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 policies. The concepts of feasibility and availability 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 "\( \times_{SDP} \)" 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:

\[
\text{Con } : \text{Set} \rightarrow \text{Set},
\]

\[
\text{Con } S = S \rightarrow \text{Set}
\]

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

\[
\text{Step } : (S : \text{Set}) \rightarrow \text{Con } S \rightarrow \text{Set}
\]

\[
\text{Step } S C = (s : S) \rightarrow C s \rightarrow S
\]

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

\[
\text{record } SDProc : \text{Set1} \ where
\]

\[
\text{constructor } SDP
\]

\[
\text{field } State : \text{Set}
\]

\[
\text{Control } : \text{Con } State
\]

\[
\text{step } : \text{Step } State \text{ Control}
\]

We can extend this idea of a sequential decision process to that of a problem by adding an additional field reward.
\[
\text{reward} : (x : \text{State}) \rightarrow \text{Control} x \rightarrow \text{Val}
\]

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 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:

\[
\text{Policy} : (S : \text{Set}) \rightarrow \text{Con} S \rightarrow \text{Set}
\]

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 \(\#_s\) and \(\#_d\) functions extract the state and step component from the SDProc respectively.

\[
\text{trajectory} : \{n : \mathbb{N}\} \rightarrow (p : \text{SDProc}) \rightarrow \text{Vec} (\text{Policy} (\#_s p) (\#_d p)) n \rightarrow \text{Vec} (\#_s p) n
\]

\[
\text{trajectory} \_\text{sys} (\_\_p) x\_0 = []
\]

\[
\text{trajectory} x\_1 : \#_s \text{sys}
\]

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

\[
pseq = \begin{align*}
\text{tryleft} &:: \text{tryleft} :: \text{right} :: \text{stay} :: \text{right} :: [] \\
n0 &:: \text{trajectory} \ I\text{-sys} \ pseq \ 0 \equiv 0 :: 0 :: 1 :: 1 :: 2 :: [] \\
n0 &\equiv \text{refl}
\end{align*}
\]

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.

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

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.

\[
\text{test1} : \text{trajectory} \ I\text{-sys} \ pseq 0 \equiv 0 :: 0 :: 1 :: 1 :: 2 :: []
\]

\[
\text{test1} \equiv \text{refl}
\]

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

\[
\text{test2} : \text{trajectory} \ 2d\text{-sys} \ 2d\text{-pseq} \ (0 , 5) \equiv (0 , 4) :: (0 , 3) :: (1 , 4) :: (1 , 4) :: (2 , 5) :: []
\]

\[
\text{test2} \equiv \text{refl}
\]

### 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
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.

\[ \text{state} : \text{Set} \]
\[ \text{state} = \mathbb{N} \]

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

\[ \text{Policy} \to \text{1d-Policy} \]

We can evaluate the system using this sequence, starting from different points. We can use \( \equiv \) and \( \text{refl} \) to assert that the system behaves as intended.

\[ \text{test1} = \text{trajectory} \quad \text{test2} = \text{trajectory} \]
\[ \text{test3} = \text{trajectory} \]

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

\[ \text{sys} : \text{SDProc} \]
\[ \text{sys} = \text{SDP} \quad \text{1d-control} \quad \text{1d-step} \]

We can turn a process into a problem we need to introduce a notion of a goal, described by a \text{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.

\[ \text{1d-reward} : \text{state} \to (x : \text{state}) \to \text{1d-control} x \to \text{state} \]

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.

\[ \text{1d-reward} \to \text{state} \]
\[ \to (x : \text{state}) \to \text{1d-control} x \]
\[ \to \text{state} \to \mathbb{N} \]

We can redefine the sequential decision process above to be a sequential decision problem simply by instantiating the \text{SDProb} record.
we move on to additional combinators we’d like to make

An Algebra of Sequential Decision Problems TyDe’19, August 18, 2019, Berlin, Germany

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 \times_{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 (#_a 2d-system) (#_c 2d-system) 5
2d-pseq = zipWith \_\_p seq 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.

