Under consideration for publication in J. Functional Programming

Contributions to a computational theory of policy advice and avoidability

NICOLA BOTTA
Potsdam Institute for Climate Impact Research, Germany
(*e-mail:* `botta@pik-potsdam.de`)

PATRIK JANSSON and CEZAR IONESCU
Chalmers University of Technology, Sweden
(*e-mail:* {`patrikj,cezar`}`@chalmers.se`)

Abstract

We present the starting elements of a mathematical theory of policy advice and avoidability. More specifically, we formalize a cluster of notions related to policy advice, such as *policy*, *viability*, *reachability*, and propose a novel approach for assisting decision making, based on the concept of *avoidability*. We formalize avoidability as a relation between current and future states, investigate under which conditions this relation is decidable and propose a generic procedure for assessing avoidability. The formalisation is constructive and makes extensive use of the correspondence between dependent types and logical propositions, decidable judgements are obtained through computations. Thus, we aim for a *computational* theory, and emphasize the role that computer science can play in global system science.

1 Introduction

This paper is a result of inter-disciplinary activities carried out in the framework of several EU-financed projects[1] in the context of Global Systems Science (GSS). It shows that dependently typed programming languages can be a useful vehicle for communication between computer scientists and scientists from other disciplines, for formalizing computable theories, and, of course, for writing provably correct software. It hopefully also points to the fact that the main role of computer science is not confined to the execution of arithmetical operations or sending data over networks, but is rather to be found in the formulation of concepts, identification and resolution of ambiguities, and, above all, in making our ideas clear.

1.1 The need for a theory of policy advice

Scientists involved in fields related to GSS, such as the study of climate change impacts, global finance, epidemics, or international policy, are often faced with the requirement

[1] Global Systems Dynamics and Policy GSDP (2010), Global Systems Rapid Assessment Tools through Constraint Functional Languages GRACeFUL (2015), Centre of Excellence for Global Systems Science CoeGSS (2015)

of acting as advisors to policy makers. For example, they are asked to contribute to the design of international emission reduction agreements (Holtsmark and Sommervoll, 2012; Carbone et al., 2009), to the introduction of a financial transaction tax at EU level (EU-FTT, European Comission (2013)), to programs for the eradication of contagious diseases, (Sandler and G. Arce M., 2002), or to efforts in combating international terrorism (Sandler and Enders, 2004).

In all these application domains, policy making is in need of rigorous scientific advice. At the moment, however, we are lacking an established theory of policy advice. More specifically, we identify three major gaps:

1) The terms used to phrase specific, concrete decision problems – for example sustainability, avoidability, policy – are devoid of precise, well established technical meanings. They are used in an informal, vague manner. The decision problems themselves are often affected by different kinds of uncertainties and tackled with different approaches.

2) There are no *accountable* contracts between advisors and decision makers. The latter do not precisely know what kind of outcomes and guarantees they can expect from implementing the advice received.

3) The proper content of policy advice is unclear. When can decision makers expect to receive advice in the form of simple sequences of actions ("do this, then that, then the other")? When do they have to expect full fledged "action rules" ("if the situation at decision step t satisfies conditions C_1 and C_2, then do action A_1, otherwise do action A_2")? These questions are essential, especially for problems in which the temporal scales of the underlying decision process are not well separated from the time required for implementing decisions.

Our main contributions are towards filling the first gap. But the theory presented in section 3 also provides some understanding of the theoretical and practical limitations of policy advice and of the kind of guarantees that decision makers can expect from advisors. And we do provide a tentative answer to the question of what the proper content of policy advice can be.

1.2 Sequential decision problems and policy advice

The main contribution of this paper is the formalisation of a cluster of concepts required for a theory of policy advice. The formalisation is rooted in optimal control theory, specifically in the study of sequential decision problems (SDPs) and their solutions by dynamic programming.

Sequential decision problems and methods for computing optimal policy sequences are at the core of many applications in economics, logistics and computing science and are, in principle, well understood (Bellman, 1957; De Moor, 1995, 1999; Gnesi et al., 1981; Botta et al., 2013a, 2017). For example, sequential decision problems appear in integrated assessment models (Research Domain III, PIK, 2013; Bauer et al., 2011) in models of international environmental agreements (Finus et al., 2003; Helm, 2003; Bauer et al., 2011; Heitzig, 2012) and in agent-based models of economic systems (Gintis, 2006, 2007; Botta et al., 2013b; Mandel et al., 2009).

The problems addressed by optimal control theory involve the control of a system evolving in time, in order to optimize a reward function over time. In sequential decision problems, time is discrete, and the controls are represented by decisions taken at each time step (hence "sequential").

In the case in which the system to be controlled is deterministic and the initial state of the system can be measured exactly, the solution of a sequential decision problem can be represented in a particularly simple form as a list of successive controls.

Most cases relevant for decision making, however, are fraught with uncertainties, both regarding the transitions of the system and the initial state. In such cases, the solution consists not of a sequence of controls, but of *policies*.

Informally, a policy is a function from states to controls: it tells which control to select when in a given state. Thus, for selecting controls over n steps, a decision maker needs a sequence of n policies, one for each step. We will give a precise definition of policy sequences and of optimal policy sequences in section 3 but, conceptually, optimal policy sequences are sequences of policies which cannot be improved by associating different controls to current and future states.

Optimal policy sequences (or, perhaps *almost* optimal policy sequences) are, for a specific decision problem, the most tangible content that policy advice can deliver for decision making. Thus, it is important that advisors make sure that stakeholders fully understand the difference between controls and policies and, therefore, between control sequences and policy sequences. In fact, the influential "rules versus discretionary measures" paper by Kydland and Prescott (1977), can be interpreted in terms of this distinction. To illustrate this difference, advisors can turn to stylized SDPs (knapsack, production lines, traffic, etc.). Traffic problems are particularly useful in this respect (the sequence of controls, "first turn left, then right, stop ten seconds at traffic light, then turn right" is liable to lead to accidents) and to exemplify the notions of state and control space. Further, it is important that both advisors and decision makers understand that, in general, policy advice cannot (and should not try to) provide, optimal sequences of controls ("optimal action plans", "optimal courses of action", etc.), because no such optimal control sequence can be computed at the time decisions have to be taken and implemented.

What can be computed at the time decisions have to be taken and implemented, however, are optimal policy sequences. A provably optimal policy sequence for a specific problem provides the decision maker with a rule for decision making and with a guarantee that, for that particular problem and at any decision step, there is no better way of making decisions given what is known at that step about the current state and the future. Again, advisors can take advantage of variations of elementary SDPs (with randomly moving obstacles, random production line failures, etc.) to illustrate the differences between deterministic and non-deterministic sequential decision problems.

1.3 The notion of "avoidability"

For many of the SDP problems we cited above, the state and the control spaces can be defined fairly rigorously. Minimal models of international agreements on greenhouse gas emissions, for instance, can be described in terms of a few state variables – perhaps greenhouse gas concentrations and certain gross domestic product measures – and of a few controls: greenhouse gas abatements, investments, etc.

The evolution of the system underlying the decision process can be affected by different kinds of uncertainty, for instance, about model parameters or measurements. Such uncertainties typically lead to non-deterministic, stochastic or fuzzy systems. The framework presented in Botta et al. (2017) for *monadic* sequential decision problems allows one to treat all these (and other) cases seamlessly. Thus, at least conceptually, uncertainties are not a serious obstacle towards a rigorous control theoretical approach for decision making in climate impact research.

But reward functions (the functions that are to be optimized) are: in most practical cases, it is not obvious how they should be defined. This is a limitation to the applicability of both control and game-theoretical approaches to climate impact research.

A common way (Finus et al., 2003; Helm, 2003) of defining reward functions is that of deriving some estimate of the costs and of the benefits associated with the particular decision process under consideration and define rewards on the basis of a cost-benefits analysis. However, there are both pragmatical and ethical concerns with this approach, see for instance Aldred (2009). These difficulties have led a number of authors to argue that, instead of basing decision making on cost-benefits analyses, it would be more sensible to focus on policies that try to avoid future possible states which are known to be potentially harmful. This is the approach exemplified in (Raven et al., 2007) but also in (Schellnhuber, 1998) where the notion of avoidability is implicit in the idea of "tolerable windows".

Some idea of avoidability is also subsumed in the notions of *mitigation* ("A human intervention to reduce the sources or enhance the sinks of greenhouse gases", (Allwood et al., 2014)) and *adaptation* ("The process of adjustment to actual or expected climate and its effects ... to moderate harm or exploit beneficial opportunities", (Allwood et al., 2014)). These are at the core of IPCC's Working Group III research: avoidability of levels of greenhouse gases reckoned to be potentially harmful for a specific human system in the case of mitigation and avoidability (realizability) of the potential harm (opportunities) from climate in the case of adaptation.

But what does it precisely mean for possible future states to be avoidable? And under which conditions is it possible to decide whether a state is avoidable or not?

If we had well understood and widely accepted notions of avoidability and a decision procedure to discriminate between avoidable and non avoidable states, policies that avoid certain future states could be computed as optimal sequences of sequential decision problems with ad-hoc reward functions. For example, one could define rewards to be zero for states which should be avoided and one elsewhere and take advantage of the framework presented in (Botta et al., 2017) to compute policies that *provably* keep the system in a *tolerable* subset of the state space.

Moreover, unambiguous notions of avoidability could help clarifying the notions of mitigation and adaptation. And a computational theory of avoidability could be a first step towards a computational theory of mitigation and adaptation.

Further, a theory of avoidability and, in particular, a generic decision procedure for assessing avoidability, could be useful in many GSS-related fields. In financial markets, even after two decades of Financial Stability Reviews, for instance, unambiguous notions (let apart operational tests) of stability are still elusive (Goodhart, 2004). Here it seems sensible to take a complementary approach and start asking in which sense and whether certain future conditions which are considered or perceived to be potentially dangerous are avoidable.

1.4 Outline

The paper is organized as follows: in section 2 we motivate and explain the basic notation adopted throughout this paper.

In section 3 we present a theory of policy advice for decision making under uncertainty. First, we introduce the notion of monadic sequential decision processes by generalizing the deterministic, the non-deterministic and the stochastic case. Then we introduce the notion of decision problem and apply a new formulation of the theory originally presented in (Botta et al., 2017) to decision making under uncertainty. Specifically, we introduce the notions of policy and policy sequence, we discuss which aspects of decision making under uncertainty need to be accounted for and how different principles of decision making – e.g., precautionary principles and expectation-based principles – can lead to different notions of optimality. We also generalize the procedure for computing provably optimal policy sequences originally presented in (Botta et al., 2017) to reward functions of generic type and demonstrate how to apply the theory to a simple decision problem in the context of climate change.

In section 4 we extend the theory of section 3 to SDPs for which a reward function is not obviously available. Further, we explain how avoidability measures could be applied in climate impact research, e.g., to operationalize notions of levity (Otto and Levermann, 2011), mitigation and adaptation. In section 5 we draw some preliminary conclusions and in section 6 we outline future work.

1.5 Related work

The relation between our work and GSS has been discussed in the first part of the introduction. Here, we situate the present paper in the context of our previous work, and relate it to similar developments in Computing Science.

This paper is based on the dependently typed theory of time-dependent, monadic SDPs originally presented in (Botta et al., 2017). A first theory of time-independent, deterministic SDPs was presented in (Botta et al., 2013a).

Our paper summarizes the work done on monadic SDPs over the last two years and improves the theory of (Botta et al., 2017) in three essential ways. First, by generalizing the return type of the reward function at the core of SDPs. This is now an arbitrary total preorder with a generic $(\oplus)$ operation. Second, by replacing all types which were expressed in terms of lifted Boolean computations in (Botta et al., 2017) by more general types. Third, by extending the original theory of monadic SDPs with a novel theory of avoidability.

Moreover, we have complemented the new formalization with a number of results which have allowed us to eliminate all postulates of the (Botta et al., 2017) theory.

The new theory – discussed in sections 3 and 4 from the perspective of policy advice – is available in the form of literate Idris files in `SequentialDecisionProblems`[2].

[2] In `https://gitlab.pik-potsdam.de/botta/IdrisLibs`

```
data ℕ : Type where
  Z : ℕ
  S : ℕ → ℕ

data Vect : ℕ → Type → Type where
  Nil  : Vect Z a
  Cons : (x : a) → (xs : Vect n a) → Vect (S n) a

head : {n : ℕ} → {A : Type} → Vect (S n) A → A
head (Cons x xs) = x

A      : Type
Sorted : Vect n A → Type
sort   : Vect n A → Vect n A

SortSpec : Type   -- a specification of sort
SortSpec = (n : ℕ) → (xs : Vect n A) → Sorted (sort xs)

sortLemma : SortSpec
sortLemma =   { a proof that sort satisfies the specification SortSpec }
```

Fig. 1. Idris syntax examples.

To the best of our knowledge, it is the first theory of policy advice which entails a computational procedure for obtaining provably optimal policies and for exhaustively investigating the possible consequences of implementing (optimal or not) policies.

Notice, however that a similar approach for solving deterministic SDPs has been proposed (De Moor, 1995) and developed in the *Algebra of Programming* book (Bird and De Moor, 1997). For a discussion of the differences and of the similarities we refer the reader to Botta et al. (2013a).

2 Preliminaries

In this paper we assume the reader knows functional programming and has some familiarity with dependent types. We use Idris (Brady, 2013) as the implementation language and in this section (starting with Fig. 1) we provide a few examples of the syntax to get readers used to Agda or Coq up to speed. We originally decided on using Idris (instead of Agda) because of its emphasis on programming, efficient compilation and interoperability with systems libraries.

Idris provides three implementations of dependent sums (generalized Cartesian products, Σ-types): *DPair*, *Exists* and *Subset*. In spite of the different names, their definitions are conceptually equivalent to

```
data Σ : (A : Type) → (P : A → Type) → Type where
  MkSigma : {A : Type} → {P : A → Type} → (x : A) → (pf : P x) → Σ A P
```

which is the one we will use throughout this paper.

In our development later, most functions will be polymorphic, using a combination of explicit and implicit type arguments. In addition to type parameters, we will also make our development generic in a number of function parameters (like *next*, *reward*, etc.). To avoid passing around the full set of parameters to all functions we will introduce these parameters as we go along and then collect them at the end.

2.1 Equality and equational reasoning

Idris has a built in heterogeneous equality type written $(a = b)$ where $a : A$ and $b : B$. The only constructor is *Refl* : $(a = a)$ and if we have in our hands a value $r : (a = b)$ we know that a and b are equal (and therefore also that A and B are equal). Here are two examples of using the equality type to postulate some desired properties about multiplication:

```
postulate Val        : Type
postulate unitMult   : (y : Val)       →  1 ∗ y = y
postulate assocMult  : (x, y, z : Val)  →  (x ∗ y) ∗ z = x ∗ (y ∗ z)
```

Idris has a special syntax for *equational reasoning*: you can string together a chain of reasoning steps to a full proof. If *p1* shows that *a1* = *a2* and *p2* shows that *a2* = *a3* then (*a1* ={ *p1* }= *a2* ={ *p2* }= *a3* *QED*) is a proof of *a1* = *a3*.

As an example we show a lemma about exponentiation: $x\hat{}m * x\hat{}n = x\hat{}(m+n)$. We prove the lemma using induction over m which means we need to implement three definitions of the following types:

```
expLemma : (x : Val) → (m : ℕ) → (n : ℕ) → (      x^m     ∗ x^n = x^(m + n))
baseCase  : (x : Val) → (n  : ℕ) →           (      x^Z     ∗ x^n = x^(Z + n))
stepCase  : (x : Val) → (m : ℕ) → (n : ℕ) → (ih : x^m     ∗ x^n = x^(m + n)) →
                                              (      x^(S m) ∗ x^n = x^((S m) + n))
```

Note that the last argument *ih* to the step case is the induction hypothesis. The main lemma just uses the base case for zero and the step case for successor and passes a recursive call to *expLemma* as the induction hypothesis.

```
expLemma x Z     n = baseCase x n
expLemma x (S m) n = stepCase x m n (expLemma x m n)
```

With this skeleton in place the proof of the base case is easy:

```
baseCase x n =
    (x^Z ∗ x^n)
  ={ Refl }=                 -- By definition of (^)
    (1 ∗ x^n)
  ={ unitMult (x^n) }=       -- Use 1 ∗ y = y for y = x^n
    (x^n)
  ={ Refl }=                 -- By definition of (+)
    (x^(Z + n))
  QED
```

and the step case is only slightly longer:

```
stepCase x m n ih =
    (x^(S m) ∗ x^n  )
  ={ Refl }=                          -- By definition of (^)
    ((x ∗ x^m) ∗ x^n)
  ={ assocMult x (x^m) (x^n) }=       -- Associativity of multiplication
    (x ∗ (x^m ∗ x^n))
  ={ cong ih }=                  -- Use the ind. hyp.: ih = expLemma x m n
    (x * x^(m + n)  )
  ={ Refl }=                     -- By definition of (^) (backwards)
    (x^(S (m + n)) )
  ={ Refl }=                     -- By definition of (+)
    (x^(S m + n)    )
  QED
```

Here we used *cong* to apply the induction hypothesis "inside" the context $x\,*$. For early examples of using the equality proof notation (in Idris' sister language Agda), see (Mu et al., 2009).

2.2 Programs and proofs

We have seen that we can represent properties as types. In contrast to languages like Coq, in Idris properties are values of type *Type* like *Bool*, $\mathbb{N}$, etc. Thus, for $a : A$ and for a property $P\;a : \mathit{Type}$, values of type $P\;a$ correspond to proofs that a fulfills $P\;a$. We sum up the correspondence between Idris and logic in Table 1.

Idris	Logic
$p : P$	p is a proof of P
FALSE (empty type)	False
non-empty type	True
$P \rightarrow Q$	P implies Q
$\Sigma\,A\,P$	there exists an x of type A such that $P\,x$ holds
$(x : A) \rightarrow P\,x$	forall x of type A, $P\,x$ holds

Table 1. Curry-Howard correspondence relating Idris and logic.

