TIPPING POINTS IN THE EARTH SYSTEM

WP6 Course on functional languages and dependently typed languages: D6.1 (D27)

TiPES: Tipping Points in the Earth System is *is a Research and Innovation action (RIA) funded by the Horizon 2020 Work programme topics "Addressing knowledge gaps in climate science, in support of IPCC reports" Start date: 1st* September 2019. End date: 31st August 2023.

The TiPES project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 820970.

About this document

Deliverable: D6.1
Work package in charge: WP6 Uncertainty and accountable policies
Actual delivery date for this deliverable: Project-month 12
Dissemination level:
The general public (PU)

Lead author(s)
Potsdam Institute for Climate Impact Research (PIK): Nicola Botta, Nuria Brede

Other contributing author(s)
Université catholique de Louvain (UCL): Michel Crucifix, Marina Martínez Montero

Reviewer(s)
University of Copenhagen (UCPH): Peter Ditlevsen

Visit us on: www.tipes.dk
Follow us on Twitter: @TiPES_H2020

Access our open access documents in Zenodo:
https://zenodo.org/communities/tipes/

Index

Summary for publication

This publication provides teaching material for an introductory course on functional and dependently typed programming and its application to verified decision-making in the context of climate science, using the computational theory of policy advice and avoidability developed by Botta et al.([1], [3]).

The course consists of 11 regular lectures and 2 extra lectures. The regular lectures consist of three main parts:

1. an introduction motivating the use of formal methods from theoretical computer science in climate science,
2. an introduction to mathematical specification, functional and dependently typed programming, and computer-verified proofs,
3. an introduction to the Botta et al. framework for decision-making under uncertainty in the context of climate science, including an example application of the framework as in [2].

The extra lectures provide some theoretical background to the topics covered in the main lectures:

1. an introduction to formal logic and the correspondence between proofs in constructive logic and programs in dependently typed programming languages,
2. an introduction to the notions of functor and monad from the perspective of category theory. These play an important role in the Botta et al. framework.

Part 1 of the regular lectures and the two extra lectures are provided as presentation slides. Parts 2 and 3 of the regular lectures are included in this document as lecture notes, but they are also available at [4] as "literate" Idris [5] source code files which can be machine-checked ("type-checked") and compiled, and from which the lecture notes can be generated automatically using the tool *lhs2tex* [6].

Work carried out

This document WP6 D6.1 (D27) contains the accompanying material for the course on functional and dependently typed programming languages given by Nicola Botta and Nuria Brede at UCL in November 25-29, 2019 and March 02-06, 2020. The course notes were prepared by Nicola Botta and Nuria Brede and benefited from the interaction with Michel Crucifix and Marina Montero Martínez during the course.

Contribution to the top-level objectives of TiPES

This deliverable contributes to the achievement of **Objective 5 (O5) - Bridge the gap between climate science and policy advice** by providing introductory course material to the formal framework which is used as basis for tasks T6.1, T6.2 and T6.3 which all work towards O5 by linking Tipping Point uncertainty and accountable decision making.

With the overall objective to employ methods from theoretical computer science towards accountable advice for decision-makers in matters of climate policy, WP6 involves a ground-breaking collaboration between climate science at UCL and theoretical computer science at PIK. The immediate role of D6.1 was to prepare the UCL personnel for the task of formally specifying sequential decision problems within the Botta et al. Framework ([1],[3]), but the course material is suitable to introduce a larger audience to verified (and thus accountable) decision making under uncertainty in the context of climate science.

References

[1] **Botta, N.**, **Jansson, P**., **Ionescu, C**. (2017). Contributions to a computational theory of policy advice and avoidability. J. Funct. Program., 27, e23.

[2] **Botta, N.**, **Jansson, P.**, **Ionescu, C.** (2018). The impact of uncertainty on optimal emission policies. Earth Syst. Dynam., 9, 525-542.

[3] **Botta, N. et al.** (2016-2020). IdrisLibs, https://gitlab.pik-potsdam.de/botta/IdrisLibs.

[4] **Botta, N. et al.** (2019-2020). TiPES D6.1 Course material,

https://gitlab.pik-potsdam.de/botta/IdrisLibs/-/tree/master/lectures/2019-%C2%AD11%2B2020%C2%AD-03.LouvainLaNeuve.TiPES_D6.1.

[5] **Brady, E.** (2013)**.** Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23(9):552–593.

[6] **Löh, A.**, lhs2tex Preprocessor , https://github.com/kosmikus/lhs2tex

Lecture 1: Decision problems in climate research

Decision problems in climate research

Nicola Botta[1,2], Nuria Brede[1]

[1]Potsdam Institute for Climate Impact Research
[2]CSE, Chalmers University of Technology

Objectives of this lecture

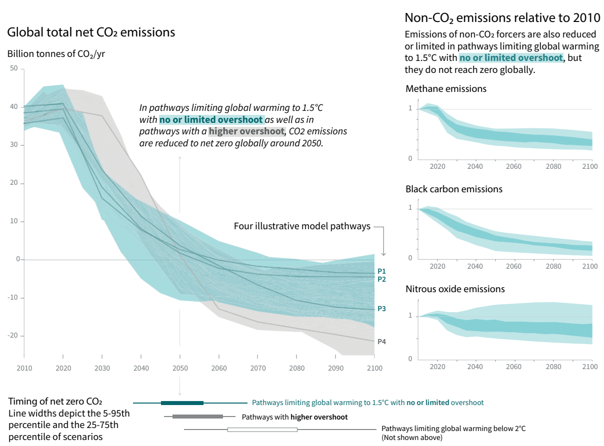
- Get acquainted with the general idea of using formal methods for climate impact research
- Learn about basic questions concerning uncertainties in sequential decision making
- Go through two basic examples of decision problems in climate research and analyze their common features

Climate research and decision making

- Climate research shall improve our understanding of the earth system and climate but also ...
- ... inform rational, transparent, accountable and, above all, good decisions!
- Best policies for a decision problem typically depend on the uncertainties that affect that problem but ...
- ... in climate research, many unavoidable uncertainties cannot be estimated through established theories or model simulations!

Example 1: emission reduction policies

- Global GHG emissions have to be reduced readily to avoid possibly unmanageable impacts of climate change:



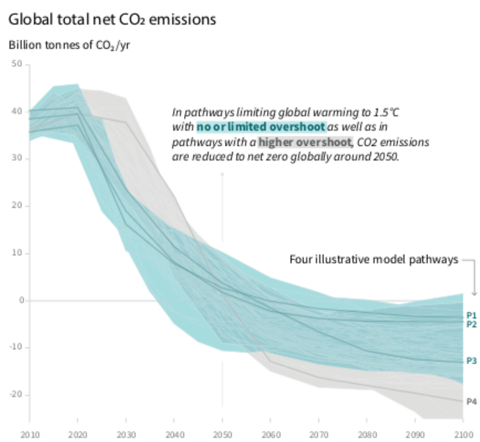
IPCC Special Report - Global Warming of 1.5 °C, Oct. 2018

Decision problems in climate research → Example 1: emission reduction policies

- Too fast reductions may compromise the wealth of one or more upcoming generations but ...
- ... they may promote a transition to societies that are more wealthy, safe, fair and manageable.
- New technologies that significantly reduce the costs of fast emission reductions may become available soon.
- Already taken decisions may not be implemented or they may be implemented with delays.

Decision problems in climate research → Example 1: emission reduction policies

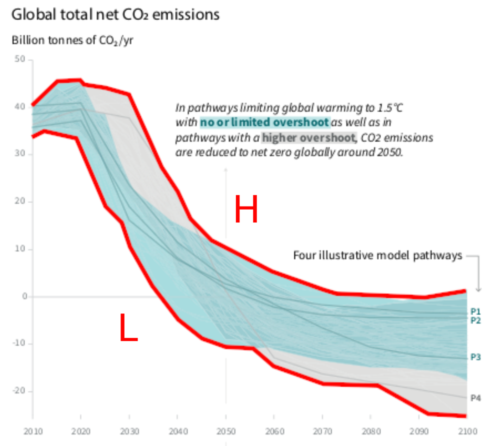
- The implication of these may is that results like this



are very useful to identify a corridor of viable options but also raise a number of difficult questions:

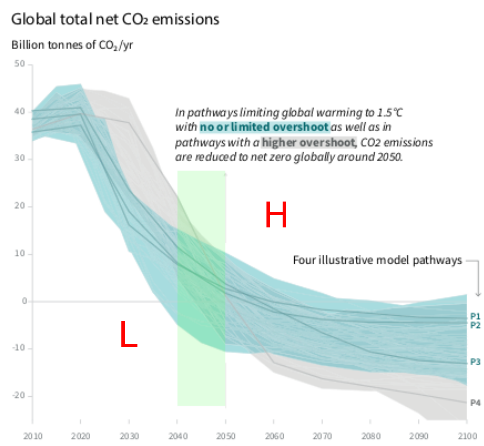
Decision problems in climate research → Example 1: emission reduction policies

- What are the risks associated with the upper (H) and lower (L) boundaries of the emissions corridor? What their costs?

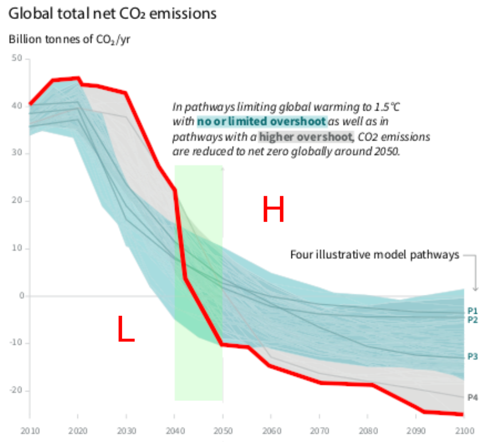


Decision problems in climate research → Example 1: emission reduction policies

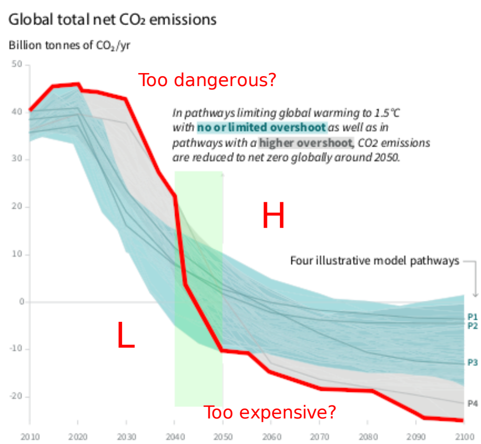
- Assume that, between 2040 and 2050, new technologies makes it possible to reduce GHG emissions at low costs:



- What are "optimal" emission paths under this scenario? What can optimal possibly mean in this context?



- Perhaps are H emissions until 2040 too dangerous? Or L emissions after 2050 too expensive?



