A computational theory of policy advice and avoidability

Nicola Botta[1], Patrik Jansson[2], Cezar Ionescu[2,1]

Abstract

We present a mathematical theory of policy advice and avoidability. More specifically, we formalize the notions of decision process, decision problem and policy, and propose a novel approach for assisting decision making. The latter is based on the idea 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 theory is both formal and constructive and makes extensive use of the correspondence between dependent types and logical propositions. Decidable judgments are obtained through computations. Thus, it is a *computational* theory. Our main motivation comes from climate impact research and our perspective is that of computing science. But the theory we propose is completely generic and we understand our contribution as one to global system science.

Keywords: dependent types, policy advice, avoidability, formalization, specification, correctness, functional languages, Idris,

1. Introduction

1.1. The need for a theory of policy advice

Besides improving our understanding of the mechanisms that determine climate impacts and of the underlying systems[3], climate impacts research has to provide answers to concrete decision problems.

[1]Potsdam Institute for Climate Impact Research
[2]Chalmers University of Technology
[3]Be these social, economic or natural

A typical example is that of emission policies. Here, what has to be decided about (or agreed upon in negotiations) are levels of greenhouse gases (GHG) emissions. There is a general agreement that current GHG emission levels should be reduced.

But it is not obvious how this goal should be achieved. Emission caps together with some form of international emissions trading is an appealing scheme but its effectiveness is in question. In a recent article entitled "International emissions trading: Good or bad?", Holtsmark & Sommervoll [1] state:

> [. . .] we find that an agreement with international emissions trading leads to increased emissions and reduced efficiency.

On the other hand, in "The case for international emission trade in the absence of cooperative climate policy", Carbone et al. [2] conclude:

> [. . .] we find that emission trade agreements can be effective [. . .]

More recently, researchers such as Holtsmark and Midttømme [3] and Heitzig [4] have argued that these conflicting messages are due to modeling shortcomings, advocating extensions such as inter-temporal optimization, modeling farsighted participants, or using methods from complex networks and systems.

A similar story can be told in other important areas in which science is expected to advise policy, such as global finance (for example, the attempt at introducing a financial transaction tax at EU level, EU-FTT), global pandemics programs to eradicate contagious diseases [5], or efforts to combat international terrorism [6].

A common trait in these application domains is that decision making is in need of rigorous scientific advice, but 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 stability, avoidability, policy – are devoid of precise, well established technical meanings. They are used in an informal, vague manner or in a normative sense. 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 "action plans"? When do they have to expect full fledged "action rules"? These questions are essential, especially for problems in which the temporal scales of the underlying decision process are not well separated from the typical times required for implementing decisions.[4]

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 a theory for sequential decision problems and policy advice. As we will see in detail in section 3, a sequential decision problem (SDP) is essentially characterized by four entities: a state space, a control space, a transition function and a reward function. In many applications all four entities explicitly depend on a discrete time.

The decision process starts in a state x_0 at an initial time t_0. The control space – the set of controls (actions, options, choices, etc.) available to the decision maker – can depend both on the initial time and state. Upon selecting a control y_0 two events take place: the system enters a new state x_1 and the decision maker receives a reward r_0.

In a deterministic decision problem, the next state x_1 is determined by a transition function which takes as arguments the time of the decision t_0, the current state x_0, and the selected control y_0. In general, the reward depends both on the "old" state and on the "new" state, and on the selected control.

In many decision problems, different controls represent different levels of consumption of resources (fuel, money) or different levels of restrictions (emission abatements) and are often associated with costs. Different current and next states often imply different levels of "running" costs or benefits (. . . of machinery, avoided damages) or outcome payoffs. Rewards depending both on the current and on the next state and on the selected controls allow one to treat all these different cases in a uniform way.

[4]This is typically the case in GHG emissions decision processes.

The idea of finite horizon SDPs is that the decision maker seeks controls that maximize the sum of the rewards collected over a finite number of steps. This characterization of sequential decision problems might appear too narrow.[5] But, as we will see in sections 2.4, it is in fact quite general. In particular, it covers the case of discounted sums and can easily be generalized to products or other binary operations.

In control theory, controls that maximize the sum of the rewards collected over a finite number of steps are called *optimal* controls. In practice, optimal controls can only be computed when a specific initial state is given and for problems in which transitions are deterministic. What is relevant for decision making – both in the deterministic case and in the non-deterministic or stochastic case – are not controls but *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 informal SDP narratives, this difference can easily be obfuscated by the ambiguity of natural languages and by the genuine difficulty of identifying, for non-trivial decision problems, suitable state and control spaces. In this case, advisors can turn to stylized SDPs (knapsack, production lines, traffic, etc.) to illustrate the difference between controls and policies. Traffic problems are particularly useful to illustrate "hidden" policy advice (first turn left, then right, ahead until traffic light, then turn right) 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

[5]Why shouldn't a decision maker be interested, for instance, in maximizing a product of rewards?

action plans", "optimal courses of action", etc.) In particular, no optimal control sequence can be computed at the time decisions have to be taken and implemented. This is simply because the control to be selected after $n > 0$ steps essentially depends on the state reached after n steps. And for non-deterministic systems, that state is not known (and cannot be computed) at the time the first decision has to be taken.[6] 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 (usually time-dependent) rule for decision making and with a guarantee that, for that particular problem and at any given time, there is no better way of making decisions given what is known (at that time) 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. SDPs in climate impact research and avoidability theory

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 [7, 8, 9]. While the theory presented in sections 3, 4 is not limited to a particular class of decision problems or applications domains, our main motivation is driven by climate impact research.

In climate impact research, sequential decision problems appear, often in a disguised form, as inter-temporal optimization problem in integrated assessment models [10, 11] but also in models of international environmental agreements [12, 13, 11, 4] and in agent-based models of economic systems

[6]There are of course particular decision problems for which an *expected* state can be computed. But fixing in advance "optimal" controls on the basis of such expected states (in other words, treating a genuinely stochastic problem as a deterministic one with a suitably modified notion of state space) is not, in general, a good idea. First, such "optimal" controls might not even be applicable in a particular realization. More importantly, such controls would not factor in the information – which state has been in fact reached after a certain number of decision steps – which *is* available in every concrete realization of a decision process. A better approach – as an alternative to the computation of a full policy – would be to consider a number of possible scenarios and compute optimal control sequences for each of them. This might provide decision makers with some guidance or what-if rules to cope with different possible future evolutions.

[14, 15, 16, 17].

For these problems and models the state and the control spaces can often be defined fairly rigorously. Minimal models of international agreements on greenhouse gas emissions, for instance, can be described in terms of a few state variable – such as selective greenhouse gas concentrations and certain gross domestic product measures – and of a few controls – e.g., greenhouse gas abatements and investments.

The transition function – the "dynamics" of the system underlying the decision process – can be affected by different kinds of uncertainty (e.g., about model parameters, empirical closures, etc.) and is often non-deterministic, stochastic or fuzzy. The framework presented in [9] 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 are: in most practical cases, it is not obvious how such functions – the fourth entity characterizing a specific SDP – should be defined. This is a limitation to the applicability of both control and game-theoretical approaches to climate impact research[7].

A common way [12, 13] 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. But there are both pragmatical and ethical concerns towards this approach, see for instance [22]. 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 [23] but also in [24] 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 green-

[7]traditionally, control-theoretical approaches focus on the temporal dimension of decision processes from the perspective of an individual decision maker. They do not explicitly account for decision problems faced by multiple players, in particular in a competitive setup. These problems are in the focus of game-theoretical approaches. The border between game-theory, control-theory and evolutionary decision making is a subject of academic research [18, 19, 20, 21].

house gases", [25]) and *adaptation* ("The process of adjustment to actual or expected climate and its effects ... to moderate harm or exploit beneficial opportunities", [25]) which 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 [9] 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 other domains than climate change. In financial markets, even after two decades of Financial Stability Reviews, for instance, unambiguous notions (let apart operational tests) of stability are still elusive [26, 27]. 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 – for instance, where significant amounts of risk are pooled and transferred through long chains of contracts – are avoidable.

1.4. Intended readership

The last remark suggests that, while our work is mainly motivated by climate impact research, our contribution is one to global system science (GSS). GSS is about developing systems, theories, languages and tools for computer-aided policy making with potentially global implications. Thus, our intended readership comprehends economists, physicists, mathematicians, and engineers.

In principle, the theory presented in sections 3 and 4 only relies on elementary notions – of sets, functions, relations, function application and composition – and does not require any advanced knowledge in either mathematics or computing science.

However, the theory is very much inspired by functional languages and makes extensive use of dependent types. In fact, modulo some syntactic sugar and a mechanism for bringing contexts in scope, sections 3 and 4 are, in fact, programs which can be type checked and compiled by the Idris system [28], see 2.14. This allows us, among others, to formalize high-level dependently typed notions and to machine check the correctness of our results, see 2.13.

Many climate scientists and global system scientists have been trained as physicists, mathematicians, or engineers and might not feel comfortable with a functional notation and with dependent types. The next section is meant for such readers. Readers with a basic understanding of dependent types can safely jump to section 3.

1.5. Outline

The paper is organized as follows: in section 2 we provide a crash-course for scientists with no background in functional programming. Here, we motivate and explain the basic notation adopted throughout this paper. We introduce the notion of dependent types and discuss the relationships between programs, types, specifications and machine checkable proofs. Fully understanding this discussion is not a pre-condition for understanding our contribution. But the discussion shall make clear why we have built our theory on the top of a dependently typed language. And perhaps it will inspire readers to further develop this approach for exploring research domains in which formal theories are found to be demanding [29, 30].

In section 3 we present our theory of SDPs and policy advice. We introduce decision processes and give examples of different kinds of uncertainty affecting decision processes and decision making. We derive the core theory and discuss the main results presented in [8, 9] from the perspective of policy advice. In particular, 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 measures. In this section we also derive a generic procedure for computing provably optimal policy sequences under different kinds of uncertainty.

In section 4 we extend the theory of SDPs and policy advice to decision problems 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 [31], mitigation and adaptation. In section 5 we draw some preliminary conclusions and in section 6 we outline future work.

2. A crash-course in functional programming for (climate) scientists

Many climate scientists, modelers in particular, are well acquainted with imperative computer programs. Somewhat simplified, an imperative program is a sequence of instructions of what a (computing) machine should do. In contrast, a functional program is a description of what a computer should compute as a mathematical function from input to output. Common to both paradigms is the ability to name and reuse patterns of computation to enable concise and precise description of complex algorithms. In this paper we use functional programming notation because it is close to mathematics and because it simplifies reasoning about programs.

We use functional programming to reduce the gap between the mathematical theory and the implementation. By insisting on using functional programming notation for our theory we obtain an implementation "for free" and we get automatic checking of partial correctness by the compiler.

Compared to using Matlab or Mathematica we gain a strong type system, in particular dependent types, which helps us describe both algorithms and their specifications in the same language.

2.1. Expressions and their types

At the core of all programming languages is a sublanguage of *expressions* like $1 + 2$, `"Hello"`, $[1, 7, 3, 8]$, etc. In functional languages this core is expressive enough to implement almost all programs you may want to write. In strongly typed languages like Idris each "valid" expression has a *type*, like $\mathbb{N}$, *String*, *List* $\mathbb{N}$, etc. The judgment $e\ :\ t$ states that the expression e has type t. Most of the power of Idris comes from its type-checker which can check these judgments for very complex expressions e and types t. In the examples below we will use a few (arbitrary but fixed) types A, B, C.

2.2. Function application and currying

In Idris (and several other functional languages like Haskell and Agda) the notation for function application is juxtaposition. You can think of it as an invisible infix operator binding more strongly than any other operator. So $f\ x$ denotes the application of the function f to the argument x.

Idris follows the usual meaning of parentheses in mathematics: to enclose a sub-expression to resolve operator precedence. The mathematical special "syntax" $f(a)$ for applying a typical mathematical function $f : A \to B$ is using parentheses in a non-standard way compared to the rest of mathematics. Fortunately, it is always possible to add extra parentheses in Idris, so $f\ (x)$ is actually also valid syntax for function application.

A function of $n > 1$ arguments is in mathematics mostly "implicitly converted" to a function taking just one $n-$tuple instead. In Idris we instead use nested function application, $(g\ x)\ y$, which can also be written $g\ x\ y$ because function application is left-associative. So g has type $A\ \to\ (B\ \to\ C)$ or simply $A\ \to\ B\ \to\ C$. This is called the *curried* form. Infix operators like $(+)$ are, just as in mathematics, a special case where a (binary) function can be written between its first and second argument: $2 + 3$.

