subset of the graph of $p\ t$.

We can take a somewhat orthogonal view and compute, for each element in $mxy\ t$, the sequence of state-control pairs of length t leading to that element from $(x_0\ **\ p_0\ x_0)$. What we obtain is a list of sequences. Formally:

$$
\begin{aligned}
&\textbf{data}\ StateCtrlSeq\ :(t\ :\ Nat)\ \to\ (n\ :\ Nat)\ \to\ Type\ \textbf{where}\\
&\quad Nil\ :\ (x\ :\ X\ t)\ \to\ StateCtrlSeq\ t\ Z\\
&\quad (::)\ :\ (x\ :\ X\ t\ **\ Y\ t\ x)\ \to\ StateCtrlSeq\ (S\ t)\ n\ \to\ StateCtrlSeq\ t\ (S\ n)
\end{aligned}
$$

$$
\begin{aligned}
&stateCtrlTrj\ :\ (t\ :\ Nat)\ \to\ (n\ :\ Nat)\ \to\\
&\qquad\qquad (x\ :\ X\ t)\ \to\ (r\ :\ Reachable\ x)\ \to\ (v\ :\ Viable\ n\ x)\ \to\\
&\qquad\qquad (ps\ :\ PolicySeq\ t\ n)\ \to\ M\ (StateCtrlSeq\ t\ n)
\end{aligned}
$$

```
stateCtrlTrj _ Z       x _ _ _          = ret (Nil x)
stateCtrlTrj t (S n) x r v (p :: ps') = fmap prepend (tagElem mx' >>= f) where
  y    : Y t x;          y    = outl (p x r v)
  mx' : M (X (S t)); mx' = step t x y
  prepend : StateCtrlSeq (S t) n → StateCtrlSeq t (S n)
  prepend xys = (x ** y) :: xys
  f : (x' : X (S t) ** so (x' ∈ mx')) → M (StateCtrlSeq (S t) n)
  f (x' ** x'ins) = stateCtrlTrj (S t) n x' r' v' ps' where
    r' : Reachable x'; r' = reachableSpec1 x r y x' x'ins
    v' : Viable n x';   v' = isInAreAllTrueSpec mx' (viable n) (outr (p x r v)) x' x'ins
```

For an initial state $x : X\ 0$ which is viable for n steps, and a policy sequence ps, *stateCtrlTrj* provides a complete and detailed information about all possible state-control sequences of length n which can be obtained by selecting controls according to ps.

For $M = List$, this information is a list of state-control sequences. For $M = SimpleProb$ it is a probability distribution of sequences. In general, it is an M-structure of state-control sequences.

If ps is an optimal policy sequence, we can search *stateCtrlTrj* for different best-case scenarios, assess their *Val*-impacts or, perhaps identify policies which are sub-optimal but easier to implement than optimal ones.

5. Conclusions and outlook

We have presented a dependently typed, generic framework for finite-horizon sequential decision problems. These include problems in which the state space and the control space depend on time 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, more generally, a monadic structure of states.

The framework supports the specification and the solution of specific sequential decision problems that is, the computation of optimal controls for an arbitrary (but finite) number of decision steps n and starting from initial states which are viable for n steps, through instantiation of an abstract context. Users of the framework are expected to implement their problem-specific context. The generic backwards induction algorithm provides them with an optimal sequence of policies for their specific problem.

This paper is part of a longer series exploring the use of dependent types for scientific computing [IJ13a] including the interplay between testing and proving [IJ13b]. We have developed parts of the library code in Agda (as well as in Idris) to explore the stronger module system and we have noticed that several notions could benefit from using the relational algebra framework (called AoPA) built up in [MKJ09]. Rewriting more of the code in AoPA style is future work.

5.1. **Generic tabulation.** The policies computed by backwards induction are provably optimal but backwards induction itself is often computationally intractable. For the cases in which $X\ t$ is finite, the framework provides a tabulated version of the algorithm which is linear in the number of time steps (not presented here).

The tabulated version is still generic but does not come with a machine-checkable proof of correctness. Nevertheless, users can apply the slow but provably correct algorithm

to "small" problems and use these results to validate the fast version. Or they can use the tabulated version for production code and switch back to the safe implementation for verification.

