An Algebra of Sequential Decision Problems

Technical Report

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: Technical Report. In *Proceedings of ACM SIGPLAN Workshop on Type-Driven Development (TyDe'19).* ACM, New York, NY, USA, 12 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*.

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

$$reward \; : \; (x \; : \; State) \; \rightarrow \; Control \; x \; \rightarrow \; Val$$

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.

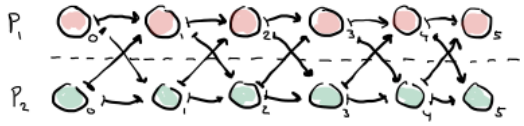


Figure 1. The product process holds components of both states and applies the step function to both components simultaneously. Each component of the next state has two incoming arrows as the policy that computes the control that is used has access to both components of the previous state.

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.

```
_×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.

5 Technical Report

The rest of this report makes up the technical report. The technical report describes further combinators, discusses time dependent processes and provide more thorough examples.

The work described in this text is the result of a project carried out at Gothenburg University by Robert Krook, under the supervision of Patrik Jansson. Robert has worked independently and met once a week with Patrik over the course of 8 weeks to discuss progress, potential bottlenecks and what to do next. Patrik has been an invaluable source of information, both regarding sequential decision problems and how to write a scientific text.

6 Example

Let us consider a sequential decision process where the state space is a one dimensional coordinate system represented by natural numbers.

```
1d-state : Set
1d-state = ℕ
```

Seeing how the state space are the natural numbers we emphasize that the coordinate system has only positive coordinates. In any given state, generally, we can choose to either increment, decrement or do nothing to the state. In the edge case where the state is *0* we can not decrement the state. We can encode this control as a type family in Agda.

```
data 1d-control : 1d-state → Set where
  Right : {n : 1d-state} → 1d-control n
  Stay  : {n : 1d-state} → 1d-control n
  Left  : {n : 1d-state} → 1d-control (suc n)
```

We implement the step function by pattern matching on the control. In the case of the *Left* control Agda recognises that the state must be a sucessor. We increment or decrement the state accordingly and leave it unchanged for the *Stay* control.

```
1d-step : (x : 1d-state) → 1d-control x → 1d-state
1d-step x       Right = suc x
1d-step x       Stay  = x
1d-step (suc x) Left  = x
```

We define a policy to be a function that given a state select a control. The policies right, stay and tryleft are all policies of this kind. tryleft is special in the sense that if the state is zero it will do nothing, as it can not go left.

```
1d-Policy = (x : 1d-state) → 1d-control x

right stay tryleft : 1d-Policy
right _    = Right
stay  _    = Stay
tryleft zero    = Stay
tryleft (suc s) = Left
```

We can parameterise a policy over a coordinate and define a policy that will select controls that moves the system towards this coordinate.

```
towards : ℕ → 1d-Policy
towards goal n with compare n goal
... | less _ _    = Right
... | equal _     = Stay
... | greater _ _ = Left
```

With the three components state, control and step, we can instantiate a sequential decision process.

```
1d-sys : SDProc
1d-sys = SDP 1d-state 1d-control 1d-step
```

A policy sequence is now just a vector of policies.

```
pseq : PolicySeq (#st 1d-sys) (#c 1d-sys) 5
pseq = tryleft :: tryleft :: right :: stay :: right :: []
```

We can evaluate the system using this sequence, starting from different points. We can use ≡ and *refl* to assert that the system behaves as intended.

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

test2 : trajectory 1d-sys pseq 5 ≡ 4 :: 3 :: 4 ::
                                   4 :: 5 :: []
test2 = refl
```

We can use the clever policy to steer the process towards a goal.

```
test3 : trajectory 1d-sys (replicate (towards 5)) 2 ≡
        3 :: 4 :: 5 :: 5 :: 5 :: []
test3 = refl
```

To turn a process into a problem we need to introduce a notion of a goal, described by a *reward* function. For our example we define the reward function to be parameterised over a target coordinate. The reward function could then reward a proposed step based on how close to the target it lands.

```
1d-reward : 1d-state
          → (x : 1d-state) → 1d-control x
          → 1d-state → ℕ
1d-reward target x₀ y x₁
  = if distance target x₁ < distance target x₀
    then 2
    else (if distance target x₀ < distance target x₁
          then 0
          else 1)
```

We can redefine the sequential decision process above to be a sequential decision problem simply by instantiating the *SDProb* record.

```
problem : 1d-state → SDProblem
problem target
  = SDProb 1d-state 1d-control
           1d-step (1d-reward target)
```

7 Combinators for sequential decision processes

Now that we've seen an example of a sequential decision process and are getting comfortable with the concept, it is suitable to move forward and see what we can do with processes. This section will explore different ways sequential decision processes can be combined in order to produce more sophisticated processes.

We already defined the product combinator, and before we move on to additional combinators we'd like to make a few notes on the product combinator. An observation to be made is that in order for the new system to exist in any state, it has to hold components of both prior states. This has the consequence that if the state space of one of the prior processes is empty, the new problems state space is also empty. Similarly, if one of the components reaches a point where there are no available controls, and thus can not progress, the other component will not be able to progress either.

Looking back at the example of the one dimensional coordinate system we find ourselves wondering if we would now indeed get a process of a two dimensional coordinate system seemingly for free. The answer, unsurprisingly, is yes.