2.3. Functions and other definitions

The ability to name and reuse expressions is at the core of programming. Idris (and basically all other programming languages) allows us to name and reuse expressions as long as we provide the type.

$$\begin{aligned}&\mathit{aNumber}\ :\ \mathbb{N}\\&\mathit{aNumber} = 1738\end{aligned}$$

Any time *aNumber* is used we can just substitute 1738. In Idris we can also have functions as expressions (called lambda-expressions):

$$\begin{aligned}&\mathit{aFun}\ :\ \mathbb{N}\ \to\ \mathbb{N}\\&\mathit{aFun} = \lambda x \Rightarrow 2 * x + 1\end{aligned}$$

or equivalently

$$\begin{aligned}&\mathit{aFun}'\ :\ \mathbb{N}\ \to\ \mathbb{N}\\&\mathit{aFun}'\ x = 2 * x + 1\end{aligned}$$

The second form is useful when we want to distinguish different cases by *pattern matching*:

$$
\begin{aligned}
&(\uparrow) : \mathbb{R} \to \mathbb{N} \to \mathbb{R}\\
&x \uparrow Z \qquad = 1\\
&x \uparrow (S\ n) = x * (x \uparrow n)
\end{aligned}
$$

The two cases (for zero and the successor of n) can be seen as equations we want to hold for the binary operator $(\uparrow)$. (Idris will match the cases from top to bottom, so if they overlap the earlier ones take control.) In addition to pattern matching this example also introduces recursion: the function being defined $(\uparrow)$ is used on the right hand side on a smaller argument $(n < S\ n)$. When defining new infix operators like $(\uparrow)$, we can also specify the precedence level to reduce the need for parentheses:

$$\textbf{infixr}\ 10 \uparrow$$

2.4. Partial application and high order functions

If an Idris function of two (or more) arguments, $g : A \to B \to C$, is applied to just one argument x we obtain a function $g\ x : B \to C$ which is a *partially applied* version of g. Thus we can view any Idris function as a 1-argument function, possibly returning a function.

We could also convert g into $h : (A, B) \to C$ by pairing up the first two arguments. Thus we can convert any (n-argument) Idris function into a 1-argument function where that one argument is an n-tuple.

For binary functions this conversion can be done generically as follows:

$$
\begin{aligned}
&\mathit{uncurry} : (A \to B \to C) \to ((A, B) \to C)\\
&\mathit{uncurry}\ f\ (a, b) = f\ a\ b
\end{aligned}
$$

The implementation is straightforward: *uncurry* takes as input a function f which takes values of type A to functions from B to C. It computes a function that takes as input a pair (a, b) by applying $f\ a$ to b. This is our first example of a *higher-order* function: a function taking another function as a parameter. The type says that *uncurry* accepts any curried 2-argument function and produces the corresponding 1-argument, pair-consuming, uncurried function.

The opposite transformation is also short and clean:

$$
\begin{aligned}
&\mathit{curry} : ((A, B) \to C) \to (A \to B \to C)\\
&\mathit{curry}\ f\ a\ b = f\ (a, b)
\end{aligned}
$$

These examples are a bit abstract, so here is a more applied example: Given a time-dependent reward function $r : \mathbb{N} \to A \to \mathbb{R}$ and a parameter *rate* : $\mathbb{R}$ we can construct a discounted reward function *dr* : $\mathbb{N} \to A \to \mathbb{R}$ by the higher-order function *discount*.

$$\begin{array}{l}
\mathit{discount} : \mathbb{R} \to (\mathbb{N} \to A \to \mathbb{R}) \to (\mathbb{N} \to A \to \mathbb{R}) \\
\mathit{discount} \;\; \mathit{rate} \;\; \mathit{reward} = \lambda t \Rightarrow \lambda x \Rightarrow (\mathit{rate} \uparrow t) * (\mathit{reward}\ t\ x)
\end{array}$$

We make heavy use of higher-order functions in our development.

2.5. Polymorphic functions and generic programs

The types presented so far have been *monomorphic*: using only specific types like $\mathbb{N}$, $\mathbb{R}$ and the example types A, B, and C. Very often certain programs turn out to work generically for a large class of types. For example, *discount*, actually works for any A and *curry* for any A, B, and C. A simpler example is the projection function *fst* for extracting the first component of a pair:

$$\begin{array}{l}
\mathit{fst} : (s, t) \to s \\
\mathit{fst} \;\; (x, y) = x
\end{array}$$

The type signature uses two *type variables* s and t and behind the scenes the definition is actually treated like this:

$$\begin{array}{l}
\mathit{fst} : \{s : \mathit{Type}\} \to \{t : \mathit{Type}\} \to (s, t) \to s \\
\mathit{fst} \;\; \{s\} \;\; \{t\} \;\; (x, y) = x
\end{array}$$

So *fst* is in fact a three-argument function taking two types and a pair and returning the first component of the pair. But the two first arguments are *implicit* arguments which can be inferred by the system in most use cases. When needed they can be supplied within { curly braces }.

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 *step*, *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.

2.6. Equality and equational reasoning

Idris supports reasoning about the equality of expressions. The claim that an expression a of type A is equal to an expression b of type B is written simply $(a = b)$. The infix operator $(=)$ used here has type $A \to B \to \mathit{Type}$

so you can see it as a family of types; for every $a : A$, and $b : B$ we have a type $(a = b)$. Almost all types in this family are empty (they have no values in them) but a few have one value written $\mathit{Refl} : (a = a)$. So 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 unitMult  : (y : ℝ) → 1 ∗ y = y
postulate assocMult : (x : ℝ) → (y : ℝ) → (z : ℝ) →
                      (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 $\mathit{a1} = \mathit{a2}$ and *p2* shows that $\mathit{a2} = \mathit{a3}$ then $(\mathit{a1} =\{\ \mathit{p1}\ \}= \mathit{a2} =\{\ \mathit{p2}\ \}= \mathit{a3}\ \mathit{QED})$ is a proof of $\mathit{a1} = \mathit{a3}$.

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

```
expLemma : (x : ℝ) → (m : ℕ) → (n : ℕ) →
           (      x ↑ m     ∗ x ↑ n = x ↑ (m + n))
baseCase : (x : ℝ) → (n : ℕ) →
           (      x ↑ Z     ∗ x ↑ n = x ↑ (Z + n))
stepCase : (x : ℝ) → (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 [32].

2.7. Dependent types

Many programming languages use types to make sure the code doesn't go wrong. But so far only a few languages (including Idris and Agda) can handle types which depend on values. Here we will describe how Idris supports dependent types.

First, note that we have already seen some dependent types in the previous subsection. For example *unitMult* $: (y : \mathbb{R}) \rightarrow (1 * y = y)$ is a function whose return type $(1 * y = y)$ depends on the value of the parameter y. In general the equality type $(x = y)$ depends on the two value parameters x and y.

Below we introduce a few other datatypes which use the dependency on values to restrict the possible values. Restrict sounds negative, but normally the restriction is used to eliminate nonsensical combinations of values, which helps to eliminate whole classes of software bugs.

2.8. Datatypes

We start with a regular (non-dependent) datatype declaration:

$$\begin{array}{l}\textbf{data}\ \mathbb{N}\ :\ \mathit{Type}\ \textbf{where}\\ \quad Z\ :\ \mathbb{N}\\ \quad S\ :\ \mathbb{N}\ \to\ \mathbb{N}\end{array}$$

This states that $\mathbb{N}$ is a type and that values of type $\mathbb{N}$ can be constructed using Z for zero and $S\ n$ for the successor of $n\ :\ \mathbb{N}$. This is the datatype of unary natural numbers which would be very inefficient to work with, but fortunately the Idris compiler often replaces operations on naturals with much more efficient operations on machine integers.

2.9. Vectors

With $\mathbb{N}$ and the syntax for **data** declarations in place we move on to the more complex example of fixed-length vectors:

$$\begin{array}{l}\textbf{data}\ \mathit{Vect}\ :\ \mathbb{N}\ \to\ \mathit{Type}\ \to\ \mathit{Type}\ \textbf{where}\\ \quad \mathit{Nil}\quad\ :\ \mathit{Vect}\ Z\ a\\ \quad \mathit{Cons}\ :\ (x\ :\ a)\ \to\ (\mathit{xs}\ :\ \mathit{Vect}\ n\ a)\ \to\ \mathit{Vect}\ (S\ n)\ a\end{array}$$

This declaration can be seen as an infinite family of simpler datatype declarations where *Vect0* A only contains 0-length vectors, etc.

$$\begin{array}{l}\textbf{data}\ \mathit{Vect0}\ :\ \mathit{Type}\ \to\ \mathit{Type}\ \textbf{where}\\ \quad \mathit{Nil0}\quad\ :\ \mathit{Vect0}\ a\\ \textbf{data}\ \mathit{Vect1}\ :\ \mathit{Type}\ \to\ \mathit{Type}\ \textbf{where}\\ \quad \mathit{Cons1}\ :(x\ :\ a)\ \to\ (\mathit{xs}\ :\ \mathit{Vect0}\ a)\ \to\ \mathit{Vect1}\ a\\ \textbf{data}\ \mathit{Vect2}\ :\ \mathit{Type}\ \to\ \mathit{Type}\ \textbf{where}\\ \quad \mathit{Cons2}\ :(x\ :\ a)\ \to\ (\mathit{xs}\ :\ \mathit{Vect1}\ a)\ \to\ \mathit{Vect2}\ a\end{array}$$

In this view it is easy to see that, even though the family as a whole (*Vect*) has two constructors, each “family member” (*Vect0*, *Vect1*, etc.) has exactly one.

A simple example of a vector based function is *head* which extracts the first element of a vector:

$$\begin{array}{l}\mathit{head}\ :\{n\ :\ \mathbb{N}\}\ \to\ \{A\ :\ \mathit{Type}\}\ \to\ \mathit{Vect}\ (S\ n)\ A\ \to\ A\\ \mathit{head}\ (\mathit{Cons}\ x\ \mathit{xs}) = x\end{array}$$

Note that *head* is only defined for non-empty vectors: vectors of length $S\ n$ for some n.

2.10. Properties as types, specifications

In regular programming the type *Bool* is often used to collect the two truth values *false* and *true*. In dependently typed programming (and constructive mathematics) we can go one step further and represent truth "values" at the type level: an empty type (called *Void* or *FALSE*) represents falsity and any inhabited type represents truth. Similarly a predicate on values of type A can be represented as a function $p : A \to \mathit{Type}$. We will use the synonym *Prop* for *Type* when used in this way.

For example, given a predicate $\mathit{Sorted} : \mathit{Vect}\ n\ A \to \mathit{Prop}$ we can partially specify a sorting function $\mathit{sort} : \mathit{Vect}\ n\ A \to \mathit{Vect}\ n\ A$ as follows:

$$
\begin{array}{l}
\mathit{SortSpec} : \mathit{Prop} \\
\mathit{SortSpec} = (n : \mathbb{N}) \to (\mathit{xs} : \mathit{Vect}\ n\ A) \to \mathit{Sorted}\ (\mathit{sort}\ \mathit{xs})
\end{array}
$$

To prove this we would implement a lemma

$$\mathit{sortLemma} : \mathit{SortSpec}$$

which is a function from vectors to "proofs of sortedness".

2.11. Existential types

Using types for propositions we can make a datatype capture the statement "there exists an x such that $P\ x$ holds". In fact, what is captured is not a classical existential quantifier but a constructive quantifier: we require a concrete witness $\mathit{wit} : a$ and a proof $\mathit{pro} : P\ \mathit{wit}$ that the proposition holds at *wit*:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exists} : \{A : \mathit{Type}\} \to (A \to \mathit{Prop}) \to \mathit{Prop}\ \textbf{where} \\
\quad \mathit{Evidence} : (\mathit{wit} : A) \to (\mathit{pro} : P\ \mathit{wit}) \to \mathit{Exists}\ P
\end{array}
$$

In addition to the logical reading you can also see *Exists P* as a general pair type where *Evidence* is the pair constructor and the two projections are *getWitness* and *getProof*.

$$
\begin{array}{llcl}
\mathit{getWitness} & : \mathit{Exists}\ \{A\}\ P & \to & A \\
\mathit{getWitness} & (\mathit{Evidence}\ \mathit{wit}\ \mathit{pro}) & = & \mathit{wit}
\end{array}
$$

$$
\begin{array}{llcl}
\mathit{getProof} & : (\mathit{evi} : \mathit{Exists}\ \{A\}\ P) & \to & P\ (\mathit{getWitness}\ \mathit{evi}) \\
\mathit{getProof} & (\mathit{Evidence}\ \mathit{wit}\ \mathit{pro}) & = & \mathit{pro}
\end{array}
$$

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

Figure 1: Curry-Howard correspondence relating Idris and logic.

Note that the second projection (*getProof*) returns a value whose type depends on the value of the first component of the pair.

When we want to refer primarily to the pair intuition we use a slight variation named *Sigma* where *Sigma* A P is isomorphic to *Exists* $\{A\}$ P. The projections are named the same but there is built-in infix syntax for the constructor: $(x ** y)\ :\ \mathit{Sigma}\ A\ B$.