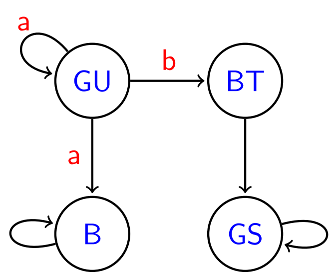
Decision making under uncertainty: basic questions

- In presence of uncertainties, the notion of "optimal path of decisions" becomes devoid of meaning!
- But the notion of optimal decision rule = optimal policy is still meaningful and relevant!
- How do optimal policies look like and what is the impact of uncertainties on optimal policies?
- How do optimal policies change if we account for the fact that technological innovations could become available later or earlier?
- Or that there is a non-zero probability of exceeding critical thresholds even if we stay within the IPCC emission corridor?
- What if decisions are not implemented according to plan or it they are delayed?
- Can we account for these uncertainties rigorously, at least for idealized decision problems?

Example 2: a generation dilemma (Heitzig et al. 2018)

- The world can be in one of four states: GU, GS, B and BT.
- B is a bad state, one in which resources are depleted and the wealth of the societies is low.
- GS is a good, safe state. In GS, plenty of resources are available, societies are wealthy and there is no risk for the world to turn into B, GU or BT.
- GU is a good but unsafe state. In GU, plenty of resources are available, societies are wealthy but there is a significant risk for the world to turn into B.
- BT is a bad but temporary state. In BT, societies are poor but it is known for certain that the next state will be GS.

- A generation in B, BT or GS has no options: the next states will be B, GS and GS, respectively.
- A generation in GU also has two options: a and b. If it picks a, the next generation will possibly be in GU again. But it can also end up in B. If it picks b, the next generation will certainly be in BT.



- What should a generation in GU do? a or b?

Examples 1 and 2: common traits

- Both decision problems have the form of a dilemma.
- In both cases, the consequences of decisions are uncertain.
- Decisions are taken sequentially as time unfolds.
- Decisions might be implemented with delays or not implemented at all.
- Can we exploit these similarities? Can we develop a method for specifying and solving these and similar decisions problems rigorously? What does this mean in the specific examples?

Example 1 revisited: a tentative specification

- A decision maker has to take a sequence of decisions about GHG emissions, one after the other.
- At each decision step, it can only pick up one of two options: L emissions or H emissions of a "safe" emission corridor.
- The decision taken by the decision maker may or may not be implemented during the time until the next decision has to be taken.
- If implemented, L emissions increase cumulated emissions less than H emissions.

- At each step, the decision maker has to choose between L and H emissions on the basis of four data:
  - The current amount of cumulated emissions.
  - The current emission level: L or H.
  - The availability of technologies for reducing emissions.
  - A state of the world which can be either good or bad .

- At the first decision step, the decision maker observes (relatively) low cumulated emissions, H current emissions, unavailable technologies and a good world.
- In this state, the probability that the world turns bad is (relatively) low.
- But if the cumulated emissions increase beyond a critical threshold, the probability that the world becomes bad steeply increases.
- Once the world has reached a bad state, there is no chance to turn back to a good state.
- Similarly, the probability that new technologies become available is low at the first decision steps. It increases steeply after 2040 or, equivalently, after a critical number of steps.
- Once available, technologies stay available for ever.

- Being in a bad world yields less benefits than being in a good world.
- L current emissions yield less benefits (more costs, less growth) than H current emissions.
- Implementing low emissions when technologies are unavailable costs more than implementing emissions when technologies are available.
- The decision maker aim at maximising a sum of the benefits over all decision steps.

More details does not mean a better understanding!

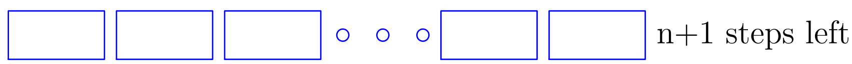
- We have introduced more details but we are still very far from a complete specification and understanding of the decision problem!
- Even if we assume that all the uncertainties that affect the problem have been specified, advising the decision maker requires answering a number of crucial questions:
  - What kind of solutions or advice can we offer to the decision maker?
  - Can we compute these solutions or advice in a rigorous way? What do we need to do so?
  - Can we guarantee that these solutions or advice are correct? What does this mean?

- We answer these questions in three steps:
  - Abstract away the details of specific decision problems.
  - Formulate a whole class of decision problems rigorously.
  - Derive generic, accountable = correct by construction solution methods for these problems.

Abstracting away the details: an informal view

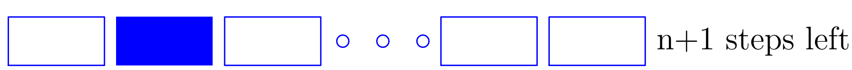
Decision problems in climate research $\rightarrow$ Abstracting away the details: an informal view

There are $n+1$ decision steps to go ...



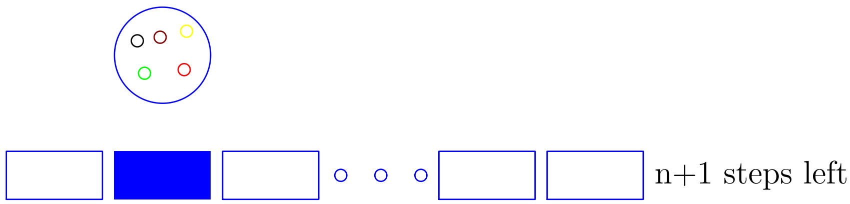
Decision problems in climate research $\rightarrow$ Abstracting away the details: an informal view

... here is the current state,



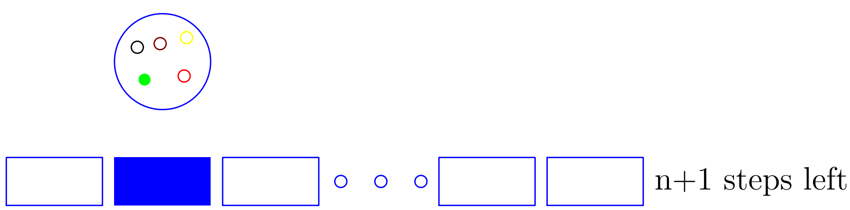
Decision problems in climate research → Abstracting away the details: an informal view

. . . here are your options.

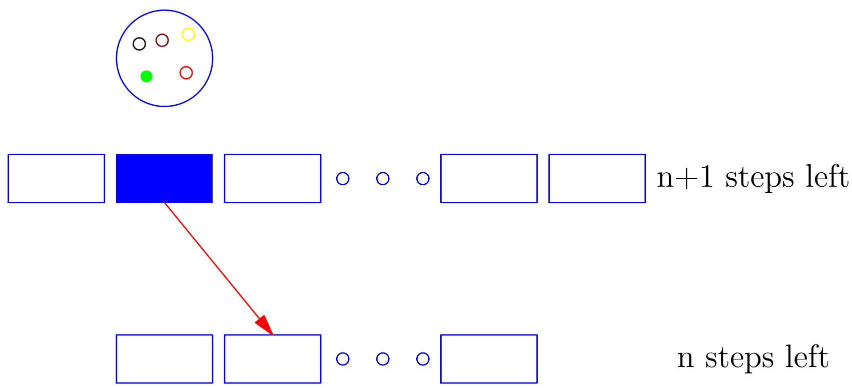


Decision problems in climate research → Abstracting away the details: an informal view

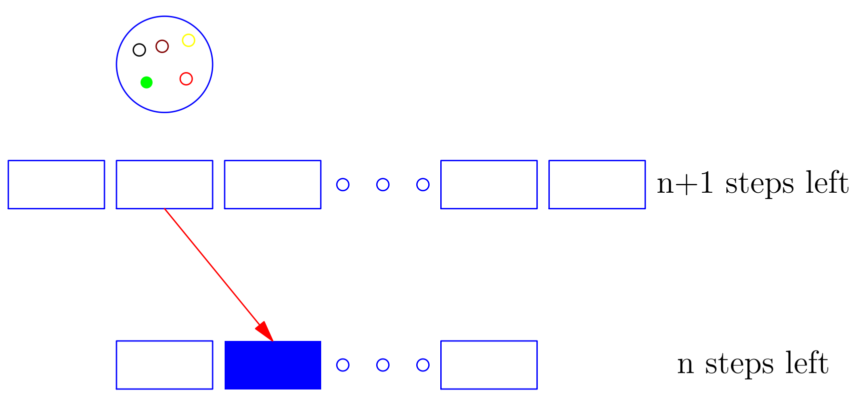
Pick one!



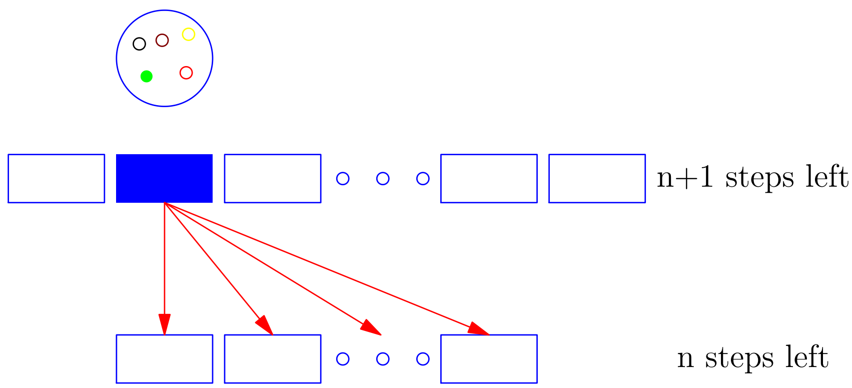
Move to a new state and . . .



. . . collect rewards and face the next decision step!



What if there are more than one next possible states?



Wrap-up

- We have seen two examples of simple but non-trivial decision problems in climate research.
- The problems have been described through informal narratives.
- We have outlined some common features and patterns but ...
- ... we are far from understanding the problems, let apart from being able to solve them!
- We need more formal problem descriptions and general, accountable methods for solving the problems.

Coming up

- In the next lecture we look at mathematical specifications.
- We learn (what it means) to specify problems through simple examples.
- We also get some elementary ideas about proof methods and a little bit of notation.

Lecture 2: Mathematical specifications

Objectives of this lecture

- Look at semi-formal mathematical problem specification
- Learn to carry out simple specification tasks through exercises
- Gain some elementary knowledge of structural induction and equational reasoning as proof method

2.1 Equations, problems and solutions

In mathematics we say that

$$x = 1$$
$$x = -1$$

are solutions of

$$x^2 = 1$$

What does this precisely mean? $x = 1$, $x = -1$ and $x^2 = 1$ are all equations. But they are in certain relations to each other. We have

$$x = 1 \Rightarrow x^2 = 1$$

and also

$$x = -1 \Rightarrow x^2 = 1$$

These implications are what justifies saying that $x = 1$ and $x = -1$ *solve* $x^2 = 1$.

The equation $x = 1$ ($x = -1$) is different from $x^2 = 1$ also from another point of view: the first equation determines *the* value of x directly, without computations.

