An Algebra of Sequential Decision Problems

Extended Abstract

Robert Krook
Computer Science and Engineering
University of Gothenburg
Sweden
guskrooro@student.gu.se

Patrik Jansson
Computer Science and Engineering
Chalmers University of Technology
Sweden
patrik.jansson@chalmers.se

ACM Reference Format:
Robert Krook and Patrik Jansson. 2019. An Algebra of Sequential Decision Problems: Extended Abstract. In *Proceedings of ACM SIGPLAN Workshop on Type-Driven Development (TyDe'19).* ACM, New York, NY, USA, 3 pages.

1 Introduction

Sequential decision processes and problems are a well established concept in decision theory, with the Bellman equation [1] as a popular choice for describing them. Botta et al [4] have formalised the notion of such problems in Idris. Using dependent types to bridge the gap between description and implementation of complex systems, for purposes of simulation, has been shown to be a good choice [5]. They have illustrated how to use their formulation to model e.g. climate impact research [3], a very relevant problem today.

Evidence based policy making (when dealing with climate change or other global systems challenges), requires computing policies which are verified to be correct. There are several possible notions of "correctness" for a policy: computing feasible system trajectories through a state space, avoiding "bad" states, or even computing optimal policys. The concepts of feasibility and avoidability have been formalised and presented in Botta et al. [2].

Although motivated by the complexity of modelling in climate impact research, we focus on simpler examples of sequential decision processes and how to combine them.

Examples: Assume that we have a process $p : SDProc$ that models something moving through a 1-D coordinate system with a natural number as the state and $+1$, 0, and -1 as actions. If the circumstances change and we need to model how something moves in a 2-D coordinate system, it would be convenient if we could reuse the one dimensional system and get the desired system for free. We seek a combinator $\_\times_{SDP}\_ : SDProc \to SDProc \to SDProc$ such that

$$p^2 = p \times_{SDP} p$$

Both p and p^2 use a fixed state space, but we can also handle time dependent processes. Assume $p' : SDProcT$ is similar to p but time dependent: not all states are available at all times, meaning p' is more restricted in the moves it can make. If we want to turn this into a process that can also move around in a second dimension, we want to be able to reuse both p' and p. We can use a combinator $\_\times^T_{SDP}\_ : SDProcT \to SDProcT \to SDProcT$ together with the trivial embedding of a time independent, as a time dependent, process $embed : SDProc \to SDProcT$.

$$p^{2\prime} = p' \times^T_{SDP} (embed\ p)$$

As a last example consider the case where we want a process that moves either in a 3-D coordinate system $p^3 = p^2 \times_{SDP} p$ or in $p^{2\prime}$. You could think of this as choosing a map in a game. Then we would want a combinator $\_\uplus^T_{SDP}\_ : SDProcT \to SDProcT \to SDProcT$ such that

$$game = p^{2\prime} \uplus^T_{SDP} (embed\ p^3)$$

These combinators, and more, make up an *Algebra of SDPs*.

2 Sequential Decision Problems

First, we formalise the notion of a Sequential Decision *Process* in Agda. A process always has a *state*, and depending on what that state is there are different *controls* that describe what actions are possible in that state. The last component of a sequential decision process is a function *step* that when applied to a state and a control for that state returns the next state. To better see the type structure we introduce a type synonym for the family of controls depending on a state:

```
Con : Set → Set₁
Con S = S → Set
```

and for the the type of step functions defined in terms of a state and a family of controls on that state:

```
Step : (S : Set) → Con S → Set
Step S C = (s : S) → C s → S
```

With these in place we define a record type for SDPs:

```
record SDProc : Set1 where
  constructor SDP
  field State   : Set
        Control : Con State
        step    : Step State Control
```

We can extend this idea of a sequential decision *process* to that of a *problem* by adding an additional field *reward*.

```
reward : (x : State) → Control x → Val
```

TyDe'19, August 18, 2019, Berlin, Germany
2019.

where *Val* is often $\mathbb{R}$. From the type we conclude that the reward puts a value on the steps taken by the step function, based on the state transition and the control used. The problem becomes that of finding the sequence of controls that produces the highest sum of rewards. Or, in more realistic settings with uncertainty (which can be modelled by a monadic step function), finding a sequence of *policies* which maximises the *expected* reward. The system presented here aims at describing finite horizon problems, meaning that the sum of rewards is over a finite list. Furthermore, rewards are usually discounted the as time passes. One action *now* is worth more than the same action a few steps later. Rewards, and problems, are not the focus of this abstract but are mentioned for completeness.