```
data Sigma : (A : Type) → (A → Type) → Type where
  MkSigma : {A : Type} → {B : A → Type} →
            (a : A) → B a → Sigma A B
```

The notation $(a\ :\ A ** B\ a)$ is often used instead of *Sigma* A B for the type of a pair of a value $a\ :\ A$ and $b\ :\ B\ a$.

2.12. Programs, proofs, totality and termination

We have seen that we can represent properties as types and in this view proofs are just values of these types. We sum up the correspondence between Idris and logic in Fig. 1. We use logic-inspired notation $\exists$ for the existential quantifier datatype.

When we embed logic in a programming language like this we have to be careful with two things: totality and termination.

A total function $f\ :\ A\ \rightarrow\ B$ is defined for all type correct inputs. A partial function would be undefined on some of A. If partial functions were allowed we could use them to prove any theorem, including patently false ones.

A simple example is the partial function $\mathit{headL}\ :\ \mathit{List}\ A\ \rightarrow\ A$ which is undefined for the empty list. Using this we can easily prove a *FALSE* theorem

```
someProofs : List FALSE
someProofs = []
surprise : FALSE
surprise = headL someProofs
```

The second potential problem is non-termination - a function may cover all cases, but still fail to terminate. The extreme case is a completely circular definition

```
circular : FALSE
circular = circular
```

Idris will warn about missing cases and potentially circular definitions if we add the keyword *total* before a function definition.

2.13. Type checking and correctness

If we use the totality checker and specify our programs using properties-as-types we get the Idris type checker to check if our program satisfies its specification. To succeed in this, dependent types are crucial, both to express the desired properties of the output and to constrain the input.

2.14. How to type check and compile sections 3 and 4

The code is available on the open source code-sharing site GitHub: `https://github.com/nicolabotta/SeqDecProbs`. In particular the code for this section is in funprogintro.lidr.

3. Monadic sequential decision problems and policy advice

3.1. Deterministic decision processes

Let's go back to the decision process sketched at the beginning of section 1. We said that the process starts in an initial state x_0 at an initial discrete time t_0. Without loss of generality we can take $t_0 : \mathbb{N}$.

States. 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 real 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.

In the example of figure 2, the state space at time t_0 is simply the set $X_0 = \{ a, b, c, d, e \}$ and the starting state $x_0 = b$. In most decision processes, the state space depends on time. Again, in the example of figure 2, the state space at time $t < 3$, $t = 4$, $t = 5$ and $t > 6$ consists of X_0 – 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

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

defining the (time dependent) state space and $X\ t$ represents the state space at time t. In the signature of X 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 obvious property that the state space – the *type* $X\ t$ – depends on the *value* t!

The notion of a state space is elementary, but identifying a suitable space space for a particular application can require careful analysis and interactions between advisors and decision makers. Sometimes it is necessary to extend an initial characterization of the state space to encompass statistical data about the decision process itself. This allows decision makers to update the data while decisions are taken and to exploit the knowledge accumulated during previous decision steps.

Controls. In many decision processes, controls represent some rate of consumption (of resources which might be limited), some production or investment rate or perhaps just different energy options.

In models of international environmental agreements of the kind discussed in [12], for instance, decision makers could select some rates of abatement of CO_2 emissions or, perhaps, emission caps. In the model presented in [4], 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

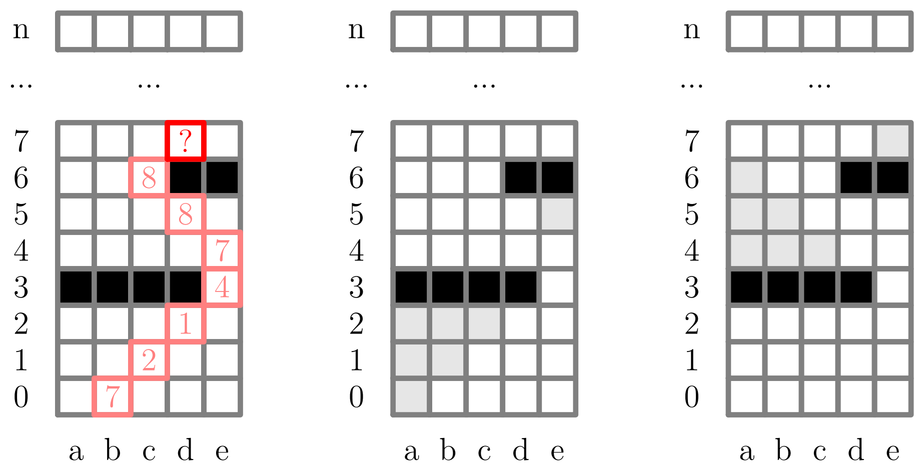

Figure 2: Possible evolution starting from b (left), states with limited viability (middle) and unreachable states (right).

disposal. [8] In general, the set of controls available to the decision maker at a given time depends both on that time and on the particular state of the process at that time. Thus, the control space can, in general, be described by a function

$$Y \; : (t \; : \; \mathbb{N}) \; \to \; (x \; : \; X \; t) \; \to \; \mathit{Type}$$

In the signature of Y we see another application of dependent types. The control space – the type of controls available at time t in x, $Y \; t \; x$ – 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

$$\mathit{step} \; : (t \; : \; \mathbb{N}) \; \to \; (x \; : \; X \; t) \; \to \; (y \; : \; Y \; t \; x) \; \to \; X \; (S \; t)$$

and $\mathit{step} \; t \; x \; y$ is the state obtained by selecting control y in state x at time t. Notice, again the type dependencies: the type of x depends on the value t;

[8] 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 states.

the type of y depends on t and on x. Finally, *step* returns a value in $X\ (S\ t)$ which is the state space at time $S\ t = 1 + t$.[9]

Rewards. As explained in the introduction, the reward obtained in a decision step depends, in general, on the current state, on the selected reward 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

$$\mathit{reward} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to (x' : X\ (S\ t)) \to \mathbb{R}$$

The return type of *reward* does not actually need to be $\mathbb{R}$. As we will see later in this section, for our theory it is enough for the return type of *reward* to be an ordered monoid.

On the other hand, the kind of generalization that can be achieved by a proper specification of the return type of *reward* is not our focus in this paper. For the rest of this article, we will trade generality for terseness of expressions and take the return type of *reward* to be $\mathbb{R}$. But we keep in mind that, here, a generalization is possible and, in fact, straightforward.

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 processes requires defining four functions: X, Y, *step* and *reward*. The first two functions define the types of the state and of the control spaces. The function *step* defines the "dynamics" of the process and *reward* its valuation.

Depending on the specific decision process, defining X, Y, *step* and *reward* might be trivial or challenging. We do not have enough experience in policy advice to discuss general methods for the specification of decision processes. In climate impact research, it is probably safe to assume that the

[9]More precisely, at step $1 + t$. Throughout this paper, we use the term "time" to simply denote a number of steps, not any "physical" (or maybe "social" ?) time. Thus, t_0, t_1, $t2$, etc. are just the first, the second, the third, etc. number of decision steps. The intuition that justifies calling the number of steps "time" is that, as implicitly coded in the signature of *step*, transitions can only increase this number.

specification of X and Y cannot be meaningfully delegated to decision makers and requires a close collaborations 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 $y : Y\ t\ x$ when in a state $x : X\ t$ at time $t : \mathbb{N}$ does not yield a unique next state $x' : X\ (S\ t)$ 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 RR (move somewhere to the right) in b at the initial time, 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, empirical closures, 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 [33] is to have *step* return a list of values instead of a single value:

$$\mathit{step} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to \mathit{List}\ (X\ (S\ t))$$

Here *List* is the type constructor already seen in section 2. It is a functor which means that we have a higher-order function

$$\mathit{fmap} : \{A, B : \mathit{Type}\} \to (A \to B) \to \mathit{List}\ A \to \mathit{List}\ B$$

which "lifts" functions from the element level to the level of lists in a way which preserves identities and composition.

We can use *fmap* to translate the uncertainty on the outcome of *step* to uncertainties on rewards:

$$\mathit{rewards} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to \mathit{List}\ \mathbb{R}$$
$$\mathit{rewards}\ t\ x\ y = \mathit{fmap}\ (\mathit{reward}\ t\ x\ y)\ (\mathit{step}\ t\ x\ y)$$

In other words, for each possible next state we have, through $\mathit{reward}\ t\ x\ y$, a corresponding possible reward. Therefore, for every $t : \mathbb{N}$, $x : X\ t$ and $y : Y\ t\ x$, 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 $t : \mathbb{N}$, $x : X\ t$ and $y : Y\ t\ x$ 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

$$step : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to List\ (X\ (S\ t))$$

with

$$step : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to Prob\ (X\ (S\ t))$$

Here *Prob A* represents a probability distribution on A: a value of type *Prob A* consists of a vector of elements of type A of arbitrary length, a vector of elements of type $\mathbb{R}$ of the same length and a proof that the latter satisfies the norming condition for probability distributions:

$$\begin{aligned}&\mathbf{data}\ Prob : Type \to Type\ \mathbf{where}\\ &\quad MkProb : \{A : Type\} \to\\ &\qquad\qquad (as : Vect\ n\ A) \to (ps : Vect\ n\ \mathbb{R}) \to sum\ ps = 1.0 \to Prob\ A\end{aligned}$$

Notice that, just like *List*, and *Vect n*, *Prob* is a functor. Its *fmap* function

$$\begin{aligned}&fmap : \{A, B : Type\} \to (A \to B) \to Prob\ A \to Prob\ B\\ &fmap\ f\ (MkProb\ as\ ps\ prf) = MkProb\ (fmap\ f\ as)\ ps\ prf\end{aligned}$$

transforms a probability distribution on A to a probability distribution on B, by applying a function that transforms elements of A into elements of B. It is easy to see that *fmap* preserves identity and function composition. As in the non-deterministic case, *step* induces, via *fmap*, a probability distribution on rewards

$$\begin{aligned}&rewards : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to Prob\ \mathbb{R}\\ &rewards\ t\ x\ y = fmap\ (reward\ t\ x\ y)\ (step\ t\ x\ y)\end{aligned}$$

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 (or an experiment, etc.) with values in A, we can represent this by a list of elements of A, i.e., an element of *List* A, and

- when, besides the possible results, we also have information about their probabilities, we can represent this by a simple probability distribution on A, i.e., an element of *Prob* 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 $M = \mathit{List}$ (no additional information), in the case of stochastic uncertainty we have $M = \mathit{Prob}$ (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) → 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 certain outcome as a singleton list:

```
certain : A → List A
certain a = [a]
```

In the case of stochastic uncertainty, we use a concentrated probability distribution:

```
certain : A → Prob A
certain a = MkProb [a] [1.0] Refl
```

The transition functions we use to represent uncertain outcomes have, as we have seen, the form

$$\mathit{step} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to M\ (X\ (S\ t))$$

It is clear how to make a transition from a given x and y at a given t: the next state will be $\mathit{step}\ t\ x\ y$. But, as opposed to the deterministic case, we now have a *collection* of states, and we cannot just apply *step* to it. Via *fmap* we can apply *step* 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 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 non-determinism:

```
reduce : List (List A) → List A
reduce = concat   -- library function that "merges" the lists together
```

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.

To summarize: in all the cases we have seen so far, and in many others, 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.

Such an M is called a *monad*, a term having its origins in category theory. Monads are ubiquitous in functional programming languages such as Haskell and Idris, and are an important way of dealing with "imperative" features such as input-output, partiality, or concurrency. The standard terminology differs a little from the one introduced here: *return* and *join* are used instead of *certain* and *reduce*, respectively.

In fact, *join* is almost always used together with *fmap* as we have explained above, and therefore both Haskell and Idris prefer to use the combinator $\gg=$ (read "bind"):

$$\begin{aligned}&(\gg=) &&: \mathit{Monad}\ M \Rightarrow M\ A \to (A \to M\ B) \to M\ B\\ &ma \gg= f &&= \mathit{join}\ (\mathit{fmap}\ f\ ma)\end{aligned}$$

Here, f is an abstraction of our *step* transition function above and $\gg=$ is the combinator that allows us to iterate the dynamical system. Thus, in *monadic decision problems* the next state is computed using $\gg=$ instead of function application. As an example, for $M = \mathit{Prob}$, $\gg=$ represents the computation of conditional probabilites.

A final remark: it is often useful to consider certainty as a limiting case of uncertainty, and, if we are to have a uniform framework, it is important that deterministic systems are special instances of monadic systems. Indeed, the type constructor

$$\begin{aligned}&\mathit{Id} : \mathit{Type} \to \mathit{Type}\\ &\mathit{Id}\ A = A\end{aligned}$$

is a monad. We have

$$\begin{aligned}&\mathit{fmap} &&: (A \to B) \to \mathit{Id}\ A \to \mathit{Id}\ B\\ &\mathit{fmap}\ f &&= f\end{aligned}$$

and

$$\begin{aligned}&\mathit{certain}\ x = x\\ &\mathit{reduce}\ \ x = x\end{aligned}$$

The bind combinator is then, as expected, just function application.

3.5. Decision problems

Consider again the decision process sketched in section 1 but now assume that the transition function is non-deterministic: $M = \mathit{List}$. We can apply the notions introduced in the previous four sections – X, Y, *step* and *reward* – to characterize the process more precisely. It starts in a state

$x_0 \; : \; X \; t_0$

at an initial time $t_0 \; : \; \mathbb{N}$. The set of controls available to the decision maker in x_0 at time t_0 is $Y \; t_0 \; x_0$. The set of states that can follow after selecting $y_0 \; : \; Y \; t_0 \; x_0$ is

$\mathit{step} \; t_0 \; x_0 \; y_0 \; : \; \mathit{List} \; (X \; (S \; t_0))$

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

$\mathit{fmap} \; (\mathit{reward} \; t_0 \; x_0 \; y_0) \; (\mathit{step} \; t_0 \; x_0 \; y_0) \; : \; \mathit{List} \; \mathbb{R}$

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:

$\mathit{meas} \; : \; \mathit{List} \; \mathbb{R} \; \rightarrow \; \mathbb{R}$

then, at least conceptually, the problem of making an optimal decision can be solved straightforwardly: for every control in $Y \; t_0 \; x_0$, we measure the value of the possible rewards for that control and select the one that yields the highest value[10].

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 $\mathit{step} \; t_0 \; x_0 \; y_0^*$ 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 (optimal or not) 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

[10]Clearly, this approach cannot, in general, be applied straightforwardly. But it surely works for finite $Y \; t_0 \; x_0$ and this is particularly relevant for applications.

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 much more flexible than decision making based on some fixed control plan. In general, taking advantage of the information that becomes available during a particular decision process allows one to achieve higher rewards. This is particularly obvious 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-priory computed "action plan". What is required here are, 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? The answer is both obvious and compelling: 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 an optimal control for step 2 to every state in $X\ t_1 = X\ (S\ t_0)$ which can be obtained by selecting y_0^* in step 1.

In fact, we can compute a function that associates to every $x_1\ :\ X\ t_1$ an optimal control $y_1^*\ :\ Y\ t_1\ x_1$. 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 : X t) → Y 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 (remember section 2.9) which is parameterized by two indexes. The first index $t : \mathbb{N}$ represents the time at which the first decision has to be taken. The second index $n : \mathbb{N}$ 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 conceptually correct but, 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 theory 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.[11] Therefore, the decision problem stated in the introduction – 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 $\mathit{meas} : M\ \mathbb{R} \to \mathbb{R}$. 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 decision theory, it is implicitly assumed that $M = \mathit{Prob}$ 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 max-min measures, or, in game-theoretical terms, “safety” strategies. Measures of possible rewards

[11]Notice, however, that such 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.

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 sequential decision problems which are not deterministic cannot, in general, be reconducted to "equivalent" deterministic problems. We have already argued for this viewpoint in the introduction. Here we can provide a more formal argument. Consider a specific decision process that is, assume that M, X, Y, and *step* are given. We can easily transform this process into a "deterministic" one

$$\begin{array}{l} \mathit{mstep} : (t : \mathbb{N}) \to (\mathit{mx} : \mathit{MX}\ t) \to (p : ((x : X\ t) \to Y\ t\ x)) \to \mathit{MX}\ (S\ t) \\ \mathit{mstep}\ t\ \mathit{mx}\ p = \mathit{join}\ (\mathit{fmap}\ (\lambda x \Rightarrow \mathit{step}\ t\ x\ (p\ x))\ \mathit{mx}) \end{array}$$

by introducing in an "equivalent" state space

$$\begin{array}{l} \mathit{MX} : (t : \mathbb{N}) \to \mathit{Type} \\ \mathit{MX}\ t = M\ (X\ t) \end{array}$$

This is possible because M is a monad and therefore has a *join* transformation. But notice that, in the new formulation, the controls are full fledged policies! This is not arbitrary or accidental: in order to apply the *step* function of the original problem[12] 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 *mstep* to be "equivalent" to *step* 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.

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

3.6. Optimal policies

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

$$\mathit{val} : (x : X\ t) \to (ps : \mathit{PolicySeq}\ t\ n) \to \mathbb{R}$$

? If $n = 0$ that is, we take zero steps, then we will collect no rewards[13] and the answer is simply zero:

$$\mathit{val}\ \{t\}\ \{n = Z\}\ x\ ps = 0$$

What if n is greater or equal to one? 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 $y : Y\ t\ x$ and an M-structure of possible next states $\mathit{step}\ t\ x\ y : M\ (X\ (S\ t))$. This is just a single next state for $M = \mathit{Id}$ (deterministic case), a list of states for $M = \mathit{List}$ (non-deterministic case) and a probability distributions on states for $M = \mathit{Prob}$ (stochastic case). In any case we know that M is a functor. Thus, we can compute, for every $x' : X\ (S\ t)$ in $\mathit{step}\ t\ x\ y$ the sum of $\mathit{reward}\ t\ x\ y\ x' : \mathbb{R}$ and of the value of making m decision steps from x' by applying the rest of the policy sequence. This yield an M-structure of $\mathbb{R}$s, one for every possible next state in $\mathit{step}\ t\ x\ y$. As discussed in the previous section, the value of such structure is measured by a measure meas:

```
$val\ \{t\}\ \{n = S\ m\}\ x\ (p :: ps) = meas\ (fmap\ f\ mx')$ **where**
  $y : Y\ t\ x$
  $y = p\ x$
  $mx' : M\ (X\ (S\ t))$
  $mx' = step\ t\ x\ y$
  $f : X\ (S\ t) \to \mathbb{R}$
  $f\ x' = reward\ t\ x\ y\ x' + val\ x'\ ps$
```

Remember that, in the introduction, we said 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

[13]In this case ps is an empty policy sequence that is, there is no policy to apply!

steps, the first one at time t. We say that $ps : PolicySeq\ t\ n$ is optimal iff for every $ps' : PolicySeq\ t\ n$ and for every $x : X\ t$, applying ps' for n decision steps from x does not yield a better value than applying ps:

$$\begin{aligned}
&OptPolicySeq : PolicySeq\ t\ n \ \to\ Prop \\
&OptPolicySeq\ \{t\}\ \{n\}\ ps = (ps' : PolicySeq\ t\ n) \ \to\ (x : X\ t) \ \to \\
&\qquad\qquad\qquad\qquad\qquad\quad So\ (val\ x\ ps' \leqslant val\ x\ ps)
\end{aligned}$$

Notice that, since $0 \leqslant 0$, the empty policy sequence (there is only one) is optimal:

$$\begin{aligned}
&nilOptPolicySeq : OptPolicySeq\ Nil \\
&nilOptPolicySeq\ ps'\ x = reflexiveFloatLTE\ 0
\end{aligned}$$

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 (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 *step* is deterministic and defined such that it simply effects the selected command: selecting L at time 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 time 2 and e at time 5 are truly "dead-ends" or, in other words, that there are no controls for these states (at time 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 associated 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 *step* 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 fulfill two additional constraints. The first constraint is that

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.[14] 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 fulfill. This is a logical consequence of the first one: if the m-th policy of a policy sequence of length n has to select controls that yield next states from which further $n - S\ m$ steps can be taken, its domain has to consist of states from which at least $n - m$ steps can be taken:

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?[15] Maybe surprisingly, the answer is positive.[16]

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 1 we consider – at the $S\ m$-th decision step at time $t = m$ – next states at time $t = S\ m$ from which at least further $n - S\ m$ steps can be taken. In

[14] And A or R in c, L, A or R in d, etc.

[15] That is, again, for arbitrary X, Y, *step* and *reward* of type $(t : \mathbb{N}) \to \mathit{Type}$, $(t : \mathbb{N}) \to (x : X\ t) \to \mathit{Type}$, $(t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to M\ (X\ (S\ t))$ and $(t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to (x' : X\ (S\ t)) \to \mathbb{R}$, respectively.

[16] We will see that a generic formulation of these two constraints is crucial for deriving a generic theory of decision making but also for formalizing the notion of avoidability.

2 we consider states at time $t = m$ from which at least $n - m$ steps can be taken. In both cases, we use a property of states – to allow a given number of further steps – to select certain subsets of the state space.

We call this property *viability*. We say that a state $x : X\ t$ 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 gray. For instance, a at time 0 is viable for at most 2 steps. At time 1, a and b are viable for 1 step and, at time 2, a, b and c are dead-ends: they are viable for 0 steps. A state which is viable for $S\ k$ steps is obviously viable for k steps: who can do more, can do less. Clearly, we can define the notion of viability recursively

Definition 1 (Viability). *Every state is viable for zero steps. A state $x : X\ t$ is viable $S\ m$ steps iff there is a control $y : Y\ t\ x$ such that $step\ t\ x\ y$ is viable m steps.*

A formalization of viability following this definition is straightforward:

```
Viable : (n : ℕ) → X t → Prop
Viable {t} Z _       = ()
Viable {t} (S m) x = ∃ (λy ⇒ Viable m (step t x y))
```

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 1 and 2:

```
Policy : (t : ℕ) → (n : ℕ) → Type
Policy t Z       = ()
Policy t (S m) = (x : X t) → Viable (S m) x →
                         (y : Y t x ** Viable m (step t x y))
```

```
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 indexes: a time t and a number of steps n. We read $p : Policy\ t\ n$ as p is a policy to make a decision at time t that supports n decision steps.

On a policy for 0 steps we have no requirements: we can take *Policy* t Z to be the singleton type.[17] But we require a policy for making a decision at time t that supports m further decision steps to associate to every state x in $X\ t$ which is viable for $S\ m$ steps a control in $Y\ t\ x$ such that *step* $t\ x\ y$ is viable for m steps.

Notice that, in contrast to the notion of policy from section 3.5, we have now a constraint on the domain (the second argument taken by *Policy*) and one on the co-domain of policies. The first constraint formalizes 1. The latter formalizes 2 and is expressed by the second element of the dependent pair returned by a policy. This consists of a control and of 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.

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 trade serendipity for viability: from a sustainability perspective it make little sense to avoid certain future states if the alternative implies 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 the constraints 1 and 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 grayed those states which, under the assumptions made in discussing the notion of viability, are not reachable. Consider, for instance, c at time 4. This state is not reachable no matter which initial state we start from. This is because from e – the only state in X 3 – we can only reach, at time 4, d (with a left move) but not a, b or c.

Computing policies for subsets of the state space that cannot be reached

[17]In Idris the singleton type is denoted by (). It contains a single element, perhaps confusingly also denoted by ().

in a decision process can imply a significant waste of resources. Consider, for instance, the decision problem sketched in figure 3. The idea here is that 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 both to the left or to the right. In other words, the decision maker faces at time 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 – marked in gray in figure 3 – that can never be reached. As the number of columns increases, the fraction of the state space that cannot be reached becomes bigger and bigger. Computing controls (optimal controls, in particular) for such states would be a waste of resources. The intuition is that (policy) advice should focus on future states which actually can happen, not on those which are unreachable. We can achieve this goal by putting forward another constraint on the domain of policies:

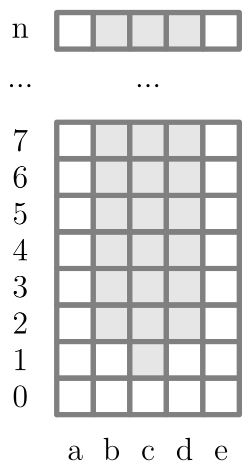

Figure 3: Bifurcation.