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

In section 11 we introduce combinators for policy sequences. Here we use the product combinator to produce a policy sequence that is compatible with the new process.

```
2d-pseq : PolicySeq (#st 2d-system) (#c 2d-system) 5
2d-pseq = zipWith _×P_ pseq pseq
```

And now we can evaluate this new process like we did with the one dimensional system.

```
run2d = trajectory 2d-system 2d-pseq

2d-test1 : run2d (0 , 5)
         ≡
         (0 , 4) :: (0 , 3) :: (1 , 4) ::
         (1 , 4) :: (2 , 5) :: (2 , 5) :: []
2d-test1 = refl

2d-test2 : run2d (5 , 5)
         ≡
         (4 , 4) :: (3 , 3) :: (4 , 4) ::
         (4 , 4) :: (5 , 5) :: (5 , 5) :: []
2d-test2 = refl
```

Functional programmers will often find they are in need of a unit, e.g when using *reduce* or other frequently appearing constructs from the functional paradigm. Before we begin implementing a unit for the product case we want to clarify what we mean by a unit. A unit to a process is one that when combined with another process, produces a process where the change at each step is exactly that of the other process.

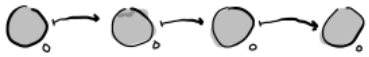


Figure 2. Illustration of the singleton process. The subscript $_0$ is meant to indicate that the state remains the same when the process advances.

What we are after is a process that will not carry any extra information, or rather one that can not alter the information it carries. Recall that in order for the state space of the product process to not be empty, both state spaces of the separate processes has to be non-empty. In order to call the step function the control space also has to be inhabited. In an effort to minimise the information the unit carries we declare its state space and control space to be singletons. The step function becomes a constant function that given the only state and the only control, will return the only state.

```
singleton : SDProc
singleton = record {
  State   = ⊤;
  Control = λ state → ⊤;
  step    = λ state → λ control → tt}
```

An example of evaluating the singleton process is illustrated in Figure 2

Taking the product of any process and the singleton process would produce a process where the only change of information during each step is that of the process which is not the singleton. Of course, the other process could itself be the singleton process also, in which case the only change in each step is exactly that of the singleton process, which is no change at all.

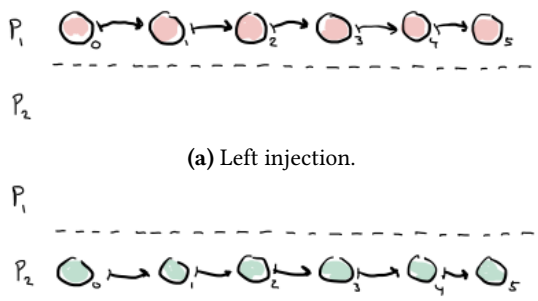
7.1 Coproduct

Seeing how we defined a product combinator of two processes, we are interested in also defining a sum combinator for processes. The approach is similar to that of the product case.

The inhabitants of the sum control is the sum of the inhabitants of the prior controls.

```
_⊎C_ : {S1 S2 : Set}
     → Con S1 → Con S2 → Con (S1 ⊎ S2)
(C1 ⊎C C2) (inj1 s1) = C1 s1
(C1 ⊎C C2) (inj2 s2) = C2 s2
```

Calculating a new step function from two prior step functions is relatively straight forward. The first input is the sum of the two states. Depending on which state the first argument



(a) Left injection.

(b) Right injection.

Figure 3. The coproduct of two processes. The process will take the shape of either of the two alternatives, but never both or a mix of the two.

belongs to, one of the prior step functions is applied to it and the second argument, the control for that state. The result of the application is then injected into the sum type using the same injection as the input.

$$
\begin{aligned}
&\_\uplus_{sf}\_ \;:\; \{S_1\ S_2\ :\ Set\} \\
&\quad \to \{C_1\ :\ Con\ S_1\} \to \{C_2\ :\ Con\ S_2\} \\
&\quad \to Step\ S_1\ C_1 \to Step\ S_2\ C_2 \\
&\quad \to Step\ (S_1 \uplus S_2)\ (C_1\ \uplus_C\ C_2) \\
&(sf_1\ \uplus_{sf}\ sf_2)\ (inj_1\ s_1)\ c_1 = inj_1\ (sf_1\ s_1\ c_1) \\
&(sf_1\ \uplus_{sf}\ sf_2)\ (inj_2\ s_2)\ c_2 = inj_2\ (sf_2\ s_2\ c_2)
\end{aligned}
$$

The sum of two problems is computed by applying the sum operators componentwise, and Figure 3 illustrate how such a problem is evaluated.

$$
\begin{aligned}
&\_\uplus_{SDP}\_ \;:\; SDProc \to SDProc \to SDProc \\
&SDP\ S_1\ C_1\ sf_1\ \uplus_{SDP}\ SDP\ S_2\ C_2\ sf_2 \\
&\quad = SDP\ (S_1 \uplus S_2)\ (C_1\ \uplus_C\ C_2)\ (sf_1\ \uplus_{sf}\ sf_2)
\end{aligned}
$$

In the case of the product process the two prior processes were not entirely independent. If one process could not progress the other process was *affected* in the sense that it too could not progress further. The sum of two processes keeps the two problems truly independent. In fact, the coproduct of two processes will start progressing from some initial state, and depending on which injection is used the other process will never advance.

A unit to the coproduct combinator is the empty process. The process has no states, no controls and the step function will return its input state. However, we will never be able to call the step function since we can not supply a state.

$$
\begin{aligned}
&empty\ :\ SDProc \\
&empty\ =\ \textbf{record}\ \{ \\
&\quad State\ \ =\ \bot; \\
&\quad Control\ =\ \lambda\ state\ \to\ \bot; \\
&\quad step\ \ =\ \lambda\ state\ \to\ \lambda\ control\ \to\ state\}
\end{aligned}
$$

Combining any process with the empty process using the coproduct combinator will produce a process that acts exactly as that of the given process. There is no way to begin advancing the empty process, and so the only available option is to select an initial state from the other process and start progressing that.