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 = \top;
  Control = \lambda state \rightarrow \top;
  step = \lambda state \rightarrow \lambda control \rightarrow \bot
}

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.

\_\_pseq : \{ S_1, S_2 : Set \}
\rightarrow Con S_1 \rightarrow Con S_2 \rightarrow Con (S_1 \uplus S_2)
(C_1 \uplus C_2) (\text{inj}_1 s_1) = C_1 s_1
(C_1 \uplus C_2) (\text{inj}_2 s_2) = C_2 s_2

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

...
will return its input state. However, we will never be able to
were not entirely independent. If one process could not
The process has no states, no controls and the step function
keeps the two problems truly independent. In fact, the co-
operators componentwise, and Figure 3 illustrate how such
In the case of the product process the two prior processes
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.
\[ \omega_{\text{set}} : \{ S_1, S_2 : \text{Set} \} \rightarrow \{ C_1 : \text{Con } S_1 \} \rightarrow \{ C_2 : \text{Con } S_2 \} \]
\[ \rightarrow \text{Step } S_1 \ C_1 \rightarrow \text{Step } S_2 \ C_2 \]
\[ \rightarrow \text{Step } (S_1 \uplus S_2) (C_1 \uplus C_2) \]
\[ (sf_1 \uplus sf_2) (inj_1 \ s_1) c_1 = \text{inj}_1 (sf_1 \ s_1 \ c_1) \]
\[ (sf_1 \uplus sf_2) (inj_2 \ s_2) c_2 = \text{inj}_2 (sf_2 \ s_2 \ c_2) \]
The sum of two problems is computed by applying the sum
operators componentwise, and Figure 3 illustrate how such
a problem is evaluated.
\[ \omega_{\text{sdp}} : \text{SDProc} \rightarrow \text{SDProc} \rightarrow \text{SDProc} \]
\[ SDP \ S_1 \ C_1 \ sf_1 \uplus sf_2 \ SDP \ S_2 \ C_2 \ sf_2 \]
\[ = \text{SDP } (S_1 \uplus S_2) (C_1 \uplus C_2) (sf_1 \uplus sf_2) \]
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.
\[ \text{empty} : \text{SDProc} \]
\[ \text{empty} = \text{record} \{ \]
\[ \text{State} = \uplus \];
\[ \text{Control} = \lambda \text{state} \rightarrow \perp ; \]
\[ \text{step} = \lambda \text{state} \rightarrow \lambda \text{control} \rightarrow \text{state} \]
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.
\[ \text{yield} : (S_1, S_2 : \text{Set}) \rightarrow \text{Set} \]
\[ s_1 \rightarrow s_2 = (s_1 \rightarrow s_2) \times (s_2 \rightarrow s_1) \]
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.
\[ \omega_{\text{cop}} : \{ S_1, S_2 : \text{Set} \} \rightarrow \text{Con } S_1 \rightarrow \text{Con } S_2 \rightarrow \text{Con } (S_1 \uplus S_2) \]
\[ (C_1 \uplus C_2)(inj_1 \ s_1) = \text{Maybe } (C_1 \ s_1) \]
\[ (C_1 \uplus C_2)(inj_2 \ s_2) = \text{Maybe } (C_2 \ s_2) \]
The new step function needs to accommodate 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 \( S_1 \rightarrow S_2 \). If the selected control is nothing the step function
will apply the appropriate component of this element to the
current state.
\[ \omega_{\text{sf}} : \{ S_1, S_2 : \text{Set} \} \{ C_1 : \text{Con } S_1 \} \{ C_2 : \text{Con } S_2 \} \rightarrow \]
\[ \text{Step } S_1 \ C_1 \rightarrow \text{Step } S_2 \ C_2 \]
\[ \rightarrow \text{Step } (S_1 \uplus S_2) (C_1 \uplus C_2) \]
\[ (sf_1 \uplus sf_2)(inj_1 \ s_1)(just \ c) = \text{inj}_1 (sf_1 \ s_1 \ c) \]
\[ (sf_1 \uplus sf_2)(inj_2 \ s_2)(just \ c) = \text{inj}_2 (sf_2 \ s_2 \ c) \]
\[ (v_1, v_2) sf_1 sf_2 (inj_1 \ s_1) \text{nothing} = \text{inj}_2 (v_1 \ s_1) \]
\[ (v_1, v_2) sf_1 sf_2 (inj_2 \ s_2) \text{nothing} = \text{inj}_1 (v_2 \ s_2) \]
Since the other operators were infix, we give a syntax declaration that mimics the same style.
\[ \text{syntax } \omega_{\text{sf}} r sf_1 sf_2 = sf_1 (r \ r) sf_2 \]
Now we can compute the yielding coproduct of two processes by applying the new operations componentwise.
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.