5.2. **Viability and reachability defaults.** As seen in the previous sections, in order to apply the framework to a specific problem, a user has to implement a problem-specific viability predicate

$$\mathit{viable} : (n : \mathit{Nat}) \to X\ t \to \mathit{Bool}$$

Attempts at computing optimal policies of length n from initial states which are not viable for at least n steps are then detected by the type checker and rejected. This guarantees that no exceptions will occur at run time. In other words: the framework will reject attempts at computing solutions of problems which are not well-posed and provide provably optimal solutions for well-posed problems.

As seen in section 4.2, for this to work, *viable* has to be consistent with the problem specific controls Y and transition function *step* that is, it has to fulfill:

$$\begin{aligned}
&\mathit{viableSpec0} : (x : X\ t) \to \mathit{Viable}\ Z\ x\\
&\mathit{viableSpec1} : (x : X\ t) \to \mathit{Viable}\ (S\ n)\ x \to \mathit{YF}\ t\ n\ x\\
&\mathit{viableSpec2} : (x : X\ t) \to \mathit{YF}\ t\ n\ x \to \mathit{Viable}\ (S\ n)\ x
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{YF} : (t : \mathit{Nat}) \to (n : \mathit{Nat}) \to X\ t \to \mathit{Type}\\
&\mathit{YF}\ t\ n\ x = (y : Y\ t\ x ** \mathit{Feasible}\ n\ x\ y)
\end{aligned}$$

and *Feasible* $n\ x\ y$ is a shorthand for *so* (*feasible* $n\ x\ y$):

$$\begin{aligned}
&\mathit{feasible} : (n : \mathit{Nat}) \to (x : X\ t) \to Y\ t\ x \to \mathit{Bool}\\
&\mathit{feasible}\ \{t\}\ n\ x\ y = \mathit{areAllTrue}\ (\mathit{fmap}\,(\mathit{viable}\ n)\ (\mathit{step}\ t\ x\ y))
\end{aligned}$$

Again, users are responsible for implementing or (if they feel confident to do so) postulating the specification.

For problems in which the control state $Y\ t\ x$ is finite for every t and x, the framework provides a default implementation of *viable*. This is based on the notion of successors:

$$\begin{aligned}
&\mathit{succs} : X\ t \to (n : \mathit{Nat} ** \mathit{Vect}\ n\ (M\ (X\ (S\ t))))\\
&\mathit{succsSpec1} : (x : X\ t) \to (y : Y\ t\ x) \to \mathit{so}\ ((\mathit{step}\ t\ x\ y) \in (\mathit{succs}\ x))\\
&\mathit{succsSpec2} : (x : X\ t) \to (mx' : M\ (X\ (S\ t))) \to\\
&\qquad \mathit{so}\ (mx' \in (\mathit{succs}\ x)) \to (y : Y\ t\ x ** mx' = \mathit{step}\ t\ x\ y)
\end{aligned}$$

Users can still provide their own implementation of *viable*. Alternatively, they can implement *succs* (and *succsSpec1*, *succsSpec2*) and rely on the default implementation of *viable* provided by the framework. This is

$$\begin{aligned}
&\mathit{viable}\ Z\ _ = \mathit{True}\\
&\mathit{viable}\ (S\ n)\ x = \mathit{isAnyBy}\ (\lambda mx \Rightarrow \mathit{areAllTrue}\ (\mathit{fmap}\,(\mathit{viable}\ n)\ mx))\ (\mathit{succs}\ x)
\end{aligned}$$

and can be shown to fulfill *viableSpec0*, *viableSpec1* and *viableSpec2*. In a similar way, the framework supports the implementation of *reachable* with a default based on the notion of predecessor.

5.3. **Outlook.** In developing the framework presented in this paper, we have implemented a number of variations of the "cylinder" problem discussed in sections 2 and 3 and a simple "Knapsack" problem instance. Discussing such examples goes beyond the scope of this paper but we plan a follow-up article focused on application examples. In particular, we want to apply the framework to study optimal emission policies in a competitive multi-agent game under threshold uncertainty in the context of climate impact research. This is the application domain that has motivated the development of the framework in the very beginning.