7.2 Yielding Coproduct

Computing the coproduct of two processes and getting a process that behaves like either of the two, without actually considering the other process, leaves us wondering what this is useful for. It would be more useful if we could jump between the two processes. To do this, we first need to define a relation between states. We define a relation on two state and define it to be a mapping from an inhabitant of one state to an inhabitant of the other.

$$
\begin{aligned}
&\_\rightleftarrows\_ \;:\; (S_1\ S_2\ :\ Set) \to Set \\
&s_1 \rightleftarrows s_2\ =\ (s_1\ \to\ s_2) \times (s_2\ \to\ s_1)
\end{aligned}
$$

Combining the two controls on the states is similar to that of the coproduct case, when looking at the type. However, instead of the new control being defined as either of the two prior ones, it is now *Maybe* either of the two previous ones. The idea is that we extend the control space to have one more inhabitant, the value *nothing*. If we select this control the process should yield in favour of the other process.

$$
\begin{aligned}
&\_\uplus^m_C\_ \;:\; \{S_1\ S_2\ :\ Set\} \\
&\quad \to Con\ S_1 \to Con\ S_2 \to Con\ (S_1 \uplus S_2) \\
&(C_1\ \uplus^m_C\ C_2)\ (inj_1\ s_1)\ =\ Maybe\ (C_1\ s_1) \\
&(C_1\ \uplus^m_C\ C_2)\ (inj_2\ s_2)\ =\ Maybe\ (C_2\ s_2)
\end{aligned}
$$

The new step function needs to accomodate for this scenario where the process should yield in favour of the other. To implement this the new step function needs to know *how* to yield. We describe how to yield by supplying an element of type $S_1 \rightleftarrows S_2$. If the selected control is *nothing* the step function will apply the appropriate component of this element to the current state.

$$
\begin{aligned}
&\uplus^m_{sf}\ :\ \{S_1\ S_2\ :\ Set\}\ \{C_1\ :\ Con\ S_1\}\ \{C_2\ :\ Con\ S_2\} \\
&\quad \to (S_1 \rightleftarrows S_2) \\
&\quad \to Step\ S_1\ C_1 \to Step\ S_2\ C_2 \\
&\quad \to Step\ (S_1 \uplus S_2)\ (C_1\ \uplus^m_C\ C_2) \\
&\uplus^m_{sf}\ \_\qquad\ sf_1\ sf_2\ (inj_1\ s_1)\ (just\ c)\ =\ inj_1\ (sf_1\ s_1\ c) \\
&\uplus^m_{sf}\ \_\qquad\ sf_1\ sf_2\ (inj_2\ s_2)\ (just\ c)\ =\ inj_2\ (sf_2\ s_2\ c) \\
&\uplus^m_{sf}\ (v_1\ ,\ \_)\ sf_1\ sf_2\ (inj_1\ s_1)\ nothing\ =\ inj_2\ (v_1\ s_1) \\
&\uplus^m_{sf}\ (\_\ ,\ v_2)\ sf_1\ sf_2\ (inj_2\ s_2)\ nothing\ =\ inj_1\ (v_2\ s_2)
\end{aligned}
$$

Since the other operators were infix, we give a syntax declaration that mimics the same style.

$$
syntax\ \uplus^m_{sf}\ r\ sf_1\ sf_2\ =\ sf_1\ \langle\ r\ \rangle\ sf_2
$$

Now we can compute the yielding coproduct of two processes by applying the new operations componentwise.

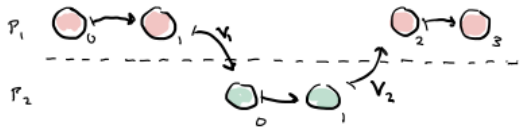


Figure 4. Illustration of the yielding coproduct process. It is capable of switching between the two processes, as illustrated by the calls to v1 and v2.

$$
\begin{aligned}
&\_\uplus^m_{SDP}\_ \;:\; (p_1 : SDProc) \to (p_2 : SDProc) \\
&\qquad \to (\#_{\mathrm{st}}\ p_1 \rightleftarrows \#_{\mathrm{st}}\ p_2) \\
&\qquad \to SDProc \\
&((SDP\ S_1\ C_1\ sf_1)\ \uplus^m_{SDP}\ (SDP\ S_2\ C_2\ sf_2))\ rel \\
&\quad = SDP\ (S_1 \uplus S_2)\ (C_1\ \uplus^m_C\ C_2)\ (sf_1 \langle\ rel\ \rangle\ sf_2)
\end{aligned}
$$

With a combinator such as this one could you model e.g a two player game. The processes would be the players and the combined process allows each to take turns making their next move. In section 11 we discuss how a policy for such a process is something of a game leader.

A unit to the yielding coproduct combinator is the same one as that for the regular coproduct combinator. If the state space is not inhabited, the process could never progress as we will not be able to call the step function. Further more, we would not be able to give a definition for a function $S_1 \to S_2$.

7.3 Interleaving processes

The next combinator we introduce is one that interleaves processes. The state of such a process holds components of both prior states, but takes turns applying the step function to each of them. This behaviour could be similar to that of a game, where two players take turns making their next move. However, the users do not know what moves the other player has made, and can therefore not make particularly smart moves. In section 11 it is reasoned that writing new policies for a process like this will be a policy that does know what move the other 'player' has made.

Similar to the product combinator the new state needs to hold components of both prior states. It should apply the step function to them one at a time, alternating between the two. In order to know which components turn it is to advance we extend the product to also hold an index.

$$
\begin{aligned}
&\_\rightleftarrows_S\_ \;:\; Set \to Set \to Set \\
&S_1 \rightleftarrows_S S_2 \;=\; Fin\ 2 \times S_1 \times S_2
\end{aligned}
$$

The control space for the interleaved process is the sum of the two prior control spaces. If the value of the first component is zero, we select the first control. If the value is one, we select the second control.

$$
\begin{aligned}
&one \;:\; Fin\ 2 \\
&one \;=\; suc\ zero
\end{aligned}
$$

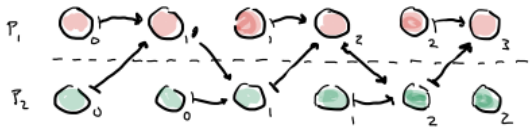


Figure 5. Illustration of two interleaved process. We want to emphasise that the state holds components of both prior states, but chooses to advance only one. The policy that chooses what control to use can however inspect both components.