Property 3. *The domain of the m-th policy of a policy sequence starting at time t has to consist of states in $X\ (t + m)$ which are reachable.*

We can easily formalize reachability if we specify what it means for a state to be the successor (or, conversely, the predecessor) of another state. For the deterministic case, this is straightforward: for every time $t\ :\ \mathbb{N}$, $x\ :\ X\ t$ is a predecessor of $x'\ :\ X\ (S\ t)$ iff there exists a control $y\ :\ Y\ t\ x$ that – under *step* – brings x to x':

$$\mathit{Pred}\ :\ X\ t\ \to\ X\ (S\ t)\ \to\ \mathit{Prop}$$
$$\mathit{Pred}\ \{t\}\ x\ x' = \exists\ (\lambda y \Rightarrow x' = \mathit{step}\ t\ x\ y)$$

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

$$
\begin{array}{l}
\mathit{Reachable} \ : \ X \ t' \ \rightarrow \ \mathit{Prop} \\
\mathit{Reachable} \, \{ t' = Z \} \ \_ \quad = () \\
\mathit{Reachable} \, \{ t' = S \ t \} \ x' = \exists \ (\lambda x \Rightarrow (\mathit{Reachable} \ x, x \, `\mathit{Pred}` \, x'))
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{Policy} \ t \ (S \ m) = (x \ : \ X \ t) \ \rightarrow \ \mathit{Reachable} \ x \ \rightarrow \ \mathit{Viable} \ (S \ m) \ x \ \rightarrow \\
\qquad\qquad\qquad\quad (y \ : \ Y \ t \ x ** \mathit{Viable} \ m \ (\mathit{step} \ t \ x \ y))
\end{array}
$$

We conclude this section by noting that, in the deterministic case, we have been able to express the notions of viability and reachability and the constraints 1, 2 and 3 generically. An immediate consequence is that we can apply the framework presented in [9] to compute provably correct optimal policies for arbitrary decision problems.

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

3.8. Viability and reachability: monadic case

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

$$
\mathit{Elem} \ : \{ A \ : \ \mathit{Type} \} \ \rightarrow \ A \ \rightarrow \ M \ A \ \rightarrow \ \mathit{Prop}
$$

and a "for all" predicate

$$
\begin{array}{l}
\mathit{All} \ : \{ A \ : \ \mathit{Type} \} \ \rightarrow \ (P \ : \ A \ \rightarrow \ \mathit{Prop}) \ \rightarrow \ M \ A \ \rightarrow \ \mathit{Prop} \\
\mathit{All} \, \{ A \} \ P \ \mathit{ma} = (a \ : \ A) \ \rightarrow \ a \, `\mathit{Elem}` \, \mathit{ma} \ \rightarrow \ P \ a
\end{array}
$$

A value of type $a \, `\mathit{Elem}` \, \mathit{ma}$ represents a proof that a is contained in ma. We require *Elem* to be consistent with the monadic interface of section 3.4 in the sense that

$$
\begin{array}{l}
\mathit{containerMonadSpec1} \ : \ a \, `\mathit{Elem}` \, (\mathit{ret} \ a) \\
\mathit{containerMonadSpec2} \ : \ \{ A \ : \ \mathit{Type} \} \ \rightarrow \ (a \ : \ A) \ \rightarrow \ (\mathit{ma} \ : \ M \ A) \ \rightarrow
\end{array}
$$

$$
\begin{aligned}
&(mma : M\ (M\ A)) \rightarrow a\ \text{`}Elem\text{`}\ ma \rightarrow \\
&ma\ \text{`}Elem\text{`}\ mma \rightarrow a\ \text{`}Elem\text{`}\ (join\ mma)
\end{aligned}
$$

All formalizes the idea that all element in the container fulfill a given property. In other words, $All\ P\ ma$ implies $P\ a$ for every $a\ \text{`}Elem\text{`}\ ma$.

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 $(a : A ** a\ \text{`}Elem\text{`}\ ma)$ and we formalize the key property by requiring a function *tagElem* which takes any $a : A$ in the container into the subset:

$$
\begin{aligned}
&tagElem &&: \{A : Type\} \rightarrow (ma : M\ A) \rightarrow M\ (a : A ** a\ \text{`}Elem\text{`}\ ma) \\
&tagElemSpec &&: \{A : Type\} \rightarrow (ma : M\ A) \rightarrow fmap\ outl\ (tagElem\ ma) = ma
\end{aligned}
$$

The specification requires *tagElem* to be a tagged identity function. For the monads of the deterministic case, of the non-deterministic case and of the stochastic case *Id*, *List* and *Prob*, *tagElem* and *tagElemSpec* are easily implemented.

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 $x : X\ t$ to be viable $S\ m$ steps iff there is a control in $Y\ t\ x$ which allows the decision maker to take m further steps no matter which state will follow after selecting y:

$$
\begin{aligned}
&Viable : \{t : \mathbb{N}\} \rightarrow (n : \mathbb{N}) \rightarrow X\ t \rightarrow Prop \\
&Viable\ \{t\}\ Z\ \_ \quad = () \\
&Viable\ \{t\}\ (S\ m)\ x = \exists\ (\lambda y \Rightarrow All\ (Viable\ m)\ (step\ t\ x\ y))
\end{aligned}
$$

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

$$
\begin{aligned}
&Pred : X\ t \rightarrow X\ (S\ t) \rightarrow Prop \\
&Pred\ \{t\}\ x\ x' = \exists\ (\lambda y \Rightarrow x'\ \text{`}Elem\text{`}\ step\ t\ x\ y)
\end{aligned}
$$

With this definition, reachability is defined exactly as in the deterministic case.

3.9. Policies and policy sequences revisited

With viability and reachability in place, formalizing the notions of policy, policy sequence and value of policy sequences for the general, monadic case is almost straightforward:

```
Policy : (t : ℕ) → (n : ℕ) → Type
Policy t Z     = ()
Policy t (S m) = (x : X t) → Reachable x → Viable (S m) x →
                 (y : Y t x ** All (Viable m) (step t x y))


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


val : (x : X t) → Reachable x → Viable n x → PolicySeq t n → ℝ
val {t} {n = Z} x r v ps = 0
val {t} {n = S m} x r v (p :: ps) = meas (fmap f (tagElem mx')) where
  y   : Y t x
  y   = outl (p x r v)
  mx' : M (X (S t))
  mx' = step t x y
  av  : All (Viable m) mx'
  av  = outr (p x r v)
  f   : (x' : X (S t) ** x' `Elem` mx') → ℝ
  f   = mkf x r v y av ps
```

As in section 3.6, we first apply the policy p and compute a control y and an M-structure of possible new states mx'. But here the application of p also yields a proof that all states in mx' are viable m steps. As we will see, this proof is crucial for computing f, the function to be mapped on *tagElem* mx'. The computation of f is delegated to a function *mkf*:

```
mkf : (x  : X t) → (r : Reachable x) → (v : Viable (S m) x) →
      (y  : Y t x) → (av : All (Viable m) (step t x y)) →
      (ps : PolicySeq (S t) m) →
      (x' : X (S t) ** x' `Elem` (step t x y)) → ℝ
mkf {t} {m} x r v y av ps (x' ** x'estep) = reward t x y x' + val x' r' v' ps where
  xpx' : x `Pred` x'
  xpx' = Evidence y x'estep
```

```
r'  : Reachable x'
r'  = Evidence x (r, xpx')
v'  : Viable m x'
v'  = av x' x'estep
```

As in section 3.6, f is a function that associates to states x' in mx' the sum of the reward of the transition from x to x' and of the value of making m further decision steps from x' according to the tail of the policy sequence $p :: ps$. But in order to compute these two values for a given x', we need to provide evidences 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 x is reachable and that it is a predecessor of x'.

The first piece of evidence is for free: it is one of the arguments of *val* and the second argument of *mkf*, r. To construct the second piece of evidence, we need to know that x' is in *step* t x y and that all states in *step* t x y are viable m steps. We have an evidence of the latter in *av*. And we compute proofs that all states in mx' are indeed in mx' by applying *tagElem* to mx'. Here is where we exploit the assumption that M is not just a monad but a monadic container.

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.

The section is a summary of the results derived in [8, 9]. We refer to the appendix for technical details and focus on the main results from an applicational 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 decision makers – be they single individuals, institutions or public stakeholders – cannot be freed from such responsibility. In turn, 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[18] and a measure, *val x r v ps* gives the value – in terms of rewards – of taking n decisions starting from a state $x : X\ t$ 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 → Prop
OptPolicySeq {t} {n} ps = (ps′ : PolicySeq t n) → (x : X t) →
                          (r : Reachable x) → (v : Viable n x) →
                          So (val x r v ps′ ⩽ val x r v ps)
```

We read this formalization of optimality for policy sequences as follows: "a policy sequence ps for making n decision steps starting from a state in $X\ t$ is optimal iff for every policy sequence ps' (of the same type as ps) and for every state in $X\ t$ which is reachable and viable for n steps, 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 ps′ x r v = reflexiveFloatLTE 0
```

Bellman's optimality principle. Bellman's optimality principle [7] 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 a decision step at time 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 $p' :: ps$:

[18] That is, given M, X, Y, *step* and *reward* of suitable types.

$$
\begin{array}{l}
\mathit{OptExt} \;:\; \mathit{PolicySeq}\;(S\;t)\;m \;\rightarrow\; \mathit{Policy}\;t\;(S\;m) \;\rightarrow\; \mathit{Prop} \\
\mathit{OptExt}\;\{t\}\;\{m\}\;\mathit{ps}\;p = (p' \;:\; \mathit{Policy}\;t\;(S\;m)) \;\rightarrow\; (x \;:\; X\;t) \;\rightarrow \\
\qquad\qquad\qquad\qquad\qquad (r \;:\; \mathit{Reachable}\;x) \;\rightarrow\; (v \;:\; \mathit{Viable}\;(S\;m)\;x) \;\rightarrow \\
\qquad\qquad\qquad\qquad\qquad \mathit{So}\;(\mathit{val}\;x\;r\;v\;(p' :: \mathit{ps}) \leqslant \mathit{val}\;x\;r\;v\;(p :: \mathit{ps}))
\end{array}
$$

In other words, if p is an optimal extension of ps we know (for sure, no matter whether the decision process is deterministic, non-deterministic, stochastic, etc.) that there is no better way of making a decision now than the one 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:

$$
\begin{array}{l}
\mathit{Bellman} \;:\; (\mathit{ps} \;:\; \mathit{PolicySeq}\;(S\;t)\;m) \;\rightarrow\; \mathit{OptPolicySeq}\;\mathit{ps} \;\rightarrow \\
\qquad\qquad\;\; (p \;\;:\; \mathit{Policy}\;t\;(S\;m)) \qquad\;\rightarrow\; \mathit{OptExt}\;\mathit{ps}\;p \;\rightarrow \\
\qquad\qquad\;\; \mathit{OptPolicySeq}\;(p :: \mathit{ps})
\end{array}
$$

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 an obvious algorithm for constructing optimal policy sequences: start with the empty policy sequence – we have seen above that this is optimal – compute an optimal extension 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 a 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 $(p :: ps) \;:\; \mathit{PolicySeq}\;t\;(S\;m)$ 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 $x \ : \ X \ t$, $r \ : \ \mathit{Reachable}\ x$ and $v \ : \ \mathit{Viable}\ (S\ m)\ x$, computes a value of type

$$\mathit{So}\ (\mathit{val}\ x\ r\ v\ (p' :: ps') \leqslant \mathit{val}\ x\ r\ v\ (p :: ps))$$

The idea is to first prove that

$$\begin{array}{l} \mathit{val}\ x\ r\ v\ (p' :: ps)\ \ \leqslant \mathit{val}\ x\ r\ v\ (p\ \ :: ps) \\ \mathit{val}\ x\ r\ v\ (p' :: ps') \leqslant \mathit{val}\ x\ r\ v\ (p' :: ps) \end{array}$$

and then apply transitivity of $\leqslant$ to deduce the result. A proof that $p' :: ps$ is not better – in terms of *val* – than $p :: ps$ can be immediately computed from the assumption that p is an optimal extension of ps. A proof that $p' :: ps'$ is not better than $p' :: ps$ can be derived from optimality of ps and from the definition of *val*. The definition of *val* implies that

$$\mathit{val}\ x\ r\ v\ (p' :: ps') \leqslant \mathit{val}\ x\ r\ v\ (p' :: ps)$$

follows from

$$\mathit{meas}\ (\mathit{fmap}\ f'\ (\mathit{tagElem}\ mx')) \leqslant \mathit{meas}\ (\mathit{fmap}\ f\ (\mathit{tagElem}\ mx'))$$

where $f', f\ : (x'\ :\ X\ (S\ t) ** x'\ \text{`}\mathit{Elem}\text{`}\ mx') \ \rightarrow\ \mathbb{R}$ and $mx'\ :\ M\ (X\ (S\ t))$ are

$$\begin{array}{ll} f' & = (\mathit{mkf}\ x\ r\ v\ y'\ av')\ ps' \\ f & = (\mathit{mkf}\ x\ r\ v\ y'\ av')\ ps \\ mx' & = \mathit{step}\ t\ x\ y \end{array}$$

and *mkf* is the function defined in section 3.9.
In the above expressions, the control y and *prf* – a proof that all states in mx' are viable m steps – are obtained by applying the policy p' to x, r and v:

$$\begin{array}{ll} y & = \mathit{outl}\ (p\ x\ r\ v) \\ \mathit{prf} & = \mathit{outr}\ (p\ x\ r\ v) \end{array}$$