The equation $x^2 = 1$ specifies a problem: that of finding *values* whose square is one. We can specify the problem a little bit more explicitly:

$$\textbf{Find } x \in \mathbb{R} \textbf{ s.t. } x^2 = 1$$

This is a first example of a *mathematical specification*. As we have seen, the problem has two solutions. We can go one step further and specify the problem of finding the square root of an arbitrary number:

Given $y \in \mathbb{R}$, **find** $x \in \mathbb{R}$ **s.t.** $x^2 = y$

This problem is not solvable. For instance, it is not solvable for $y = -1$. We say that the specification is *infeasible*. The problem here is that the requirements on y are too weak. We can obtain a *feasible* specification if we require y to be non-negative:

Given $y \in \mathbb{R}$, $0 \leqslant y$, **find** $x \in \mathbb{R}$ **s.t.** $x^2 = y$

The problem can now be solved, at least in principle. In practice, computing square roots of arbitrary numbers can be very difficult if we pretend to fulfill $x^2 = y$ exactly. When doing real computations, we typically accept that this equation will only be satisfied up to a certain tolerance. We do not want to deal with these kind of problems here.

2.2 Functions as solutions

The last specification does not say anything about which root shall be found for a given y. For instance, if we just want to look at four input values, say y takes the values $[1, 0, 9, 4]$, then all of

$[-1, 0, 3, 2]$, $[1, 0, -3, -2]$, $[1, 0, 3, 2]$, $[-1, 0, -3, -2]$

are acceptable results according to that specification. Sometimes we want to be more precise and require the solution of a problem to be a *function*. In mathematics, we specify a function by giving its signature and its definition. For instance

$$\begin{aligned} &double \ : \ \mathbb{N} \ \rightarrow \ \mathbb{N} \\ &double \ n = 2 * n \end{aligned}$$

We say that double is of type $\mathbb{N} \rightarrow \mathbb{N}$ or that double maps natural numbers to natural numbers. For $f \ : \ A \rightarrow B$, A and B are called the *domain* and the *codomain* of f.

Notation: we denote function application $f\ (x)$ by juxtaposition $f\ x$!

We can specify the problem of finding a function that computes a square root of arbitary non-negative numbers as e.g.

Find $\sqrt{\ } \ : \ \mathbb{R} \ \rightarrow \ \mathbb{R}$ **s.t.** $\forall y \in \mathbb{R}, \ \ 0 \leqslant y \Rightarrow (\sqrt{y})^2 = y$

or equivalently as

Find $\sqrt{\ } \ : \ \mathbb{R}_{\geq 0} \ \rightarrow \ \mathbb{R}$ **s.t.** $\forall y \in \mathbb{R}_{\geq 0}, \ \ (\sqrt{y}) * (\sqrt{y}) = y$

This problem has two solutions: one function that always computes the negative root and one that always computes the positive square root.

Exercise 2.1. Specify the problem of finding a function that computes both square roots.

Remark: In this subsection, we followed Morgan's introduction to programming from specification [3]. There you can also find more examples and exercises.

2.3 Domain-specific notions

Mathematical specifications can also be applied to clarify notions that are specific to a given application domain. For example:

- What does it mean for $f : X \to Y$ to be a function?
- What does it mean for a strategy to be dominant?
- What does it mean for a climate state to be avoidable?

Often, giving precise answers is not easy. Sometimes, it turns out that we want a whole *family of notions*, not just one notion. The context of the emission problem discussed in lecture 1, for instance, is

- Emission reductions imply different costs and benefits for different countries.
- The highest global benefits are obtained if most countries reduce emissions by certain (optimal, fair, ...) country-specific amounts.
- In this situation most countries have a free-ride opportunity!

The most paradigmatic example of this situation is perhaps the two-players prisoner's dilemma

	D	C
D	(1,1)	(3,0)
C	(0,3)	(2,2)

Table 1: Payoff matrix

Which property makes (D, D) stable and yet undesirable strategies?

Let $S = \{D, C\}$ and $p_1, p_2 : S \times S \to \mathbb{R}$ payoff functions. A **strategy profile** $(x, y) \in S \times S$ is a **Nash equilibrium** iff $\forall x', y' \in S$, $p_1\ (x', y) \leqslant p_1\ (x, y)$ and $p_2\ (x, y') \leqslant p_2\ (x, y)$.

Remark: Note that the definition of Nash equilibrium depends on a binary operator $\leqslant$. (Which properties should this binary operator reasonably have?)

Exercise 2.2. Modify the payoffs of (C, C) in Table 1 for (C, C) to become a Nash equilibrium.

Exercise 2.3. Generalize the notion of Nash equilibrium to an arbitrary number of players.

Let X denote a set of states that a decision maker can observe. For instance, X could be a tuple of numbers that represent aggregated measures or indicators of wealth, inequality, environmental stress, etc.

Let Y denote the options available to the decision maker. For simplicity, we assume that she has the same options in all states $x \in X$.

Functions that associate an option to every state are called *policies*.

Let $val : X \to Y \to \mathbb{R}$ be a *value function*: $val\ x\ y$ denotes the value of taking decision y in state x.

A policy $p : X \to Y$ is called *optimal* w.r.t to val if it yields controls that are better or as good as any other control for all states.

Exercise 2.4. Give a mathematical specification of the notion of optimality for policies.

If Y is finite and non-empty, one can implement

$$
\begin{aligned}
max &: (Y \to \mathbb{R}) \to \mathbb{R} \\
argmax &: (Y \to \mathbb{R}) \to Y
\end{aligned}
$$

that fulfill

$$
\begin{aligned}
&\forall f : Y \to \mathbb{R},\ \forall y \in Y,\ f\ y \leqslant max\ f \\
&\forall f : Y \to \mathbb{R},\ f\ (argmax\ f) = max\ f
\end{aligned}
$$

Exercise 2.5. Find a function $p : (X \to Y \to \mathbb{R}) \to (X \to Y)$ that such that $p\ val$ is an optimal policy w.r.t to val for arbitrary val. Prove that $p\ val$ is indeed optimal.

Exercise 2.6. Let $\mathit{Fin}\ n$ be the set of natural numbers smaller than n:

$$\begin{aligned}
&\mathit{Fin}\ 0 = \{\,\}\\
&\mathit{Fin}\ 1 = \{0\}\\
&\mathit{Fin}\ 2 = \{0, 1\}\\
&\ldots\\
&\mathit{Fin}\ n = \{0 \ldots n - 1\}
\end{aligned}$$

Give a mathematical specification of the notion of finiteness for a set X. Begin with

A set X is **finite** iff ...

Exercise 2.7. Apply the specification of finiteness from Exercise 2.6 to show that the two elements set $X = \{\mathit{Up}, \mathit{Down}\}$ is finite.

2.4 Mathematical specifications and modelling

In agent-based models of green growth (opinion formation, consume, etc.) it is common to equip a set of agents with certain features. Thus, for instance, agents can be employed or unemployed

$$\mathit{status} : \mathit{Agent} \to \{\mathit{Employed}, \mathit{Unemployed}\}$$

... have certain incomes and

$$\mathit{income} : \mathit{Agent} \to \mathbb{R}_{\geq 0}$$

... consume behaviors

$$\mathit{buy} : \mathit{Agent} \to \mathit{Prob}\ \{\mathit{GreenCar}, \mathit{BrownCar}, \mathit{NoCar}\}$$

Here $\mathit{Prob}\ X$ represents finite probability distributions over an arbitrary set X. Let $\mathit{Event}\ X = X \to \mathit{Bool}$ and

$$\mathit{prob} : \mathit{Prob}\ X \to \mathit{Event}\ X \to [0, 1]$$

be the generic function that computes the probability of an event $e : \mathit{Event}\ X$ according to a given probability distribution. Thus, for $d \in \mathit{Prob}\ X$ and $e \in \mathit{Event}\ X$

$prob\ d\ e$

represents the probability of e according to d. We want to implement an agent-based model in which agents with higher incomes are more likely to buy green cars than agents with lower incomes.

We also would like to specify that unemployed agents are less likely to buy a brown car than employed agents.

Exercise 2.8. Express these model requirements as mathematical specifications using the model-specific functions $status$, $income$, buy and the generic function $prob$.

2.5 Equational reasoning

Equational Reasoning is the proof method encouraged by the "Algebra of Programming" community [1] ($\rightsquigarrow$ see L3E2) for reasoning about systematic, correctness preserving program transformations. Originally this form of calculation with programs was done on a semi-formal meta-level (by semi-formal we mean: on paper, not inside an implemented type theory/proof assistant).

It comes with a distinctive style of presenting proofs with justification of every transformation step (just as one would do in school when solving equations).

A very important ingredient of this algebraic approach to program correctness is *structural induction*. Here, we will look at a simple example using this technique, presented in equational reasoning style.

We will prove a property of exponentiation.

The exponentiation with natural numbers fulfills the properties

(1) $\forall x \in \mathbb{R},\ x^0 = 1$
(2) $\forall x \in \mathbb{R},\ m \in \mathbb{N},\ x^{1+m} = x * x^m$

From (1), (2) and the algebraic properties of $*$ and $+$ we can show that

$$\forall x \in \mathbb{R},\ m, n \in \mathbb{N},\ x^{m+n} = x^m * x^n$$

The proof is by induction on m. We first show the base case ($m = 0$)

$$\forall x \in \mathbb{R},\ n \in \mathbb{N},\ x^{0+n} = x^0 * x^n$$

Then we prove the induction step ($m \Rightarrow 1 + m$)

$$\forall x \in \mathbb{R},\ n \in \mathbb{N},\ x^{m+n} = x^m * x^n$$
$$\Rightarrow$$
$$\forall x \in \mathbb{R},\ n \in \mathbb{N},\ x^{(1+m)+n} = x^{1+m} * x^n$$

The proofs are obtained by *equational reasoning*. Let's start with the "difficult" (induction step) case:

$$
\begin{aligned}
& x^{(1+m)+n} \\
& \quad = \quad \{ \text{ Associativity of } + \} \\
& x^{1+(m+n)} \\
& \quad = \quad \{ \text{ Property (2) } \} \\
& x * x^{m+n} \\
& \quad = \quad \{ \text{ Induction hypothesis } \} \\
& x * (x^{m} * x^{n}) \\
& \quad = \quad \{ \text{ Associativity of } * \} \\
& (x * x^{m}) * x^{n} \\
& \quad = \quad \{ \text{ Property (2) } \} \\
& x^{1+m} * x^{n}
\end{aligned}
$$

Exercise 2.9. Prove the base case.

2.6 Coming up

The next lecture is an introduction to functional programming.

We use Idris [2] as a specification and programming language.