$$
\begin{aligned}
&\_\rightleftarrows_C\_ \;:\; \{S_1\ S_2 : Set\} \\
&\qquad \to Con\ S_1 \to Con\ S_2 \to Con\ (S_1 \rightleftarrows_S S_2) \\
&(C_1 \rightleftarrows_C C_2)\ (zero\ ,\ s_1\ ,\ s_2) \;=\; C_1\ s_1 \\
&(C_1 \rightleftarrows_C C_2)\ (one\ ,\ s_1\ ,\ s_2) \;=\; C_2\ s_2
\end{aligned}
$$

Defining a new step function in terms of the two previous ones is done by pattern matching on the state. Specifically we are interested in the first component, the index. If the index is zero we apply the first step function to the second component of the state, leave the last component unchanged and increment the index by one. Similarly if the index is zero we apply the second step function to the last component, leave the second one unchanged and decrement the index by one.

$$
\begin{aligned}
&\_\rightleftarrows_{\mathrm{sf}}\_ \;:\; \{S_1\ S_2 : Set\} \\
&\qquad \to \{C_1 : Con\ S_1\} \to \{C_2 : Con\ S_2\} \\
&\qquad \to Step\ S_1\ C_1 \to Step\ S_2\ C_2 \\
&\qquad \to Step\ (S_1 \rightleftarrows_S S_2)\ (C_1 \rightleftarrows_C C_2) \\
&(sf_1 \rightleftarrows_{\mathrm{sf}} sf_2)\ (zero\ ,\ s_1\ ,\ s_2)\ c \;=\; (one\ ,\ sf_1\ s_1\ c\ ,\ s_2) \\
&(sf_1 \rightleftarrows_{\mathrm{sf}} sf_2)\ (suc\ zero\ ,\ s_1\ ,\ s_2)\ c \;=\; (zero\ ,\ s_1\ ,\ sf_2\ s_2\ c) \\
&(sf_1 \rightleftarrows_{\mathrm{sf}} sf_2)\ (suc\ (suc\ ())\ ,\ \_\ ,\ \_)
\end{aligned}
$$

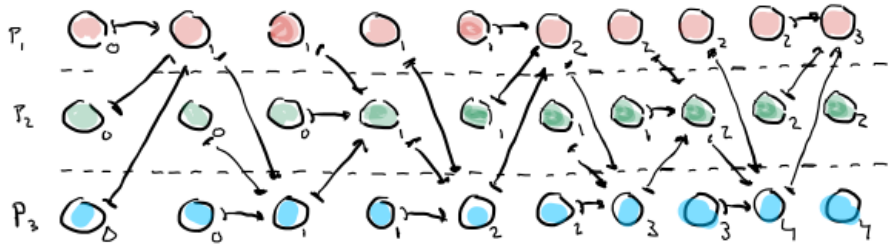
Combining two processes to capture this interleaved behaviour is once again simply done by combining the components componentwise.

$$
\begin{aligned}
&\_\rightleftarrows_{\mathrm{SDP}}\_ \;:\; SDProc \to SDProc \to SDProc \\
&SDP\ S_1\ C_1\ sf_1 \rightleftarrows_{\mathrm{SDP}} SDP\ S_2\ C_2\ sf_2 \\
&\quad = SDP\ (S_1 \rightleftarrows_S S_2)\ (C_1 \rightleftarrows_C C_2)\ (sf_1 \rightleftarrows_{\mathrm{sf}} sf_2)
\end{aligned}
$$

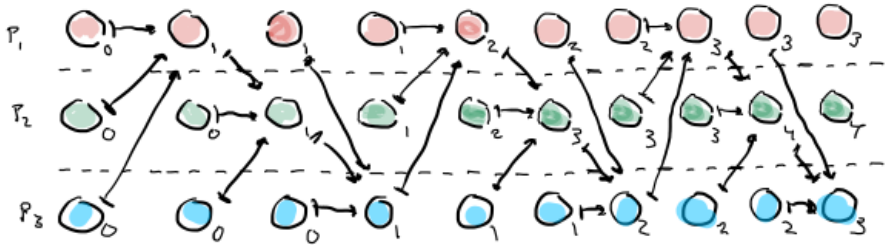
The final process behaves as illustrated in figure 5.

Defining a unit for the interleaved process is not possible. Where the initial process would advance e.g five steps, the interleaved process would need ten steps to take that component to the same state. We can not give a generic process that when interleaved with another process acts as a unit.

The way we define the interleaved combinator might not be optimal. Combining more than two processes will produce potentially unexpected behaviour. If we combine three processes using this combinator the resulting system would be one where one of the processes advance half the time, and the other two only a quarter of the time each.



(a) If we interleave two processes and then interleave the resulting process with a third we get a situation like this. They are not properly interleaved.



(b) This is the interleaved behaviour we might expect for three processes. A round robin behaviour that gives the processes equally many turns.

Figure 6. Illustrations of why the interleaved combinator might not behave as one would expect. Again the two incoming arrows illustrate that the policy that selects the control has access to all components and can base the choice of control on them.

This does not necessarily mean that the combinator described in Figure 6 is wrong, but rather that there is another combinator we could implement that would have this other behaviour.

8 Time dependent processes

Imagine a process where the state space can vary over time. If we consider the example with the one dimensional coordinate system, if the process is time dependent it could disallow some states at certain points in time. Below we illustrate how this is defined in Agda.

```
ConT : (ℕ → Set) → Set₁
ConT S = (t : ℕ) → S t → Set

StepT : (S : ℕ → Set) → ConT S → Set
StepT S C = (t : ℕ) → (s : S t) → C t s → S (suc t)

record SDProcT : Set₁ where
  constructor SDPT
  field
    State   : ℕ → Set
    Control : ConT State
    step    : StepT State Control
```

The state is now dependent on a parameter $t : \mathbb{N}$, which allows the state to take on alternate forms. In section 9.1 we illustrate what this means.