3 Monadic sequential decision problems and policy advice

3.1 Deterministic decision processes

We have argued that, if a decision process is deterministic and the initial state can be measured exactly, solutions of the corresponding decision problem can be represented in a particularly simple form as lists of successive controls. In this section, we formalize the notion of deterministic decision processes.

States A deterministic decision process starts in an initial state x_0 at an initial decision step t_0. Without loss of generality, we can take $t_0 : \mathbb{N}$ and $t_0 = 0$.

The type of x_0 – the state space at t_0 or, in other words, the set of possible initial values – represents all information available to the decision maker at t_0. In a decision process like those underlying models of international environmental agreements, x_0 could just be a tuple of numbers representing some estimate of the greenhouse gas (GHG) concentration in the atmosphere (and, perhaps, in other earth system components), some measure of the gross domestic product, and possibly other model variables, see also the example from section 3.12.

In the example of Figure 2, the state space at step t_0 is simply the set $\{a, b, c, d, e\}$ and the starting state is b. In most decision processes, the state space depends on the decision step. Let's focus on the picture on the left of Figure 2 (we will come to the other two pictures later): the state space at step $t \leqslant 2$, $t = 4$, $t = 5$ and $t \geqslant 7$ consists of the five columns a to e. But at $t = 3$ the state space is just column e and at $t = 6$ only columns a, b and c constitute the state space. In general, a decision process is characterized by a function

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

defining the state space and *State t* represents the state space at decision step t. In the signature of *State* we see a first application of dependent types. In a language in which types were not allowed to be predicated on values, it would not be possible to express the important property that the type *State t* depends on the value t!

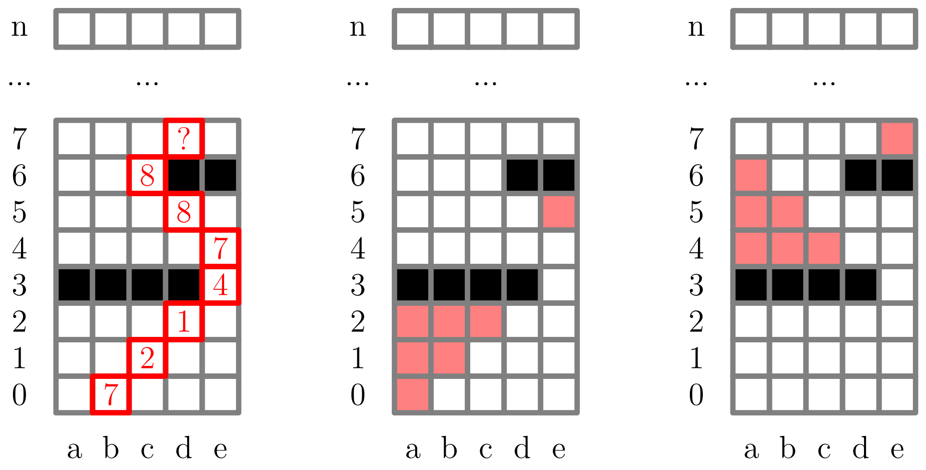

Fig. 2. Possible evolution starting from b (left), states from which less than three decision steps can be done (middle, pale red) and states that cannot be reached, no matter what the initial state at step 0 is and which controls are selected (right, pale red). The black squares represent unavailable columns: only e and a, b, c belong to *State* 3 and *State* 6, respectively.

Controls In a sequential decision process, the decision maker is required to select a control (action, option, choice, etc.) at each decision step. In many applications, controls represent some rate of consumption (of resources which might be limited), some production or investment rate or perhaps different energy options.

In models of international environmental agreements of the kind discussed in (Finus et al., 2003), decision makers could select some rates of abatement of CO_2 emissions, see example from section 3.12. In the model presented in (Heitzig, 2012), controls could be requests for entering or exiting a coalition or a market.

In the example of Figure 2, the controls are, for all but the first and the last column, L to move to the left of the current column, A to stay at the current column and R to move to the right of the current column. In the first column only A and R belong to the control space and in the last column the control space only consists of A and L.

In defining the control space for a particular decision process, it is important to carefully identify which options the decision makers have at their disposal and, we should add, "want" to dispose of. It is easy to imagine decision problems, typical examples are steering problems or negotiations problems, in which decision makers consciously decide to exclude certain control options, e.g. to avoid potentially unmanageable future states.

In general, the set of controls available to the decision maker at a given decision step depends both on that step and on the particular state of the process at that step. Thus, the control space can, in general, be described by a function

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

In the signature of *Ctrl* we see another application of dependent types. The type *Ctrl t x* depends on the values t and x.

Transition functions In deterministic decision processes, the current state and the control selected at the current state together determine the next state. Thus, a deterministic decision process is characterized by a function

$$\mathit{next} : (t : \mathbb{N}) \to (x : \mathit{State}\ t) \to (y : \mathit{Ctrl}\ t\ x) \to \mathit{State}\ (S\ t)$$

and *next t x y* is the state obtained by selecting control y in state x at step t. Notice, again the type dependencies: the type of x depends on the value t; the type of y depends on t and on x. Finally, *next* returns a value in *State* $(S\ t)$ which is the state space at step $S\ t = t + 1$.

Rewards We have mentioned in the introduction that, in SDPs, the decision maker seeks controls that maximize a reward function. This is expressed as a "sum" of rewards, one for each decision step. The notion of sum should be understood in a broad sense: every monoid is, in principle, suitable for defining a reward function. The reward obtained in a single decision step depends, in general, on the current state, on the selected control and on the next state. In models of international environmental agreements, for instance, rewards are computed on the basis of abatement costs and of avoided climate impact damages. Abatement costs certainly depend on the abatement level and, e.g., when the state space also represents available technologies, on the current state. Avoided damages might depend both on the current state and on the next state. In general, a decision process is characterized by a function

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

and *reward* $t\ x\ y\ x'$ represents the value of selecting control y in state x at decision step t *and* moving to state x'. The return type of *reward* is just a type $\mathit{Val} : \mathit{Type}$.

Decision processes Before moving to the next section, let us summarize the results obtained so far for deterministic decision processes. We have seen that specifying one such process requires defining four functions: *State*, *Ctrl*, *next* and *reward*. Of course, this implies defining *Val*, the type of rewards. The first two functions define the types of the state and of the control spaces. The function *next* defines the "dynamics" of the process and *reward* its valuation.

Depending on the specific decision process, defining *State*, *Ctrl*, *next* and *reward* might be trivial or challenging. In climate impact research, it is probably safe to assume that the specification of *State* and *Ctrl* cannot be meaningfully delegated to decision makers and requires a close collaboration between these, domain experts and perhaps modelers.

3.2 Non-deterministic decision processes

The difference between deterministic and non-deterministic decision processes is that, in the second case, selecting a control $y : Ctrl\ t\ x$ when in a state $x : State\ t$ at step $t : \mathbb{N}$ does not yield a unique next state $x' : State\ (S\ t)$ but a whole set of *possible* next states. For instance, a non-deterministic process similar to the one sketched on the left of Figure 2 could be one that, when selecting a control *SR* (defined as "move somewhere to the right") in *b* at the initial decision step, yields a move to *c* or to *d* or perhaps *e*.

Non-deterministic decision processes account for uncertainties in the decision process ("fat-finger" errors in trading games, uncertainty about the effectiveness of controls, etc.), in the transition function (uncertainties about modeling assumptions, observations, etc.) or in the reward function.

There are many ways to account for these and other kinds of uncertainty in the formalization of sequential decision processes but one that has turned out to be particularly simple and effective (Ionescu, 2009) is to have the transition function return a list of values instead of a single value:

$$nexts : (t : \mathbb{N}) \to (x : State\ t) \to (y : Ctrl\ t\ x) \to List\ (State\ (S\ t))$$

Because *List* is a functor, we have a higher-order function *fmap* which propagates uncertainty on the outcome of *nexts* to rewards:

$$rewards : (t : \mathbb{N}) \to (x : State\ t) \to (y : Ctrl\ t\ x) \to List\ Val$$
$$rewards\ t\ x\ y = fmap\ (reward\ t\ x\ y)\ (nexts\ t\ x\ y)$$

In other words, for each possible next state we have, through *reward t x y*, a corresponding possible reward. Therefore, for every $t : \mathbb{N}$, $x : State\ t$ and $y : Ctrl\ t\ x$, we have a unique list of possible rewards. Before further discussing the formalization of non-deterministic decision processes, let's move to the stochastic case.

3.3 Stochastic decision processes

The difference between non-deterministic and stochastic decision processes is that, in the second case and for a given $t : \mathbb{N}$, $x : State\ t$ and $y : Ctrl\ t\ x$ we do not only know the possible next states but also their probabilities. Building upon the non-deterministic case discussed above, we can easily formalize the stochastic case by replacing

$$nexts : (t : \mathbb{N}) \to (x : State\ t) \to (y : Ctrl\ t\ x) \to List\ (State\ (S\ t))$$

with

$$nexts : (t : \mathbb{N}) \to (x : State\ t) \to (y : Ctrl\ t\ x) \to SimpleProb\ (State\ (S\ t))$$

Here *SimpleProb A* represents a finite probability distribution on *A*. A value of type *SimpleProb A* is constructed in terms of two values:

```
data SimpleProb  : Type → Type where
  MkSimpleProb : {A : Type} →
                 (aps : List (A, NonNegRational)) →
                 sumMapSnd aps = 1 →
                 SimpleProb A
```

The first value is a list of pairs of type $(A, NonNegRational)$. In Idris, *NonNegRational* is not a pre-defined numerical type. For an implementation of non-negative rational numbers, please see `NonNegRational`[2].

The second value is a proof that the sum of the probabilities in the list is one. Thus, for instance *MkSimpleProb* [(`'a'`, 1/6), (`'b'`, 1/3), (`'a'`, 1/6), (`'c'`, 1/3)] *Refl* represents a probability distribution on characters in which the probability of `'a'`, `'b'` and `'c'` is $1/3$ and the probability of all other characters is zero. Just like *List* and *Vect n*, *SimpleProb* is a functor. Its *fmap* function

```
fmap : {A, B : Type} → (A → B) → SimpleProb A → SimpleProb B
```

transforms a probability distribution on A into a probability distribution on B, by applying a function that transforms elements of A into elements of B. The implementation of *fmap* is conceptually straightforward. It is documented in `NonNegRational/MonadicOperators`[2]. As in the non-deterministic case, *nexts* induces, via *fmap*, a probability distribution on rewards

```
rewards : (t : ℕ) → (x : State t) → (y : Ctrl t x) → SimpleProb Val
rewards t x y = fmap (reward t x y) (nexts t x y)
```

3.4 Monadic decision processes

In the previous two subsections, we have seen two representations of uncertainty:

- when we only know the possible results of a transition with values in A, we can represent this by a list of elements of A, i.e., a value of type *List A*, and
- when we also have information about the probabilities of the results, we can represent this by a value of type *SimpleProb A*.

Other representations of uncertainty are possible. For example, we might want to describe the quality of possible results of a transition, by using fuzzy sets (e.g., we might want to talk about "big increases in global temperature", "satisfactory economic growth", and so on). Or we might want to combine various representations of uncertainty: say, fuzziness in one dimension with non-determinism in another.

In all these cases, we represent uncertainty of outcomes of type A by some structure of type $M\,A$, that combines possible results with some information about the uncertainty. In the case of non-determinism we have $M = List$ (no additional information), in the case of stochastic uncertainty we have $M = SimpleProb$ (elements with probabilities), and so on.

In all these cases, we can find a function *fmap* which transforms representations of uncertainty of outcomes of type A to representations of uncertainty of outcomes of type B by using a function at the element level

```
fmap : {A, B : Type} → (A → B) → M A → M B
```

in a way which preserves identities and compositions. In other words, the structures with which we represent uncertainty are *functorial*.

Moreover, in all these cases, we have a way of expressing that an outcome is certain. In the case of non-determinism, we do this by wrapping the outcome as a singleton list:

$$
\begin{aligned}
&\mathit{certain} : \{A : \mathit{Type}\} \to A \to \mathit{List}\, A\\
&\mathit{certain}\; a = [a]
\end{aligned}
$$

In the case of stochastic uncertainty, we use a concentrated probability distribution, etc. The transition functions we use to represent uncertain outcomes all have the form

$$\mathit{nexts} : (t : \mathbb{N}) \to (x : \mathit{State}\; t) \to (y : \mathit{Ctrl}\; t\; x) \to M\; (\mathit{State}\; (S\; t))$$

Thus, as opposed to the deterministic case, a decision step yields a M-structure – a value of type $M\; t$ for some type t – of states, and we cannot just apply *nexts* to it again. Via *fmap* we can apply *nexts* to the elements *inside* the structure, but then we end up with a “second-order” uncertainty: we obtain a structure of structures of states. We appear to have lost the basic operation of a discrete dynamical system, namely the ability to iterate the transition function in a uniform fashion.

In fact, however, in all the cases we have seen so far we can reduce a “second-order” representation of uncertainty to a “first-order” one. For example, in the case of lists:

$$
\begin{aligned}
&\mathit{reduce} : \{A : \mathit{Type}\} \to \mathit{List}\; (\mathit{List}\; A) \to \mathit{List}\; A\\
&\mathit{reduce} = \mathit{concat}
\end{aligned}
$$

Similarly, we reduce probabilities of probabilities on A to just probabilities on A, fuzzy sets of fuzzy sets to just fuzzy sets, and so on. In all cases, the reduction satisfies some simple laws, such as, for all $\mathit{ma} : M\; A$

$$\mathit{reduce}\; (\mathit{certain}\; \mathit{ma}) = \mathit{ma}$$

This can be paraphrased as: certainty about an uncertain representation (denoted by *ma*) can be reduced to just the uncertain representation.

Thus, in general, uncertainty about outcomes of type A is represented by a structure of type $M\; A$, where the type constructor $M : \mathit{Type} \to \mathit{Type}$ satisfies the following properties:

- it is a functor (i.e., we have a function *fmap* lifting functions from elements to functions on M-structures)
- we have a way of representing certain outcomes ($\mathit{certain} : A \to M\; A$)
- we have a way of reducing “second-order” uncertainty ($\mathit{reduce} : M\; (M\; A) \to M\; A$)
- these items are related by a small number of simple equations.

Readers familiar with Haskell or category theory will have recognised that these conditions imply that M is a *monad* (Moggi, 1991; Spivey, 1990) and that *certain* and *reduce* are just domain-specific names for *return* and *join*. The equations referred to above can be found in, e.g., section 10.3 of Bird (1998); for their interpretation in the context of dynamical systems, see Ionescu (2009). In particular, the combinator $\gg\!=$ (bind), which can be defined in terms of *fmap* and *join* by

$$
\begin{aligned}
&(\gg\!=) \quad : \{A, B : \mathit{Type}\} \to M\; A \to (A \to M\; B) \to M\; B\\
&\mathit{ma} \gg\!= f = \mathit{join}\; (\mathit{fmap}\; f\; \mathit{ma})
\end{aligned}
$$

describes how to compute the transition from an uncertain state to the next, using a transition function that defines only how to compute an uncertain state from a certain one: for $f : A \to M\ B$ we have $(\gg\!= f) : M\ A \to M\ B$.

In particular, for a stochastic system with state space A, $f\ a$ is the conditional probability distribution over the next state, given that the current state is a. If A is finite, f is traditionally represented by a matrix, whose (i,j)th entry represents the probability that the next state is j, given that the current one is i. An uncertain state is then represented by a vector, whose ith component is the probability that the state is i. The next uncertain state is then computed by vector-matrix multiplication, and this corresponds to the monadic bind combinator for the probability monad.

The reason for using a function of type $A \to M\ A$ rather than $M\ A \to M\ A$ is that the former is usually easier to define. If one cannot define the transition even when the current state is completely known, then one can also not define it in the more difficult case in which the current state is uncertain (otherwise, one could just "forget" elements of the complete description).

A final remark: in decision problems, it is useful to recover certainty as a limiting case of uncertainty and deterministic systems as special instances of monadic systems. Our formalisation handles that gracefully: the identity functor given by $Id\ A = A$ is a monad, for which *fmap*, *certain* (or *return*), and *reduce* (or *join*) are all identity functions and the bind combinator is just (flipped) function application.

3.5 Decision problems

Consider again a non-deterministic decision process ($M = List$) starting in

$x_0 : State\ t_0$

The set of controls available to the decision maker in x_0 at step t_0 is $Ctrl\ t_0\ x_0$. The set of states that can follow after selecting $y_0 : Ctrl\ t_0\ x_0$ is

$nexts\ t_0\ x_0\ y_0 : List\ (State\ (S\ t_0))$

Each of the states in $nexts\ t_0\ x_0\ y_0$ represents a *possible* next state and for each of these states we have a corresponding possible reward:

$fmap\ (reward\ t_0\ x_0\ y_0)\ (nexts\ t_0\ x_0\ y_0) : List\ Val$

If we are to take one single step, and if we have a means of measuring the value of the possible rewards obtained by selecting a specific control:

$meas : List\ Val \to Val$

then, at least conceptually, the problem of making an optimal decision can be solved straightforwardly: for every control in $Ctrl\ t_0\ x_0$, we measure the value of the possible rewards for that control and select the one that yields the highest value. Clearly, this approach cannot, in general, be applied straightforwardly. But it surely works for finite $Ctrl\ t_0\ x_0$ and this is particularly relevant for applications.

But what if we are to take decisions for two or more steps? What does it mean for a decision in step 2 to be "optimal"?

The problem we face is that, even if we were able to select an optimal control y_0^* at step 1 (whatever this means!) we would not be able to even precisely state which controls are available at step 2, let alone which ones are optimal! This is because, for each possible outcome in *nexts* t_0 x_0 y_0^* we would have potentially different sets of controls and potentially different optimal choices.

The argument shows that, except for the deterministic case where a decision at step 1 implies a unique next state, it does not make sense to ask for a specific decision (let alone an "optimal" decision) at step 2 without knowing the outcome of step 1: what is optimal at step 2 very much depends on which of the possible states actually occurs in a particular realization.