\[ \text{one} : 	ext{Fin} 2 \]
\[ \text{one} = \text{suc zero} \]

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.

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 \( S_1 \rightarrow S_2 \).

![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.](image)

![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.](image)
(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.

\[
\begin{align*}
ConT & : (\mathbb{N} \rightarrow \text{Set}) \rightarrow \text{Set} \\
ConT S & = (t : \mathbb{N}) \rightarrow S t \rightarrow \text{Set} \\
StepT & : (S : \mathbb{N} \rightarrow \text{Set}) \rightarrow ConT S \rightarrow \text{Set} \\
StepT S C & = (t : \mathbb{N}) \rightarrow (s : S t) \rightarrow C t s \rightarrow S (\text{succ} t)
\end{align*}
\]

**record** SDProcT : Set₁** where**

**constructor** SDPT

**field**

State : \(\mathbb{N} \rightarrow \text{Set}\)
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.

\[
\begin{align*}
\text{embed} & : \text{SDProc} \rightarrow \text{SDProcT} \\
\text{embed} \ (SDP \ S \ C \ sf) & = \ SDPT \ (\lambda \ _ \rightarrow S) \ (\lambda \ _ \rightarrow C) \ (\lambda \ _ \rightarrow sf)
\end{align*}
\]

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.

\[
\begin{align*}
data \ \text{Fin} & : \mathbb{N} \rightarrow \text{Set} \text{ where} \\
\text{zero} & : \{ n : \mathbb{N} \} \rightarrow \text{Fin (succ n)} \\
\text{succ} & : \{ n : \mathbb{N} \} \ (i : \text{Fin n}) \rightarrow \text{Fin (succ n)}
\end{align*}
\]

From this definition we see that zero is an element of Fin \(n\) for any \(n > 0\). The constructor succ takes an element of type Fin \(n\) and returns an element of type Fin (succ \(n\)). We illustrate the type for a couple of different \(n\)'s in figure 7.

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

What if we want an element of the successor type without using the succ 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 (succ \(n\)).
we initially cannot decrement the state.

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

\[
\begin{align*}
\text{inject} \colon & \forall \{ m \} \to \text{Fin } m \to \text{Fin } (\text{suc } m) \\
\text{inject } & \text{zero } = \text{ zero } \\
\text{inject } (\text{suc } i) &= \text{ suc } (\text{inject } i)
\end{align*}
\]

Now, what if we find ourselves in a situation where we have an element of type \(\text{Fin } (\text{suc } n)\), and we want to return its predecessor, but of the successor type \(\text{Fin } (\text{suc } (\text{suc } n))\)? What we want to do is given an element \(\text{suc } x\), return \(x\). We can’t do this as is since the element \(\text{suc } x\) is of type \(\text{Fin } (\text{suc } n)\), the element \(x\) is of type \(\text{Fin } n\). To get the proper type we need to invoke \(\text{inject }_1\) 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 \(\text{suc } \text{zero}\). 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 \(n + 1\).