Solutions

Exercise 2.1:

The specification of a function *allsqrts* that returns the set of all possible square roots of a positive real number:

$$\textbf{Find } allsqrts : \mathbb{R} \rightarrow \mathcal{P}\,\mathbb{R} \ \textbf{ s.t. } \forall y \in \mathbb{R}_{\geq 0},\ \ \forall x \in \mathbb{R}, x^2 = y \Leftrightarrow x \in allsqrts\ (y)$$

Exercise 2.2:

	D	C
D	(1,1)	(3,0)
C	(0,3)	(3,3)

Exercise 2.3:

Let $n \in \mathbb{N}$ be the number of players and S_i, $i \in \{1, ..., n\}$ the strategy set of the i-th player. Let $p_i : S_1 \times S_2 \times ... \times S_n \rightarrow \mathbb{R}$, $i \in \{1, ..., n\}$ the payoff function of the i-th player. A **strategy profile** $(x_1, ..., x\_n) \in S_1 \times ... \times S_n$ is a **Nash equilibrium** iff $\forall i \in \{1, ..., n\}$ and $\forall x_i' \in S_i$, $p_i\ (x_1, ..., x_{i-1}, x_i', x_{i+1}, ...x\_n) \leqslant p_i\ (x_1, ..., x_{i-1}, x_i, x_{i+1}, ...x\_n)$.

Exercise 2.4:

$$p : X \rightarrow Y \ \textbf{optimal} \text{ iff } \forall x : X, \forall y : Y,\ val\ x\ y \leqslant val\ x\ (p\ x)$$

Exercise 2.5:

With

$$p : (X \rightarrow Y \rightarrow \mathbb{R}) \rightarrow (X \rightarrow Y)$$
$$p\ val\ x = argmax\ (val\ x)$$

and for an arbitrary $x : X$, one has

$$\forall\ y \in Y,\ \ (val\ x)\ y \leqslant max\ (val\ x)$$

because of (a) with $f = val\ x$. Because of (b) and, again, for $f = val\ x$, one has

$$max\ (val\ x) = (val\ x)\ (argmax\ (val\ x))$$

Thus, we conclude

$$\forall \ x \ : \ X, \forall \ y \in Y, \ (val \ x) \ y \leqslant (val \ x) \ (argmax \ (val \ x))$$

But $argmax \ (val \ x) = p \ val \ x$ by definition of p and thus we have

$$\forall \ x \ : \ X, \forall \ y \in Y, \ val \ x \ y \leqslant val \ x \ (p \ val \ x)$$

that is, $p \ val$ is optimal w.r.t. val as required.

Exercise 2.6:

A set X is **finite** iff ...

$$\ldots \ \exists \ n \in \mathbb{N}, \exists \ f \ : \ X \ \rightarrow \ Fin \ n, \ \ isIso \ f$$

where for a function $f \ : \ A \ \rightarrow \ B$

$$isIso \ f \Leftrightarrow \exists \ g \ : \ B \ \rightarrow \ A, \ \ f \circ g = id \wedge g \circ f = id$$

Exercise 2.7:

In order to show $finite \ \{ Up, Down \}$, we first have to choose a natural number:

Choose $n := 2$

then choose a function $f \ : \{ Up, Down \} \ \rightarrow \ Fin \ 2$:

Choose
$f \ : \{ Up, Down \} \ \rightarrow \ Fin \ 2$, where
$f \ Up = 0 \ f \ Down = 1$

and show that f is an isomorphism according to definition given above. I.e. we have to choose a function $g \ : \ Fin \ 2 \ \rightarrow \ \{ Up, Dpwn \}$ and show that f and g are mutual inverses.

Choose $g \ : \ Fin \ 2 \ \rightarrow \ \{ Up, Down \}$, where
$g \ 0 = Up \ g \ 1 = Down$

Now show (pointwise) that

$f \circ g = id$ by
$\forall x \in Fin \ 2, f \ (g \ x) = x$

Case $x = 0$:

$f\ (g\ 0)) = f\ \mathit{Up} = 0$ by definition of f and g

Case $x = 1$:

$f\ (g\ 1) = f\ \mathit{Down} = 1$ by definition of f and g
and
$g \circ f = \mathit{id}$ by
$\forall x \in \{\mathit{Up}, \mathit{Down}\}, g\ (f\ x) = x$

Case $x = \mathit{Up}$:

$g\ (f\ \mathit{Up})) = g\ 0 = \mathit{Up}$ by definition of f and g

Case $x = \mathit{Down}$:

$g\ (f\ \mathit{Down}) = g\ 1 = \mathit{Down}$ by definition of f and g

Exercise 2.8:

Let $\mathit{eGreen}\ : \{\mathit{GreenCar}, \mathit{BrownCar}, \mathit{NoCar}\} \ \rightarrow\ \mathit{Bool}$ be an event such that

$$\mathit{eGreen}\ \mathit{GreenCar} = \mathit{true}$$

and
$\mathit{eBrown}\ : \{\mathit{GreenCar}, \mathit{BrownCar}, \mathit{NoCar}\} \ \rightarrow\ \mathit{Bool}$ be an event such that

$$\mathit{eBrown}\ \mathit{BrownCar} = \mathit{true}$$

Then we require that the following implications hold:

$\forall a_1, a_2 \in \mathit{Agent}$,
$\mathit{income}\ (a_1) > \mathit{income}\ (a_2)$
$\Rightarrow \mathit{prob}\ (\mathit{buy}\ (a_1))\ \mathit{eGreen} > \mathit{prob}\ (\mathit{buy}\ (a_2))\ \mathit{eGreen}$

$\forall a_1, a_2 \in \mathit{Agent}$,
$\mathit{status}\ (a_1) = \mathit{Unemployed} \wedge \mathit{status}\ (a_2) = \mathit{Employed}$
$\Rightarrow \mathit{prob}\ (\mathit{buy}\ (a_1))\ \mathit{eBrown} < \mathit{prob}\ (\mathit{buy}\ (a_2))\ \mathit{eBrown}$

Exercise 2.9:

x^{0+n}

$\quad = \quad \{$ Zero left neutral element of $+$ $\}$

x^n

$\quad = \quad \{$ One left neutral element of $*$ $\}$

$1 * x^n$

$\quad = \quad \{$ Property (1) $\}$

$x^0 * x^n$

References

[1] Richard S. Bird and Oege de Moor. *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall, 1997.

[2] Edwin Brady. Programming in Idris : a tutorial, 2013.

[3] Carroll Morgan. *Programming from specifications*. Prentice Hall International Series in computer science. Prentice Hall, 1990.

Lecture 3: A crash course in functional programming

Objectives of this lecture

- Get acquainted with the basic usage of functions, function composition, and higher order functions in functional programming
- Learn about primitive and inductive data types in Idris
- Get to know two forms of polymorphism and learn how one of them is related to generic programming
- Learn about the *List* and *Vector* data types

3.1 Imperative and functional languages

Traditionally, computer scientists use(d) to distinguish a number of different programming *paradigms* (e.g. precedural, object-oriented, functional or declarative). As of today these distinctions have become somewhat blurred, with modern programming languages often integrating features from different paradigms.

However, it still seems valid to contrast against each other an *imperative* and a *functional way of thinking* about the programs.

Climate scientists and modelers are often well acquainted with *imperative* programming languages/style of programming.

Very roughly, imperative programming is a method of specifying what a computing machine shall do in terms of *instructions* and *execution procedures*.

In functional programming, one specifies what a computing machine shall do in terms of *functions* and their *application* and *composition*, with an emphasis on inductive definitions and algebraic structure.

In this lecture we are going to learn some basics of FP using Idris [2] as a language.

Idris is a strongly typed functional programming language. A prototype implementation appeared in 2008, the current implementation began in 2011. Thus, Idris is a relatively young language. However, its roots are much older and in fact reach back to the quest for a logical foundation of mathematics in the beginning of the 20th century. ($\rightsquigarrow$ L4E1 for historical background)

3.2 Expressions and their types

At the core of all programming languages is a sublanguage of *expressions* like

$1 + 2$
`"Hello"`
$[1, 7, 3, 8]$

$$\lambda x \Rightarrow 2 * x + 1$$

In functional languages this core is expressive enough to implement almost all programs you may want to write.

In strongly typed languages like Idris each *valid* expression has a *type*. The judgment $e : t$ states that the expression e has type t.

$$\begin{array}{ll} 1+2 & : \mathbb{N} \\ \texttt{"Hello"} & : \mathit{String} \\ [1,7,3,8] & : \mathit{List}\ \mathbb{N} \\ \lambda x \Rightarrow 2 * x & : \mathit{Integer} \ \to \ \mathit{Integer} \end{array}$$

Most of the power of Idris comes from its type-checker which can check these judgments for very complex expressions e and types t.

Remark: Note that the above "code" in the pdf-document is "prettified" by preprocessing and slightly differs from the actual Idris code. E.g.

```
\x => t
```

is printed as

$$\lambda\ x \Rightarrow t$$

instead.

3.3 Function application and currying

In Idris (and several other functional languages like Haskell and Agda) the notation for function application is juxtaposition. Thus,

$$f\ x$$

instead of

$$f\ (x)$$

denotes the application of the function f to the argument x. Apart from this notational difference, parantheses in Idris play the same role as in mathematics: enclosing sub-expressions to resolve operator precedence.

A function of $n > 1$ arguments in mathematics is usually considered as a function taking one $n-$tuple as argument. In Idris, we often use nested function application

$$(g\ x)\ y$$

instead of

$$g\ (x, y)$$

That is, functions "of n arguments" take one argument at a time, resulting in a function "of $n-1$ arguments" which in turn is applied to the next input. (Both ways of looking at functions have their merit, though, and we will come back to the relation between the two.)

As an aside, $(g\ x)\ y$ can also be written $g\ x\ y$ because function application is left-associative.

Exercise 3.1. What is the type of g?

Here a, b and c are arbitrary types. Examples of functions that take two arguments are infix operators like $(+)$. Infix operators can be written between their first and second arguments:

$$(+)\ 1\ 2 = 1 + 2$$

3.4 Function composition and higher order functions

Function composition is another example of an infix operator. In Idris (but also in Haskell, Agda and plain mathematics), function composition is denoted by a dot:

$$(\circ) : (b \to c) \to (a \to b) \to a \to c$$
$$(\circ)\ g\ f\ x = g\ (f\ x)$$

Function composition is an example of a *higher order* function. It takes two functions and returns their composite.

Higher order functions take one or more functions as arguments and/or return a function. Every function of two or more arguments can be partially applied and is thus a higher order function. Thus, if

$$g : a \to (b \to c) = a \to b \to c$$

then

$$g\ x : b \to c$$

We can easily turn g into an equivalent function g' that takes as arguments pairs

$$g' : (a, b) \to c$$
$$g'\ (x, y) = g\ x\ y$$

In Idris (an most functional languages) two high order functions convert between the two forms:

$$uncurry : (a \to b \to c) \to (a, b) \to c$$

$$curry : ((a, b) \to c) \to a \to b \to c$$

Remark: Unfortunately, in Idris (as in Haskell), the product type constructor and the product term constructor are both denoted by $(\cdot, \cdot)$. So in the above, (a, b) is the product type which could be thought of as cartesian product $a \times b$, while (x, y) is a pair of values, where x is of type a and y of type b.