Decision making that takes into account the facts as they unfold during a particular realization of the decision process is not only more flexible than decision making based on a fixed control plan. In general, taking advantage of the information that becomes available during a particular decision process yields higher rewards. This is particularly clear if one considers decision processes like those underlying activities such as driving, lecturing, playing a competitive game or negotiating a price. No one would seriously consider tackling such activities by blindly following some fixed, a priori computed "action plan". What is required here is, on one hand, the capability to recognize which situations or states actually occur and, on the other hand, rules that tell one which actions to take for every possible situation or state.

But, if policy advice cannot be about recommending static decision plans and delivering scenarios according to such plans, what should then be the content of policy advice?

To answer this question consider again the two-step decision process outlined above. As we have seen, we cannot say which decision should be taken at step 2 without having performed step 1. But we certainly can compute (again, in principle and with the same caveats mentioned for the case of step 1) an optimal control for every possible outcome of step one. That is, we can compute a function that associates a best control for step 2 to every state in *State* t_1 which can be obtained by selecting y_0^* in step 1.

In fact, we can compute a function that associates to every x_1 : *State* t_1 an optimal control y_1^* : *Ctrl* t_1 x_1. In control theory such functions are called policies and we argue that the main content of policy advice – what advisors are to provide to decision makers – are policies, perhaps, in practice, policy "explanations" or narratives. More precisely, if a decision process unfolds over n steps what is required for decision making under uncertainty are n policies, one for each decision step. Formally:

```
Policy   : (t : ℕ) → Type
Policy t = (x : State t) → Ctrl t x

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

Notice that a policy sequence is a dependent vector which is parameterized by two indices. The first index t : ℕ represents the step at which the first decision has to be taken. The second index n : ℕ gives the length of the policy sequence or, equivalently, the number of policies of the sequence. Thus, a policy sequence of length n assists decision making over n steps.

These notions of policy and policy sequence are, as we will see in the next sections, too simplistic. In order to derive a generic method for computing optimal policies, we will have to refine these notions. This is done in section 3.9. In the next two sections we formalize optimality and introduce two fundamental notions: reachability and viability. These will be the basis for the notion of avoidability presented in section 4.

We conclude this section with three remarks. The first one is that, if we have a policy sequence of length n and a measure *meas* for the value of the possible rewards, we can compute the value (in terms of the sum of measures of possible rewards) of making n decision steps according to that sequence.

Clearly, the computation is not completely straightforward: at the m-th decision step, the value of applying the $n-m$ policies left after m decision steps has to be computed for every possible "next" state! This generates a M-structure of values which has to be measured with *meas*. We discuss such computation in detail in sections 3.6 and 3.9. In spite of its computational complexity, the problem of maximizing the sum of the rewards obtained over n decision steps can be phrased as the problem of finding a policy sequence of length n whose value is at least as good as the value of every other possible policy sequence.

The second remark follows directly from the first one: a particular decision problem is characterized, among others, by a monad M and by a measure $meas : M\ Val \to Val$. The monad characterizes the kind of uncertainties inherent in the decision process. If there are no uncertainties, M is simply *Id*, the identity monad. The measure *meas* characterizes how the decision maker values such uncertainties. In many textbooks on dynamic programming, it is implicitly assumed that $M = SimpleProb$ and *meas* is the expected value measure. Often, this is a sensible assumption. But other measures are possible. In decision problems in climate impact research, for instance, one might want to apply measures which are informed by other guidelines than the maximization of the expected value. Typical examples are worst-case measures, or, in game-theoretical terms, "safety" strategies. Measures of possible rewards have to satisfy a monotonicity condition, see section 3.10. It is a responsibility of advisors to clarify the role of measures in non deterministic SDPs and to make sure that decision makers understand the implications of adopting different principles of measurement on the outcome of a decision process.

The third remark is that SDPs which are not deterministic cannot, in general, be reconducted to "equivalent" deterministic problems. Consider a specific decision process that is, assume that M, *State*, *Ctrl*, and *nexts* are given. We can easily transform this process into a "deterministic" one

$$mnexts : (t : \mathbb{N}) \to (mx : MState\ t) \to (p : ((x : State\ t) \to Ctrl\ t\ x)) \to MState\ (S\ t)$$
$$mnexts\ t\ mx\ p = join\ (fmap\ (\lambda x \Rightarrow nexts\ t\ x\ (p\ x))\ mx)$$

by introducing an "equivalent" state space

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

This is possible because M is a monad and therefore has a *join* transformation. But notice that, in the new formulation, the third argument of *mnexts* are values of type $(x : State\ t) \to Ctrl\ t\ x$. Thus, the policies of the original problem play the role of controls in its deterministic formulation! This is not arbitrary or accidental: in order to apply the *nexts* function of the original problem[3] to the states in *mx*, we have to compute a control (of the original process) for each such state. Therefore we need a policy. The transformation has not brought any practical advantage over the original formulation. Even worse, it has brought the obligation of answering two questions: what does it mean for *mnexts* to be "equivalent" to *nexts* and how to introduce an "equivalent" decision problem by means of a suitable *mreward* function.

Fortunately, there is no need to reformulate monadic decision problems. As we will see in the next section, the notion of policy is strong enough to allow all monadic problems (deterministic, non-deterministic, stochastic, etc.) to be tackled with a uniform, seamless approach. This allows decision makers to select controls on the basis of whatever states will occur in actual realizations in a provably optimal way and according to a notion of optimality which is intuitively understandable and computationally compelling.

3.6 Optimal policies

What is the value, in terms of rewards, of making n decision steps from some initial state $x : \mathit{State}\ t$ by applying the policy sequence $ps : \mathit{PolicySeq}\ t\ n$? More formally: how do we compute *val*?

```
val : (x : State t) → (ps : PolicySeq t n) → Val
```

If $n = 0$ that is, we take zero steps, then we will collect no rewards. In this case *ps* is an empty policy sequence, there is no policy to apply and the answer is simply zero:

```
val {t} {n = Z} x ps = zero
```

Here, we assume that $\mathit{Val} : \mathit{Type}$ is equipped with a zero value $\mathit{zero} : \mathit{Val}$. What if we are to make one or more decision steps? In this case $n = S\ m$ for some $m : \mathbb{N}$ and the policy sequence consists of a first policy, say p, and of a possibly empty tail. We can make a first decision by applying the policy p to the initial value x. This yields a control $y : \mathit{Ctrl}\ t\ x$ and an M-structure of possible next states $\mathit{nexts}\ t\ x\ y : M\ (\mathit{State}\ (S\ t))$. M is a functor. Thus, we can compute, for every $x' : \mathit{State}\ (S\ t)$ in $\mathit{nexts}\ t\ x\ y$ the sum of $\mathit{reward}\ t\ x\ y\ x' : \mathit{Val}$ and of the value of making m decision steps from x' by applying the rest of the policy sequence. This yields an M-structure of *Val*s, one for every possible next state in $\mathit{nexts}\ t\ x\ y$. As discussed in the previous section, the value of such a structure is measured by a function $\mathit{meas} : M\ \mathit{Val} \to \mathit{Val}$:

```
val {t} {n = S m} x (p :: ps) = meas (fmap f mx') where
  y   : Ctrl t x;              y   = p x
  f   : State (S t) → Val;     f x' = reward t x y x' ⊕ val x' ps
  mx' : M (State (S t));       mx' = nexts t x y
```

In the introduction, we argued that an optimal policy sequence is, informally, a policy sequence that cannot be further improved. We can now formalize this intuition. Consider a policy sequence *ps* for n decision steps, starting at t. We say that $ps : \mathit{PolicySeq}\ t\ n$ is

[3] There is little else we can do except for applying *nexts* if the new process has to be, in some meaningful sense, "equivalent" to the original one.

optimal iff for every $x : State\ t$ and for every $ps' : PolicySeq\ t\ n$, applying ps' for n decision steps from x does not yield a better value than applying ps[4]:

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

The notion of optimality of policy sequences requires *Val* to be equipped with a binary "comparison" relation

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

Notice that, if $(\sqsubseteq)$ is reflexive

$$reflexive_{\sqsubseteq} : (a : Val) \to a \sqsubseteq a$$

the empty policy sequence (there is only one) is optimal:

$$
\begin{array}{l}
nilOptPolicySeq : OptPolicySeq\ Nil \\
nilOptPolicySeq\ x\ ps' = reflexive_{\sqsubseteq}\ zero
\end{array}
$$

This is a trivial but important observation. It is a consistency check for the notion of optimality introduced above and, as we will see in section 3.10, the base case for a generic form of backwards induction for computing optimal policy sequences.

3.7 Viability and reachability (the deterministic case)

The notions of policy and policy sequence introduced in section 3.5 are conceptually correct but, for practical purposes, of little use.

Let's consider again the decision problem sketched in Figure 2. For concreteness, assume that the transition function is deterministic and defined such that it simply effects the selected command: selecting L at step 0 in b yields a, selecting A yields b and selecting R yields c and so on. Also, assume that states like a, b and c at step 2 and e at step 5 are truly "dead-ends" or, in other words, that there are no controls for these states (at step 2 and 5, respectively).

Consider the head of a policy sequence $p :: ps$ of length $n = S\ m \geqslant 3$ for this problem. According to the notions of policy and policy sequence introduced in section 3.5, the types of p and ps are $Policy\ 0$ and $PolicySeq\ 1\ m$. Thus, p is a function that associates a control to each of the initial states a, b, c, d and e. There is nothing preventing p to choose L in b

$$p\ b = L$$

But a policy which is the head of a sequence of policies for 3 or more steps cannot select a move to the left for the initial state b! This would lead, for *next* defined as outlined above, to a at $t = 1$ and, from there, to a dead-end no matter what ps at step 2 prescribes. In other words, such a policy sequence would not allow, in general, to take more than 2 steps. To avoid such situations, policies which are elements of policy sequences have to fulfil two additional constraints. The first constraint is that

[4] Remember that, as explained in section 2, in Idris properties are values of type *Type*. Thus *OptPolicySeq ps* is a predicate: it explains (defines) what it means for *ps* to be optimal. In turn, values of type *OptPolicySeq ps* are optimality proofs for *ps*.

Property 1
The m-th policy of a policy sequence of length $n > m$ has to select controls that yield next states from which at least further $n - S\,m$ steps can be taken.

The above rule requires p (the 0-th policy of $p :: ps$) to select R in b. But what shall p select in a? There is no control in a that leads to next states from which at least two more steps can be taken!

The point is simply that a cannot belong to the domain of p. This leads us to the second constraint that policies which are elements of sequences supporting a given number of decision steps have to fulfil. This is a logical consequence of the first one:

Property 2
The domain of the m-th policy of a policy sequence of length $n > m$ has to consist of states from which at least $n - m$ steps can be taken.

Viability Can we formulate these two constraints generically, that is, independently of the particular decision problem at stake or, in other words, for arbitrary *State*, *Ctrl*, *nexts* and *reward* of the appropriate types? Maybe surprisingly, the answer is positive.

Let's consider, first, the deterministic case $M = Id$. Properties 1 and 2 express constraints for the co-domain and for the domain of policies. These constraints are specified in terms of particular subsets of the state space: in Property 1 we consider, at the $S\,m$-th decision step of a sequence of n steps starting at decision step t, next states at step $t + S\,m$ from which at least further $n - S\,m$ steps can be taken. In Property 2 we consider states at decision step $t = m$ from which at least $n - m$ steps can be taken. In both cases, we use a property of states – that of allowing a given number of further steps – to select certain subsets of the state space.

We call this property *viability*. We say that a state x : *State t* is viable for k steps if it is possible, by selecting suitable controls, to take at least k further steps starting from x.

In the middle of Figure 2 we have represented states which are viable for less then three steps in pale red. For instance, a at step 0 is viable for 2 decision steps. At step 1, a and b are viable for 1 step and, at step 2, a, b and c are dead-ends: they are viable for 0 steps. A state which is viable for $S\,k$ steps is also viable for k steps. We can specify the notion of viability in terms of two properties:

Definition 1 (*Viability*)
Every state is viable for zero steps. A state x : *State t* is viable for $S\,m$ steps iff there exists a control y : *Ctrl t x* such that *next t x y* is viable for m steps.

A formalization of this notion is straightforward:

```
Viable              : (n : ℕ) → State t → Type
viableBaseCase      : (x : State t) → Viable Z x
viableToGoodCtrl    : (x : State t) → Viable (S n) x → GoodCtrl t x n
viableFromGoodCtrl  : (x : State t) → GoodCtrl t x n → Viable (S n) x
```

Here, *GoodCtrl* captures the existential constraint on controls:

```
Good          : (t : ℕ) → (x : State t) → (n : ℕ) → (Ctrl t x) → Type
Good t x n y  = Viable {t = S t} n (next t x y)
```

```
GoodCtrl        : (t : ℕ) → (x : State t) → (n : ℕ) → Type
GoodCtrl t x n = Σ (Ctrl t x) (Good t x n)
```

Thus, for instance, *viableToGoodCtrl* ensures that for states which are viable *S n* steps, we can compute a "good" control: one that yields a next state which is viable *n* steps. It is probably worth mentioning that, instead of specifying *Viable*, we could define it

```
Viable {t} Z     x = Unit
Viable {t} (S n) x = GoodCtrl t x n
```

Such a definition would trivially fulfill the specification. The reason why we prefer to let *Viable* undefined in the theory is efficiency: when applying the theory to specific decision problems, it is often possible to provide more efficient, non-inductive, problem-specific implementations.

Policies revisited With the notion of viability in place, we can refine the formalization of policy and policy sequence introduced in section 3.5 to account for the constraints expressed in Properties 1 and 2:

```
Policy : (t : ℕ) → (n : ℕ) → Type
Policy t Z     = Unit
Policy t (S m) = (x : State t) → Viable (S m) x → GoodCtrl t x m
data PolicySeq : (t : ℕ) → (n : ℕ) → Type where
  Nil  : PolicySeq t Z
  (::) : Policy t (S n) → PolicySeq (S t) n → PolicySeq t (S n)
```

A policy is now parameterized on two indices: a decision step counter *t* and a number of steps *n*. We read *p* : *Policy t n* as "*p* is a policy to make decisions at step *t* that support *n* decision steps".

On a policy for 0 steps we have no requirements: we can take *Policy t Z* to be the singleton type.[5] But we require a policy for making a decision at step *t* that supports *m* further decision steps to associate to every state *x* in *State t* which is viable for *S m* steps a control in *Ctrl t x* such that *next t x y* is viable for *m* steps.

Notice that, in contrast to the notion of policy from section 3.5, we now have a constraint on the states to which policies can be applied. This enforces Property 2. We also have a constraint on the controls returned by policies. These have to be "good" controls. A good control is just a control paired with a proof (a guarantee for the decision maker) that that control yields a next state from which a suitable number of further steps can be taken. Thus, Property 1 is enforced by the second element of the dependent pair returned by policies.

As we will see in section 4, the notion of viability is crucial not only for building a sound theory of decision making. When considering policies that avoid potentially harmful future states, one has to be careful not to run into alternative states that lead to dead-ends.

Reachability In the beginning of this section, we have argued that the notions of policy and policy sequence introduced in section 3.5 were conceptually correct but that, in order to be useful, three problems had to be solved. We have formulated two of them through

[5] In Idris the singleton type is denoted by *Unit*. It contains a single element, denoted by ().

Property 1 and Property 2 for the deterministic case. We have seen that addressing these problems is mandatory to make sure that policies for n decision steps do not lead to dead-ends. We have solved these problems for the deterministic case and derived a notion of viability which, if decidable, allows advisors to make precise statements about the capability of states (current or future) to sustain future decision steps. We now turn our attention to the third problem.

Consider, again, the decision process sketched in Figure 2. On the right-hand side of the figure we have coloured in pale red those states which, under the assumptions that decision makers can only move one column to the left or to the right or stay in the same column (these are also the assumption used for colouring partially viable states in the middle of Figure 2), cannot be reached. Thus, for instance, c at decision step 4 cannot be reached because there is no control that allows to move from e, the only state in *State* 3, to c : *State* 4.

Computing policies for subsets of the state space that cannot be reached in a decision process can imply a significant waste of resources. Consider, for instance, the decision problem sketched in Figure 3.

Here all columns are valid and there are no dead-ends. But the set of controls available to the decision maker is more limited than in the example of Figure 2. In a and e, the only control available to the decision maker is A. In b and d, the decision maker can only select L and R, respectively. The only state in which the decision maker truly faces a decision problem is c. Here, it can move to the left or to the right. In other words, the decision maker faces at step zero and in c a dilemma but has otherwise no choices.

The decision problem models a bifurcation: for $t > 1$, the system is either in a or in e no matter what the initial condition was. Thus, there is a wedge of states that cannot be reached. This is marked in pale red in Figure 3. As the number of columns increases, the fraction of the state space that cannot be reached becomes bigger and bigger. Policy advice should focus on future states which actually can happen. We can achieve this goal by putting forward a third constraint on policies:

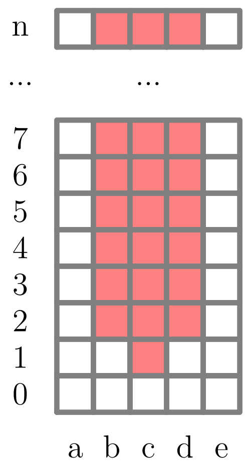

Fig. 3. Bifurcation.

Property 3

The domain of the m-th policy of a policy sequence starting at step t has to consist of states in *State* $(t+m)$ which are reachable.

The notion of reachability is in a certain sense dual to the notion of viability: in the deterministic case, the intuition is that every state at the initial decision step is reachable and that a state x' : *State* $(S\,t)$ is reachable iff it has a reachable predecessor and there exists a control that allows the decision maker to move from there to x':

$$
\begin{array}{ll}
\mathit{Reachable} & : \mathit{State}\ t' \ \to\ \mathit{Type} \\
\mathit{reachableBaseCase} & : (x : \mathit{State}\ Z) \ \to\ \mathit{Reachable}\ x \\
\mathit{reachableForward} & : (x : \mathit{State}\ t) \ \to\ \mathit{Reachable}\ x \ \to\ (y : \mathit{Ctrl}\ t\ x) \ \to\ \mathit{Reachable}\ (\mathit{next}\ t\ x\ y) \\
\mathit{reachableBackward} & : (x' : \mathit{State}\ (S\ t)) \ \to\ \mathit{Reachable}\ x' \ \to\ \Sigma\ (\mathit{State}\ t)\ (\lambda x \Rightarrow x\ \text{`}\mathit{ReachablePred}\text{`}\ x')
\end{array}
$$

Explaining what it means for $x : State\ t$ to be a reachable predecessor of $x' : State\ (S\ t)$ is straightforward:

$$\begin{array}{ll}
Pred & : State\ t \rightarrow State\ (S\ t) \rightarrow Type \\
Pred\ \{t\}\ x\ x' & = \Sigma\ (Ctrl\ t\ x)\ (\lambda y \Rightarrow x' = next\ t\ x\ y) \\
ReachablePred & : State\ t \rightarrow State\ (S\ t) \rightarrow Type \\
ReachablePred\ x\ x' & = (Reachable\ x, x\ `Pred`\ x')
\end{array}$$

Policies revisited again We can now further refine our notion of policy by requiring it to take values in reachable subsets of the state space:

$$Policy\ t\ (S\ m) = (x : State\ t) \rightarrow Reachable\ x \rightarrow Viable\ (S\ m)\ x \rightarrow GoodCtrl\ t\ x\ m$$

We conclude this section by noting that, in the deterministic case, we have been able to express the notions of viability and reachability and Properties 1, 2 and 3 generically. An immediate consequence is that we can apply the framework presented in (Botta et al., 2017) to compute provably correct optimal policies for arbitrary decision problems.

In the next section we show how to extend the notions of reachability and viability (and the corresponding notions of policy and policy sequence) to the general, monadic case.

3.8 Viability and reachability (the monadic case)

Consider again the monads for the deterministic case, for the non-deterministic case and for the stochastic case: *Id*, *List* and *SimpleProb*. These are not just monads but *container* monads. A monadic container M has, in addition to the monadic interface, a membership predicate, a predicate assessing non-emptiness and a "for all" predicate:

$$\begin{array}{ll}
Elem & : \{A : Type\} \rightarrow A \rightarrow M\ A \rightarrow Type \\
NotEmpty & : \{A : Type\} \rightarrow M\ A \rightarrow Type \\
All & : \{A : Type\} \rightarrow (P : A \rightarrow Type) \rightarrow M\ A \rightarrow Type
\end{array}$$

We will write $a \in ma$ for *Elem a ma* for readability. A value of type $a \in ma$ represents a proof that a is contained in ma. We require *Elem*, *NotEmpty* and *All* to fulfil the natural conditions

$$\begin{array}{ll}
allElemSpec0 & : \{A : Type\} \rightarrow \{P : A \rightarrow Type\} \rightarrow \\
 & \quad (a : A) \rightarrow (ma : M\ A) \rightarrow All\ P\ ma \rightarrow a \in ma \rightarrow P\ a \\
elemNotEmptySpec0 & : \{A : Type\} \rightarrow \\
 & \quad (a : A) \rightarrow (ma : M\ A) \rightarrow a \in ma \rightarrow NotEmpty\ ma \\
elemNotEmptySpec1 & : \{A : Type\} \rightarrow \\
 & \quad (ma : M\ A) \rightarrow NotEmpty\ ma \rightarrow \Sigma\ A\ (\lambda a \Rightarrow a \in ma)
\end{array}$$

This interface is similar to the Haskell class *Foldable* which provides *elem*, *null* and *all*, but we use functions returning *Type* instead of *Bool*.

A key property of monadic containers is that if we map a function $f : A \rightarrow B$ over a container ma, f will only be used on values in the subset of A which are in ma. We model the subset as $\Sigma\ A\ (\lambda a \Rightarrow a \in ma)$ and we formalize the key property by requiring a function *tagElem* which takes any $a : A$ in the container into the subset:

$$\begin{array}{ll}
tagElem & : \{A : Type\} \rightarrow (ma : M\ A) \rightarrow M\ (\Sigma\ A\ (\lambda a \Rightarrow a \in ma)) \\
tagElemSpec & : \{A : Type\} \rightarrow (ma : M\ A) \rightarrow fmap\ outl\ (tagElem\ ma) = ma
\end{array}$$

The specification requires *tagElem ma* to just add a tag to the elements of *ma*. For the monads *Id*, *List* and *SimpleProb*, *tagElem* and *tagElemSpec* are easily implemented. We used *tagElem* as part of the interface of container monads in an earlier paper (Botta et al., 2017) but we have not found it elsewhere.

Viability and reachability The notion of viability for the deterministic case expressed necessary and sufficient conditions for being able to perform a given number of steps from a given state. We extend this notion to the monadic case by defining a state x : *State t* to be viable *S m* steps iff there is a control in *Ctrl t x* which allows the decision maker to take *m* further steps no matter which state will follow after selecting *y*. Thus, the notion of viability is unchanged but the conditions required for a control to be "good" are stronger:

```
Good          : (t : ℕ) → (x : State t) → (n : ℕ) → (Ctrl t x) → Type
Good t x n y  = (NotEmpty (nexts t x y), All (Viable {t = S t} n) (nexts t x y))
GoodCtrl      : (t : ℕ) → (x : State t) → (n : ℕ) → Type
GoodCtrl t x n = Σ (Ctrl t x) (Good t x n)
```

We read the specification of *Viable* for the monadic case as: "a state *x* at step *t* is viable for *S m* steps if there is a control in *Ctrl t x* such that all states in *nexts t x y* are viable for *m* steps". With the notion of monadic container, it is straightforward to formalize the predecessor relation in the monadic case

```
Pred            : State t → State (S t) → Type
Pred {t} x x′ = Σ (Ctrl t x) (λy ⇒ x′ ∈ nexts t x y)
```

and to define reachability analogously to the deterministic case

```
Reachable          : {t′ : ℕ} → State t′ → Type
reachableBaseCase  : (x : State Z) → Reachable x
reachableForward   : (x : State t) → Reachable x → (y : Ctrl t x) → All Reachable (nexts t x y)
reachableBackward  : (x′ : State (S t)) → Reachable x′ → Σ (State t) (λx ⇒ x ‘ReachablePred‘ x′)
```

3.9 Policies and policy sequences revisited

With viability and reachability in place, the notions of policy and policy sequence for the general, monadic case are formally identical to the deterministic case:

```
Policy : (t : ℕ) → (n : ℕ) → Type
Policy t Z     = Unit
Policy t (S m) = (x : State t) → Reachable x → Viable (S m) x → GoodCtrl t x m

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

Computing the value of policy sequences for the general, monadic case is almost straightforward, too. To this end, it is useful to introduce the set of *possible* next states:

```
PossibleNextState : (x : State t) → (y : Ctrl t x) → Type
PossibleNextState {t} x y = Σ (State (S t)) (λx′ ⇒ x′ ∈ (nexts t x y))
```

With this notion in place, we can implement *val* and the helper function *sval* by mutual recursion:

```
mutual
  val : (x : State t) → Reachable x → Viable n x → PolicySeq t n → Val
  val {t} {n = Z} x r v ps           = zero
  val {t} {n = S m} x r v (p :: ps) = meas (fmap (sval x r v gy ps) (tagElem mx')) where
    gy   : GoodCtrl t x m;   gy   = p x r v
    y    : Ctrl t x;         y    = ctrl gy
    mx'  : M (State (S t));  mx'  = nexts t x y

  sval : (x : State t) → (r : Reachable x) → (v : Viable (S m) x) →
         (gy : GoodCtrl t x m) → (ps : PolicySeq (S t) m) →
         PossibleNextState x (ctrl gy) → Val
  sval {t} {m} x r v gy ps (MkSigma x' x'emx') = reward t x y x' ⊕ val x' r' v' ps where
    y    : Ctrl t x;               y    = ctrl gy
    mx'  : M (State (S t));        mx'  = nexts t x y
    ar'  : All Reachable mx';      ar'  = reachableForward x r y
    av'  : All (Viable m) mx';     av'  = allViable gy
    r'   : Reachable x';           r'   = allElemSpec0 x' mx' ar' x'emx'
    v'   : Viable m x';            v'   = allElemSpec0 x' mx' av' x'emx'
```

As in section 3.6, we pattern-match on the length of the policy sequence. For policy sequences consisting of a first policy p and of a tail policy sequence ps, we first apply p and compute a control y and an M-structure of possible new states mx'.

Here, *ctrl* and *allViable* are projections. They extract from a good control gy for a state x at step t the control y and the associated proof that all states in $mx' = nexts\ t\ x\ y$ are viable m steps, see appendix A. The proof is crucial for computing $sval\ x\ r\ v\ gy\ ps$, the function to be mapped on $tagElem\ mx'$.

In the above implementation of *val*, *sval* plays the role of f in the implementation of *val* presented in section 3.6. Thus, $sval\ x\ r\ v\ gy\ ps$ associates to possible next states x' in mx' the sum of the reward from the transition from x to x' and of the value of making m further decision steps from x' according to ps.

In order to compute these two values for a given x', we need to provide evidence that x' is reachable and viable m steps. These proofs are coded in r' and v'. We prove that x' is reachable by providing two pieces of evidence: that all elements of mx' are reachable and that x' is an element of mx'.

We know that all elements of mx' are reachable because x is reachable and because of *reachableForward*. We know that x' is an element of mx' because we have built such evidence by applying *tagElem* to mx'. Here we fully exploit the assumption that M is a monadic container.

A similar argument allows us to establish that x' is viable for m steps. This is a necessary condition for computing $val\ x'\ r'\ v'\ ps$ and motivates the "strong" notion of viability discussed in the previous section.

3.10 A framework for monadic sequential decision problems

In this section we introduce the computational core of our theory: first, we formalize the notion of optimality for policy sequences. Then we formulate Bellman's original principle of optimality. Finally, we derive a generic method for computing optimal policy sequences and show that the method yields optimal policies for arbitrary sequential decision problems. We refer to the appendix for technical details and focus on the main results from an application perspective.

Optimality of policy sequences In the previous section we have expressed *a value* in terms of the sum of the *possible* rewards over *n* decision steps of taking decisions according to a policy sequence *ps* : *PolicySeq t n* through *val*.

The emphasis here is on *a value* and *possible*. As explained at the end of section 3.5, in order to compute the value of *ps*, the decision maker has to adopt *a* measure *meas* for estimating the rewards associated to the *possible* outcomes of the decision steps. Decision makers who are measuring chances according to a precautionary principle might end up taking very different decisions from decision makers that measure chances according to their expected value.

The responsibility of adopting a measure is a crucial one and it is a responsibility of the policy advisors to make stakeholders aware of the importance of consciously adopting a measure, to provide alternatives, and to explain the consequences of adopting different criteria.

But, given a decision problem that is, given *M*, *State*, *Ctrl*, *nexts* and *reward* of suitable types, and a measure, *val x r v ps* gives the value, in terms of rewards, of taking *n* decisions starting from a state *x* : *State t* which is reachable and viable for *n* steps and following the policy sequence *ps* : *PolicySeq t n*. Under these premises, it is clear what it means for *ps* to be optimal:

```
OptPolicySeq : PolicySeq t n → Type
OptPolicySeq {t} {n} ps = (x : State t) → (r : Reachable x) → (v : Viable n x) →
                          (ps' : PolicySeq t n) → val x r v ps' ⊑ val x r v ps
```

We read this notion of optimality for policy sequences as follows: "a policy sequence *ps* for making *n* decision steps starting from a state in *State t* is optimal iff for every state in *State t* which is reachable and viable for *n* steps and for every policy sequence *ps′* of the same type as *ps*, the value of *ps* is at least as high as the value of *ps′*". Just as for our simple-minded formalization of policy sequence from section 3.6, also here we can prove that the empty policy sequence is optimal:

```
nilOptPolicySeq : OptPolicySeq Nil;   nilOptPolicySeq x r v ps' = reflexive⊑ zero
```

Bellman's optimality principle Bellman's optimality principle (Bellman, 1957) can be expressed through the notion of optimal extension. Being an optimal extension is a property of a policy. It is relative to a policy sequence. The idea is that a policy *p* for decision step *t* is an optimal extension of a policy sequence *ps* for *n further* decision steps iff for every policy *p′*, the value of *p* :: *ps* is at least as high as the value of *p′* :: *ps*:

```
OptExt : PolicySeq (S t) m → Policy t (S m) → Type
OptExt {t} {m} ps p = (x : State t) → (r : Reachable x) → (v : Viable (S m) x) →
                      (p' : Policy t (S m)) → val x r v (p' :: ps) ⊑ val x r v (p :: ps)
```

Notice that optimal extensions are, in fact, pre-extensions: if *ps* is a sequence of policies for making *m* decision steps starting from step *S t* and *p* is an optimal extension of *ps*, then *p* :: *ps* is a policy sequence for making *S m* decisions starting from step *t*. This will give us "backwards induction" later: we build up our sequence of policies step by step from the last towards the first step.

If p is an optimal extension of $ps : PolicySeq\ (S\ t)\ m$ we know (for sure, no matter whether the decision process is deterministic, non-deterministic, stochastic, etc.) that there are no better ways of making decisions at step t than those indicated by p, given that we will make decisions in the future according to ps. The last conditional is crucial for expressing Bellman's principle. This can be stated as:

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

We read Bellman's principle as follows: for every policy sequence ps and policy p, if ps is an optimal policy sequence and p an optimal extension of ps, then $p :: ps$ is optimal.

Bellman's principle is particularly important because it embodies a simple algorithm for constructing optimal policy sequences: start with the empty policy sequence. As seen above, this is optimal. Then, compute an optimal extension of the empty policy sequence and proceed from there. This algorithm is called backwards induction and we derive a generic and provably correct implementation in the next section.

For the moment, it is important to understand that Bellman's principle reduces the problem of computing optimal policy sequences for n steps to the problem of computing n optimal extensions. This is crucial because of two reasons. The first one is that computing optimal extensions is, in principle, straightforward. We discuss this problem at the end of this section. The second reason is that Bellman's principle suggests that, if we can compute optimal extensions with complexity independent of the length of the policy sequence to be extended, the complexity of computing optimal policy sequences is linear in the number of steps. This is important because it makes a rigorous approach towards policy advice applicable to real problems.

But does Bellman's principle hold? The answer is positive and, in principle, known since 1957. Here, we implement a machine checkable proof. Proving that the policy sequence $(p :: ps)$ is optimal, given that ps is optimal and that p is an optimal extension of ps, means implementing a function that, for every $p' :: ps'$ (with p' and ps' of the same type as p and ps, respectively) and for every $x\ :\ State\ t$, $r\ :\ Reachable\ x$ and $v\ :\ Viable\ (S\ m)\ x$, computes a value of type

$$val\ x\ r\ v\ (p' :: ps') \sqsubseteq val\ x\ r\ v\ (p :: ps)$$

Let $vax = val\ x\ r\ v$ for brevity. The idea is to first prove that

$$vax\ (p' :: ps') \sqsubseteq vax\ (p' :: ps), \quad vax\ (p' :: ps) \sqsubseteq vax\ (p :: ps)$$

From these premises and assuming $(\sqsubseteq)$ to be a preorder (we have already assumed $(\sqsubseteq)$ to be reflexive in the computation of *nilOptPolicySeq*)

$$transitive_{\sqsubseteq}\ :\ \{a, b, c\ :\ Val\}\ \to\ a \sqsubseteq b\ \to\ b \sqsubseteq c\ \to\ a \sqsubseteq c$$

we can immediately deduce the result. A proof of the second part —that the value of $p' :: ps$ is at most as good as that of $p :: ps$— can be immediately computed from the assumption that p is an optimal extension of ps. A proof of the first part —that $p' :: ps'$ is not better than

$p' :: ps$— can be derived from the optimality of ps and from the definition of val. From this inequality

$$vax\ (p' :: ps') \sqsubseteq vax\ (p' :: ps)$$

follows from

$$meas\ (fmap\ svax'\ (tagElem\ mx')) \sqsubseteq meas\ (fmap\ svax\ (tagElem\ mx'))$$

where $svax', svax : PossibleNextState\ x\ (ctrl\ gy') \to Val$ and $mx' : M\ (State\ (S\ t))$ are

$$svax' = sval\ x\ r\ v\ gy'\ ps', \quad svax = sval\ x\ r\ v\ gy'\ ps, \quad mx' = nexts\ t\ x\ y$$

and $sval$ is the function defined in section 3.9. In the above expressions, the control y' is obtained by applying the policy p' to x, r and v:

$$gy' = p'\ x\ r\ v, \quad y' = ctrl\ gy'$$

Thus, the question is whether we can deduce

$$meas\ (fmap\ svax'\ (tagElem\ mx')) \sqsubseteq meas\ (fmap\ svax\ (tagElem\ mx'))$$

from the optimality of ps. For this, we need two additional assumptions:

$$\begin{aligned}
&monotonePlus_{\sqsubseteq} : a \sqsubseteq b \to c \sqsubseteq d \to (a \oplus c) \sqsubseteq (b \oplus d)\\
&measMon : \{A : Type\} \to (f : A \to Val) \to (g : A \to Val) \to\\
&\qquad ((a : A) \to f\ a \sqsubseteq g\ a) \to\\
&\qquad (ma : M\ A) \to meas\ (fmap\ f\ ma) \sqsubseteq meas\ (fmap\ g\ ma)
\end{aligned}$$

The first requirement is clear. The monotonicity condition for *meas* was originally discovered by Ionescu (2009) in a formalization of “vulnerability” as a *measure of possible future harm*. It is a natural condition that all meaningful measures should satisfy. It is easy to see that the expected value measure and “worst case” measures satisfy this condition. As for other specifications of the monadic container interface already discussed, *measMon* is only required to hold for $A = PossibleNextState\ x\ (ctrl\ gy')$ for *Bellman* to hold.

In appendix B, we give a full machine checkable proof of Bellman’s principle for a generic M, that is, independently of whether the decision problem is deterministic, stochastic, non-deterministic or something else.