The Agda type \(\text{Fin } n\) 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.

\[
\begin{align*}
1d-\text{control } & (n : \mathbb{N}) \to 1d-\text{state } n \to \text{Set} \\
1d-\text{control } n \text{ zero } &= \text{ ZAction} \\
1d-\text{control } n (\text{suc } x) &= \text{ SAction}
\end{align*}
\]

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.

\[
\begin{align*}
1d-\text{step } & (n : \mathbb{N}) \to (x : 1d-\text{state } n) \\
& \to (y : 1d-\text{control } n x) \to 1d-\text{state } (\text{suc } n)
\end{align*}
\]

Now the entire system has been defined and we can package it as a \(\text{SDProcT}\).

\[
\begin{align*}
\text{finsystem } & : \text{SDProcT} \\
\text{finsystem } &= \text{SDPT 1d-state } 1d-\text{control } 1d-\text{step}
\end{align*}
\]

### 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 \(\text{Con’}\).

\[
\begin{align*}
\text{Con’ } & : \text{Con } \mathbb{N} \to \text{Set}_t \\
\text{Con’ } S & = (t : \mathbb{N}) \to \text{Con } (S t)
\end{align*}
\]

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.

\[
\begin{align*}
\times_\text{S-} & : (S_1, S_2 : \text{Con } \mathbb{N}) \to \text{Con } \mathbb{N} \\
S_1 \times_\text{S-} S_2 & = \lambda t \to S_1 t \times S_2 t
\end{align*}
\]

The product combinator for two controls should produce a new \(\text{Con’}\) on \(S_1 \times_\text{S-} S_2\) defined in terms of two controls \(\text{Con’ } S_1\) and \(\text{Con’ } S_2\). The defining equation is similar to the time independent case, but the extra parameter time is given as the first argument.
\[ x_{\text{Con}} : \{ S_1, S_2 : \text{Con } \mathbb{N} \} \]
\[ \rightarrow \text{Con'} S_1 \rightarrow \text{Con'} S_2 \rightarrow \text{Con'} (S_1 \times \text{C}) \]
\[ \text{(C \times C)} \text{ time } (s_1, s_2) = C_1 \text{ time } s_1 \times C_2 \text{ time } s_2 \]

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

\[ \text{Step : } (S : \text{Con } \mathbb{N}) \rightarrow \text{Con'} S \rightarrow \text{Set} \]
\[ \text{Step } S C = (t : \mathbb{N}) \rightarrow (s : S t) \rightarrow C \text{ t s } \rightarrow S (\text{Suc } t) \]

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

\[ x_{\text{sf}} : \{ S_1, S_2 : \text{Con } \mathbb{N} \} \]
\[ \rightarrow \{ C_1 : \text{Con'} S_1 \} \rightarrow \{ C_2 : \text{Con'} S_2 \} \]
\[ \rightarrow \text{Step } S_1 C_1 \rightarrow \text{Step } S_2 C_2 \]
\[ \rightarrow \text{Step } (S_1 \times \text{C}) (C_1 \times C_2) \]
\[ (\text{sf}_1 \times_{\text{sf}} \text{sf}_2) \text{ time state control} \]
\[ = \text{sf}_1 \text{ time (proj state) (proj control)} , \]
\[ \text{sf}_2 \text{ time (proj state) (proj control)} \]

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

\[ x_{\text{SDP}} : \text{SDProcT} \rightarrow \text{SDProcT} \rightarrow \text{SDProcT} \]
\[ \text{SDPT } S_1 C_1 \text{ sf}_1 \times_{\text{sdp}} \text{SDPT } S_2 C_2 \text{ sf}_2 \]
\[ = \text{SDPT } (S_1 \times S_2) (C_1 \times \text{C}) (\text{sf}_1 \times_{\text{sf}} \text{sf}_2) \]

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.

\[ x_{\text{sdp}} : (S_1, S_2 : \text{Con } \mathbb{N}) \rightarrow \text{Con } \mathbb{N} \]
\[ s_1 \times s_2 = \lambda t \rightarrow s_1 t \land s_2 t \]