Exercise 3.2. Define *uncurry* and *curry*.

Exercise 3.3. Show with equational reasoning (as introduced in lecture 2) that $\mathit{uncurry} \circ \mathit{curry} = \mathit{curry} \circ \mathit{uncurry} = \mathit{id}$.

If $g' : (a, b) \to c$, $\mathit{curry}\ g' : a \to b \to c$ is called the *curried* form of g'. Similarly $\mathit{uncurry}\ g : (a, b) \to c$ is called the *uncurried* form of $g : a \to b \to c$. [1]

3.5 Polymorphism

Some functions can be used for more than one type of data – they are *polymorphic*. However, in Idris and similar languages, one distinguishes between two conceptually different forms of polymorphism.

3.5.1 Parametric polymorphism and generic programs

The notion of *parametric polymorphims* relates to functions whose defiition does not depend on the structure of the underlying datatypes. (In category theoretical terms, they can be seen as *natural transformations* [6], $\rightsquigarrow$ see LE32)

Function composition, *curry* and *uncurry* are examples of this kind of *polymorphic* function. The symbols a, b and c in

$$(\circ) : (b \to c) \to (a \to b) \to a \to c$$

denote *implicit type variables*. The judgment is in fact an abbreviation of

$$(\circ) : \{a, b, c : \mathit{Type}\} \to (b \to c) \to (a \to b) \to a \to c$$

which is itself an abbreviation of

$$\begin{aligned}(\circ) : \{a : \mathit{Type}\} \to \{b : \mathit{Type}\} \to \{c : \mathit{Type}\} \to \\ (b \to c) \to (a \to b) \to a \to c\end{aligned}$$

[1] The names stem from Haskell Brooks Curry (1900-1982) but the idea that every function of more than one variable can be understood as a (higher order) of only one variable was apparently first used by Frege [5] and studied further by Schönfinkel [4]. But Curry's combinatory logic [3] ws more influential with respect to functional programming which might explain why Reynold's chose this name in his influential 1972 paper [1] which then was adopted by the community.

We say that $(\circ)$ is a *generic* program. It computes the composition $f \circ g$ of any two functions f and g as long as the domain of f coincides with the *codomain* of g.

A simpler example of polymorphic functions are the projection functions for pairs:

$$
\begin{array}{lcl}
\mathit{fst} : (s,t) & \to & s \quad \text{-- Remark: } s \times t \to s \\
\mathit{fst}\ (x,y) & = & x
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{snd} : (s,t) & \to & t \\
\mathit{snd}\ (x,y) & = & y
\end{array}
$$

Here too, s and t are implicit type variables and the above are abbreviations for

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

and similarly for snd. In Idris we can afford to write abbreviated forms because the type checker can often infer the type of implicit arguments. When needed, such types can be supplied within curly braces.

Polymorphic functions are central to generic programming. Generic programming is a methodology that aims at improving sofware reuse and correctness while at the same time reducing documentation efforts.

IdrisLibs [?] provides, among others, generic methods for specifying and solving sequential decision problems.

3.5.2 Aside: Constrained polymorphism and type classes

There often is another form of polymorphism available in modern programming languages which concerns the overloading of operator symbols. In the context of functional programming, this is often referred to as *ad-hoc polymorphism* and provided by the so-called *type classes*. In Idris these go by the name of *interfaces*. As we do not need them for now, we just mention this concept for completeness and in contrast to the concept of parametric polymorphism.

3.6 Data types

Beside providing methods to define, compose and apply functions, most functional programming languages support the definition of new data types.

We can introduce new types that extend the language via inductive definitions like

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

which defines the type of natural numbers in Idris. The definition coincides with the usual inductive definition of $\mathbb{N}$: It states that

* $\mathbb{N}$ is a type.

* Z (zero) is a value of type $\mathbb{N}$.

* $\forall\ n\ :\ \mathbb{N}$, $S\ n$ (the successor of n) is a value of type $\mathbb{N}$.

Z and S are called the *data constructors* of $\mathbb{N}$. Data constructors are *disjoint*: no natural number can be both zero and a successor. Moreover every natural number is either zero or the successor of another natural number. These properties make it possible to define *total* functions via *pattern matching*:

$$
\begin{array}{l}
plus\ :\ \mathbb{N}\ \to\ \mathbb{N}\ \to\ \mathbb{N} \\
plus\ Z \qquad n = n \\
plus\ (S\ m)\ n = S\ (plus\ m\ n)
\end{array}
$$

(This amounts to specifying functions by recursion equations which are accepted by the type checker, if it can guarantee termination of the recursive calls because of a syntactic monotonicity criterion.)

3.7 Lists

$$
\begin{array}{l}
\textbf{data}\ List\ :\ Type\ \to\ Type\ \textbf{where} \\
\quad Nil\ :\ List\ a \\
\quad (::)\ :\ a\ \to\ List\ a\ \to\ List\ a
\end{array}
$$

* $\forall\ a\ :\ Type$, $List\ a$ is a type.

* $\forall\ a\ :\ Type$, Nil (the empty list) is a value of type $List\ a$.

* $\forall\ a\ :\ Type,\ x\ :\ a,\ xs\ :\ List\ a$, $x :: xs$ is a value of type $List\ a$.

Thus, for instance

$$
\begin{array}{l}
xs\ :\ List\ \mathbb{N} \\
xs = Nil
\end{array}
$$

$$
\begin{array}{l}
ys\ :\ List\ \mathbb{N} \\
ys = Z :: ((S\ Z) :: Nil)
\end{array}
$$

Notation: we usually write $[3, 0, 1]$ instead of $(S\ (S\ (S\ Z))) :: (Z :: ((S\ Z) :: Nil))$.

In Idris, lists come with a number of useful predefined functions and abbreviations:

3.7.1 List comprehension

$$
\begin{array}{l}
Idris > [0\,..\,3] \\
[0, 1, 2, 3]\ :\ List\ Integer
\end{array}
$$

$$\mathit{Idris} > [5\mathinner{..}2]$$
$$[5,4,3,2] \;:\; \mathit{List}\ \mathit{Integer}$$

$$\mathit{Idris} > [2 * n \mid n \leftarrow [1\mathinner{..}9]]$$
$$[2,4,6,8,10,12,14,16,18] \;:\; \mathit{List}\ \mathit{Integer}$$

$$\mathit{Idris} > \mathit{map}\ (2*)\ [1\mathinner{..}9]$$
$$[2,4,6,8,10,12,14,16,18] \;:\; \mathit{List}\ \mathit{Integer}$$

Exercise 3.4. What is the type of map?

3.7.2 Basic operations

$$\mathit{length} \;:\; \mathit{List}\ a \;\to\; \mathbb{N}$$

Exercise 3.5. Implement length.

$$\begin{array}{l}
(+\!\!+) \;:\; \mathit{List}\ a \;\to\; \mathit{List}\ a \;\to\; \mathit{List}\ a \\
(+\!\!+)\ \mathit{Nil}\qquad\quad \mathit{ys} = \mathit{ys} \\
(+\!\!+)\ (x :: \mathit{xs})\ \mathit{ys} = x :: (\mathit{xs} +\!\!+ \mathit{ys})
\end{array}$$

Exercise 3.6. What is the result of $[3,1] +\!\!+ [2,0,1]$? Give a computational proof of your conjecture (i.e. by step-wise evaluation according to the definition of $(+\!\!+)$).

$$\begin{array}{l}
\mathit{concat} \;:\; \mathit{List}\ (\mathit{List}\ a) \;\to\; \mathit{List}\ a \\
\mathit{concat}\ \mathit{Nil}\qquad\quad\; = \mathit{Nil} \\
\mathit{concat}\ (\mathit{xs} :: \mathit{xss}) = \mathit{xs} +\!\!+ \mathit{concat}\ \mathit{xss}
\end{array}$$

$$\begin{array}{l}
\mathit{map} \;:\; (a \;\to\; b) \;\to\; \mathit{List}\ a \;\to\; \mathit{List}\ b \\
\mathit{map}\ f\ \mathit{Nil}\qquad\; = \mathit{Nil} \\
\mathit{map}\ f\ (x :: \mathit{xs}) = f\ x :: \mathit{map}\ f\ \mathit{xs}
\end{array}$$

Exercise 3.7. Show that $map\ id = id$.

Exercise 3.8. Show that $map\ (f \circ g) = map\ f \circ map\ g$.

3.8 Vectors

In many cases, one would like to operate with lists of specific lengths.

For instance, require a function

$$zip : List\ a \to List\ b \to List\ (a, b)$$

to only take arguments of the same length. This can be done by encoding the length of a list in its type:

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

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

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

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

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

$$\begin{aligned}&head : Vect\ (S\ n)\ a \to a\\ &head\ (x :: xs) = x\end{aligned}$$

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

Exercise 3.9. Implement a $tail$ function that computes the tail of a non-empty vector.

Exercise 3.10. It is easy to see that $\forall n : \mathbb{N}$, $a : \mathit{Type}$, $v : \mathit{Vect}\ (S\ n)\ a$, $\mathit{head}\ v :: \mathit{tail}\ v = \mathrm{v}$. Give a formal proof (like in section 2.5). What happens in the case $v = \mathit{Nil}$?

3.9 Coming up

The next lecture will be an introduction to dependently-typed programming and theorem proving.