It is easy to see that f' is point-wise not greater than f that is $f'\ z \leqslant f\ z$ for all z. This follows from the definitions of f', f, from the optimality of ps, that is

$$\mathit{val}\ x'\ r'\ v'\ ps' \leqslant \mathit{val}\ x'\ r'\ v'\ ps$$

and from the fact that $+$ is monotone on $\mathbb{R}$ that is $y \leqslant z \rightarrow x + y \leqslant x + z$ for all $x, y, z : \mathbb{R}$. But in order to deduce

$$meas\ (fmap\ f'\ (tagElem\ mx')) \leqslant meas\ (fmap\ f\ (tagElem\ mx'))$$

from $f' \leqslant f$, we have to assume that *meas* fulfills

```
measMon : {A : Type} →
          (f : A → ℝ) → (g : A → ℝ) →
          ((a : A) → So (f a ⩽ g a)) →
          (ma : M A) → So (meas (fmap f ma) ⩽ meas (fmap g ma))
```

This monotonicity condition[19] 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

$$A = (x' : X\ (S\ t) ** x'\ `Elem`\ mx')$$

for *Bellman* to hold. In appendix Appendix A, we give a 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:

```
optExt : PolicySeq (S t) n → Policy t (S n)
postulate optExtLemma : (ps : PolicySeq (S t) n) → OptExt ps (optExt ps)
```

Then a generic backwards induction procedure for computing optimal policy sequences can be implemented as follows:

```
bi : (t : ℕ) → (n : ℕ) → PolicySeq t n
bi t Z     = Nil
bi t (S n) = (optExt ps :: ps) where
  ps : PolicySeq (S t) n
  ps = bi (S t) n
```

[19]Originally introduced by C. Ionescu in [33] in a different context: that of formalizing the notion of "vulnerability" as a *measure of possible future harm.*

It is easy to see that *bi t n* yields optimal policy sequences for every time step *t* and number of decision steps *n*. It surely does so for *n* equal to zero because, as seen above, 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:

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

Notice how the induction hypothesis – optimality of *ps* – is obtained through a recursive call to *biLemma*. 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 Appendix A. 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*:

```
optExt : PolicySeq (S t) n → Policy t (S n)
optExt {t} {n} ps = p where
  p : Policy t (S n)
  p x r v = argmax g where
    g : (y : Y t x ** All (Viable n) (step t x y)) → ℝ
    g (y ** av) = meas (fmap f (tagElem (step t x y))) where
      f : (x' : X (S t) ** x' `Elem` (step t x y)) → ℝ
      f = mkf x r v y av ps
```

In the implementation above, a best "feasible control" – a value in the co-domain of p, $(y \, : \, Y \; t \; x \ast\ast \mathit{All} \; (\mathit{Viable} \; n) \; (\mathit{step} \; t \; x \; y))$ – is obtained by maximizing the function g. For a feasible control $(y \ast\ast \mathit{av})$, $g \; (y \ast\ast \mathit{av})$ yields the value (measured by *meas*) of making a single step with y and taking n further decision steps according to the policy sequence *ps*.

Thus, the computation of an optimal extension always implies solving a maximization problem. The theories for solving such problems constitute an important sub-domain of numerical analysis, combinatorics and interval arithmetic. They go well beyond the scope of the theory presented here. We formalize the requirements needed for computing optimal extensions of arbitrary policy sequences in terms of the following specification:

$$
\begin{array}{ll}
\mathit{max} & : \{A : \mathit{Type}\} \to (f : A \to \mathbb{R}) \to \mathbb{R} \\
\mathit{argmax} & : \{A : \mathit{Type}\} \to (f : A \to \mathbb{R}) \to A \\
\mathit{maxSpec} & : \{A : \mathit{Type}\} \to (f : A \to \mathbb{R}) \to (a : A) \to \mathit{So} \; (f \; a \leqslant \mathit{max} \; f) \\
\mathit{argmaxSpec} & : \{A : \mathit{Type}\} \to (f : A \to \mathbb{R}) \to \mathit{max} \; f = f \; (\mathit{argmax} \; f)
\end{array}
$$

As usual, for *optExt* to actually compute an optimal extension p of an arbitrary policy sequence *ps* that is, for *optExtLemma* to be implementable, we only need the above specification to hold for A equal to the co-domain of p. Depending on the specific application, implementing *max*, *argmax*, *maxSpec* and *argmaxSpec* can be quite difficult or even impossible. It is certainly straightforward for the case in which the set of feasible controls is finite. We give a machine checkable proof of *optExtLemma* in appendix Appendix B.

3.11. Sequential decision problems and policy advice

In the previous section we have presented a theory for specifying and solving sequential decision problems under different kinds of uncertainty. In this theory, a decision problem is specified by giving five entities:

- A monadic container M specifying the kind of uncertainty affecting the decision problem. For problems with no uncertainties $M = \mathit{Id}$.

- A function $X \, : (t \, : \, \mathbb{N}) \to \mathit{Type}$ specifying the state space – what the decision maker can observe – for every time $t \, : \, \mathbb{N}$.

- A function $Y \, : (t \, : \, \mathbb{N}) \to (x \, : \, X \; t) \to \mathit{Type}$ specifying the control space – the options available to the decision maker – for every time $t \, : \, \mathbb{N}$ and for every state $x \, : \, X \; t$.

- A transition function $\mathit{step} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to M\ (X\ (S\ t))$ specifying the consequences of selecting a control in a given state and at a given time for every time $t : \mathbb{N}$, for every state $x : X\ t$ and for every control $y : Y\ t\ x$.

- A reward function $\mathit{reward} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to (x' : X\ (S\ t)) \to \mathbb{R}$ specifying the reward obtained by entering a new state upon selecing a control in a given state and at a given time, for every time $t : \mathbb{N}$, for every state $x : X\ t$, for every control $y : Y\ t\ x$ and for every new state $x' : X\ (S\ t)$.

The theory only requires M to be a monadic container and does not impose any restriction (or implicit assumption) on X, Y, *step* and *reward* except for those implicit in their signature.

The theory supports a disciplined, accountable approach towards policy advice: First, it explains what decision makers and advisors have to specify for policy advice to be accountable. 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 computing provably optimal policies.

The last result holds under two additional assumptions: that decision makers and advisors agree on a monotone measure $\mathit{meas} : M\ \mathbb{R} \to \mathbb{R}$ for estimating the value of uncertain rewards and that they provide *max* and *argmax* methods for solving local maximization problems that fulfill the *maxSpec* and *argmaxSpec* specification given in the last section[20].

While backwards induction – since Bellman's original contribution in 1957– has been routinely implemented and applied to a vast number of decision problems in, among others, economics, bioinformatics and computing science, 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 question:

1. Can the theory deliver more given the specification of a decision problem?

[20]The latter is, in principle, a strong assumptions. But it cannot be avoided and decision theories that do not explicitely 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.

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 are extremely useful for assisting decision making. They can be applied to refine – 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 from 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.[21]

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 [33].

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 generate an M-structure of state-control pairs. But what are sequences of state-control pairs? These can be introduced in a similar way as policy sequences:

```
data StateCtrlSeq : (t : ℕ) → (n : ℕ) → Type where
  Nil : (x : X t) → StateCtrlSeq t Z
  (::) : (x : X t ** Y t x) → 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

[21] 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.

sequences of length n. For example, for $M = \mathit{Prob}$, 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:

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

We give an implementation of *stateCtrlTrj* in appendix Appendix C. 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.

4. Policy advice and avoidability

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

$$\mathit{reward} \; : (t \; : \; \mathbb{N}) \; \to \; (x \; : \; X \; t) \; \to \; (y \; : \; Y \; t \; x) \; \to \; (x' \; : \; X \; (S \; t)) \; \to \; \mathbb{R}$$

In order to specify a decision problem, *reward* has to be defined for every time $t \; : \; \mathbb{N}$, for every state $x \; : \; X \; t$, for every control $y \; : \; Y \; t \; x$ and for every "possible" next state $x' \; : \; X \; (S \; t)$. The idea is that $\mathit{reward} \; t \; x \; y \; x'$ gives the value of selecting y in x at time t *and* then entering state x'.

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 numebr of steps. We could also try to constrain x' to take values in the support of $\mathit{step} \; t \; x \; y$.[22] But still, *reward* has to be defined for a decision problem to be specified.

In many application domains – in particular in climate impact research – it is not very difficult to specify the state spaces X, the control spaces Y and the transition function *step* of a particular decision process. 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.[23]

[22]The $x' \; : \; X \; (S \; t)$ such that $x' \; \text{`}\mathit{Elem}\text{`} \; \mathit{step} \; t \; x \; y$.

[23]A way of re-formulating this question is to ask whether rewards could be defined in terms of something less questionable.

4.1. Policy advice and 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.[24]

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 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[25] 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

[24]For instance, because – in these states – certain "climate" variables or certain "socio-economic" variables exceed critical thresholds.

[25]We should probably be more careful and replace "what it means" by "what it can mean" here: we expect – as for the case of equilibrium in game theory – that a whole family of notions is needed to capture the idea of avoidability in the context of decision theory.

maker can only move to the left, ahead or to the right, column b, c, d and e), column e at time 3 is unavoidable. This is simply because column e has no alternative: is 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 $x'' : X\ t''$ and $x : X\ t$. We explain what it means for x'' to be reachable from x by considering two cases:

$$\begin{array}{l} \mathit{ReachableFrom} : X\ t'' \to X\ t \to \mathit{Prop} \\ \mathit{ReachableFrom}\ \{t'' = Z\} \quad \{t\}\ x''\ x = (t = Z, x = x'') \\ \mathit{ReachableFrom}\ \{t'' = S\ t'\}\ \{t\}\ x''\ x = \\ \quad \mathit{Either}\ (t = S\ t', x = x'')\ (\exists\ (\lambda x' \Rightarrow (x'\ \text{`}\mathit{ReachableFrom}\text{`}\ x, x'\ \text{`}\mathit{Pred}\text{`}\ x''))) \end{array}$$

The first case is one in which t'' is equal to zero. In this case t also has to be equal to zero[26] and x has to be equal to x''. This formalizes 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, for time zero.

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 $x'' : X\ t''$ is reachable from $x : X\ t$, then it is the case that $t'' \geqslant t$:

$$\begin{array}{l} \mathit{reachableFromLemma} : (x'' : X\ t'') \to (x : X\ t) \to \\ \qquad\qquad\qquad\qquad x''\ \text{`}\mathit{ReachableFrom}\text{`}\ x \to t''\ \text{`}\mathit{GTE}\text{`}\ t \end{array}$$

We prove *reachableFromLemma* in appendix Appendix D.

[26]Remember that we are formalizing a notion of reachability in the future, not in the past. Therefore t'' cannot be smaller than t: $t'' \geqslant t$. For $t'' = Z$, $t'' \geqslant t$ implies $t = Z$.

4.3. Avoidability

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

$$\begin{aligned}&\mathit{Alternative} : (x : X\ t) \to (x' : X\ t') \to (m : \mathbb{N}) \to (x'' : X\ t') \to \mathit{Prop}\\ &\mathit{Alternative}\ x\ x'\ m\ x'' = (x''\ \text{`}\mathit{ReachableFrom}\text{`}\ x, \mathit{Viable}\ m\ x'', \mathit{Not}\ (x'' = x'))\end{aligned}$$

$$\begin{aligned}&\mathit{AvoidableFrom} : (x' : X\ t') \to (x : X\ t) \to\\ &\qquad\qquad x'\ \text{`}\mathit{ReachableFrom}\text{`}\ x \to \mathit{Viable}\ n\ x' \to \mathit{Prop}\\ &\mathit{AvoidableFrom}\ \{t'\}\ \{n\}\ x'\ x\ r\ v = \exists\ (\mathit{Alternative}\ x\ x'\ n)\end{aligned}$$

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 good – in viability terms – as x'.[27]

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

[27]Obviously, the requirement does not prevent x'' to be better than x': a state which is viable for more than n steps is certainly viable for n steps.

[28]The most obvious exception to this rule has probably been 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 – though refined – is consistent with that usage.

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 responsability 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 of course 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 particulary bad from the point of view of sustainability can be avoided.[29]. In this case the property of x' being unsustainable could be expressed – in a suitable DSL – 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':

$$\begin{aligned}
&\mathit{AvoidableFrom} : (x' : X\ t') \to (x : X\ t) \to \\
&\qquad\qquad x'\ \text{‘}\mathit{ReachableFrom}\text{‘}\ x \to (m : \mathbb{N}) \to \mathit{Prop} \\
&\mathit{AvoidableFrom}\ \{t'\}\ x'\ x\ r\ m = \exists\ (\mathit{Alternative}\ x\ x'\ m)
\end{aligned}$$

The generalization introduces a family of avoidability notions through the additional parameter m. For $m = n$ we recover the original notion. The parameter m allows one to strenghten or to weaken the viability requirements the alternative state has to fulfill. 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.

[29]Given a (factual or hypothetical) “current” state x, given that x' is reachable from x, etc.

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 built upon avoidability. In climate impact research, for instance, *mitigation* and *adaptation* [25] 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* [31]. 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'[30] 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.[31]

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 $X\ t$ from being functions of real variables!

Thus, the question of decidability has to be slightly rephrased. What we have to ask is under which additional assumption our notion of avoidability becomes decidable. Before answering this question, it is necessary to clarify

[30]Again, given a current state x, etc.

[31]With a decision procedure, one could compute avoidability orderings. For instance, a state could be defined to be less avoidable than another state if the number of its alternative states is smaller than the number of alternatives of the other state. With avoidability orderings and provided that orderings or measures of possible harm can be established, one could construct levity orderings. This kind of “operationalization” could be applied to other notions that crucially depend on avoidability.

the notion of decidability. Decidability is a trait of a predicate. A predicate is just a function that returns a type. For instance

$$\mathit{Reachable} : X\ t' \to \mathit{Prop}$$

is a predicate. Another example of a predicate is

$$\mathit{Viable} : (n : \mathbb{N}) \to X\ t \to \mathit{Prop}$$

Both *Reachable* and *Viable* return a type. But *Reachable* takes only one (explicit) argument while *Viable* takes two arguments. A predicate that takes one argument $p : A \to \mathit{Prop}$ is called decidable if for every $a : A$ one can compute a value of type $\mathit{Dec}\ (p\ a)$:

```
Dec1 : {A : Type} → (P : A → Prop) → Prop
Dec1 {A} P = (a : A) → Dec (P a)
```

Similarly, a predicate that takes two arguments $p : A \to B \to \mathit{Prop}$ is called decidable if for every $a : A$ and for every $b : B$, one can compute a value of type $\mathit{Dec}\ (p\ a\ b)$:

```
Dec2 : {A, B : Type} → (P : A → B → Prop) → Prop
Dec2 {A} {B} P = (a : A) → (b : B) → Dec (P a b)
```

The extensions for properties that take three, four, etc. argumenst are obvious. For completeness, it is useful to also introduce a synonym for *Dec*:

```
Dec0 : Prop → Prop
Dec0 P = Dec P
```

In the above definitions, *Dec* is a type constructor. For a type P, a value of type $\mathit{Dec}\ P$ can be constructed in two ways: by just providing a value of type P or by providing a function that maps every value of type P to the empty type:

```
data Dec : Prop → Prop where
  Yes : {P : Prop} → (prf    : P)         → Dec P
  No  : {P : Prop} → (contra : P → Void)  → Dec P
```

The idea is that if a predicate $p : A \to \mathit{Prop}$ is decidable, then we have, for every $a : A$ either an evidence for $p\ a$ – this is just a value of type

$p\ a$ wrapped by *Yes* – or a demonstration that an evidence for $p\ a$ yields a contradiction. This is a function of type $p\ a\ \rightarrow\ \mathit{Void}$ wrapped by *No*.

With these preliminaries it becomes clear that our notion of avoidability is decidable if we can implement a function that returns a value of type $\mathit{Dec}\ (\mathit{AvoidableFrom}\ x'\ x\ r\ v\ m)$ for every x', x, r, v and m of the appropriate types. This is – because of our definition of avoidability – a value of type

$$\mathit{Dec}\ (\exists\ (\mathit{Alternative}\ x\ x'\ m))$$

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 richer set of alternative states. Such orderings could be combined with measures of possible harm to construct specific reward function, e.g., ones that assign low rewards 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 for whatever reason questionable.

In a nutshell, decidability could allow scientific advisors to apply in an accountable fashion principles (of levity, avoidance, safety) that – for instance in climate impact research – are considered to be relevant but that, to the best of our knowledge, have not so far been operationalized.

4.5. Finite types and decidability

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

$$\begin{aligned}&\mathit{AvoidableFrom}\ :\ (x'\ :\ X\ t')\ \rightarrow\ (x\ :\ X\ t)\ \rightarrow\\ &\qquad\qquad\qquad x'\ `\mathit{ReachableFrom}`\ x\ \rightarrow\ (m\ :\ \mathbb{N})\ \rightarrow\ \mathit{Prop}\\ &\mathit{AvoidableFrom}\ \{t'\}\ x'\ x\ r\ m = \exists\ (\mathit{Alternative}\ x\ x'\ m)\end{aligned}$$

This notion explains $x' : X\ t'$ to be avoidable from $x : X\ t$ if there exists a state $x'' : X\ t'$ such that $\mathit{Alternative}\ x\ x'\ m\ x''$. Thus, a decision procedure for avoidability has to provide, for every x', x, r and m either a value of type $\exists\ (\mathit{Alternative}\ x\ x'\ m)$ or a contradiction.[32] A value of type $\exists\ (\mathit{Alternative}\ x\ x'\ m)$ is just a state x'' in $X\ t'$ together with a proof that x'' is an alternative to x'. This is a value of type $\mathit{Alternative}\ x\ x'\ m\ x''$. Thus, a minimal condition for avoidability to be decidable is that $\mathit{Alternative}\ x\ x'\ m\ x''$ is decidable.[33] The intuition is that decidability of $\mathit{Alternative}\ x\ x'\ m\ x''$ is also sufficient if $X\ t'$ is finite.

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

$$\begin{aligned}\mathit{finiteDecLemma} : \{A : \mathit{Type}\} &\to \{P : A \to \mathit{Prop}\} \to \\ \mathit{Finite}\ A &\to \mathit{Dec1}\ P \to \mathit{Dec}\ (\exists\ P)\end{aligned}$$

We read the lemma as follows: if A is a finite type and $p : A \to \mathit{Prop}$ is decidable, then $\exists\ p$ 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 $\mathit{Fin}\ n$

$$\begin{aligned}&\mathit{Finite} : \mathit{Type} \to \mathit{Prop}\\ &\mathit{Finite}\ A = \exists\ (\lambda n \Rightarrow \mathit{Iso}\ A\ (\mathit{Fin}\ n))\end{aligned}$$

We do not explain 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 *finiteDecLemma* here but just a proof sketch: the idea is that all the values of a finite type can be put in a list. Thus, if A is finite, all the values of type A can be put in a list, say $\mathit{as} : \mathit{List}\ A$. Then the question of whether there exists a value in A which fullfills a decidable predicate p can be answered by considering the elements of *as* one-by-one. If *as* is the empty list, then, no matter what p is, there will be no element in *as* that fulfills p. In this case we can easily implement a contradiction that is, a function that given a value of type $\exists\ p$ computes a value of the empty type. If *as* consists of a head a and of a tail as', then we can apply the decision procedure for $p\ a$[34]. If the procedure returns a

[32] A function that, give one such values, produces a value of type *Void*.

[33] For every x, x', etc.

[34] This procedure is obtained by applying the second argument of the lemma – a function of type $(a : A) \to \mathit{Dec}\ (p\ a)$ – to a.

Yes prf, we just return a together with *prf*. If it returns a *No contra*, we proceed analyzing as'. Eventually we will end up with a value of type $\exists\ p$ or with a value of type *Not* $(\exists\ p)$, wrapped by *Yes* and *No*, respectively.

4.6. Decidability of avoidability, continued

In the last section we have shown that, if *Alternative* $x\ x'\ m\ x''$ is decidable for every $x''\ :\ X\ t'$ and $X\ t'$ is finite, then avoidability of x' is decidable.

The next and last step is to discuss under which conditions *Alternative* $x\ x'\ m\ x''$ is decidable. This is pretty straightforward: *Alternative* $x\ x'\ m\ x''$ is just a synonym for three conditions:

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

Thus, we have to understand under which conditions x'' ‘*ReachableFrom*‘ x, *Viable* $m\ x''$ and *Not* $(x'' = x')$ are decidable.

A necessary and sufficient condition for *Not* $(x'' = x')$ to be decidable is that equality in $X\ t'$ (both x'' and x' are states in $X\ t'$) is decidable:

$$\mathit{decEqX}\ : (x\ :\ X\ t)\ \to\ (x'\ :\ X\ t')\ \to\ \mathit{Dec}\ (x = x')$$

This is a basic requirement that most practical applications fulfill: if states cannot be distinguished from each other, decision makers will have a very hard time implementing no matter which policy! What about reachability and viability? Let's look at viability first. We have introduced *Viable* in section 3.8:

```
Viable : {t : ℕ} → (n : ℕ) → X t → Prop
Viable {t} Z      _ = ()
Viable {t} (S m) x = ∃ (λy ⇒ All (Viable m) (step t x y))
```

Thus, 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 $x\ :\ X\ t$:

$$\mathit{decViable}\ : (n\ :\ \mathbb{N})\ \to\ (x\ :\ X\ t)\ \to\ \mathit{Dec}\ (\mathit{Viable}\ n\ x)$$

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

$$\mathit{decViable}\ Z\ x = \mathit{Yes}\ ()$$

For $n = S\ m$ we have decidability of an existential type. Provided $Y\ t\ x$ is finite and we have a decision procedure for *All p as* for an arbitrary M-structure *as* and for a decidable predicate p

```
decAll : {A : Type} → (P : A → Prop) → Dec1 P →
         (as : M A) → Dec (All P as)
finY : (t : ℕ) → (x : X t) → Finite (Y t x)
```

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

```
decViable {t} (S m) x = finiteDecLemma fY dAll where
  fY : Finite (Y t x)
  fY = finY t x
  dAll : Dec1 (λy ⇒ All (Viable m) (step t x y))
  dAll y = decAll (Viable m) (decViable m) (step t x y)
```

A similar argument shows that, if, again, $Y\ t\ x$ is finite and we have a decision procedure for *a ‘Elem‘ as* for arbitrary a and *as*

```
decElem : {A : Type} → (a : A) → (as : M A) → Dec (a ‘Elem‘ as)
```

then *Pred* is decidable:

```
decPred : (x : X t) → (x′ : X (S t)) → Dec (x ‘Pred‘ x′)
decPred {t} x x′ = finiteDecLemma fY dElem where
  fY : Finite (Y t x)
  fY = finY t x
  dElem : Dec1 (λy ⇒ x′ ‘Elem‘ (step t x y))
  dElem y = decElem x′ (step t x y)
```

From here and using decidability of conjunctions and disjunctions

```
decPair   : {P, Q : Prop} → Dec P → Dec Q → Dec (P, Q)
decEither : {P, Q : Prop} → Dec P → Dec Q → Dec (Either P Q)
```

it is easy to see that *ReachableFrom* is decidable, too:

```
decReachableFrom : (x″ : X t″) → (x : X t) → Dec (x″ ‘ReachableFrom‘ x)
decReachableFrom {t″ = Z} {t} x″ x = decPair dp dq where
  dp : Dec (t = Z)
  dp = decEqNat t Z
  dq : Dec (x = x″)
  $dq = decEqX\ x\ x''$
$decReachableFrom\ \{t'' = S\ t'\}\ \{t\}\ x''\ x = decEither\ dp\ dq$ **where**
  $dp\ :\ Dec\ (t = S\ t', x = x'')$
  $dp = decPair\ (decEqNat\ t\ (S\ t'))\ (decEqX\ x\ x'')$
  $dq\ :\ Dec\ (\exists\ (\lambda x' \Rightarrow (x'\ `ReachableFrom`\ x, x'\ `Pred`\ x'')))$
  $dq = finiteDecLemma\ fX\ dRP$ **where**
    $fX\ :\ Finite\ (X\ t')$
    $fX = finX\ t'$
    $dRP\ :\ Dec1\ (\lambda x' \Rightarrow (x'\ `ReachableFrom`\ x, x'\ `Pred`\ x''))$
    $dRP\ x' = decPair\ drf\ dpred$ **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* and *All* are decidable, then *Viable*, *Pred*, *ReachableFrom* are decidable and therefore avoidability is decidable.