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.

\[ x_{\text{C}} : \{ S_1, S_2 : \text{Con } \mathbb{N} \} \]
\[ \rightarrow \text{Con'} S_1 \rightarrow \text{Con'} S_2 \rightarrow \text{Con'} (S_1 \times \text{C}) \]
\[ \text{(C \times C)} \text{ time } \{(\text{inj}_1 s_1) \rightarrow C_1 \text{ time } s_1; \]
\[ \{(\text{inj}_2 s_2) \rightarrow C_2 \text{ time } s_2 \} \]

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.

\[ \omega_{\text{sdp}} : \text{SDProcT} \rightarrow \text{SDProcT} \rightarrow \text{SDProcT} \]
\[ \text{SDPT } S_1 C_1 \text{ sf}_1 \omega_{\text{sdp}} \text{SDPT } S_2 C_2 \text{ sf}_2 \]
\[ = \text{SDPT } (S_1 \times S_2) (C_1 \times \text{C}) (\text{sf}_1 \omega_{\text{sf}} \text{sf}_2) \]

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.

\[ \omega_{\text{sdp}} : (S_1, S_2 : \text{Con } \mathbb{N}) \rightarrow \text{Set} \]
\[ s_1 \rightarrow s_2 = ((t : \mathbb{N}) \rightarrow s_1 t \rightarrow s_2 (\text{Suc } t)) \times \]
\[ ((t : \mathbb{N}) \rightarrow s_2 t \rightarrow s_1 (\text{Suc } t)) \]

The first change from the coproduct combinator is again that the control space is extended to contain also the \text{nothing} constructor.

\[ \omega_{\text{C}} : \{ S_1, S_2 : \text{Con } \mathbb{N} \} \]
\[ \rightarrow \text{Con'} S_1 \rightarrow \text{Con'} S_2 \rightarrow \text{Con'} (S_1 \times \text{C}) \]
\[ \text{(C \times C)} \text{ time } (\text{inj}_1 s_1) = \text{Maybe } (C_1 \text{ time } s_1) \]
\[ (C_1 \times \text{C}) \text{ time } (\text{inj}_2 s_2) = \text{Maybe } (C_2 \text{ time } s_2) \]

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

\[ \omega_{\text{C}} : \{ S_1, S_2 : \text{Con } \mathbb{N} \} \]
\[ \rightarrow \{ C_1 : \text{Con'} S_1 \} \rightarrow \{ C_2 : \text{Con'} S_2 \} \rightarrow S_1 \rightarrow S_2 \]
\[ \rightarrow \text{Step } S_1 C_1 \rightarrow \text{Step } S_2 C_2 \]
\[ \rightarrow \text{Step } (S_1 \times \text{C}) (C_1 \times \text{C}) \]
\[ \omega_{\text{C}} : \text{sf}_1 \text{sf}_2 \text{ time } (\text{inj}_1 s_1) (\text{just } c_1) = \]
\[ \text{inj}_1 (\text{sf}_1 \text{time } s_1 c_1) \]
we can reuse the given policy for the other process.

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.

An Algebra of Sequential Decision Problems

\[ \omega^m_{\mathcal{D}} (r_1, \_ ) sf_1 sf_2 \text{ time (inj}_1 s_1) \text{ nothing } = \]
\[ \text{inj}_2 (r_1 \text{ time } s_1) \]
\[ \omega^m_{\mathcal{D}} - sf_1 sf_2 \text{ time (inj}_2 s_2) \text{ (just } c_2) = \]
\[ \text{inj}_2 (sf_2 \text{ time } s_2 c_2) \]
\[ \omega^m_{\mathcal{D}} (\_ , r_2) sf_1 sf_2 \text{ time (inj}_2 s_2) \text{ nothing } = \]
\[ \text{inj}_1 (r_2 \text{ time } s_2) \]

Again we provide an infix operator to be consistent.

syntax \[ \omega^m_{\mathcal{D}} r sf_1 sf_2 \]