Solutions

Exercise 3.1:

$$g : a \rightarrow (b \rightarrow c) = a \rightarrow b \rightarrow c$$

Exercise 3.2:

$$uncurry\ g\ (x, y) = g\ x\ y$$

$$curry\ g'\ x\ y = g'\ (x, y)$$

Exercise 3.3:

$$\begin{aligned}
&(uncurry \circ curry)\ g'\ (x, y)\\
&\quad = \quad \{\ \text{Def. composition}\ \}\\
&uncurry\ (curry\ g')\ (x, y)\\
&\quad = \quad \{\ \text{Def. uncurry}\ \}\\
&(curry\ g')\ x\ y\\
&\quad = \quad \{\ \text{Def. curry}\ \}\\
&g'\ (x, y)
\end{aligned}$$

$$\begin{aligned}
&(curry \circ uncurry)\ g\ x\ y\\
&\quad = \quad \{\ \text{Def. composition}\ \}\\
&curry\ (uncurry\ g)\ x\ y\\
&\quad = \quad \{\ \text{Def. curry}\ \}\\
&(uncurry\ g)\ (x, y)\\
&\quad = \quad \{\ \text{Def. uncurry}\ \}\\
&g\ x\ y
\end{aligned}$$

Exercise 3.4:

$$map : (a \rightarrow b) \rightarrow List\ a \rightarrow List\ b$$

Exercise 3.5:

$$\begin{aligned}
&length\ Nil &&= Z\\
&length\ (x :: xs) &&= S\ (length\ xs)
\end{aligned}$$

Exercise 3.6:

$$[3,1] ⧺ [2,0,1] = [3,1,2,0,1]$$

$$\begin{aligned}
&[3,1] ⧺ [2,0,1] \\
&\quad = \quad \{ \text{ Syntax } \} \\
&(3 :: (1 :: \mathit{Nil})) ⧺ (2 :: (0 :: (1 :: \mathit{Nil}))) \\
&\quad = \quad \{ \text{ Def. } (++), \text{ case } 2 \} \\
&3 :: ((1 :: \mathit{Nil}) ⧺ (2 :: (0 :: (1 :: \mathit{Nil})))) \\
&\quad = \quad \{ \text{ Def. } (++), \text{ case } 2 \} \\
&3 :: (1 :: (\mathit{Nil} ⧺ (2 :: (0 :: (1 :: \mathit{Nil}))))) \\
&\quad = \quad \{ \text{ Def. } (++), \text{ case } 1 \} \\
&3 :: (1 :: (2 :: (0 :: (1 :: \mathit{Nil})))) \\
&\quad = \quad \{ \text{ Syntax } \} \\
&[3,1,2,0,1]
\end{aligned}$$

Exercise 3.7:

The Nil case:

$$\begin{aligned}
&\mathit{map}\ \mathit{id}\ \mathit{Nil} \\
&\quad = \quad \{ \text{ Def. of } \mathit{map}, \text{ case } 1 \} \\
&\mathit{Nil} \\
&\quad = \quad \{ \text{ Def. of } \mathit{id} \text{ on lists } \} \\
&\mathit{id}\ \mathit{Nil}
\end{aligned}$$

The $(::)$ case:

$$\begin{aligned}
&\mathit{map}\ \mathit{id}\ (x :: \mathit{xs}) \\
&\quad = \quad \{ \text{ Def. of } \mathit{map}, \text{ case } 2 \} \\
&\mathit{id}\ x :: \mathit{map}\ \mathit{id}\ \mathit{xs} \\
&\quad = \quad \{ \text{ Def. of } \mathit{id} \text{ on list elements } \} \\
&x :: \mathit{map}\ \mathit{id}\ \mathit{xs} \\
&\quad = \quad \{ \text{ induction hypothesis } \} \\
&x :: \mathit{xs} \\
&\quad = \quad \{ \text{ Def. of } \mathit{id} \text{ on lists } \} \\
&\mathit{id}\ (x :: \mathit{xs})
\end{aligned}$$

Exercise 3.8:

The Nil case:

$$
\begin{aligned}
& map\ (f \circ g)\ Nil \\
& \quad = \quad \{ \text{ Def. of } map \text{, case 1 } \} \\
& Nil \\
& \quad = \quad \{ \text{ Def. of } map \text{, case 1, reverse } \} \\
& map\ f\ Nil \\
& \quad = \quad \{ \text{ Def. of } map \text{, case 1, reverse, replace } \} \\
& map\ f\ (map\ g\ Nil) \\
& \quad = \quad \{ \text{ Def. composition } \} \\
& (map\ f \circ map\ g)\ Nil
\end{aligned}
$$

The (::) case:

$$
\begin{aligned}
& map\ (f \circ g)\ (x :: xs) \\
& \quad = \quad \{ \text{ Def. of } map \text{, case 2 } \} \\
& (f \circ g)\ x :: map\ (f \circ g)\ xs \\
& \quad = \quad \{ \text{ Def. of } \circ \} \\
& f\ (g\ x) :: map\ (f \circ g)\ xs \\
& \quad = \quad \{ \text{ induction hypothesis } \} \\
& f\ (g\ x) :: (map\ f \circ map\ g)\ xs \\
& \quad = \quad \{ \text{ Def. of } \circ \} \\
& f\ (g\ x) :: map\ f\ (map\ g\ xs) \\
& \quad = \quad \{ \text{ Def. of } map \text{, case 2 } \} \\
& map\ f\ (g\ x :: map\ g\ xs) \\
& \quad = \quad \{ \text{ Def. of } map \text{, case 2 } \} \\
& map\ f\ (map\ g\ (x :: xs)) \\
& \quad = \quad \{ \text{ Def. of } \circ \} \\
& (map\ f \circ map\ g)\ (x :: xs)
\end{aligned}
$$

Exercise 3.9:

$$
\begin{aligned}
& tail\ :\ Vect\ (S\ n)\ a\ \rightarrow\ Vect\ n\ a \\
& tail\ (x :: xs) = xs
\end{aligned}
$$

Exercise 3.10:

In an formal-logic proof, we would need to treat case $v = Nil$ which leads to a contradiction, since Nil is of type $Vect\ Z\ a \neq Vect\ S\ n\ a$. If this is not necessary in Idris, then because the type-checker is able to treat such impossible cases for us.

So we just have to look at the case $vs = v :: vs'$:

$$
\begin{aligned}
& head\ (v :: vs') :: tail\ (v :: vs') \\
& = \quad \{ \text{ Def. of head } \} \\
& v :: tail\ (v :: vs') \\
& = \quad \{ \text{ Def. of tail } \} \\
& v :: vs' \\
& = \quad \{ \text{ Hypothesis } vs = v :: vs' \ \} \\
& vs
\end{aligned}
$$

In Idris the following suffices, though (the above is just the work the type-checker is doing in the background and the programmer is doing in her head...):

$$
\begin{aligned}
& headTailId\ : (n\ :\ \mathbb{N})\ \to\ (a\ :\ Type)\ \to\ (vs\ :\ Vect\ (S\ n)\ a) \\
& \quad \to\ head\ vs :: tail\ vs = vs \\
& headTailId\ n\ a\ (v :: vs') = Refl
\end{aligned}
$$

References

[1] Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, Dec 1998.

[2] Edwin Brady. Programming in Idris : a tutorial, 2013.

[3] H.B. Curry and R. Feys. *Combinatory Logic*. Number v. 1 in Combinatory Logic. North-Holland Publishing Company, 1958.

[4] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, pages 305–316, 1924.

[5] W. Orman van Quine. *Introduction to Moses Schönfinkel's "Bausteine der mathematischen Logik"*, pages 355–357. Source books in the history of the sciences. Harvard University Press, 1967.

[6] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.

Lecture 4: Dependent types and machine-checkable specifications

Objectives of this lecture

- Get acquainted with dependent types
- Learn how to formulate mathematical specifications using dependent types and do first proofs in Idris
- Deepen theoretical background about correspondence between logic and type theory, the importance of totality and termination, and the issue of extensional equality of functions.

4.1 Dependent types

In a nutshell, *dependent types* are types that depend on values. We have already seen examples of dependent types in lecture 3. For instance, both the type of the argument of

$$tail : Vect\ (S\ n)\ a \ \rightarrow\ Vect\ n\ a$$

and the type of its result depend on the *values* $n : \mathbb{N}$ and $a : Type$: the type $Vect$ is a dependent type.

Notation: Remember that the above is in fact an abbreviation for

$$tail : \{n : \mathbb{N}\} \ \rightarrow\ \{a : Type\} \ \rightarrow\ Vect\ (S\ n)\ a \ \rightarrow\ Vect\ n\ a$$

(It is save to think about this as a logical statement $\forall n : \mathbb{N}, \forall a : Type, Vect\ (S\ n)\ a \Rightarrow Vect\ n\ a$.)

Other examples of dependently typed functions from lecture 3 are $(+\!\!+)$, $concat$ and map. We have also seen dependently typed data constructors.

4.2 Equality types

Important and natural examples of dependent types are equality types.

Idris has a built-in type for *propositional equality*.

But let us look at *boolean equality tests* first.

In Idris, many predefined types come equipped with equality *tests*:

$Idris > 2 + 1 == 3$
$True : Bool$

$Idris > [1, 2, 3] == [2, 1, 3]$
$False : Bool$

$Idris > True == False$
$False : Bool$

The tests are all called (==) and return Boolean values. Not all Idris types can be test compared for equality.

Exercise 4.1. Give an example of a type whose values cannot be compared for equality.

For all predefined types that can be compared for equality, (==) is defined as one would expect. But nothing would prevent one to define an equality test for Booleans like for instance

$$(==) : Bool \to Bool \to Bool$$
$$(==)\ b_1\ b_2 = False$$

This would yield

$$Idris > True == True$$
$$False : Bool$$

Equality tests can only be evaluated at *run time* and their results may or may not reflect the equality (or inequality) of their arguments.

But Idris also supports logical reasoning about the equality or the inequality of expressions at *type check time*. This is the role of the built-in propositional equality type briefly mentioned above. Type checking is done before a program is actually compiled and, thus, well before the program can be executed.

For instance

$$p : 2 + 1 = 3$$

is a legal Idris declaration. It represents a claim that the expression $2+1$ is equal to the expression 3. A proof of such claim is just an implementation of p:

$$p = Refl$$

The claim that an expression x of type a is equal to an expression y of type b represented by the type $(x = y)$.

The infix operator $(=)$ used here has type $a \to b \to Type$. For every $x : a$, and $y : b$ we have a type $(x = y)$. This type depends on the values x and y. Thus, it is a dependent type. Conceptually, it is defined as:

$$\textbf{data}\ (=) : a \to b \to Type\ \textbf{where}$$
$$\quad Refl : x = x$$

where $Refl$ stands for "reflexivity". For most $(x = y)$ types there are no values: they are *empty* types. But a few have one value written $Refl : (a = a)$.

$Refl$ can be used to constuct a value of type $(x = y)$ iff x and y can be reduced to the same expression. Thus, for instance, the claim

$$q \;:\; 2 + 1 = 0$$

can be formulated but it cannot be implemented. The only way of implementing a proof would be by

$$q = Refl$$

and this triggers a *type check error*. The program cannot be compiled.

4.3 Negation, logical impossibility

While Idris does not allow to implement a proof q that 2 + 1 equals 0, it makes it easy to show that such a q is an absurdity:

$$notq \;:\; Not\ (2 + 1 = 0)$$
$$notq\ Refl\ impossible$$

Here Not is the function

$$Not \;:\; Type \;\to\; Type$$
$$Not\ a = a \;\to\; Void$$

and $Void$ is a type with no constructors:

data $Void \;:\; Type$ **where**

Thus, a value of type $Void$ represents a logical impossibility. Idris provides a built-in rule for "ex falso sequitur quodlibet" called $void$:

$$void \;:\; Void \;\to\; a$$

Thus, if we have a value of type T and one of type $Not\ T$, we can prove everything:

$$T \;:\; Type$$
$$t \;:\; T$$
$$nt \;:\; Not\ T$$

$$oneEqZero \;:\; 1 = 0$$
$$oneEqZero = void\ (nt\ t)$$

Back to the implementation of $notq$. There, $impossible$ is a keyword.