A time independent process can be embedded as a time dependent process. The embedding is a process that disregards the time parameter.

```
embed : SDProc → SDProcT
embed (SDP S C sf)
  = SDPT (λ _ → S) (λ _ → C) (λ _ → sf)
```

9 A discussion on the *Fin* type

Before we move on to an example of a time dependent process, we need to briefly present the *Fin* type and its properties. The *Fin n* type (for any natural number *n*) has exactly *n* elements.

```
data Fin : ℕ → Set where
  zero : {n : ℕ}              → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

From this definition we see that *zero* is an element of *Fin n* for any $n > zero$. The constructor *suc* takes an element of type *Fin n* and returns an element of type *Fin* (*suc n*). We illustrate the type for a couple of different n's in figure 7.

We illustrate what the *suc* constructor does in figure 8a. It takes an element of type *Fin n*, and returns the sucessor element of type *Fin* (*suc n*).

What if we want an element of the sucessor type without using the suc constructor? We might wish to simply 'promote' the type of an element. In figure 8d it becomes clear that all elements of type *Fin n* are also elements of *Fin* (*suc n*).

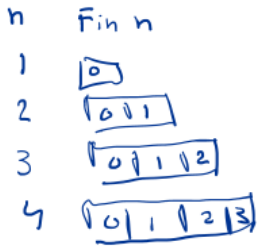


Figure 7. An illustration of the *Fin n* type. We emphasise that there are exactly *n* inhabitants of *Fin n*.

To do this promoting we use the function inject_1, which is illustrated in figure 8b.

```
inject₁ : ∀ {m} → Fin m → Fin (suc m)
inject₁ zero    = zero
inject₁ (suc i) = suc (inject₁ i)
```

Now, what if we find ourselves in a situation where we have an element of type *Fin* (*suc n*), and we want to return its predecessor, but of the sucessor type *Fin (suc (suc n))*? What we want to do is given an element *suc x*, return *x*. We can't do this as is since the element *suc x* is of type *Fin* (*suc n*), the element *x* is of type *Fin n*. To get the proper type we need to invoke $\textit{inject}_1$ twice, which is illustrated in figure 8c.

9.1 Time dependent example

Looking back at the time independent example, we reflect on the choice of state. The natural numbers seemed, and were, a reasonable choice. With the time dependent process at our disposal however we notice a source of ineffectiveness.

We consider the case where the process is always evaluated with *zero* as the initial state. After 1 step we could either have stayed or we went right, meaning the state is now *zero* or *suc zero*. After 2 steps we could have gone left, stayed or gone right. In figure 9 this edge case is illustrated, and we note that the number of possible states after n steps is $n + 1$.

Figure 9. In the edge case the state space grows slower, as we initially can not decrement the state.

If we consider the example from earlier but restrict it to starting in state zero, we could define this process as follows.

```
1d-state : ℕ → Set
1d-state n = Fin (suc n)
```

The Agda type *Fin n* is a type with at most *n* elements. Using this type gives us a more precise definition of what the possible states are.

The controls are identical to those in the time independent case.

```
1d-control : (n : ℕ) → 1d-state n → Set
1d-control n zero    = ZAction
1d-control n (suc x) = SAction
```

The step function says the same thing as in the previous example, but it says it a little differently. If the state is zero there is only two available controls, and we update the state like we did previously. However, if the state is greater than *zero* we need to change the types as described in section 9. For the left control the result has to be injected twice, and for the stay control it has to be injected once.

```
1d-step : (n : ℕ) → (x : 1d-state n)
        → (y : 1d-control n x) → 1d-state (suc n)
1d-step n zero    ZS = zero
1d-step n zero    ZR = suc zero
1d-step n (suc x) SL = inject₁ (inject₁ x)
1d-step n (suc x) SS = inject₁ (suc x)
1d-step n (suc x) SR = suc (suc x)
```

Now the entire system has been defined and we can package it as a *SDProcT*.

```
finsystem : SDProcT
finsystem = SDPT 1d-state 1d-control 1d-step
```

10 Combinators for the Time Dependent Case

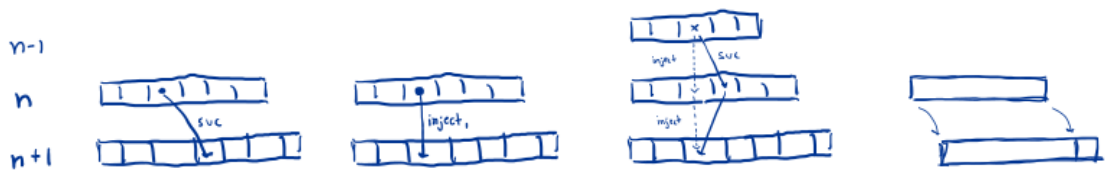
Before we move on we want to highlight that the state now is dependent on the natural numbers. The controls of these time dependent processes are time dependent themselves. We capture this reasoning in the definition of *Con′*.

```
Con′ : Con ℕ → Set₁
Con′ S = (t : ℕ) → Con (S t)
```

Now on to the product combinator for the time dependent case. To combine two states that are time dependent we compute a new time dependent state that is the product of applying the prior states to the time.