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.

\[ \omega^m_{\mathcal{SDP}} : (p_1 p_2 : \mathcal{SDProcT}) \rightarrow (\#^a_1 p_1) \mapsto (\#^a_2 p_2) \]
\[ \rightarrow \mathcal{SDProcT} \]
\[ \omega^m_{\mathcal{SDP}} (SDPT S_1 C_1 sf_1) (SDPT S_2 C_2 sf_2) r \]
\[ = \mathcal{SDPT} (S_1 \_ \_ S_2) (C_1 \_ \_ C_2) (sf_1 \ (r \ ) sf_2) \]

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 \( \text{suc } n \), while the other is not changed at all.

\section{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 \text{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 simple policies.

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

\[ P : (S : \text{Set}) \rightarrow (C : S \rightarrow \text{Set}) \rightarrow \text{Set} \]
\[ P S C = (s : S) \rightarrow C s \]

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.

\[ \preceq \_ \_ p_\_ : \{S_1 S_2 : \text{Set}\} \{C_1 : \text{Con } S_1\} \{C_2 : \text{Con } S_2\} \]
\[ \rightarrow P S_1 C_1 \rightarrow P S_2 C_2 \]
\[ \rightarrow P (S_1 \times S_2) (C_1 \_ \_ C_2) \]
\[ (p_1 \preceq_\_ p_2) (s_1 , s_2) = p_1 s_1 , p_2 s_2 \]

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.

\[ \preceq \_ \_ p_\_ : \{S_1 S_2 : \text{Set}\} \{C_1 : \text{Con } S_1\} \{C_2 : \text{Con } S_2\} \]
\[ \rightarrow P S_1 C_1 \rightarrow P S_2 C_2 \]
\[ \rightarrow P (S_1 \_ \_ S_2) (C_1 \_ \_ C_2) \]
\[ (p_1 \preceq_\_ p_2) (s_1) = p_1 s_1 , p_2 s_2 \]

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 \( (s : \text{State}) \rightarrow \text{Control } s \), 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 \( ((a , b) : \text{A} \times \text{B}) \rightarrow \text{Control } (a , b) \). We know from currying that this is the same as \( (a : \text{A}) \rightarrow (b : \text{B}) \rightarrow \text{Control } (a , b) \).

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.

\[
\forall \mapsto \times : \{ S_1, S_2 : \text{Set} \}
\begin{align*}
\{ C_1 : \text{Con } S_1 \} & \{ C_2 : \text{Con } S_2 \} \\
\rightarrow P (S_1 \uplus S_2) (C_1 \uplus C_2) \\
\rightarrow P S_1 C_1 \times P S_2 C_2 \\
\forall \mapsto \times \text{ policy } = (\lambda s_1 \mapsto \text{policy } (\text{inj}_1 s_1), \\
\lambda s_2 \mapsto \text{policy } (\text{inj}_2 s_2)
\end{align*}
\]

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

\[
\forall \mapsto \times : \{ S_1, S_2 : \text{Set} \}
\begin{align*}
\{ C_1 : \text{Con } S_1 \} & \{ C_2 : \text{Con } S_2 \} \\
(p : P (S_1 \uplus S_2) (C_1 \uplus C_2)) \\
\rightarrow (\text{state} : S_1 \uplus S_2) \\
\rightarrow \times \mapsto \forall (\mapsto \times p) \text{ state } \equiv p \text{ state}
\end{align*}
\]

\[
\forall \mapsto \times (\text{inj}_1 \_ ) = \text{refl} \\
\forall \mapsto \times (\text{inj}_2 \_ ) = \text{refl}
\]

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 combin-
ing 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 in-
teresting 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 en-
tirely clear how problems should be combined. The
problems need to offer the same type of reward, nor-
malised 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 addi-
tional 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 pro-
duce a list of possible next states and associate a prob-
ability to each of them. Modeling such uncertainties
vastly increases the amount of problems that can be
modeled using this framework. When combining pro-
cesses 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.