It recognizes an impossible pattern matching (remember that $2 + 1$ is just an abbreviation for $plus\ (S\ (S\ Z))\ (S\ Z)$ which, in turn, reduces to $S\ (S\ (S\ Z))$ and that constructors are disjoint) and yields a contradiction – that is, a value of type $Void$.

4.4 Properties, propositions and types

Dependent types can also be used to encode properties and propositions. For instance, with

$$\begin{aligned}&\mathit{Domain} : (a \to b) \to \mathit{Type}\\ &\mathit{Domain}\ \{a\}\ f = a\end{aligned}$$

we can express what it means for an arbitrary function $f : a \to b$ to be injective

$$\begin{aligned}&\mathit{Injective} : (a \to b) \to \mathit{Type}\\ &\mathit{Injective}\ f = (x, y : \mathit{Domain}\ f) \to f\ x = f\ y \to x = y\end{aligned}$$

This is almost a word-by-word translation of the corresponding mathematical specification:

$f : a \to b$ *injective* iff $\forall x, y \in \mathit{Dom}\ f,\ f\ x = f\ y \Rightarrow x = y$.

Exercise 4.2. Recall the notion of *optimality of policies* from lecture 2:

$p : X \to Y$ **optimal** iff $\forall x : X, \forall y : Y,\ \mathit{val}\ x\ y \leqslant \mathit{val}\ x\ (p\ x)$

Here X and Y were sets (states, options) and $\mathit{val} : X \to Y \to \mathbb{R}$ denoted a value function. Take

```
X   : Type            -- the type of states
Y   : Type            -- the type of options
val : X → Y → ℝ       -- a value function
```

and implement a dependently typed specification of the notion of optimality for policies through an Idris function of type $(X \to Y) \to \mathit{Type}$.

4.5 Existential types

In many mathematical specifications we find fragments of the form $\exists x \in X$ **s.t.** ... For instance

Let $n \in \mathbb{N}$. $d \in \mathbb{N}$ is a **divisor** of n iff $\exists q \in \mathbb{N}$, **s.t.** $d * q = n$.

Because in DTLs we can encode propositions as types, we can define a data type that represents the statement "there exists an x such that $\mathit{prop}\ x$ holds".

```
data Exists : (a : Type) → (pro : a → Type) → Type where
  Evidence : (wit : a) → (prf : pro wit) → Exists a pro
```

Now we can specify what it means for a natural number to be a divisor:

$$\begin{aligned}&\mathit{Divisor} : (d : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{Type}\\&\mathit{Divisor}\ d\ n = \mathit{Exists}\ \mathbb{N}\ (\lambda q \Rightarrow d * q = n)\end{aligned}$$

and give a proof that 3 is a divisor of 6

$$\begin{aligned}&\mathit{threeDivisorSix} : \mathit{Divisor}\ 3\ 6\\&\mathit{threeDivisorSix} = \mathit{Evidence}\ 2\ \mathit{Refl}\end{aligned}$$

Exercise 4.3. In *Evidence* 2 *Refl*, *Refl* asserts the equality of two expressions. What are the expressions on the LHS and on the RHS of this equality? Proceed by unfolding the definitions in *Divisor* 3 6 and *Evidence* 2 *refl*.

Remark: The notion of *existence* encoded by *Exists* is constructive (evidential). A value of type *Exists a pro* can only be constructed by giving a concrete witness $wit : a$ and a proof $prf : pro\ wit$ that *pro* holds at *wit*.

Remark: The Idris definition of *Exists* is slightly different: the first argument of *Exists* is implicit.

In addition to the logical reading one can also see values of type *Exists a pro* as *dependent pairs* where *Evidence* is the pair constructor and the two projections are *getWitness* and *getProof*:

$$\begin{aligned}&\mathit{getWitness} : \mathit{Exists}\ a\ pro \to a\\&\mathit{getWitness}\ (\mathit{Evidence}\ wit\ prf) = wit\end{aligned}$$

$$\begin{aligned}&\mathit{getProof} : (evi : \mathit{Exists}\ a\ pro) \to pro\ (\mathit{getWitness}\ evi)\\&\mathit{getProof}\ (\mathit{Evidence}\ wit\ prf) = prf\end{aligned}$$

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

When we want to underline the dependent pair interpretation we use a data type called Σ.

Remark: Idris treats values of its pre-defined types *Exists a pro* and $\Sigma\ a\ pro$ slightly differently but we do not need to be concerned with these differences here.

4.6 Specifications and program correctness

One important application of dependently typed programming is for writing programs that are *correct by construction*.

There are two methodologies for assessing the correctness of programs: *testing* and *proving* [3].

Testing is well suited for showing the *presence* of errors. Proving is good at showing their *absence*. Thus, the methodologies are complementary.

Proving the correctness of a program requires two steps. First, one has to specify what it means for the program to be correct. Second, one has to exhibit a proof of correctness.

In non dependently typed languages, both steps have to be undertaken in a suitable formal language (external to the programming language). Specification languages [2, 1, 4] and formal methods of program derivation provide specific support for one or both steps.

In dependently typed languages, we do not need to rely on external specification languages. We can use the same language to

- Specify a program P.
- Implement P.
- Prove that P fulfills its specification.

Let us look at an example for the type of binary trees defined by

$$\begin{array}{l}\textbf{data}\ \mathit{BinTree}\ :\ \mathit{Type}\ \to\ \mathit{Type}\ \textbf{where}\\ \quad \mathit{Leaf}\ :\ a\ \to\ \mathit{BinTree}\ a\\ \quad \mathit{Branch}\ :\ \mathit{BinTree}\ a\ \to\ \mathit{BinTree}\ a\ \to\ \mathit{BinTree}\ a\end{array}$$

In Idris we can *specify what it means* for the following function

$$\mathit{mapBinTree}\ :\ (a\ \to\ b)\ \to\ \mathit{BinTree}\ a\ \to\ \mathit{BinTree}\ b$$

to be correct

$$\begin{array}{l}\mathit{mapBinTreeSpec1}\ :\ (b_t\ :\ \mathit{BinTree}\ a)\ \to\ \mathit{mapBinTree}\ \mathit{id}\ b_t = \mathit{id}\ b_t\\ \mathit{mapBinTreeSpec2}\ :\ (f\ :\ b\ \to\ c)\ \to\ (g\ :\ a\ \to\ b)\ \to\\ \qquad\qquad\qquad\qquad \mathit{mapBinTree}\ (f \circ g) = \mathit{mapBinTree}\ f \circ \mathit{mapBinTree}\ g\end{array}$$

, *implement* $\mathit{mapBinTree}$

$$\begin{array}{ll}\mathit{mapBinTree}\ f\ (\mathit{Leaf}\ x) & = \mathit{Leaf}\ (f\ x)\\ \mathit{mapBinTree}\ f\ (\mathit{Branch}\ l\ r) & = \mathit{Branch}\ (\mathit{mapBinTree}\ f\ l)\ (\mathit{mapBinTree}\ f\ r)\end{array}$$

and *prove that the implementation fulfills its specification*:

$$\begin{array}{l}\mathit{cong2}\ :\ \{a_1, a_2\ :\ a\}\ \to\ \{b_1, b_2\ :\ b\}\ \to\ \{f\ :\ a\ \to\ b\ \to\ c\}\ \to\\ \qquad\quad (a_1 = a_2)\ \to\ (b_1 = b_2)\ \to\ f\ a_1\ b_1 = f\ a_2\ b_2\\ \mathit{cong2}\ \mathit{Refl}\ \mathit{Refl} = \mathit{Refl}\end{array}$$

$$\mathit{mapBinTreeSpec1base}\ :\ (x\ :\ a)\ \to\ \mathit{mapBinTree}\ \mathit{id}\ (\mathit{Leaf}\ x) = \mathit{id}\ (\mathit{Leaf}\ x)$$

$$
\begin{aligned}
mapBinTreeSpec1base\ x = {} & (mapBinTree\ id\ (Leaf\ x)) \\
& = \{\ Refl\ \} = \\
& \quad (Leaf\ (id\ x)) \\
& = \{\ Refl\ \} = \\
& \quad (Leaf\ x) \\
& = \{\ Refl\ \} = \\
& \quad (id\ (Leaf\ x)) \\
& QED
\end{aligned}
$$

$$
\begin{aligned}
mapBinTreeSpec1step\ :\ & (l\ :\ BinTree\ a)\ \to\ (r\ :\ BinTree\ a)\ \to \\
& (ihl\ :\ mapBinTree\ id\ l = id\ l)\ \to\ (ihr\ :\ mapBinTree\ id\ r = id\ r)\ \to \\
& mapBinTree\ id\ (Branch\ l\ r) = id\ (Branch\ l\ r)
\end{aligned}
$$

$$
\begin{aligned}
& mapBinTreeSpec1step\ l\ r\ ihl\ ihr = (mapBinTree\ id\ (Branch\ l\ r)) \\
& \quad = \{\ Refl\ \} = \\
& \quad\ \ (Branch\ (mapBinTree\ id\ l)\ (mapBinTree\ id\ r)) \\
& \quad = \{\ cong2\ ihl\ ihr\ \} = \\
& \quad\ \ (Branch\ (id\ l)\ (id\ r)) \\
& \quad = \{\ Refl\ \} = \\
& \quad\ \ (Branch\ l\ r) \\
& \quad = \{\ Refl\ \} = \\
& \quad\ \ (id\ (Branch\ l\ r)) \\
& \quad QED
\end{aligned}
$$

$$
\begin{aligned}
& mapBinTreeSpec1\ (Leaf\ x) \qquad\ \ = mapBinTreeSpec1base\ x \\
& mapBinTreeSpec1\ (Branch\ l\ r) = mapBinTreeSpec1step\ l\ r\ ihl\ ihr\ \textbf{where} \\
& \quad ihl\ = mapBinTreeSpec1\ l \\
& \quad ihr = mapBinTreeSpec1\ r
\end{aligned}
$$

There is, however, a subtle difference between the statement of $mapBinTreeSpec1$ and $mapBinTreeSpec2$ above which we need to address. The first is stated as a *pointwise* property, while the second is stated as an *extensional equality of functions*. Abstractly, for given functions $f, g\ :\ a\ \to\ b$, the first is a statement of the form

(1) $\forall x\ :\ b, f\ x = g\ x$

while the second has the form

(2) $f = g$.

In the intuitionistic logic underlying Idris, (2) ⇒ (1) is provable, but (1) ⇒ (2) is not (as the theory has models in which the latter implication does not hold).