The work presented here naturally raises a number of questions. A first one is related with the notion of reward function

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

As mentioned in our previous paper [BIB13], we have taken *reward* to return values of type *Float* but it is clear that this assumption can be weakened. A natural question here is what specification the return type of *reward* has to fulfill for the framework to be implementable.

A second question is directly related with the notion of viability discussed above. According to this notion, a necessary (and sufficient) condition for a state $x : X\ t$ to be viable $S\ n$ steps is that there exists a control $y : Y\ t\ x$ such that all states in $\mathit{step}\ t\ x\ y$ are viable n steps.

Remember that $\mathit{step}\ t\ x\ y$ is an M-structure of values of type $X\ (S\ t)$. In the stochastic cases, $\mathit{step}\ t\ x\ y$ is a probability distribution. Its support is the set of all states that can be reached from x with non-zero probability by selecting y. Our notion of viability requires all such states to be viable n steps no matter how small their probabilities might actually be. It is clear that under our notion of viability small perturbations of a perfectly deterministic transition function can easily turn a well-posed problem into an ill-posed one. The question here is whether there is a natural way of weakening the viability notion that allows one to preserve well-posedness in the limit for vanishing probabilities of non-viable states.

Another question comes from the notion of aggregation measure introduced in section 4.3. As mentioned there, in the stochastic case *meas* is often taken to be the expected value function. Can we construct other suitable aggregation measures? What is their impact on optimal policy selection?

Finally, the formalization presented here has been implemented on the top of ad-hoc extensions of the Idris standard library. Beside application (framework) specific software components — e.g., for implementing the context of sequential decision problems or tabulated versions of backwards induction — we have implemented data structures to represent bounded natural numbers, finite probability distributions, and setoids. We also have implemented a number of operations on data structures of the standard library, e.g., point-wise modifiers for functions and vectors and filter operations with guaranteed non-empty output.

From a software engineering perspective, an interesting question is how to organize such extensions in a software layer which is independent of specific applications (in our case the components that implement the framework for sequential decision problems) but still not part of the standard library. To answer this question we certainly need a better understanding of the scope of different constructs for structuring programs: modules, parameter blocks, records and type classes.

For instance it is clear that from our viewpoint – that of the developers of the framework – the specifications ($\in$), *isInAreAllTrueSpec* of section 4 demand a more polymorphic formulation. On the other hand, these functions are specifications: in order to apply the

framework, users have to implement them. How can we avoid over-specification while at the same time minimizing the requirements we put on the users?

Tackling such questions would obviously go beyond the scope of this paper and must be deferred to future work.

Acknowledments: 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.

References

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, 1997.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ber95] P. Bertsekas, D. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, Mass., 1995.

[BIB13] Nicola Botta, Cezar Ionescu, and Edwin Brady. Sequential decision problems, dependently-typed solutions. In *Proceedings of the Conferences on Intelligent Computer Mathematics (CICM 2013), "Programming Languages for Mechanized Mathematics Systems Workshop (PLMMS)"*, volume 1010 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[Bra13] Edwin Brady. Programming in Idris : a tutorial, 2013.

[CSRL01] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to algorithms*. Mc-Graw-Hill, second edition, 2001.

[dM95] O. de Moor. A generic program for sequential decision processes. In *In PLILPS '95 Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 1–23. Springer, 1995.

[IJ13a] Cezar Ionescu and Patrik Jansson. Dependently-typed programming in scientific computing: Examples from economic modelling. In Ralf Hinze, editor, *24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, volume 8241 of *LNCS*, pages 140–156. Springer, 2013.

[IJ13b] Cezar Ionescu and Patrik Jansson. Testing versus proving in climate impact research. In *Proc. TYPES 2011*, volume 19 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41–54, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Ion09] Cezar Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin, 2009.

[MKJ09] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *Journal of Functional Programming*, 19:545–579, 2009.

[RND77] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial algorithms: Theory and Practice*. Prentice Hall, 1977.

[SG98] Sutton R. S. and Barto A. G. *Reinforcement learning: An Introduction*. MIT Press, Cambridge, MA, 1998.