Backwards induction Assume that we have a procedure for computing an optimal extension of a policy sequence:

$$\begin{aligned}
&optExt : PolicySeq\ (S\ t)\ n \to Policy\ t\ (S\ n)\\
&\textbf{postulate}\ optExtLemma : (ps : PolicySeq\ (S\ t)\ n) \to OptExt\ ps\ (optExt\ ps)
\end{aligned}$$

The postulate is a strong new assumption and we will come back to it later. Under this assumption, a generic backwards induction procedure for computing optimal policy sequences can be implemented as follows:

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

It is easy to see that *backwardsInduction t n* yields optimal policy sequences for arbitrary t and number of decision steps n. It surely does so for n equal to zero because the empty policy sequence is optimal. Assume *ps* : *PolicySeq* (*S t*) *n* is optimal. Bellman's optimality principle shows that (*optExt ps* :: *ps*) : *PolicySeq t* (*S n*) is also optimal. A machine checkable proof can be implemented easily:

```
backwardsInductionLemma : (t : ℕ) → (n : ℕ) → OptPolicySeq (backwardsInduction t n)
backwardsInductionLemma t Z     = nilOptPolicySeq
backwardsInductionLemma t (S n) = Bellman ps ops p oep where
  ps  : PolicySeq (S t) n;  ps  = backwardsInduction (S t) n
  ops : OptPolicySeq ps;    ops = backwardsInductionLemma (S t) n
  p   : Policy t (S n);     p   = optExt ps
  oep : OptExt ps p;        oep = optExtLemma ps
```

Notice how the induction hypothesis – the optimality of *ps* – is obtained through a recursive call to *backwardsInductionLemma*. The lemma shows that, in order to implement a provably correct, generic procedure for computing optimal policy sequences, two ingredients are crucial: Bellman's optimality principle and the capability of computing optimal extensions of arbitrary policy sequences. We have given a machine checkable proof of Bellman's principle in appendix B. In the next section we derive a generic procedure for computing optimal extensions.

Can we compute optimal extensions? Conceptually, computing an optimal extension *p* of a policy sequence *ps* is straightforward. We can define the policy *p* by computing, for every state *x* (which is reachable and viable for *S n* steps), a "best" value in the co-domain of *p*:

```
cval : (x : State t) → (r : Reachable x) → (v : Viable (S n) x) → (ps : PolicySeq (S t) n) →
       GoodCtrl t x n → Val
cval {t} x r v ps gy = meas (fmap (sval x r v gy ps) (tagElem mx')) where
  y   : Ctrl t x;             y   = ctrl gy
  mx' : M (State (S t));   mx' = nexts t x y
optExt : PolicySeq (S t) n → Policy t (S n)
optExt {t} {n} ps = p where
  p : Policy t (S n);   p x r v = cvalargmax x r v ps
```

In the implementation above, *cval x r v ps* is a function that computes the value (measured by *meas*) of applying a (good) control fo make a first decision step in *x* and then *n* further steps according to *ps*.

We construct an optimal extension *p* of *ps* by computing a best "good" control – a value of type *GoodCtrl t x n* – for each (reachable and viable) state. The computation is done by *cvalargmax*. In turn, *cvalargmax x r v ps* computes a good control that maximizes *cval x r v ps*. Thus, the computation of an optimal extension always implies solving a maximization problem for each (reachable and viable) state. We formalize the requirements that *cvalargmax* has to fulfill as:

```
cvalargmax     : (x : State t) → (r : Reachable x) → (v : Viable (S n) x) →
                 (ps : PolicySeq (S t) n) → GoodCtrl t x n
cvalmax        : (x : State t) → (r : Reachable x) → (v : Viable (S n) x) →
                 (ps : PolicySeq (S t) n) → Val
cvalargmaxSpec : (x : State t) → (r : Reachable x) → (v : Viable (S n) x) →
                 (ps : PolicySeq (S t) n) →
```

$$\begin{array}{lll} & & \mathit{cvalmax}\ x\ r\ v\ ps = \mathit{cval}\ x\ r\ v\ ps\ (\mathit{cvalargmax}\ x\ r\ v\ ps) \\ \mathit{cvalmaxSpec} & : & (x : \mathit{State}\ t) \to (r : \mathit{Reachable}\ x) \to (v : \mathit{Viable}\ (S\ n)\ x) \to \\ & & (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to (gy : \mathit{GoodCtrl}\ t\ x\ n) \to \\ & & (\mathit{cval}\ x\ r\ v\ ps\ gy) \sqsubseteq (\mathit{cvalmax}\ x\ r\ v\ ps) \end{array}$$

The reason for using these very specific functions, instead of more general *max* and *argmax*, is that optimisation is, in most cases, not computable. The assumptions on *cvalmax* and *cvalargmax* are the minimal requirements for the computability of optimal extensions. Anything more general risks being non-implementable.

Depending on the specific application, implementing the above specification can be quite difficult or even impossible. It is certainly straightforward for the case in which the set of feasible controls is finite and *Val* is a total preorder. For this case, we provide ready-to-use implementations in `Opt`[2]. We give a machine checkable proof of *optExtLemma* in appendix C.

Theories for solving optimization problems constitute an important sub-domain of numerical analysis, combinatorics and interval arithmetic. They go well beyond the scope of the theory presented here.

3.11 Towards a theory of policy advice

In the previous section we have presented a theory for specifying and solving sequential decision problems under different kinds of uncertainty.

Syntactically, the theory has been introduced through a very limited number of Idris constructs: forward declarations like *M* and *Viable*, fully defined functions like *GoodCtrl*, *Policy* and data types like *PolicySeq*. In particular, we have not made use of records, interfaces, namespaces and parameterized blocks.

Roughly speaking, forward declarations represent theory assumptions. Fully defined functions and data type represent notions, algorithms and results.

Thus, for instance, *OptPolicySeq* is a fully defined function. It explains what it means for a policy sequence to be optimal. *optExt* is also a fully defined function. It implements an algorithm for computing optimal extensions of arbitrary policy sequences. In contrast, $\mathit{reflexive}_{\sqsubseteq}$ is a forward declaration of a function. It requires $(\sqsubseteq)$ (another forward declaration) to be reflexive.

The idea is that users apply the theory by filling in all forward declarations. This allows them to use the theory's fully implemented functions, e.g., to compute machine checked optimal policy sequences.

When considering the theory's assumptions, it is useful to distinguish between a *core* theory and a *full* theory. The core theory contains only those assumptions, notions and algorithms which are needed to fully specify a concrete SDP and to compute optimal policy sequences for that problem. The full theory contains those assumptions and intermediate results (theorems) which are needed to prove that the methods implemented in the core theory are indeed correct.

We provide a complete list of the assumptions of the core theory and of the full theory in appendix D. The core and the full theory are available as literate Idris files at `SequentialDecisionProblems`[2]. The code presented in this section is available in the same repository in `papers/JFP2016`[2].

In the rest of this section, we summarize the most important features of the theory from the perspective of policy advice. We also discuss two questions on limitations and possible extensions of the theory.

In a nutshell, the (core) theory allows to define a specific decision problem by implementing five functions: *M*, *State*, *Ctrl*, *nexts* and *reward*. The (full) theory requires *M* to be a monadic container but does not impose any restriction (or implicit assumption) on the other functions except for those implicit in their signature.

It promotes a disciplined, accountable approach towards policy advice in a threefold way. First, the theory explains what decision makers and advisors have to provide for a decision problem to be unambiguously specified. Second, it explains what it means for policy sequences to be optimal and which guarantees decision makers can expect from implementing optimal policies. Third the theory provides a backwards induction procedure for actually computing optimal policies.

The last result holds under two additional assumptions: that decision makers and advisors agree on a monotone measure $meas : M\ Val \rightarrow Val$ for estimating the value of uncertain rewards and that they provide a method for maximizing the function $cval\ x\ r\ v\ ps$ (for arbitrary x, r, v, and ps) that meets the specification given in the last section.

The latter is a strong assumption. But it cannot be avoided and decision theories that do not explicitly mention this assumption, most likely sweep it under the rug. For example the finiteness assumption is introduced in many applications through a discretization of the control space.

Since Bellman's original contribution in 1957, backwards induction has been routinely implemented and applied to a vast number of decision problems in, among others, economics, bioinformatics and computing science. But our theory is, to the best of our knowledge, the first one that entails a generic, machine checkable implementation. A new theory raises two obvious questions:

1. Can the theory deliver more given the specification of a decision problem?
2. Can the theory demand less for the specification of a decision problem?

The answer to the first question is positive: given a decision problem, we can provide more than optimal policy sequences. In particular, we can provide different notions of monadic trajectories and methods for computing the possible future evolutions resulting from selecting controls according to a given sequence of policies, optimal or not.

These notions can be applied to refine and give precise meanings to the idea of "scenario". The related methods allow advisors to automatically generate consistent and provably complete samples of possible future evolutions. In decision problems with a limited number of options and severe uncertainties, optimal policy sequences can be expected not to be unique. For these problems, decision makers can take advantage of consistent and complete scenarios, e.g. to estimate the impact of different optimal policies according to criteria which are not captured by the notion of optimality characterizing the decision problem. Thus, for instance, a decision maker might not be able (or allowed) to modify the notion of optimality underlying the decision process but still have preferences on optimal policy sequences. He could for instance prefer an optimal policy sequence in which the highest rewards come immediately after the first decision steps to an optimal policy sequence in which the highest rewards come towards the end of the decision procedure, e.g., to increase his chances at being re-elected.

A comprehensive theory of trajectories and scenarios and computational methods for generating such trajectories and scenarios and for combining systems characterized by different kinds of uncertainty have been originally proposed by C. Ionescu and we refer the interested reader to (Ionescu, 2009).

Here, we just outline a generic procedure for computing an M-structure of (all) possible future evolutions under a given policy sequence. To this end, it is important to recognize that a policy sequence naturally generates an M-structure of state-control sequences. These new sequences can be introduced in a similar way as policy sequences:

```
data StateCtrlSeq : (t : ℕ) → (n : ℕ) → Type where
  Nil : (x : State t) → StateCtrlSeq t Z
  (::) : Σ (State t) (Ctrl t) → StateCtrlSeq (S t) n → StateCtrlSeq t (S n)
```

The idea is that, if we are given a sequence of policies *ps* for *n* steps and some initial state *x*, we can construct an M-structure of possible state-control sequences of length *n*. For example, for $M = SimpleProb$, we obtain a probability distribution of state-control sequences representing all possible evolutions of the system given the controls implied by *ps* and starting from *x*:

```
possibleStateCtrlSeqs : (x : State t) → Reachable x → Viable n x →
                        (ps : PolicySeq t n) → M (StateCtrlSeq t n)
```

We give an implementation of *possibleStateCtrlSeqs* in appendix E. If observations of initial states are themselves uncertain, one does not have a well defined initial state. Instead one has an M-structure of possible initial states. Even in this case, we can compute all possible state-control sequences entailed by a sequence of policies:

```
morePossibleStateCtrlSeqs : (mx : M (State t)) → All Reachable mx → All (Viable n) mx →
                            (ps : PolicySeq t n) → M (StateCtrlSeq t n)
morePossibleStateCtrlSeqs {t} {n} mx ar av ps = (tagElem mx) >>= f where
  f : Σ (State t) (λx ⇒ x ∈ mx) → M (StateCtrlSeq t n)
  f (MkSigma x xemx) = possibleStateCtrlSeqs x r v ps where
    r : Reachable x;  r = allElemSpec0 x mx ar xemx
    v : Viable n x;   v = allElemSpec0 x mx av xemx
```

Here, we have used the fact that M is a monad and, therefore, we have an explicit rule, $\gg=$, for computing how uncertainties on states are propagated by transition functions.

The answer to the second question raised above – whether the theory can demand less for the specification of a decision problem – is also positive. The key idea lies in the notion of avoidability and is the subject of the second part of this work.

3.12 A stylized greenhouse gases emission problem

We demonstrate how the theory presented in this section can be applied to specify and solve decision problems under uncertainty with a simple example. This is a stylized stochastic greenhouse gases (GHG) emission problem of the kind studied, among others, by Webster (2000, 2008).

A complete, commented implementation of the problem is available at `EmissionsGame1` in `SequentialDecisionProblems/applications`[2]. Here, we skip over a number of details related to, among others, export/import, auto implicits and name qualification rules.

The idea of the problem is that, at each decision step, a decision maker has to select a certain level of emissions or emission reductions. In Webster (2000), for example, the decision maker can select one of a handful of levels. For simplicity, here we assume that the decision maker has only two options: freezing or increasing emissions. Thus, the control space at any decision step and for any state is

$Ctrl\ t\ x = FreezeOrIncrease$

Here, *FreezeOrIncrease* is a type with only two values: *Freeze* and *Increase*. At each decision step, the decision maker has to make up her choice on the basis of two pieces of information: an amount of cumulated emissions and a state of the world. The latter can be either good or bad.

Again, for simplicity, we assume that, at each decision step, emissions can be increased by zero or one unit. Thus, after *t* decision step, the amount of cumulated emissions can be at most *t*. We represent cumulated emissions through values of type *Fin n*[6]. Thus

$State\ t = (Fin\ (S\ t), GoodOrBad)$

The idea is that the game starts with zero cumulated emissions and with the world in the good state. In these conditions, the risk for the world to turn bad is low. But if the cumulated emissions increase beyond a critical threshold, the probability that the world becomes bad increases. If the world is in the bad state, there is no chance to come back to the good state.

Thus, our idealized problem is stochastic and $M = SimpleProb$. We have provided suitable defaults for deterministic, non-deterministic and stochastic decision problems. These can be included in applications by importing suitable auxiliary modules.

Thus, in our example, we define *M*, *fmap*, *Elem*, *NotEmpty*, *All*, *elemNotEmptySpec0*, *elemNotEmptySpec1*, *tagElem*, *allElemSpec0* from the core theory by importing `FastStochasticDefaults` from `SequentialDecisionProblems`[2].

In contrast to `StochasticDefaults` which defines *SimpleProb* in terms of non-negative rational numbers, `FastStochasticDefaults` implements simple probability distributions in terms of non-negative, double precision floating point numbers and a weaker "sum one" requirement: the sum of the probabilities of a "sum one" probability distribution is only required to be strictly positive. This allows fast arithmetic and the computation of optimal policy sequences for a sizable number of decision steps.

Importing `FastStochasticDefaults` also injects proper definitions for *return*, $(\gg=)$, *finiteAll*, *finiteNotEmpty*, *decidableAll* and *decidableNotEmpty* in the local scope. These entities are declared in `Utils`, another auxiliary module. They are proof obligations in specific deduction skeletons.

Thus, for instance, one skeleton derives the finiteness of *AllViable* from the finiteness of *All* and *Viable*. From the finiteness of *AllViable* and of *NotEmpty*, another skeleton

[6] In Idris, *Fin n* is a datatype representing the natural numbers between 0 and $n-1$. Thus, there are no values of type *Fin* 0 and the single value of type *Fin* 1 represents the natural number 0. *Fin* 2 has values representing 0 and 1 and so on.

derives the finiteness of *Good* from which, assuming finiteness of controls, finiteness of good controls follows.

Additional assumptions, deduction skeletons and default implementations are part of the infrastructure that we have built to support the specification of sequential decision problems.

In our specific case, the additional assumptions are finiteness of states and controls and the imported modules automatically fill in all the assumptions of the theory apart from *State*, *Ctrl* and from those defined in the rest of this section.

We mentioned that, in our stylized GHG emission problem, if the cumulated emissions increase beyond a critical threshold, the probability that the state of the world turns to bad increases. We encode this idea in our transition function. We represent the critical threshold with *cr* : *Double* and denote the probabilities of staying in a good world when the cumulated emissions are below and above the critical threshold by *p1*, *p2* : *NonNegDouble*:

```
cr = 0.0;   p1 = cast 0.99;   p2 = cast 0.10
```

With the transition probabilities in place, the transition function of the emission problem is easily defined by pattern matching on the state of the world and on the decision maker's decision

```
nexts t (e, Good) Freeze =
  let goodState = (weaken e, Good) in
  let badState  = (weaken e, Bad)  in
  if (fromFin e ⩽ cr) then mkSimpleProb [(goodState, p1), (badState, one − p1)]
                      else mkSimpleProb [(goodState, p2), (badState, one − p2)]
nexts t (e, Good) Increase =
  let goodState = (FS e, Good) in
  let badState  = (FS e, Bad)  in
  if (fromFin e ⩽ cr) then mkSimpleProb [(goodState, p1), (badState, one − p1)]
                      else mkSimpleProb [(goodState, p2), (badState, one − p2)]
nexts t (e, Bad) Freeze =
  let badState = (weaken e, Bad) in mkSimpleProb [(badState, one)]
nexts t (e, Bad) Increase =
  let badState = (FS e, Bad)     in mkSimpleProb [(badState, one)]
```

In the code above, *FS* is a function that computes the successor of a value of type *Fin n* and *weaken* is a function that embeds a value of type *Fin n* in *Fin* (*S n*) by leaving the corresponding natural number unchanged.

In order to complete the specification of our emission problem, we have to define the reward function. Remember that *Val*, the return type of *reward* has to have a $\oplus$ combinator, a *zero* value and a total preorder $\sqsubseteq$. We our example these are the standard addition, zero and the smaller or equal relation on non-negative double precision floating point numbers. With these definitions in place, we can proceed to define the reward function of the problem. The idea is that being in a good world yields one unit of benefits per step and being in a bad world yields half of those benefits:

```
badOverGood : NonNegDouble;   badOverGood = cast 0.5
```

Emitting greenhouse gases also brings benefits. These are a fraction of the step benefits in a good world. Further, freezing emissions brings less benefits than increasing emissions:

$$\begin{aligned}
&\mathit{freezeOverGood} &&: \mathit{NonNegDouble}; \quad \mathit{freezeOverGood} &&= \mathit{cast}\ 0.1\\
&\mathit{increaseOverGood} &&: \mathit{NonNegDouble}; \quad \mathit{increaseOverGood} &&= \mathit{cast}\ 0.3
\end{aligned}$$

Finally, the reward function is defined by pattern matching on the control and on the next possible states of the world:

$$\begin{aligned}
&\mathit{reward}\ t\ x\ \mathit{Freeze}\ \ (e, \mathit{Good}) = \mathit{one} && + \mathit{one} * \mathit{freezeOverGood}\\
&\mathit{reward}\ t\ x\ \mathit{Increase}\ (e, \mathit{Good}) = \mathit{one} && + \mathit{one} * \mathit{increaseOverGood}\\
&\mathit{reward}\ t\ x\ \mathit{Freeze}\ \ (e, \mathit{Bad}) \ \ = \mathit{one} * \mathit{badOverGood} && + \mathit{one} * \mathit{freezeOverGood}\\
&\mathit{reward}\ t\ x\ \mathit{Increase}\ (e, \mathit{Bad}) \ \ = \mathit{one} * \mathit{badOverGood} && + \mathit{one} * \mathit{increaseOverGood}
\end{aligned}$$

To measure the possible rewards we use the expected value

$$\mathit{meas} = \mathit{expectedValue}; \quad \mathit{measMon} = \mathit{monotoneExpectedValue}$$

Completing the specification of our problem requires defining *Viable*, *viableToGoodCtrl*, *finiteViable*, *decidableViable*, *Reachable*, *reachableForward*, *decidableReachable*, *finiteCtrl*, *finiteState* and two functions to show states and control. For this example, *Viable t n* is simply the unit type and the implementations of *viableToGoodCtrl*, *finiteViable*, etc. are rather trivial. We do not discuss them here.