5. Conclusions

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

The theory is motivated by decision problems in climate impact research but can obviously 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.

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.[35] The extension is based on the idea of avoidability. We have proposed a family of avoidability notions and a tentative formalization of sustainability. In the last section, we have 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 X and Y cannot be meaningfully delegated to decision makers and requires a close collaborations 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 example 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.

Appendix A. Bellman's principle

Proving *Bellman* is almost straightforward:

[35] Be this because such analyses are considered to be too simplistic or because of methodological reasons.

```
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 (p' :: ps') x r v = transitiveFloatLTE s4 s5 where
    y' : Y t x
    y' = outl (p' x r v)
    mx' : M (X (S t))
    mx' = step t x y'
    av' : All (Viable m) mx'
    av' = outr (p' x r v)
    f' : (x' : X (S t) ** x' `Elem` mx') → ℝ
    f' = mkf x r v y' av' ps'
    f : (x' : X (S t) ** x' `Elem` mx') → ℝ
    f = mkf x r v y' av' ps
    s1 : (x' : X (S t)) → (r' : Reachable x') → (v' : Viable m x') →
         So (val x' r' v' ps' ⩽ val x' r' v' ps)
    s1 x' r' v' = ops ps' x' r' v'
    s2 : (z : (x' : X (S t) ** x' `Elem` mx')) → So (f' z ⩽ f z)
    s2 (x' ** x'emx') = monotoneFloatPlusLTE (reward t x y' x') (s1 x' r' v') where
      xpx' : x `Pred` x'
      xpx' = Evidence y' x'emx'
      r' : Reachable x'
      r' = Evidence x (r, xpx')
      v' : Viable m x'
      v' = av' x' x'emx'
    s3 : So (meas (fmap f' (tagElem mx')) ⩽ meas (fmap f (tagElem mx')))
    s3 = measMon f' f s2 (tagElem mx')
    s4 : So (val x r v (p' :: ps') ⩽ val x r v (p' :: ps))
    s4 = s3
    s5 : So (val x r v (p' :: ps) ⩽ val x r v (p :: ps))
    s5 = oep p' x r v
```

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

```
So (val x r v (p' :: ps') ⩽ val x r v (p :: ps))
```

for arbitrary $p' :: ps'$, x, r and v. This is finally done by applying transitivity of $\leqslant$ 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

```
So (val x' r' v' ps' ⩽ val x' r' v' ps)
```

for arbitrary $x'\ :\ X\ (S\ t)$ 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 $+$ w.r.t. $\leqslant$. 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*, *mkf* is the function defined as in section 3.9.

Appendix B. Optimal extensions

In order to give an implementation of *optExtLemma*, it is useful to slightly rewrite the implementation of *optExt* given in section 3.10. In particular, it is useful to rewrite the local definition of g in

```
optExt : PolicySeq (S t) n → Policy t (S n)
optExt {t} {n} ps = p where
  p : Policy t (S n)
  p x r v = argmax g where
    g : (y : Y t x ** All (Viable n) (step t x y)) → ℝ
    g (y ** av) = meas (fmap f (tagElem (step t x y))) where
      f : (x' : X (S t) ** x' `Elem` (step t x y)) → ℝ
      f = mkf x r v y av ps
```

through a call to an auxiliary global function *mkg*:

```
mkg : (x : X t) → (r : Reachable x) → (v : Viable (S n) x) →
  (ps : PolicySeq (S t) n) →
  (y : Y t x ** All (Viable n) (step t x y)) → ℝ
mkg {t} {n} x r v ps yav = meas (fmap f (tagElem (step t x (outl yav)))) where
  f : (x' : X (S t) ** x' `Elem` (step t x (outl yav))) → ℝ
  f = mkf x r v (outl yav) (outr yav) ps
```

where, in the definition of *mkg*, we have again used the definition of *mkf* introduced in section 3.9. With *mkg*, the implementation of *optExt* becomes:

```
optExt : PolicySeq (S t) n → Policy t (S n)
optExt {t} {n} ps = p where
  p : Policy t (S n)
  p x r v = argmax g where
    g : (y : Y t x ** All (Viable n) (step t x y)) → ℝ
    g = mkg x r v ps
```

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

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

where we have denoted *optExt ps* by *p*. This immediately follows from the definition of *optExt* and from the specification of *max* and *argmax*. From *maxSpec*, we know that, for every feasible control *yav*, $g\ yav \leqslant max\ g$. This holds, in particular, for $yav = p'\ x\ r\ v$:

$$g\ (p'\ x\ r\ v) \leqslant max\ g$$

From *argmaxSpec*, we know that $max\ g = g\ (argmax\ g)$. Therefore

$$g\ (p'\ x\ r\ v) \leqslant g\ (argmax\ g)$$

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

$$g\ (p'\ x\ r\ v) \leqslant g\ (p\ x\ r\ v)$$

The result follows from the definition of *g*. In the implementation of *optExtLemma*, *s3* to *s6* 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 8 lines of the program:

```
optExtLemma : (ps : PolicySeq (S t) n) → OptExt ps (optExt ps)
optExtLemma {t} {n} ps p' x r v = s6 where
  p    : Policy t (S n)
  p    = optExt ps
  yav  : (y : Y t x ** All (Viable n) (step t x y))
  yav  = p x r v
  y    : Y t x
  y    = outl yav
```

```
av   : All (Viable n) (step t x y)
av   = outr yav
yav' : (y : Y t x ** All (Viable n) (step t x y))
yav' = p' x r v
y'   : Y t x
y'   = outl yav'
av'  : All (Viable n) (step t x y')
av'  = outr yav'
g    : (y : Y t x ** All (Viable n) (step t x y)) → ℝ
g    = mkg x r v ps
f    : (x' : X (S t) ** x' `Elem` (step t x y)) → ℝ
f    = mkf x r v y av ps
f'   : (x' : X (S t) ** x' `Elem` (step t x y')) → ℝ
f'   = mkf x r v y' av' ps
s1   : So (g yav' ⩽ max g)
s1   = maxSpec g yav'
s2   : So (g yav' ⩽ g (argmax g))
s2   = replace {P = λz ⇒ So (g yav' ⩽ z)} (argmaxSpec g) s1
s3   : So (g yav' ⩽ g yav)
s3   = s2
s4   : So (mkg x r v ps yav' ⩽ mkg x r v ps yav)
s4   = s3
s5   : So (meas (fmap f' (tagElem (step t x y'))) ⩽ meas (fmap f (tagElem (step t x y))))
s5   = s4
s6   : So (Val x r v (p' :: ps) ⩽ Val x r v (p :: ps))
s6   = s5
```

Appendix C. State-control trajectories

```
stateCtrlTrj : (x : X t) → (r : Reachable x) → (v : Viable n x) →
               (ps : PolicySeq t n) → M (StateCtrlSeq t n)
stateCtrlTrj {t} {n = Z} x r v Nil = ret (Nil x)
stateCtrlTrj {t} {n = S m} x r v (p :: ps') =
  fmap g (bind (tagElem mx') f) where
    y   : Y t x
    y   = outl (p x r v)
    mx' : M (X (S t))
    mx' = step t x y
    av  : All (Viable m) mx'
av   = outr (p x r v)
g    : StateCtrlSeq (S t) n → StateCtrlSeq t (S n)
g    = ((x ** y)::)
f    : (x' : X (S t) ** x' `Elem` mx') → M (StateCtrlSeq (S t) m)
f (x' ** x'estep) = stateCtrlTrj {n = m} x' r' v' ps' where
  xpx' : x `Pred` x'
  xpx' = Evidence y x'estep
  r'   : Reachable x'
  r'   = Evidence x (r, xpx')
  v'   : Viable m x'
  v'   = av x' x'estep
```

Appendix D. Reachability from a given state

```
reachableFromLemma : (x'' : X t'') → (x : X t) → x'' `ReachableFrom` x → t'' `GTE` t
reachableFromLemma {t'' = Z} {t = Z}    x'' x prf =
  LTEZero
reachableFromLemma {t'' = Z} {t = S m} x'' x (prf1, prf2) =
  void (uninhabited u) where
    u : Z = S m
    u = trans (sym prf1) Refl
reachableFromLemma {t'' = S t'} {t = Z} x'' x prf =
  LTEZero
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 (Evidence x' (prf1, prf2))) =
  s3 where
    s1 : t' `GTE` t
    s1 = reachableFromLemma x' x prf1
    s3 : S t' `GTE` t
    s3 = idSuccPreservesLTE t t' s1
```

Acknowledgements

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.

[1] Bjart Holtsmarkark and Dag Einar Sommervoll, International emissions trading: Good or bad?, Economics Letters 117 (2012) 362364.

[2] Jared C. Carbone and Carsten Helm and Thomas F. Rutherford, The case for international emission trade in the absence of cooperative climate policy, Journal of Environmental Economics and Management 58 (2009) 266280.

[3] Bjart Holtsmarkark and Kristoffer Midttømme, The dynamics of linking permit markets, working paper (2013).
URL `http://www.sv.uio.no/econ/english/research/unpublished-works/working-papers/2015/memo022015.html`

[4] 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`

[5] T. Sandler, D. G. A. M., A conceptual framework for understanding global and transnational public goods for health, Fiscal Studies 23 (2002) 195–222.

[6] T. Sandler, W. Enders, An economic perspective on transnational terrorism, European Journal of Political Economy 20 (2004) 301–316.

[7] R. Bellman, Dynamic Programming, Princeton University Press, 1957.

[8] N. Botta, C. Ionescu, 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)", Vol. 1010 of CEUR Workshop Proceedings, CEUR-WS.org, 2013.
URL `http://dblp.uni-trier.de/db/conf/mkm/cicmws2013.html#Botta13`

[9] N. Botta, P. Jansson, C. Ionescu, D. R. David R. Christiansen, E. Brady, Sequential decision problems, dependent types and generic solutions, Submitted to LMCS (Logical Methods in Computer Science), August 2014 (2014).

[10] P. Research Domain III, 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`.

[11] B. N., B. L., H. M., L. M., L. G., L. M., P. R., S. J., L. S., K. A., G. A., K. D., REMIND: The equations (2011).

[12] M. Finus, E. van Ierland, R. Dellink, Stability of Climate Coalitions in a Cartel Formation Game, FEEM Working Paper No. 61.2003 (2003). URL `http://ssrn.com/abstract=447461`

[13] C. Helm, International emissions trading with endogenous allowance choices, Journal of Public Economics 87 (2003) 27372747.

[14] H. Gintis, The emergence of a Price System from Decentralized Bilateral Exchange, B. E. Journal of Theoretical Economics 6 (2006) 1302–1322.

[15] H. Gintis, The Dynamics of General Equilibrium, Economic Journal 117 (2007) 1280–1309.

[16] N. Botta, A. Mandel, M. Hofmann, S. Schupp, 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, 2013.

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

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

[19] H. Peyton Young, The evolution of conventions, Econometrica 61 (1993) 57–84.

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

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

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

[23] P. Raven, R. Bierbaum, 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`

[24] H. J. Schellnhuber, Discourse: Earth system analysis - the scope of the challenge, in: H. Schellnhuber, V. Wenzel (Eds.), Earth System Analysis: Integrating Science for Sustainability, Springer, Berlin/Heidelberg, 1998, pp. 3–195.

[25] A. J.M., V. Bosetti, N. Dubash, L. Gmez-Echeverri, 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, J. Minx (Eds.), Climate Change 2014: Mitigation of Climate Change. Contribution of Working Group III to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change, Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, 2014, pp. 33–51.

[26] Defining financial stability.
URL `http://www.clevelandfed.org/research/topics/finstability/definition.cfm`

[27] 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`

[28] E. Brady, Idris, a general purpose dependently typed programming language: Design and implementation.
URL `http://www.idris-lang.org/documentation/`

[29] C. Ionescu, P. Jansson, Dependently-typed programming in scientific computing: Examples from economic modelling, in: R. Hinze (Ed.), 24th Symposium on Implementation and Application of Functional Languages (IFL 2012), Vol. 8241 of LNCS, Springer, 2013, pp. 140–156. doi:10.1007/978-3-642-41582-1_9.

[30] C. Ionescu, P. Jansson, Testing versus proving in climate impact research, in: Proc. TYPES 2011, Vol. 19 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013, pp. 41–54. doi:10.4230/LIPIcs.TYPES.2011.41.

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

[32] S.-C. Mu, H.-S. Ko, P. Jansson, Algebra of programming in Agda: dependent types for relational program derivation, Journal of Functional Programming 19 (2009) 545–579. doi:10.1017/S0956796809007345.

[33] C. Ionescu, Vulnerability modelling and monadic dynamical systems, Ph.D. thesis, Freie Universität Berlin (2009).