```
_×ₛ_ : (S₁ S₂ : Con ℕ) → Con ℕ
s₁ ×ₛ s₂ = λ t → s₁ t × s₂ t
```

The product combinator for two controls should produce a new *Con′* on $S_1 \times_S S_2$ defined in terms of two controls $\textit{Con}'\ S_1$ and $\textit{Con}'\ S_2$. The defining equation is similar to the time independent case, but the extra parameter time is given as the first argument.



(a) (b) (c) (d)

Figure 8. Illustrations of how to embed elements of type *Fin n* in the sucessor type *Fin* (*suc n*).

$$
\begin{aligned}
&\_\times_C\_ \;:\; \{S_1\ S_2 \;:\; Con\ \mathbb{N}\} \\
&\qquad \to Con'\ S_1 \to Con'\ S_2 \to Con'\ (S_1 \times_S S_2) \\
&(C_1 \times_C C_2)\ time\ (s_1\ ,\ s_2) \;=\; C_1\ time\ s_1 \times C_2\ time\ s_2
\end{aligned}
$$

Again we capture the type of the step function in a type *Step*. *Step* accepts a state and a control and returns a type. The type is that of the step function for time dependent processes.

$$
\begin{aligned}
&Step \;:\; (S \;:\; Con\ \mathbb{N}) \;\to\; Con'\ S \;\to\; Set \\
&Step\ S\ C \;=\; (t \;:\; \mathbb{N}) \to (s \;:\; S\ t) \;\to\; C\ t\ s \;\to\; S\ (suc\ t)
\end{aligned}
$$

Combining two such step functions is similar to the time independent case. The only different is that we have an extra parameter *time*, and we must apply the step functions to this *time* parameters.

$$
\begin{aligned}
&\_\times_{sf}\_ \;:\; \{S_1\ S_2 \;:\; Con\ \mathbb{N}\} \\
&\qquad \to \{C_1 \;:\; Con'\ S_1\} \to \{C_2 \;:\; Con'\ S_2\} \\
&\qquad \to Step\ S_1\ C_1 \to Step\ S_2\ C_2 \\
&\qquad \to Step\ (S_1 \times_S S_2)\ (C_1 \times_C C_2) \\
&(sf_1 \times_{sf} sf_2)\ time\ state\ control \\
&\quad = sf_1\ time\ (proj_1\ state)\ (proj_1\ control)\ , \\
&\qquad sf_2\ time\ (proj_2\ state)\ (proj_2\ control)
\end{aligned}
$$

Finally, combining two time dependent sequential decision processes is done by applying the combinators componentwise.

$$
\begin{aligned}
&\_\times_{SDP}\_ \;:\; SDProcT \to SDProcT \to SDProcT \\
&SDPT\ S_1\ C_1\ sf_1 \times_{SDP} SDPT\ S_2\ C_2\ sf_2 \\
&\quad = SDPT\ (S_1 \times_S S_2)\ (C_1 \times_C C_2)\ (sf_1 \times_{sf} sf_2)
\end{aligned}
$$

Just as the product combinator, the defining equation for the coproduct combinator is similar to its time independent counterpart. The difference is again that the parameters are applied to the time.

$$
\begin{aligned}
&\_\uplus_S\_ \;:\; (S_1\ S_2 \;:\; Con\ \mathbb{N}) \to Con\ \mathbb{N} \\
&s_1 \uplus_S s_2 \;=\; \lambda\ t \to s_1\ t \uplus s_2\ t
\end{aligned}
$$

The time dependent sum combinator for controls pattern matches on what injection was used, and applies the associated control to the time and the state.

$$
\begin{aligned}
&\_\uplus_C\_ \;:\; \{S_1\ S_2 \;:\; Con\ \mathbb{N}\} \\
&\qquad \to Con'\ S_1 \to Con'\ S_2 \to Con'\ (S_1 \uplus_S S_2) \\
&(C_1 \uplus_C C_2)\ time \;=\; \lambda\ \{(inj_1\ s_1) \to C_1\ time\ s_1; \\
&\qquad\qquad\qquad\qquad (inj_2\ s_2) \to C_2\ time\ s_2\}
\end{aligned}
$$

Combining the step functions to produce one defined for the new process is, similarly to the time independent case, done by pattern matching on the state. If the state is injected with the first injection, we apply the first step function, and similarly for the second injection.

$$
\begin{aligned}
&\_\uplus_{sf}\_ \;:\; \{S_1\ S_2 \;:\; Con\ \mathbb{N}\} \\
&\qquad \to \{C_1 \;:\; Con'\ S_1\} \to \{C_2 \;:\; Con'\ S_2\} \\
&\qquad \to Step\ S_1\ C_1 \to Step\ S_2\ C_2 \\
&\qquad \to Step\ (S_1 \uplus_S S_2)\ (C_1 \uplus_C C_2) \\
&(sf_1 \uplus_{sf} sf_2)\ time\ (inj_1\ s_1)\ c \;=\; inj_1\ (sf_1\ time\ s_1\ c) \\
&(sf_1 \uplus_{sf} sf_2)\ time\ (inj_2\ s_2)\ c \;=\; inj_2\ (sf_2\ time\ s_2\ c)
\end{aligned}
$$

Again we combine two processes by applying the component combinators componentwise.

$$
\begin{aligned}
&\_\uplus_{SDP}\_ \;:\; SDProcT \to SDProcT \to SDProcT \\
&SDPT\ S_1\ C_1\ sf_1 \uplus_{SDP} SDPT\ S_2\ C_2\ sf_2 \\
&\quad = SDPT\ (S_1 \uplus_S S_2)\ (C_1 \uplus_C C_2)\ (sf_1 \uplus_{sf} sf_2)
\end{aligned}
$$

To combine two time dependent processes into a yielding coproduct we begin by describing the component that relates the states in one process to states in the other.

$$
\begin{aligned}
&\_\rightleftarrows\_ \;:\; (S_1\ S_2 \;:\; Con\ \mathbb{N}) \to Set \\
&s_1 \rightleftarrows s_2 \;=\; ((t \;:\; \mathbb{N}) \to s_1\ t \to s_2\ (suc\ t)) \times \\
&\qquad\qquad ((t \;:\; \mathbb{N}) \to s_2\ t \to s_1\ (suc\ t))
\end{aligned}
$$

The first change from the coproduct combinator is again that the control space is extended to contain also the *nothing* constructor.

$$
\begin{aligned}
&\_\uplus^m_C\_ \;:\; \{S_1\ S_2 \;:\; Con\ \mathbb{N}\} \\
&\qquad \to Con'\ S_1 \to Con'\ S_2 \to Con'\ (S_1 \uplus_S S_2) \\
&(C_1 \uplus^m_C C_2)\ time\ (inj_1\ s_1) \;=\; Maybe\ (C_1\ time\ s_1) \\
&(C_1 \uplus^m_C C_2)\ time\ (inj_2\ s_2) \;=\; Maybe\ (C_2\ time\ s_2)
\end{aligned}
$$

In contrast to the coproduct case, the new step function will switch which process is executing if the control is the *nothing* constructor, and otherwise, depending on which injection was used, apply one of the previous step functions.

$$
\begin{aligned}
&\uplus^m_{sf} \;:\; \{S_1\ S_2 \;:\; Con\ \mathbb{N}\} \\
&\qquad \to \{C_1 \;:\; Con'\ S_1\} \to \{C_2 \;:\; Con'\ S_2\} \to S_1 \rightleftarrows S_2 \\
&\qquad \to Step\ S_1\ C_1 \to Step\ S_2\ C_2 \\
&\qquad \to Step\ (S_1 \uplus_S S_2)\ (C_1 \uplus^m_C C_2) \\
&\uplus^m_{sf}\ \_ \qquad sf_1\ sf_2\ time\ (inj_1\ s_1)\ (just\ c_1) \;= \\
&\quad inj_1\ (sf_1\ time\ s_1\ c_1)
\end{aligned}
$$

$$\begin{aligned}
&\uplus^m_{sf}\ (r_1\ ,\ _)\ sf_1\ sf_2\ time\ (inj_1\ s_1)\ nothing\ =\\
&\quad inj_2\ (r_1\ time\ s_1)\\
&\uplus^m_{sf}\ _\qquad sf_1\ sf_2\ time\ (inj_2\ s_2)\ (just\ c_2)\ =\\
&\quad inj_2\ (sf_2\ time\ s_2\ c_2)\\
&\uplus^m_{sf}\ (_\ ,\ r_2)\ sf_1\ sf_2\ time\ (inj_2\ s_2)\ nothing\ =\\