The last part of `EmissionsGame1` is a minimal program for computing an optimal sequence of policies and for printing all the state-control sequences and all the rewards that can result from the application of those policies. The file can be compiled with `make emissionsgame1` from the command line. For five decision steps, it produces the output:

```
enter number of steps:
 Nat: 5
thanks!
computing optimal policies ...
computing optimal controls ...
possible state-control sequences:
 [([((0,G),F) ([((0,G),F) ([((0,G),F) ([((0,G),F) ([((0,G),F) ([((0,G),F)
    ((0,G),F)   ((0,G),F)   ((0,G),F)   ((0,G),F)   ((0,G),F)   ((0,B),I)
    ((0,G),F)   ((0,G),F)   ((0,G),F)   ((0,G),F)   ((0,B),I)   ((1,B),I)
    ((0,G),F)   ((0,G),F)   ((0,G),F)   ((0,B),I)   ((1,B),I)   ((2,B),I)
    ((0,G),I)   ((0,G),I)   ((0,B),I)   ((1,B),I)   ((2,B),I)   ((3,B),I)
    ((1,G), )   ((1,B), )   ((1,B), )   ((2,B), )   ((3,B), )   ((4,B), )
   ],0.95)     ],0.01)     ],0.01)     ],0.01)     ],0.01)     ],0.01)
 ]
possible rewards:
 [(5.7, 0.95) (5.2, 0.01) (4.7, 0.01) (4.4, 0.01) (4.1, 0.01) (3.8, 0.01)]
done!
```

Here $(0, G)$ represents the initial state: zero cumulated emissions and a world in a good state. We have six possible state-control sequences starting from $(0, G)$. These are represented by the columns of the `possible state-control sequences` table.

As required by *possibleStateCtrlSeqs*, the possible state-control sequences are a probability distribution. Thus, each column represents a possible outcome of applying the optimal policies *ps* and the probability of that outcome.

The results show that, in a good world, optimal policies require to freeze emissions in all but the last decision step. Conversely, in a bad world, freezing emissions is never an optimal option.

The first column indicates that freezing policies keep the world in a good state with a probability of 95%. The probability of ending up in a bad world is 5%. This probability is equally distributed over the remaining possible trajectories. This is an artifact of having rounded the results to two decimals, the actual probabilities are slightly different. The last table shows the possible rewards and their probabilities.

We do not further comment the results but notice that, even for this oversimplified example, the framework allows to systematically investigate the logical consequences (in terms of optimal emission policies) of a number of assumptions.

Thus, for instance, increasing the uncertainty below the critical threshold has the effect of making freezing policies sub-optimal: if the probability of staying in a good world when the cumulated emissions are below the critical threshold diminishes from 99% to about 65%, freezing emissions becomes sub-optimal no matter whether in a good or in a bad world state.

Importing the full theory guarantees that the results obtained, be these expected or surprising, are logical consequences of the problem specification (for instance, as discussed above, of the specification of transition probabilities) and not of computational artifacts.

4 Policy advice and avoidability

The major weakness of the theory presented in the previous section is that it relies on a reward function:

$$\mathit{reward} : (t : \mathbb{N}) \to (x : X\,t) \to (y : \mathit{Ctrl}\,t\,x) \to (x' : X\,(S\,t)) \to \mathit{Val}$$

In order to specify a decision problem, *reward* has to be defined for every time $t : \mathbb{N}$, for every state $x : \mathit{State}\ t$, for every control $y : \mathit{Ctrl}\ t\ x$ and for every "possible" next state $x' : \mathit{State}\ (S\ t)$.

We could try to be a little bit more precise and only require *reward* to be defined for states which are reachable and viable for a given number of steps. We could also try to constrain x' to be a possible next state. But still, *reward* has to be defined for a decision problem to be specified.

As our GHG emission problem example demonstrates, specifying the state spaces *State*, the control spaces *Ctrl* and the transition function *nexts* of a particular decision process is often straightforward. But the notion of rewards (payoffs, utility, etc.) is more problematic. We do not want to discuss here the reasons of such difficulties. As mentioned in the introduction, they can be practical, ethical or perhaps just operational.

Instead, we ask ourselves whether a theory of policy advice and decision making can be built without relying on the notion of rewards. A way of re-formulating this question is to ask whether rewards could be defined in terms of something less questionable.

4.1 Avoidability

Consider, for concreteness, the problem of designing abatement policies for GHG emissions. Here, the first and foremost concern is to envisage sequences of policies that avoid certain future states which are considered to be potentially harmful, for instance, because in these states certain "climate" variables or certain "socio-economic" variables exceed critical thresholds. In our stylized example, such states were lumped together in the notion of a “bad” world.

If we knew that a policy sequence provably avoids (or provably avoids with a probability above a given threshold) these potentially harmful states and if such a policy sequence was implementable at “low” costs, it would be foolish not to adopt it.

The argument suggests that, in many decision problems, avoidability is a relevant notion which could be fruitfully applied to inform policy advice.

But what does it mean for a future possible state to be avoidable? The question is crucial because, in absence of a clear understanding of what it means for a state to be avoidable, one very first concern of policy advice – namely that of avoiding potentially harmful future states – is void of meaning.

Before attempting a formalization of the notion of avoidability, it is useful to fix a few intuitions: First notice that, in contrast to the notions of reachability and viability put forward in the previous sections, the notion of avoidability is necessarily a relative one. Whether a future state, say a state that can possibly occur in 10 decision steps from now is avoidable or not certainly depends on the current state.

Thus, avoidability is a relation between states. More precisely, it is a relation between states at a given time and states at some later times. Another remark is that we are interested in the avoidability of “possible” future states. We do not care what it means for states that are not reachable to be avoidable. The other way round: we are interested in the avoidability of states which are reachable from a given (e.g., current) state. The latter notion of reachability is again a relative one.

A third remark is that the notion of avoidability entails the notion of an alternative. Consider again Figure 2: for all initial states from which at least three steps can be made (these are, under the assumption that the decision maker can only move to the left, ahead or to the right, columns b, c, d and e), column e at time 3 is unavoidable. This is simply because column e has no alternative: is the only state that can happen at time 3.

Finally consider, again in Figure 2, columns c and d at time 5. Are these states avoidable? There are certainly alternatives: a, b and e. Columns a and b, however, are not reachable from any initial state. Column e is reachable but is a dead-end: it is only viable for zero steps. Should we conclude that columns c or d are unavoidable? We think that, at least for one notion of avoidability, this should be the case: alternatives shall be at least as viable as the state to be avoided.

4.2 Reachability from a state

We have argued that, in order to formalize a notion of avoidability, we need to explain what it means for a state to be reachable from a given state. Consider two states $x'' : \mathit{State}\ t''$ and $x : \mathit{State}\ t$. We explain what it means for x'' to be reachable from x by considering two cases:

```
ReachableFrom : State t'' → State t → Type
ReachableFrom {t'' = Z}    {t} x'' x = (t = Z, x = x'')
ReachableFrom {t'' = S t'} {t} x'' x =
  Either (t = S t', x = x'') (Σ (State t') (λx' ⇒ (x' `ReachableFrom` x, x' `Pred` x'')))
```

The first case is one in which t'' is equal to zero. Remember that we are formalizing a notion of reachability in the future. Therefore t'' cannot be smaller than t: $t'' \geqslant t$. For $t'' = Z$, $t'' \geqslant t$ implies $t = Z$. Thus, t also has to be equal to zero and x has to be equal to x''. This formalizes, at time zero, the intuition that a state at a given time is reachable from a state *at the same time* if and only if the two states are equal.

The second case explains what it means for x'' to be reachable from x for the case in which t'' is not zero. In this case, t'' is the successor of a time t' and we have two cases: either $t = t''$ and $x = x''$ or x'' has a predecessor which is reachable from x.

It is easy to show that the above definition is consistent with our intuition that, if $x'' : \mathit{State}\ t''$ is reachable from $x : \mathit{State}\ t$, then it is the case that $t'' \geqslant t$:

$$\mathit{reachableFromLemma} : (x'' : \mathit{State}\ t'') \rightarrow (x : \mathit{State}\ t) \rightarrow x''\ \text{`}\mathit{ReachableFrom}\text{`}\ x \rightarrow t'' \geqslant t$$

We prove *reachableFromLemma* in appendix F.

4.3 Avoidability

We are now ready to formalize the notion of avoidability discussed in section 4.1: a state $x' : \mathit{State}\ t'$ which is reachable from a state $x : \mathit{State}\ t$ and viable for n steps is avoidable from x if there exists an alternative state $x'' : \mathit{State}\ t'$ which is also reachable from x and viable for n steps:

$$\begin{aligned}
&\mathit{Alternative} : (x : \mathit{State}\ t) \rightarrow (m : \mathbb{N}) \rightarrow (x' : \mathit{State}\ t') \rightarrow (x'' : \mathit{State}\ t') \rightarrow \mathit{Type}\\
&\mathit{Alternative}\ x\ m\ x'\ x'' = (x''\ \text{`}\mathit{ReachableFrom}\text{`}\ x, \mathit{Viable}\ m\ x'', \mathit{Not}\ (x'' = x'))\\
&\mathit{AvoidableFrom} : (x' : \mathit{State}\ t') \rightarrow (x : \mathit{State}\ t) \rightarrow x'\ \text{`}\mathit{ReachableFrom}\text{`}\ x \rightarrow \mathit{Viable}\ n\ x' \rightarrow \mathit{Type}\\
&\mathit{AvoidableFrom}\ \{t'\}\ \{n\}\ x'\ x\ r\ v = \Sigma\ (\mathit{State}\ t')\ (\lambda x'' \Rightarrow \mathit{Alternative}\ x\ n\ x'\ x'')
\end{aligned}$$

The above formalization explains what it means for a state x' to be avoidable given a "current" state x. It is a more or less word-by-word translation of the informal notion discussed in section 4.1. It requires, for x' to be avoidable, the existence of an alternative state x'' which is at least as viable as x'.

Thus, for instance, in the stylized GHG decision problem discussed at the end of section 3, "bad" states of the world can be avoided from conditions in which the world is in a good state.

The viability constraint in this notion of avoidability is essential, for instance for policy advice which has to be informed by sustainability principles. In developing the theory presented in this paper, we have consciously refrained from using, in the formal framework, terms which are prominently used in specific application domains, in particular in climate impact research.[7] Thus, we have denoted the capability of a state to support a certain number of future evolution steps with "viability" and not with "sustainability".

The rationale behind our approach is that it is in a domain specific theory that domain specific notions, for instance the notion of sustainability in climate impact decision problems, are to be given a meaning. This is done in terms of domain-independent notions (for

[7] An exception to this rule is the vocabulary used in discussing our stylized GHG emissions problem and the usage of the term "policy" which is widely used in a number of application domains. We feel that its usage here is justified: policy is a standard notion in control theory and our notion of policy is consistent with that usage.

instance, those proposed here) and the translation is usually referred to as a domain-specific language (DSL).

Our work has been inspired by climate impact research, but our main goal has been to provide a framework of domain-independent notions. It is a responsibility of the developers of a DSL for climate impact research – a team that necessarily has to include climate scientists and decision makers – to give meaning to notions like sustainability in a suitable DSL.

But we have to ask ourselves whether our domain-independent notions are flexible enough to support such a DSL. And since our main motivation comes from climate impact research, our notions should be at least able to support a DSL for this domain.

From this angle, the notion of avoidability outlined above is perhaps too narrow. Consider, again, the problem of designing abatement policies for GHG emissions. Here it seems natural for a decision maker to raise the question whether a future state x' which is considered to be particularly bad from the point of view of sustainability can be avoided given a (factual or hypothetical) "current" state x, given that x' is reachable from x, etc. In this case the property of x' being unsustainable could be expressed by the property of x' being viable only for a limited number of steps. Perhaps x' is to be avoided because it is only viable for zero steps like for instance states a, b and c at time 2 in Figure 2. In this case the intuition is that a meaningful alternative to x' should be more viable than x'. We can capture this idea by dropping the requirement that the alternative state has to be as viable as x'. This is easily done by replacing *Viable n* x' in the argument list of *AvoidableFrom* with a natural number m and by requiring the alternative to x' to be viable for m steps:

$$\begin{array}{l} \mathit{AvoidableFrom} : (x' : \mathit{State}\ t') \to (x : \mathit{State}\ t) \to x'\ \text{`}\mathit{ReachableFrom}\text{`}\ x \to (m : \mathbb{N}) \to \mathit{Type} \\ \mathit{AvoidableFrom}\ \{t'\}\ x'\ x\ r\ m = \Sigma\ (\mathit{State}\ t')\ (\lambda x'' \Rightarrow \mathit{Alternative}\ x\ m\ x'\ x'') \end{array}$$

Thus, the generalization introduces a family of avoidability notions through the additional parameter m. This parameter allows one to strengthen or to weaken the viability requirements that the alternative state has to fulfil. This gives advisors more flexibility to adapt the notion of avoidability to the specific decision problem. For a given decision problem, it allows stakeholders to investigate the consequences of weaker and stronger notions of avoidability.

4.4 Decidability of avoidability

Beside formalizing notions of avoidability, an avoidability theory has to answer the question of whether such notions are decidable. This is crucial for applications.

Knowing what it means for future states to be avoidable is essential to give content to notions that build upon avoidability. In climate impact research, for instance, *mitigation* and *adaptation* (Allwood et al., 2014) depend on the notion of avoidability. They take on different meanings as the underlying notion of avoidability changes. Another notion that depends on that of avoidability is *levity* (Otto and Levermann, 2011). In a nutshell, the idea is that a future state that is potentially very harmful and easily avoidable (perhaps because there are many alternative states) has a high levity. The rationale behind this notion is normative: policies should try to avoid states with high levity values. Obviously, different notions of avoidability imply different notions of levity.

For applications, however, it is often important to be able to assess whether a given future state x' – again, given a current state x, etc. – is avoidable or not. In other words, it is important to have a decision procedure which allows one to discriminate between states which are avoidable and states which are not avoidable.

Decidability does not, in general, come for free. A typical example is that of equality. We have a very clear notion of what it means for two functions to be equal: they have to have the same value at every point. But, in general, we do not have a decision procedure for equality of functions. For functions on real numbers, for instance, we do not have a decision procedure even if we restrict ourselves to equality on a closed interval.

The example makes clear that, if we do not introduce additional requirements, there is little hope for avoidability to be decidable: nothing so far prevents *State t* from being functions of real variables! A minimal requirement is that equality on states is decidable

```
decEqState : (x : State t) → (x' : State t') → Dec (x = x')
```

and we expect most practical applications to fulfil this requirement: if states cannot be distinguished from each other, decision makers will have a very hard time implementing no matter which policy! In the specification above, *Dec* is the standard decidability notion from Idris' prelude:

```
data Dec : Type → Type where
  Yes : {P : Type} → (p  : P)         → Dec P
  No  : {P : Type} → (np : P → Void) → Dec P
```

The idea is that if a predicate P : *Type* is decidable, then we have either an evidence – this is just a value p : P wrapped by *Yes* – or a function np : $P \rightarrow$ *Void* wrapped by *No*. In Idris, *Void* represents the empty type.

Thus, our notion of avoidability is decidable if we can implement a function that returns a value of type *Dec* (*AvoidableFrom* x' x r m) for every x', x, r, and m of the appropriate types. In the next section we discuss under which conditions we can implement such a function and provide an implementation. We conclude this section with two remarks.

An important consequence of decidability is that one can implement a Boolean test. Thus, if avoidability is decidable, decision makers could rely on a test that provably returns *True* if a state x' is avoidable from x and *False* if x' is not avoidable. This could be very useful, for instance in negotiations.

A second implication of avoidability being decidable is that one could easily derive avoidability orderings and use these to compute provably optimal precautionary policies. For instance, one could say that a state x is more avoidable than y if x has a bigger set of alternative states. Such orderings could be combined with measures of possible harm to construct, e.g., reward functions that assign low values to states which are highly avoidable and are possibly very harmful. This would support a more disciplined and more transparent approach towards policy advice, in particular for decision problems in which realistic estimates of costs and benefits are lacking or questionable.

Decidability allows scientific advisors to suggest accountable decision making using core principles, such as levity, avoidance and safety in climate impact research.

4.5 Finite types and decidability

Consider again the notion of avoidability introduced in the last section:

$$\begin{array}{l} \mathit{AvoidableFrom} : (x' : \mathit{State}\ t') \to (x : \mathit{State}\ t) \to x'\ \text{`}\mathit{ReachableFrom}\text{`}\ x \to (m : \mathbb{N}) \to \mathit{Type} \\ \mathit{AvoidableFrom}\ \{t'\}\ x'\ x\ r\ m = \Sigma\ (\mathit{State}\ t')\ (\lambda x'' \Rightarrow \mathit{Alternative}\ x\ m\ x'\ x'') \end{array}$$

This notion explains $x' : \mathit{State}\ t'$ to be avoidable from $x : \mathit{State}\ t$ if there exists a state $x'' : \mathit{State}\ t'$ such that $\mathit{Alternative}\ x\ m\ x'\ x''$. Thus, a decision procedure for avoidability has to provide, for every x', x, r and m either an alternative state or a contradiction. In Idris, a contradiction is a function that, given a $x'' : \mathit{State}\ t'$ and a value of type $\mathit{Alternative}\ x\ m\ x'\ x''$, produces a value of the empty type. Thus, a minimal condition for avoidability to be decidable is that $\mathit{Alternative}\ x\ m\ x'\ x''$ is decidable for every x, m, etc. The intuition is that decidability of $\mathit{Alternative}\ x\ m\ x'\ x''$ is also sufficient if $\mathit{State}\ t'$ is finite.