A policy is a function from states to controls:

```
Policy : (S : Set) → Con S → Set
Policy S C = (s : S) → C s
```

Given a list of policies to apply, one for each time step, we can compute the trajectory of a process from a starting state. Here the $\#_{st}$ and $\#_{sf}$ functions extract the state and step component from the *SDProc* respectively.

```
trajectory :     {n : ℕ} → (p : SDProc)
           →     Vec (Policy (#st p) (#c p)) n
           →     #st p → Vec (#st p) n
trajectory sys [] x₀ = []
trajectory sys (p :: ps) x₀ = x₁ :: trajectory sys ps x₁
  where x₁ : #st sys
        x₁ = (#sf sys) x₀ (p x₀)
```

As an example of a trajectory computation we return to the one dimensional process *1d-sys* (called just *p* in the intro) and an example policy sequence *pseq*. Ideally *pseq* is the result of an optimization computed using Bellmans backwards induction. Here we just illustrate one trajectory:

```
pseq = tryleft :: tryleft :: right :: stay :: right :: []
test1 : trajectory 1d-sys pseq 0 ≡ 0 :: 0 :: 1 :: 1 :: 2 :: []
test1 = refl
```

In an applied setting many trajectories would be computed to explore the system behaviour. This brief example is fully presented in an accompanying technical report [6].

In this abstract we focus on non-monadic, time-independent, sequential decision processes, but the algebra extends nicely to the more general case.

3 The Product Combinator

To compute p^2 we need to define a *product* combinator for SDPs. We illustrate what this combinator does in Figure 1. The state of the product of two processes is the cartesian product of the two separate states.

Given two control families, we can compute the control family for pairs of states. The inhabitants (the controls) of each family member are pairs of controls for the two state components.

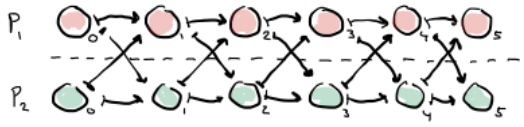

Figure 1. The product process holds components of both states and applies the step function to both components simultaneously. Each component of the next state has two incoming arrows as the policy that computes the control that is used has access to both components of the previous state.

```
_×C_ : {S₁ S₂ : Set} →
       Con S₁ → Con S₂ → Con (S₁ × S₂)
(C₁ ×C C₂) (s₁ , s₂) = C₁ s₁ × C₂ s₂
```

Given two step functions we can define a new step function for the product process by returning the pair computed by applying the individual step functions to the corresponding components of the input.

```
_×sf_ : {S₁ S₂ : Set} {C₁ : Con S₁} {C₂ : Con S₂}
      → Step S₁ C₁ → Step S₂ C₂
      → Step (S₁ × S₂) (C₁ ×C C₂)
(sf₁ ×sf sf₂) (s₁ , s₂) (c₁ , c₂) = (sf₁ s₁ c₁ , sf₂ s₂ c₂)
```

Finally, we can compute the product of two sequential decision processes by applying the combinators componentwise.

```
_×SDP_ : SDProc → SDProc → SDProc
(SDP S₁ C₁ sf₁) ×SDP (SDP S₂ C₂ sf₂)
  = SDP (S₁ × S₂) (C₁ ×C C₂) (sf₁ ×sf sf₂)
```

To illustrate how the combinator works we apply it to the system (*1d-sys*) mentioned previously.

```
2d-system = 1d-sys ×SDP 1d-sys
```

Now *2d-system* is a process of two dimensions rather than one, as illustrated by the type of *test2*.

```
2d-pseq = zipWith _×P_ pseq pseq
test2 : trajectory 2d-system 2d-pseq (0 , 5)
    ≡ (0 , 4) :: (0 , 3) :: (1 , 4) :: (1 , 4) :: (2 , 5) :: []
test2 = refl
```

4 Wrapping up

In the technical report [6] we present more combinators for time dependent and time independent processes and policies. We implement the example of a coordinate system described above, and make it even more precise as a time dependent process. Future work includes generalising to monadic SDPs and applying our combinators to the green house gas emission problem [3].

We thank the anonymous reviewers for their helpful comments and the Agda developers for a great tool!

References

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

[2] Nicola Botta, Patrik Jansson, and Cezar Ionescu. 2017. Contributions to a computational theory of policy advice and avoidability. *Journal of Functional Programming* 27 (2017), 1–52. https://doi.org/10.1017/S0956796817000156

[3] N. Botta, P. Jansson, and C. Ionescu. 2018. The impact of uncertainty on optimal emission policies. *Earth System Dynamics* 9, 2 (2018), 525–542. https://doi.org/10.5194/esd-9-525-2018

[4] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 05 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[5] Cezar Ionescu and Patrik Jansson. 2013. Dependently-typed programming in scientific computing: Examples from economic modelling. In *24th Symposium on Implementation and Application of Functional Languages (IFL 2012) (LNCS)*, Ralf Hinze (Ed.), Vol. 8241. Springer, 140–156. https://doi.org/10.1007/978-3-642-41582-1_9

[6] Robert Krook and Patrik Jansson. 2019. *An Algebra of Sequential Decision Problems*. Technical Report. Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden. http://www.cse.chalmers.se/~patrikj/papers/AlgSDP_Krook_Jansson_2019_TechReport.pdf.