&\quad inj_1\ (r_2\ time\ s_2)
\end{aligned}$$

Again we provide an infix operator to be consistent.

$$syntax\ \uplus^m_{sf}\ r\ sf_1\ sf_2\ =\ sf_1\ \langle\ r\ \rangle\ sf_2$$

To create a yielding coproduct we use the same combinator for the state space, but use the new modified combinators for the control space and step function.

$$\begin{aligned}
&\uplus^m_{SDP}\ :\ (p_1\ p_2\ :\ SDProcT) \to (\#_{\mathrm{st}}\ p_1)\ \rightleftarrows\ (\#_{\mathrm{st}}\ p_2)\\
&\qquad \to SDProcT\\
&\uplus^m_{SDP}\ (SDPT\ S_1\ C_1\ sf_1)\ (SDPT\ S_2\ C_2\ sf_2)\ r\\
&\quad =\ SDPT\ (S_1\ \uplus_S\ S_2)\ (C_1\ \uplus^m_C\ C_2)\ (sf_1\ \langle\ r\ \rangle\ sf_2)
\end{aligned}$$

When we try to implement the interleaved combinator for the time dependent case we run into some problems. The main problem is that since the step function only advances one of the state components, the other one will be of the wrong type. At time n one of the components get advanced to a state in time $suc\ n$, while the other is not changed at all.

11 Policy Combinators

Now that we have a way of reusing sequential decision processes to create more sophisticated processes, we want to reuse existing policy sequences also. As the function *trajectory* which observes a process accepts a process and a policy sequence as input, this would simplify combining and observing processes without having to write any new policy sequences. We start by combining single policies.

We remind the reader that a policy is defined in terms of a state and a control.

$$\begin{aligned}
&P\ :\ (S\ :\ Set) \to (C\ :\ S \to Set) \to Set\\
&P\ S\ C\ =\ (s\ :\ S) \to C\ s
\end{aligned}$$

A policy for a product process defined in terms of two policies for the individual processes, is created by taking a pair of the two previous policies applied to the components of the state.

$$\begin{aligned}
&_\times_P_\ :\ \{S_1\ S_2\ :\ Set\}\ \{C_1\ :\ Con\ S_1\}\ \{C_2\ :\ Con\ S_2\}\\
&\qquad \to P\ S_1\ C_1 \to P\ S_2\ C_2\\
&\qquad \to P\ (S_1 \times S_2)\ (C_1\ \times_C\ C_2)\\
&(p_1\ \times_P\ p_2)\ (s_1\ ,\ s_2)\ =\ p_1\ s_1\ ,\ p_2\ s_2
\end{aligned}$$

A policy for the sum of two processes is defined by pattern matching on the state. If the pattern matches on the left injection, we can reuse the previous policy defined on that state. Similarly, if the pattern matches on the right injection we can reuse the given policy for the other process.

$$\begin{aligned}
&_\uplus_P_\ :\ \{S_1\ S_2\ :\ Set\}\ \{C_1\ :\ Con\ S_1\}\ \{C_2\ :\ Con\ S_2\}\\
&\qquad \to P\ S_1\ C_1 \to P\ S_2\ C_2\\
&\qquad \to P\ (S_1 \uplus S_2)\ (C_1\ \uplus_C\ C_2)\\
&(p_1\ \uplus_P\ p_2)\ (inj_1\ s_1)\ =\ p_1\ s_1\\
&(p_1\ \uplus_P\ p_2)\ (inj_2\ s_2)\ =\ p_2\ s_2
\end{aligned}$$

Reusing policies for the yielding coproduct is similar to that of the regular coproduct. The only difference is that when reusing the old policy the result must be wrapped in the *just* constructor.

$$\begin{aligned}
&_\uplus^m_P_\ :\ \{S_1\ S_2\ :\ Set\}\ \{C_1\ :\ Con\ S_1\}\ \{C_2\ :\ Con\ S_2\}\\
&\qquad \to P\ S_1\ C_1 \to P\ S_2\ C_2\\
&\qquad \to P\ (S_1 \uplus S_2)\ (C_1\ \uplus^m_C\ C_2)\\
&(p_1\ \uplus^m_P\ p_2)\ (inj_1\ s_1)\ =\ just\ (p_1\ s_1)\\
&(p_1\ \uplus^m_P\ p_2)\ (inj_2\ s_2)\ =\ just\ (p_2\ s_2)
\end{aligned}$$

To combine two policies for an interleaved process we recall that the control space changes when the index changes. When the index is 0 the control space is that of the first process, and when it is 1 the control space is that of the second process. Similarly, in order to reuse the two previous policies we must then pattern match on the state and see what the index is. If the index is *zero*, we reuse the first policy. If it is *suc zero*, we reuse the second policy.

$$\begin{aligned}
&_\rightleftarrows_P_\ :\ \{S_1\ S_2\ :\ Set\}\\
&\qquad\qquad \{C_1\ :\ Con\ S_1\}\ \{C_2\ :\ Con\ S_2\}\\
&\qquad \to P\ S_1\ C_1 \to P\ S_2\ C_2\\
&\qquad \to P\ (S_1\ \rightleftarrows_S\ S_2)\ (C_1\ \rightleftarrows_C\ C_2)\\
&(p_1\ \rightleftarrows_P\ p_2)\ (zero\ ,\ s_1\ ,\ _)\qquad =\ p_1\ s_1\\
&(p_1\ \rightleftarrows_P\ p_2)\ (suc\ zero\ ,\ _\ ,\ s_2)\ =\ p_2\ s_2\\
&(p_1\ \rightleftarrows_P\ p_2)\ (suc\ (suc\ ())\ ,\ _)
\end{aligned}$$

12 A note on Policies

With these policy combinators defined, we make a few observations. These observations intend to illustrate some of the meanings behind policies.

12.1 Product State Policies

Recall a policy for a product state. The type of this policy is $(s\ :\ State) \to Control\ s$, where State is the type of states and Control is the type family of controls. Since s is a product , the signature actually is something like $((a\ ,\ b)\ :\ A \times B) \to Control\ (a\ ,\ b)$. We know from currying that this is the same as $(a\ :\ A) \to (b\ :\ B) \to Control\ (a\ ,\ b)$.

In the case of the product combinator the control space was the product of the separate state components control spaces. Where a policy for the individual process would base the choice of control on the state, the policy for a product process will select a control for the same state component, but base the choice on both state components.

In the case of the interleaved process we related the situation to that of a game, where players take turns making their

moves. In such a scenario a policy could be something of a game leader, making the best choices for each component based both components. The policy can, after all, make a decision for one of the components based on the state of both components.

12.2 Sum state implies Product Policy

Another interesting observation to make is that a policy for a process with a sum state, e.g a policy for a coproduct, is a pair of policies for the separate processes. We can make this concrete with the following two definitions.