This intuition is correct and certainly does not depend on anything specific to *State*. We can afford to be a little bit more general and formulate

$$\begin{array}{l} \mathit{finiteDecSigmaLemma} : \{A : \mathit{Type}\} \to \{P : A \to \mathit{Type}\} \to \\ \qquad\qquad\qquad\qquad \mathit{Finite}\ A \to ((a : A) \to \mathit{Dec}\ (P\ a)) \to \mathit{Dec}\ (\Sigma\ A\ P) \end{array}$$

We read the lemma as follows: if A is a finite type and $P : A \to \mathit{Type}$ is decidable, then $\Sigma\ A\ P$ is decidable. We have to explain what it means for a type A to be finite. The idea is that A is finite if there exists a natural number n such that A is isomorphic to $\mathit{Fin}\ n$

$$\mathit{Finite} : \mathit{Type} \to \mathit{Type}; \quad \mathit{Finite}\ A = \Sigma\ \mathbb{N}\ (\lambda n \Rightarrow \mathit{Iso}\ A\ (\mathit{Fin}\ n))$$

We do not detail here the notions of an isomorphism and of *Fin*. These would introduce technicalities that add little to the theory proposed here. In the same spirit, we do not provide a formal proof of *finiteDecSigmaLemma* here but the idea is clear: a finite type A of cardinality n can be represented by a value of type $\mathit{Vect}\ n\ A$ and the question of whether there exists a value in A which fulfils a decidable predicate can be answered by linear search on a vector representation of A.

4.6 Decidability of avoidability, continued

In the last section we have shown that, if $\mathit{Alternative}\ x\ m\ x'\ x''$ is decidable for every $x'' : \mathit{State}\ t'$ and $\mathit{State}\ t'$ is finite, then avoidability of x' is decidable. The next and last step is to discuss under which conditions $\mathit{Alternative}\ x\ m\ x'\ x''$ is decidable. This is pretty straightforward: $\mathit{Alternative}\ x\ m\ x'\ x''$ is just a synonym for three conditions:

$$\mathit{Alternative}\ x\ m\ x'\ x'' = (x''\ \text{`}\mathit{ReachableFrom}\text{`}\ x, \mathit{Viable}\ m\ x'', \mathit{Not}\ (x'' = x'))$$

Thus, we have to understand under which conditions $x''\ \text{`}\mathit{ReachableFrom}\text{`}\ x$, $\mathit{Viable}\ m\ x''$ and $\mathit{Not}\ (x'' = x')$ are decidable. A necessary and sufficient condition for $\mathit{Not}\ (x'' = x')$ to be decidable is that equality in $\mathit{State}\ t'$ (both x'' and x' are states in $\mathit{State}\ t'$) is decidable. As already mentioned, this is a very natural assumption, posited via *decEqState*. What about reachability and viability? Let's look at viability first. We have introduced *Viable* in section 3.8 through the specification

$$\begin{array}{ll} \mathit{Viable} & : (n : \mathbb{N}) \to \mathit{State}\ t \to \mathit{Type} \\ \mathit{viableBaseCase} & : (x : \mathit{State}\ t) \to \mathit{Viable}\ Z\ x \end{array}$$

```
viableToGoodCtrl   : (x : State t) → Viable (S n) x → GoodCtrl t x n
viableFromGoodCtrl : (x : State t) → GoodCtrl t x n → Viable (S n) x
```

An implementation of *Viable* that fulfils this specification is

```
Viable : (n : ℕ) → State t → Type
Viable {t} Z     _ = ()
Viable {t} (S m) x = GoodCtrl t x m
```

A decision procedure for *Viable n x* is a function that computes a value of type *Dec* (*Viable n x*) for every $n : \mathbb{N}$, $t : \mathbb{N}$ and $x : State\ t$:

```
decViable : (n : ℕ) → (x : State t) → Dec (Viable n x)
```

Can we implement such a function for *Viable* defined as above? The case *n* equal to zero is trivial: by definition, every state is viable for zero steps:

```
decViable Z _ = Yes ()
```

For $n = S\ m$, *Viable n x* is equal to *GoodCtrl t x m*. Provided *Ctrl t x* is finite and we have a decision procedure for *Good*

```
finCtrl : (x : State t) → Finite (Ctrl t x)
decGood : (x : State t) → (n : ℕ) → (y : Ctrl t x) → Dec (Good t x n y)
```

we can easily complete the implementation of *decViable* and obtain decidability of *Viable*:

```
decViable {t} (S m) x = finiteDecSigmaLemma (finCtrl x) (decGood x m)
```

In section 3.8 we have explained that controls which are good for performing *m* further decision steps yield non-empty *M*-structures of possible next states such that all states in such structures support at least *m* further decision steps

```
Good t x m y = (NotEmpty (nexts t x y), All (Viable {t = S t} m) (nexts t x y))
```

Thus, decidability of *Good* can be reduced to decidability of *NotEmpty* and, provided that *Viable* is decidable, to decidability of *All*. It might appear that we are stuck in a circular argument: decidability of *Viable* requires decidability of *Good* which requires decidability of *Viable* .... Fortunately, this is not the case. If the control space is finite, it is enough for *NotEmpty* and *All* to be decidable for *M*-structures of states

```
decAll      : (P : (State t) → Type) → ((x : State t) → Dec (P x)) →
              (mx : M (State t)) → Dec (All P mx)
decNotEmpty : (mx : M (State t)) → Dec (NotEmpty mx)
```

for deriving decidability of *Good* and of *Viable* inductively:

```
mutual
  decGood : (x : State t) → (m : ℕ) → (y : Ctrl t x) → Dec (Good t x m y)
  decGood {t} x m y = decPair (decNotEmpty mx′) (decAll (Viable m) (decViable m) mx′) where
    mx′ : M (State (S t));   mx′ = nexts t x y
  decViable : (n : ℕ) → (x : State t) → Dec (Viable n x)
  decViable Z _ = Yes ()
  decViable {t} (S m) x = finiteDecSigmaLemma (finCtrl x) (decGood x m)
```

A similar argument shows that, if, again, *Ctrl t x* is finite and we have a decision procedure for $x \in mx$ for arbitrary $x : State\ t$ and $mx : M\ (State\ t)$

```
decElem : (x : State t) → (mx : M (State t)) → Dec (x ∈ mx)
```

then the predecessor relation *Pred* is decidable:

```
decPred : (x : State t) → (x' : State (S t)) → Dec (x `Pred` x')
decPred {t} x x' = finiteDecSigmaLemma (finCtrl x) (λy ⇒ decElem x' (nexts t x y))
```

From here and using decidability of conjunctions and disjunctions, it is easy to see that *ReachableFrom* is decidable, too:

```
decReachableFrom : (x'' : State t'') → (x : State t) → Dec (x'' `ReachableFrom` x)
decReachableFrom {t'' = Z} {t} x'' x = decPair dp dq where
  dp : Dec (t = Z);          dp = decEq t Z
  dq : Dec (x = x'');        dq = decEqState x x''
decReachableFrom {t'' = S t'} {t} x'' x = decEither dp dq where
  dp : Dec (t = S t', x = x'');   dp = decPair (decEq t (S t')) (decEqState x x'')
  dq : Dec (Σ (State t') (λx' ⇒ (x' `ReachableFrom` x, x' `Pred` x'')))
  dq = finiteDecSigmaLemma fState dRP where
    fState : Finite (State t');   fState = finState t'
    dRP : (x' : State t') → Dec (x' `ReachableFrom` x, x' `Pred` x'')
    dRP x' = decPair drf dpred where
      drf   : Dec (x' `ReachableFrom` x);   drf   = decReachableFrom x' x
      dpred : Dec (x' `Pred` x'');          dpred = decPred x' x''
```

We can summarize the results of this section in the following result: for finite state and control spaces, if equality on states and the monadic container queries *Elem*, *NotEmpty* and *All* are decidable, then *Viable* and *ReachableFrom* are decidable and therefore avoidability is decidable.

4.7 Further thoughts

We have motivated our formalization of the notion of avoidability with the need of tackling decision problems in which estimates of rewards are unavailable or are considered to be problematic. If, however, reward (cost) estimates are available and we have computed sequences of policies that provably avoid certain future states, we can ask ourselves what is the (minimal, maximal, average, etc.) cost of avoiding such states.

This question can be answered by applying the *possibleStateCtrlSeqs* function discussed at the end of section 3.11 (and implemented in appendix E) to our policy sequence. Of course, cost estimates will be relative to a "current" state.

For a sequence of policies for performing n decision steps and a current (viable and reachable) state, *possibleStateCtrlSeqs* computes an M-structure of state-control sequences of length n. Each sequence represents a possible consequence of applying those policies from that current state. If the decision problem was stochastic and therefore M is a probability distribution, each sequence will be associated to a specific probability. Computing the cost of a sequence of state-control pairs is straightforward:

```
valStateCtrlSeq : (t : ℕ) → (n : ℕ) → StateCtrlSeq t n → Val
valStateCtrlSeq t Z       (Nil x) = zero
valStateCtrlSeq t (S Z) ((MkSigma x y) :: (Nil x')) = reward t x y x'
valStateCtrlSeq t (S (S m)) ((MkSigma x y) :: (MkSigma x' y') :: xys) =
  reward t x y x' ⊕ valStateCtrlSeq (S t) (S m) ((MkSigma x' y') :: xys)
```

Mapping *valStateCtrlSeq* on the state-control sequences computed by *possibleStateCtrlSeqs* yields an M-structure of possible costs. Now one can compute minimal, maximal, average cost of avoiding certain future states or whatever other cost measure as desired.

5 Conclusions

In the first part of this paper, we have outlined a theory of decision making for sequential decision problems.

The theory is motivated by decision problems in climate impact research but can also be applied to other domains. It supports a disciplined, accountable approach towards policy advice and a rigorous treatment of decision problems under different kinds of uncertainty. These encompass (but are not limited to) deterministic (no uncertainty), non-deterministic and stochastic uncertainty.

The theory requires decision problems to be specified in terms of four entities: a state space, a decision space, a transition function and a reward function. It gives precise meaning(s) to notions which, in informal approaches towards policy advice and decision making are often unclear. In particular, the theory explains the notions of decision process, decision problem, policy, policy sequence and optimality of policy sequences. It also provides decision makers with a generic procedure for computing provably optimal policy sequences. Thus, the theory makes an accountable approach toward policy advice possible. Examples of computations of optimal policies and scenarios for variations (deterministic, non-deterministic, stochastic) of the problem sketched in Figures 2 and 3 are available in `SequentialDecisionProblems/examples`[2]

In contrast to game theoretical approaches where multiple players are treated explicitly and the temporal dimension is often treated implicitly, the approach proposed here emphasizes a control theoretical perspective. The focus is on the temporal dimension and the point of view is that of an individual decision maker. Notice that this does not imply that we can only apply the theory to problems where there is only a single decision maker. While decision problems with multiple players (and, perhaps, with free-riding opportunities for individual decision makers as they commonly appear, for instance, in models of international environmental agreements) cannot be described explicitly in the theory, they can certainly can be *modeled*. Thus, for instance, we could apply the theory to investigate the effectiveness of measures designed to incentivate (coerce, enforce) coordination in a competitive game with a "Tragedy of the Commons" like structure (Hardin, 1968), by setting up a chain of (possibly sequential) decision problems. In each such problems, the transition function for the single "representative" decision maker explicitly represented in our theory would encode the collective behavior of the other decision makers. Modeling competitive games with multiple, different decision makers would be more complicated but possible. The solution of a decision problem would inform the transition function of the next problem in the chain, and so on. The study of the evolution of coordination – in particular, under uncertainty – would be an interesting subject at the border between game theory, control theory and evolutionary decision making (Ellison, 1993; Peyton Young, 1993; Ellison, 1995; Peyton Young, 2001).

In the second part of our paper, we have worked towards extending our theory to decision problems for which a reward function is not obviously available or for which notions of optimality based on costs-benefits analyses are questionable. The extension is based on the idea of avoidability. We have proposed a family of avoidability notions and discussed under which conditions avoidability is decidable. We have also sketched how decidable notions of avoidability could be used to derive avoidability measures.

Avoidability measures could be applied in climate impact research, e.g., to operationalize notions of levity, mitigation and adaptation. These notions are considered to be crucial in policy advice but, to the best of our knowledge, have not so far been formalized. We consider our theory as a first step in this direction.

6 Future work

In section 3.1 we noted that "In climate impact research, it is probably safe to assume that the specification of *State* and *Ctrl* cannot be meaningfully delegated to decision makers and requires a close collaboration between these, domain experts and perhaps modelers". As future work we would like to develop a Domain Specific Language to support the specification of Sequential Decision Problems (SDPs). The aim would be to A) make it easier for domain experts to describe a problem in a way that fits the theory developed here and B) develop a collection of simple examples and reusable combinators to build more complex SDPs.

Our algorithms for solving SDPs are based on computable policies. In section 3.5 we wrote "In control theory such functions are called policies and we argue that the main content of policy advice – what advisors are to provide to decision makers – are policies, perhaps, in practice, policy 'explanations' or narratives". Future work includes investigating how to provide (or even parse) "text approximations" of policies using natural language technology.

A Auxiliary functions

In section 3.9, we have used the following auxiliary functions:

```
ctrl : GoodCtrl t x n → Ctrl t x
ctrl (MkSigma y _) = y

allViable : (y : GoodCtrl t x n) → All (Viable n) (nexts t x (ctrl y))
allViable (MkSigma _ p) = snd p
```

B Bellman's principle

The proof of Bellman's principle from section 3.10 in full:

```
Bellman : (ps : PolicySeq (S t) m) → OptPolicySeq ps →
          (p  : Policy t (S m))    → OptExt ps p → OptPolicySeq (p :: ps)
Bellman {t} {m} ps ops p oep = opps where
  opps : OptPolicySeq (p :: ps)
  opps x r v (p' :: ps') = transitive⊑ s4 s5 where
    gy'  : GoodCtrl t x m;              gy'  = p' x r v
    y'   : Ctrl t x;                    y'   = ctrl gy'
    mx'  : M (State (S t));             mx'  = nexts t x y'
```

```
av'  : All (Viable m) mx';                       av' = allViable gy'
f'   : PossibleNextState x (ctrl gy') → Val;  f'  = sval x r v gy' ps'
f    : PossibleNextState x (ctrl gy') → Val;  f   = sval x r v gy' ps
s1   : (x' : State (S t)) → (r' : Reachable x') → (v' : Viable m x') →
         val x' r' v' ps' ⊑ val x' r' v' ps
s1 x' r' v' = ops x' r' v' ps'
s2   : (z : PossibleNextState x (ctrl gy')) → (f' z) ⊑ (f z)
s2 (MkSigma x' x'emx') =
   monotonePlus⊑ (reflexive⊑ (reward t x y' x')) (s1 x' r' v') where
     ar' : All Reachable mx';  ar' = reachableForward x r y'
     r'  : Reachable x';       r'  = allElemSpec0 x' mx' ar' x'emx'
     v'  : Viable m x';        v'  = allElemSpec0 x' mx' av' x'emx'
s3    : meas (fmap f' (tagElem mx')) ⊑ meas (fmap f (tagElem mx'))
s3    = measMon f' f s2 (tagElem mx')
s4    : val x r v (p' :: ps') ⊑ val x r v (p' :: ps);   s4 = s3
s5    : val x r v (p' :: ps) ⊑ val x r v (p :: ps);     s5 = oep x r v p'
```

In the above implementation we construct a function *opps* that returns a value of type

$$val\ x\ r\ v\ (p' :: ps') \sqsubseteq val\ x\ r\ v\ (p :: ps)$$

for arbitrary $p' :: ps'$, x, r and v. This is finally done by applying transitivity of $(\sqsubseteq)$ to *s4* and *s5*. The computation of *s5* is trivial and follows directly from the fourth argument of *Bellman*, *oep*. This is a proof that p is an optimal extension of *ps*. In order to compute *s4*, we proceed as outlined in section 3.9: we first apply optimality of *ps* to deduce that

$$val\ x'\ r'\ v'\ ps' \sqsubseteq val\ x'\ r'\ v'\ ps$$

for arbitrary $x' : State\ (S\ t)$ which are reachable and viable m steps. This is done in *s1*. Then we show that f' is point-wise smaller than f by applying monotonicity of $(\oplus)$ w.r.t. $(\sqsubseteq)$. This is encoded in *s2*. Finally we apply the monotonicity of *meas* to compute *s3* which is equal to *s4* by definition of *val*. Notice that, in the implementation of *Bellman*, *sval* is the function defined as in section 3.9.

C Optimal extensions

We have to show that, for every policy sequence $ps : PolicySeq\ (S\ t)\ n$, the policy $p = optExt\ ps : Policy\ t\ (S\ n)$ is an optimal extension of *ps*. This means showing that, for every $p' : Policy\ t\ (S\ n)$, $x : State\ t$, $r : Reachable\ x$ and $v : Viable\ (S\ n)\ x$, one has

$$val\ x\ r\ v\ (p' :: ps) \sqsubseteq val\ x\ r\ v\ (p :: ps)$$

This immediately follows from the definition of *optExt* and from the specification of *cvalmax* and *cvalargmax*. From *cvalmaxSpec*, we know that, for every good control gy', $cval\ x\ r\ v\ ps\ gy' \sqsubseteq cvalmax\ x\ r\ v\ ps$. This holds, in particular, for $gy' = p'\ x\ r\ v$:

$$cval\ x\ r\ v\ ps\ (p'\ x\ r\ v) \sqsubseteq cvalmax\ x\ r\ v\ ps$$

From *cvalargmaxSpec*, we know that $cvalmax\ x\ r\ v\ ps = cval\ x\ r\ v\ ps\ (cvalargmax\ x\ r\ v\ ps)$. Therefore

$$cval\ x\ r\ v\ ps\ (p'\ x\ r\ v) \sqsubseteq cval\ x\ r\ v\ ps\ (cvalargmax\ x\ r\ v\ ps)$$

But, by definition of *optExt*, $cvalargmax\ x\ r\ v\ ps$ is just $p\ x\ r\ v$. Therefore

$cval\ x\ r\ v\ ps\ (p'\ x\ r\ v) \sqsubseteq cval\ x\ r\ v\ ps\ (p\ x\ r\ v)$

The result follows from the definition of *cval*. In the implementation of *optExtLemma*, *s3* to *s5* are trivial consequences of *s2*. They are written explicitly here to improve understandability but we could as well define *optExtLemma* $\{t\}$ $\{n\}$ *ps* p' *x r v* to be equal to *s2* and erase the last 3 lines of the program:

```
optExtLemma : (ps : PolicySeq (S t) n) → OptExt ps (optExt ps)
optExtLemma {t} {n} ps x r v p' = s5 where
  p   : Policy t (S n);                        p   = optExt ps
  gy  : GoodCtrl t x n;                        gy  = p x r v
  y   : Ctrl t x;                              y   = ctrl gy
  av  : All (Viable n) (nexts t x y);          av  = allViable gy
  gy' : GoodCtrl t x n;                        gy' = p' x r v
  y'  : Ctrl t x;                              y'  = ctrl gy'
  av' : All (Viable n) (nexts t x y');         av' = allViable gy'
  f   : PossibleNextState x (ctrl gy) → Val;   f   = sval x r v gy ps
  f'  : PossibleNextState x (ctrl gy') → Val;  f'  = sval x r v gy' ps
  s1  : cval x r v ps gy' ⊑ cvalmax x r v ps;  s1  = cvalmaxSpec x r v ps gy'
  s2  : cval x r v ps gy' ⊑ cval x r v ps (cvalargmax x r v ps)
  s2  = replace {P = λz ⇒ (cval x r v ps gy' ⊑ z)} (cvalargmaxSpec x r v ps) s1
  s3  : cval x r v ps gy' ⊑ cval x r v ps gy;                                     s3 = s2
  s4  : meas (fmap f' (tagElem (nexts t x y'))) ⊑ meas (fmap f (tagElem (nexts t x y))); s4 = s3
  s5  : val x r v (p' :: ps) ⊑ val x r v (p :: ps);                                s5 = s4
```

D Core theory and full theory: the assumptions

We list all parameters and functions that have to be defined to specify and solve a sequential decision problem (core theory). Specifications that are not strictly necessary to instantiate the theory, for instance *viableBaseCase* from section 3, are not listed here.

```
M            : Type → Type
fmap         : {A, B : Type} → (A → B) → M A → M B
Elem         : {A : Type} → A → M A → Type
NotEmpty     : {A : Type} → M A → Type
All          : {A : Type} → (P : A → Type) → M A → Type
allElemSpec0 : {A : Type} → {P : A → Type} →
                (a : A) → (ma : M A) → All P ma → a ∈ ma → P a
tagElem      : {A : Type} → (ma : M A) → M (Σ A (λa ⇒ a ∈ ma))

Val  : Type
zero : Val
(⊕)  : Val → Val → Val
(⊑)  : Val → Val → Type

meas : M Val → Val

State  : (t : ℕ) → Type
Ctrl   : (t : ℕ) → (x : State t) → Type
nexts  : (t : ℕ) → (x : State t) → (y : Ctrl t x) → M (State (S t))
reward : (t : ℕ) → (x : State t) → (y : Ctrl t x) → (x' : State (S t)) → Val
```

$$\begin{aligned}
&\mathit{Viable} &&: (n : \mathbb{N}) \to \mathit{State}\ t \to \mathit{Type}\\
&\mathit{viableToGoodCtrl} &&: (x : \mathit{State}\ t) \to \mathit{Viable}\ (S\ n)\ x \to \mathit{GoodCtrl}\ t\ x\ n
\end{aligned}$$

$$\begin{aligned}
&\mathit{Reachable} &&: \mathit{State}\ t' \to \mathit{Type}\\
&\mathit{reachableForward} &&: (x : \mathit{State}\ t) \to \mathit{Reachable}\ x \to (y : \mathit{Ctrl}\ t\ x) \to \mathit{All}\ \mathit{Reachable}\ (\mathit{nexts}\ t\ x\ y)
\end{aligned}$$

$$\begin{aligned}
&\mathit{cvalargmax} : (x : \mathit{State}\ t) \to (r : \mathit{Reachable}\ x) \to (v : \mathit{Viable}\ (S\ n)\ x) \to\\
&\qquad\qquad (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{GoodCtrl}\ t\ x\ n
\end{aligned}$$

The following parameters and functions (full theory) are sufficient to implement a machine checkable proof that the results of the core theory are correct:

$$\begin{aligned}
&\mathit{reflexive}_{\sqsubseteq} &&: (a : \mathit{Val}) \to a \sqsubseteq a\\
&\mathit{transitive}_{\sqsubseteq} &&: (a : \mathit{Val}) \to (b : \mathit{Val}) \to (c : \mathit{Val}) \to a \sqsubseteq b \to b \sqsubseteq c \to a \sqsubseteq c\\
&\mathit{monotonePlus}_{\sqsubseteq} &&: \{a, b, c, d : \mathit{Val}\} \to a \sqsubseteq b \to c \sqsubseteq d \to (a \oplus c) \sqsubseteq (b \oplus d)
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{cvalmax} &&: (x : \mathit{State}\ t) \to (r : \mathit{Reachable}\ x) \to (v : \mathit{Viable}\ (S\ n)\ x) \to\\
& &&\ \ (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{Val}\\
&\mathit{cvalargmaxSpec} &&: (x : \mathit{State}\ t) \to (r : \mathit{Reachable}\ x) \to\\
& &&\ \ (v : \mathit{Viable}\ (S\ n)\ x) \to (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to\\
& &&\ \ \mathit{cvalmax}\ x\ r\ v\ ps = \mathit{cval}\ x\ r\ v\ ps\ (\mathit{cvalargmax}\ x\ r\ v\ ps)\\
&\mathit{cvalmaxSpec} &&: (x : \mathit{State}\ t) \to (r : \mathit{Reachable}\ x) \to\\
& &&\ \ (v : \mathit{Viable}\ (S\ n)\ x) \to (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to\\
& &&\ \ (gy : \mathit{GoodCtrl}\ t\ x\ n) \to (\mathit{cval}\ x\ r\ v\ ps\ gy) \sqsubseteq (\mathit{cvalmax}\ x\ r\ v\ ps)
\end{aligned}$$

E State-control trajectories

The implementation of *possibleStateCtrlSeqs* from section 3.11:

$$\begin{aligned}
&\mathit{possibleStateCtrlSeqs} : (x : \mathit{State}\ t) \to (r : \mathit{Reachable}\ x) \to (v : \mathit{Viable}\ n\ x) \to\\
&\qquad\qquad\qquad\qquad (ps : \mathit{PolicySeq}\ t\ n) \to M\ (\mathit{StateCtrlSeq}\ t\ n)\\
&\mathit{possibleStateCtrlSeqs}\ \{t\}\ \{n = Z\}\quad x\ r\ v\ \mathit{Nil} \qquad = \mathit{ret}\ (\mathit{Nil}\ x)\\
&\mathit{possibleStateCtrlSeqs}\ \{t\}\ \{n = S\ m\}\ x\ r\ v\ (p :: ps') =\\
&\quad \mathit{fmap}\ g\ (\mathit{bind}\ (\mathit{tagElem}\ mx')\ f)\ \textbf{where}\\
&\qquad y \quad : \mathit{Ctrl}\ t\ x; &&y \quad = \mathit{ctrl}\ (p\ x\ r\ v)\\
&\qquad mx' : M\ (\mathit{State}\ (S\ t)); &&mx' = \mathit{nexts}\ t\ x\ y\\
&\qquad av \ \ : \mathit{All}\ (\mathit{Viable}\ m)\ mx'; &&av \ \ = \mathit{allViable}\ (p\ x\ r\ v)\\
&\qquad g \quad : \mathit{StateCtrlSeq}\ (S\ t)\ m \to \mathit{StateCtrlSeq}\ t\ (S\ m); &&g \quad = ((\mathit{MkSigma}\ x\ y)::)\\
&\qquad f \quad : \Sigma\ (\mathit{State}\ (S\ t))\ (\lambda x' \Rightarrow x' \in mx') \to M\ (\mathit{StateCtrlSeq}\ (S\ t)\ m)\\
&\qquad f\ (\mathit{MkSigma}\ x'\ x'emx') = \mathit{possibleStateCtrlSeqs}\ \{n = m\}\ x'\ r'\ v'\ ps'\ \textbf{where}\\
&\qquad\quad ar : \mathit{All}\ \mathit{Reachable}\ mx'; \quad ar = \mathit{reachableForward}\ x\ r\ y\\
&\qquad\quad r' : \mathit{Reachable}\ x'; \qquad\quad\ \ r' = \mathit{allElemSpec0}\ x'\ mx'\ ar\ x'emx'\\
&\qquad\quad v' : \mathit{Viable}\ m\ x'; \qquad\quad\ \ v' = \mathit{allElemSpec0}\ x'\ mx'\ av\ x'emx'
\end{aligned}$$

F Reachability from a given state

The implementation of *reachableFromLemma* from section 4.2:

```
reachableFromLemma : (x'' : State t'') → (x : State t) → x'' ‘ReachableFrom‘ x → t'' ⩾ t
reachableFromLemma {t'' = Z}    {t = Z}    x'' x prf                            = LTEZero
reachableFromLemma {t'' = S t'} {t = Z}    x'' x prf                            = LTEZero
reachableFromLemma {t'' = Z}    {t = S m}  x'' x (prf1, prf2)                   = void (uninhabited (sym prf1))
reachableFromLemma {t'' = S t'} {t = S t'} x'' x (Left (Refl, prf2))            = eqInLTE (S t') (S t') Refl
reachableFromLemma {t'' = S t'} {t = t}    x'' x (Right (MkSigma x' (prf1, prf2))) = s2 where
  s1 : t' ⩾ t;    s1 = reachableFromLemma x' x prf1
  s2 : S t' ⩾ t;  s2 = idSuccPreservesLTE t t' s1
```

Acknowledgements

The authors thank the JFP reviewers, whose comments have lead to significant improvements of the original manuscript.

The work presented in this paper heavily relies on free software, among others on Idris, Agda, GHC, git, vi, Emacs, LaTeX and on the FreeBSD and Debian GNU/Linux operating systems. It is our pleasure to thank all developers of these excellent products.

This work was partially supported by the projects GRACeFUL (grant agreement No 640954) and CoeGSS (grant agreement No 676547), which have received funding from the European Unions Horizon 2020 research and innovation programme.

Bibliography

J. Aldred. Ethics and climate change cost-benefit analysis: Stern and after, 2009. URL `https://ideas.repec.org/p/lnd/wpaper/442009.html#cites`.

J. Allwood, V. Bosetti, N. Dubash, L. Gmez-Echeverri, and C. von Stechow. Glossary. In O. Edenhofer, R. Pichs-Madruga, Y. Sokona, E. Farahani, S. Kadner, K. Seyboth, A. Adler, I. Baum, S. Brunner, P. Eickemeier, B. Kriemann, J. Savolainen, S. Schlmer, C. von Stechow, T. Zwickel, and J. Minx, editors, *Climate Change 2014: Mitigation of Climate Change. Contribution of Working Group III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change*, pages 33–51. Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, 2014.

N. Bauer, L. Baumstark, M. Haller, M. Leimbach, G. Luderer, M. Lueken, R. Pietzcker, J. Strefler, S. Ludig, A. Koerner, A. Giannousakis, and D. Klein. REMIND: The equations, 2011. URL `https://www.pik-potsdam.de/research/sustainable-solutions/models/remind/remind-equations.pdf`.

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

R. Bird. *Introduction to Functional Programming using Haskell*. International Series in Computer Science. Prentice Hall, second edition edition, 1998.

R. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, Hemel Hempstead, 1997.

N. Botta, C. Ionescu, and E. 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, 2013a. URL `http://dblp.uni-trier.de/db/conf/mkm/cicmws2013.html#Botta13`.

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

N. Botta, P. Jansson, C. Ionescu, D. R. Christiansen, and E. Brady. Sequential decision problems, dependent types and generic solutions. *Logical Methods in Computer Science*, 13(1), Mar. 2017. .

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013. ISSN 1469-7653. . URL `http://journals.cambridge.org/article_S095679681300018X`.

J. C. Carbone, C. Helm, and T. F. Rutherford. The case for international emission trade in the absence of cooperative climate policy. *Journal of Environmental Economics and Management*, 58:266–280, 2009.

CoeGSS. Center of Excellence for Global Systems Science. http://coegss.eu/, 2015. Accessed: 2015-12-30.

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.

O. De Moor. Dynamic programming as a software component. *Proc. 3rd WSEAS Int. Conf. Circuits, Systems, Communications and Computers (CSCC 1999)*, pages 4–8, 1999.

G. Ellison. Learning, Local Interaction, and Coordination. *Econometrica*, 61 (5):1047–71, September 1993. URL `http://ideas.repec.org/a/ecm/emetrp/v61y1993i5p1047-71.html`.

G. Ellison. Basins of Attraction, Long-Run Equilibria, and the Speed of Step-by-Step Evolution. Technical report, MIT, Department of Economics, Working Paper No. 96-4, 1995. URL `http://ssrn.com/abstract=139523`.

European Comission. Proposal for a Financial Transaction Tax, 2013. URL `http://ec.europa.eu/taxation_customs/taxation-financial-sector_en#prop`.

M. Finus, E. van Ierland, and R. Dellink. Stability of climate coalitions in a cartel formation game. FEEM Working Paper No. 61.2003, 2003. URL `http://ssrn.com/abstract=447461`.

H. Gintis. The emergence of a price system from decentralized bilateral exchange. *B. E. Journal of Theoretical Economics*, 6:1302–1322, 2006.

H. Gintis. The Dynamics of General Equilibrium. *Economic Journal*, 117:1280–1309, 2007.

S. Gnesi, U. Montanari, and A. Martelli. Dynamic programming as graph searching: An algebraic approach. *Journal of the ACM (JACM)*, 28(4):737–751, 1981.

C. Goodhart. Some new directions for financial stability? Per Jacobsson lecture, Zurich, 27 June 2004, 2004. URL `http://www.bis.org/events/agm2004/sp040627.htm`.

GRACeFUL. Global systems Rapid Assessment tools through Constraint FUnctional Languages. https://www.graceful-project.eu/, 2015. Accessed: 2015-12-30.

GSDP. Global Systems Dynamics and Policy. http://www.gsdp.eu/, 2010. Accessed: 2015-12-30.

G. Hardin. The Tragedy of the Commons. *Science*, 162(3859):1243–1248, 1968.

J. Heitzig. Bottom-up strategic linking of carbon markets: Which climate coalitions would farsighted players form?, 2012. URL `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=2119219`.

C. Helm. International emissions trading with endogenous allowance choices. *Journal of Public Economics*, 87:2737–2747, 2003.

B. Holtsmark and D. E. Sommervoll. International emissions trading: Good or bad? *Economics Letters*, 117:362–364, 2012.

C. Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin, 2009.

F. E. Kydland and E. C. Prescott. Rules rather than discretion: The inconsistency of optimal plans. *Journal of Political Economy*, 85(3):473–91, June 1977. URL `https://ideas.repec.org/a/ucp/jpolec/v85y1977i3p473-91.html`.

A. Mandel, S. Fürst, W. Lass, F. Meissner, and C. Jaeger. Lagom generiC: an agent-based model of growing economies. *ECF working paper*, 1, 2009.

E. Moggi. Notions of computation and monads. *Information and computation*, 93(1): 55–92, 1991.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *Journal of Functional Programming*, 19:545–579, 2009.

F. E. L. Otto and A. Levermann. Levity — a concept for complementing climate policy strategies, 2011. URL `http://www.osti.gov/eprints/topicpages/documents/record/666/1527922.html`.

H. Peyton Young. The evolution of conventions. *Econometrica*, 61:57–84, 1993.

H. Peyton Young. *Individual Strategy and Social Structure: An Evolutionary Theory of Institutions*. Princeton University Press, 2001.

P. Raven, R. Bierbaum, and J. Holdren. Confronting climate change: Avoiding the unmanageable and managing the unavoidable. UN-Sigma Xi Climate Change Report, 2007. URL `https://www.sigmaxi.org/programs/critical-issues-in-science/un-sigma-xi-climate-change-report`.

Research Domain III, PIK. ReMIND-R. ReMIND-R is a global multi-regional model incorporating the economy, the climate system and a detailed representation of the energy sector. `http://www.pik-potsdam.de/research/sustainable-solutions/models/remind`, 2013.

T. Sandler and W. Enders. An economic perspective on transnational terrorism. *European Journal of Political Economy*, 20:301–316, 2004.

T. Sandler and D. G. Arce M. A conceptual framework for understanding global and transnational public goods for health. *Fiscal Studies*, 23:195–222, 2002.

H. J. Schellnhuber. Discourse: Earth system analysis - the scope of the challenge. In H. Schellnhuber and V. Wenzel, editors, *Earth System Analysis: Integrating Science for Sustainability*, pages 3–195. Springer, Berlin/Heidelberg, 1998.

M. Spivey. A functional theory of exceptions. *Science of computer programming*, 14(1): 25–42, 1990.

M. D. Webster. The curious role of ”learning” in climate policy: Should we wait for more data? Technical report, MIT Joint Program on the Science and Policy of Global Change, Report No. 67, 2000.

M. D. Webster. Incorporating path dependency into decision-analytic methods: An application to global climate-change policy. *Decision Analysis*, 5(2):60–75, 2008.