```
⊎ ↦ × :  {S₁ S₂ : Set}
         {C₁ : Con S₁} {C₂ : Con S₂}
      → P (S₁ ⊎ S₂) (C₁ ⊎C C₂)
      → P S₁ C₁ × P S₂ C₂
⊎ ↦ × policy = (λ s₁ → policy (inj₁ s₁)) ,
               λ s₂ → policy (inj₂ s₂)
× ↦ ⊎ :  {S₁ S₂ : Set}
         {C₁ : Con S₁} {C₂ : Con S₂}
      → P S₁ C₁ × P S₂ C₂
      → P (S₁ ⊎ S₂) (C₁ ⊎C C₂)
× ↦ ⊎ (p₁ , p₂) = λ {(inj₁ s₁) → p₁ s₁;
                     (inj₂ s₂) → p₂ s₂}
```

Then we can further solidify this statement by showing that they are equal by functional extensionality.

```
∀⊎ ↦ × :  {S₁ S₂ : Set}
          {C₁ : Con S₁} {C₂ : Con S₂}
          (p : P (S₁ ⊎ S₂) (C₁ ⊎C C₂))
       → (state : S₁ ⊎ S₂)
       → × ↦ ⊎ (⊎ ↦ × p) state ≡ p state
∀⊎ ↦ × _ (inj₁ _) = refl
∀⊎ ↦ × _ (inj₂ _) = refl
```

13 Conclusions and future work

We have shown that sequential decision processes can quite comfortably be combined. Things start to get interesting when we consider writing new policies rather than combining existing policies. As exemplified in section 11 a policy for a product state is essentially a policy for one process parameterised over another. This becomes particularly interesting when we look at the interleaved process where we only wish to advance one component but we now have information about the other also. Using backwards induction to compute optimal policy sequences could perhaps use this extra information to produce very clever sequences.

There are many avenues left to explore for future work.

- Here we have mainly focused on sequential decision *processes*, and not so much on *problems*. It is not entirely clear how problems should be combined. The problems need to offer the same type of reward, normalised to some range and then combined in some way. Even when the type of the reward is the natural numbers, combining them is not unambiguous as there are many ways to combine natural numbers.
- There are definitely many more combinators than those presented by us in this text. Here we focused mainly on generic combinators that required no additional information and then briefly touched the yielding coproduct, which did require additional information in order to switch processes.
- Both processes and problems can be parameterised over a monad. A processes step function can e.g produce a list of possible next states and associate a probability to each of them. Modeling such uncertainties vastly increases the amount of problems that can be modeled using this framework. When combining processes like these combining the step functions becomes less trivial.
- Many claims and ideas in this text has been described but not formalised. As an example the notion of a unit and its properties could be formalised, and then the units presented here could be proven to be correct units. We are also interested in implementing an n-ary combinator for the interleaved combinator, as well as an interleaved combinator for the time dependent case.