If we want to prove a statement such as this, we thus cannot avoid to postulate that pointwise equality of functions implies extensional equality of functions:

Idris	Logic
$p\ :\ pro$	p is a proof of pro
$Void$ (empty type)	False
() (singleton type)	True
$p\ \rightarrow\ q$	p implies q
$Exists\ a\ pro$	there exists a wit such that $pro\ wit$ holds
$(x\ :\ a)\ \rightarrow\ pro\ x$	forall x of type a, $pro\ x$ holds

Table 1: Curry-Howard correspondence relating Idris and logic on the type level.

$$funext\ : (f, g\ :\ a\ \rightarrow\ b)\ \rightarrow\ ((x\ :\ a)\ \rightarrow\ f\ x = g\ x)\ \rightarrow\ f = g$$

Using this postulate, we can first prove the pointwise statement

$$mapBinTreeSpec2a\ :\ (b_t\ :\ BinTree\ a)\ \rightarrow\ (f\ :\ b\ \rightarrow\ c)\ \rightarrow\ (g\ :\ a\ \rightarrow\ b)\ \rightarrow$$
$$mapBinTree\ (f \circ g)\ b_t = (mapBinTree\ f \circ mapBinTree\ g)\ b_t$$

and then use $funext$ to derive $maBinTreeSpec2$ as stated above.

Exercise 4.4. Implement first $mapBinTreeSpec2a$, then use $funext$ to prove $mapBinTreeSpec2$.

4.7 Programs, proofs, totality and termination

We have seen that we can represent properties as types and in this view proofs are just values of these types.

We sum up (a part of) the correspondence between Idris and logic in Table 1. (This correspondence goes in fact much deeper than conveyed by the table. ⇝ see L4E1)

When we embed logic in a programming language, we have to be careful about two notions: *totality* and *termination*. (Non-termination of programs can be seen as counterpart to logical paradoxes, ⇝ see L4E1.)

A total function $f\ :\ a\ \rightarrow\ b$ is defined for all type correct inputs.

A partial function would be undefined on some of $x\ :\ a$.

Partial functions can be very useful. But proofs shall always be total.

If we allowed partial functions to silently compromise the totality of proofs, we could easily prove any theorem, including patently false ones. Consider the function $headL\ :\ List\ a\ \rightarrow\ a$

$$\begin{array}{l} partial \\ headL\ :\ List\ a\ \rightarrow\ a \\ headL\ (x :: xs) = x \end{array}$$

This is a partial function because it is not defined for the empty list. Using *headL* we could easily "prove" that every natural number is zero:

$$\mathit{aNecessarilyEmptyList} \ : \ \mathit{List}\ \mathit{Void}$$
$$\mathit{aNecessarilyEmptyList} = [\,]$$

$$\mathit{surprise} \ : (n \ : \ \mathbb{N}) \ \rightarrow \ n = 0$$
$$\mathit{surprise}\ n = \mathit{void}\ (\mathit{headL}\ \mathit{aNecessarilyEmptyList})$$

The Idris type checker realizes that we are trying to fool the system and that *surprise* cannot be total.

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

$$\mathit{circular} \ : \ \mathit{Void}$$
$$\mathit{circular} = \mathit{circular}$$

Idris will warn about missing cases and potentially non-terminating loops in definitions that are required to be *total*.

4.8 Type checking and correctness

With dependently typed languages, we can require the type checker to verify that a certain program implementation is correct with respect to its specification.

This methodology yields programs that are *correct by construction*. This is the highest standard we can aim for in programming.

Crucial components of the methodology are *totality* and *termination* checks.

Termination checks are necessarily conservative: failures to pass the tests mean that the program might not terminate, not that it will not terminate.

Conversely, a program that passes a termination test will always terminate, at least in principle. Of course, memory limitations and hardware failures can always in practice prevent a computation from terminating.

Beyond providing dependent types and totality and termination checks, Idris supports a programming methodology that aims at increasing the correctness of programs *incrementally*.

The idea is to first fulfill program specifications *conditionally* on the basis of suitable postulates. These are then eliminated stepwise, eventually leading to unconditional correctness proofs.

4.9 Coming up

In the next lecture, we will start looking at the formalization of dynamical systems and the problem of decision making under uncertainty.

Solutions

Exercise 4.1:

The type of functions from natural numbers to *Bool*.

Exercise 4.2:

$$\begin{aligned}&\mathit{Optimal} : (X \to Y) \to \mathit{Type}\\&\mathit{Optimal}\ p = (x : X) \to (y : Y) \to \mathit{val}\ x\ y \leqslant \mathit{val}\ x\ (p\ x)\end{aligned}$$

What is the type of $\leqslant$?

Exercise 4.3:

The expressions are $3 * 2$ on the LHS and 6 on the RHS

Exercise 4.4:

$$\begin{aligned}\mathit{mapBinTreeSpec2a} &: (b_t : \mathit{BinTree}\ a) \to (f : b \to c) \to (g : a \to b) \to\\&\qquad \mathit{mapBinTree}\ (f \circ g)\ b_t = (\mathit{mapBinTree}\ f \circ \mathit{mapBinTree}\ g)\ b_t\end{aligned}$$

$$\begin{aligned}\mathit{mapBinTreeSpec2base} &: (x : a) \to (f : b \to c) \to (g : a \to b) \to\\&\qquad \mathit{mapBinTree}\ (f \circ g)\ (\mathit{Leaf}\ x) =\\&\qquad (\mathit{mapBinTree}\ f \circ \mathit{mapBinTree}\ g)\ (\mathit{Leaf}\ x)\\\mathit{mapBinTreeSpec2base}\ x\ f\ g &= (\mathit{mapBinTree}\ (f \circ g)\ (\mathit{Leaf}\ x))\\&= \{\mathit{Refl}\} =\\&\quad (\mathit{Leaf}\ ((f \circ g)\ x))\\&= \{\mathit{Refl}\} =\\&\quad (\mathit{Leaf}\ (f\ (g\ x)))\\&= \{\mathit{Refl}\} =\\&\quad (\mathit{mapBinTree}\ f\ (\mathit{Leaf}\ (g\ x)))\\&= \{\mathit{Refl}\} =\\&\quad (\mathit{mapBinTree}\ f\ (\mathit{Leaf}\ (g\ x)))\\&= \{\mathit{Refl}\} =\\&\quad (\mathit{mapBinTree}\ f\ (\mathit{mapBinTree}\ g\ (\mathit{Leaf}\ x)))\\&= \{\mathit{Refl}\} =\\&\quad ((\mathit{mapBinTree}\ f \circ \mathit{mapBinTree}\ g)\ (\mathit{Leaf}\ x)) >\\&\mathit{QED}\\\mathit{mapBinTreeSpec2step} &: (l : \mathit{BinTree}\ a) \to (r : \mathit{BinTree}\ a) \to (f : b \to c) \to (g : a \to b) \to\end{aligned}$$

$$
\begin{aligned}
&\qquad\qquad (iHl \; : \; mapBinTree \; (f \circ g) \; l = (mapBinTree \; f \circ mapBinTree \; g) \; l) \; \to \\
&\qquad\qquad (iHr \; : \; mapBinTree \; (f \circ g) \; r = (mapBinTree \; f \circ mapBinTree \; g) \; r) \; \to \\
&\qquad\qquad mapBinTree \; (f \circ g) \; (Branch \; l \; r) = (mapBinTree \; f \circ mapBinTree \; g) \; (Branch \; l \; r) \\
&mapBinTreeSpec2step \; l \; r \; f \; g \; ihl \; ihr \\
&\qquad\qquad = (mapBinTree \; (f \circ g) \; (Branch \; l \; r)) \\
&\qquad\qquad = \{ \, Refl \, \} = \\
&\qquad\qquad\quad (Branch \; (mapBinTree \; (f \circ g) \; l) \; (mapBinTree \; (f \circ g) \; r)) \\
&\qquad\qquad = \{ \, cong2 \; ihl \; ihr \, \} = \\
&\qquad\qquad\quad (Branch \\
&\qquad\qquad\qquad ((mapBinTree \; f \circ mapBinTree \; g) \; l) \\
&\qquad\qquad\qquad ((mapBinTree \; f \circ mapBinTree \; g) \; r)) \\
&\qquad\qquad = \{ \, Refl \, \} = \\
&\qquad\qquad\quad (mapBinTree \; f \; (Branch \; (mapBinTree \; g \; l) \; (mapBinTree \; g \; r))) \\
&\qquad\qquad = \{ \, Refl \, \} = \\
&\qquad\qquad\quad ((mapBinTree \; f \circ mapBinTree \; g) \; (Branch \; l \; r)) \\
&\qquad\qquad QED
\end{aligned}
$$

$$
\begin{aligned}
&mapBinTreeSpec2a \; (Leaf \; x) \qquad\;\; f \; g = mapBinTreeSpec2base \; x \; f \; g \\
&mapBinTreeSpec2a \; (Branch \; l \; r) \; f \; g = mapBinTreeSpec2step \; l \; r \; f \; g \; ihl \; ihr \; \textbf{where} \\
&\quad ihl \; = mapBinTreeSpec2a \; l \; f \; g \\
&\quad ihr = mapBinTreeSpec2a \; r \; f \; g
\end{aligned}
$$

$$
\begin{aligned}
&helper \; : (f \; : \; b \; \to \; c) \; \to \; (g \; : \; a \; \to \; b) \; \to \; (b_t \; : \; BinTree \; a) \; \to \\
&\quad mapBinTree \; (f \circ g) \; b_t = (mapBinTree \; f \circ mapBinTree \; g) \; b_t \\
&helper \; f \; g \; b_t = mapBinTreeSpec2a \; b_t \; f \; g
\end{aligned}
$$

$$
\begin{aligned}
&mapBinTreeSpec2 \; f \; g = funext \; (mapBinTree \; (f \circ g)) \; (mapBinTree \; f \circ mapBinTree \; g) \\
&\quad (helper \; f \; g)
\end{aligned}
$$

References

[1] Richard S. Bird and Oege de Moor. *Algebra of programming.* Prentice Hall International series in computer science. Prentice Hall, 1997.

[2] Manfred Broy, editor. *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*, volume 118 of *NATO ASI Series.* Springer, 1993.

[3] Cezar Ionescu and Patrik Jansson. Testing versus proving in climate impact research. In *Proc. TYPES 2011*, volume 19 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41–54, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] Carroll Morgan. *Programming from specifications.* Prentice Hall International Series in computer science. Prentice Hall, 1990.

Lecture 5: Time-discrete dynamical systems

With basic ideas about problem specification and the support of a specification language, we turn back to the problem of understanding decision making under uncertainty.

As a first step, we look at *time discrete deterministic dynamical systems*. This notion is fundamental in modelling, in particular in earth system modelling. This is because of two reasons:

- Continuous dynamical systems have typically to be *approximated*. This is done in terms of discrete systems.
- Deterministic dynamical systems are special cases of more general non-deterministic, stochastic, fuzzy, etc. systems.

5.1 Discrete deterministic dynamical systems

A time discrete deterministic dynamical system (in short, a *deterministic system*) on a set X is a function of type $X \to X$.

Remember that in dependently typed languages sets and propositions are encoded through types. Thus, a natural formalization of the notion is

$$
\begin{aligned}
&\mathit{DetSys} : \mathit{Type} \to \mathit{Type}\\
&\mathit{DetSys}\ X = X \to X
\end{aligned}
$$

We can also introduce the notion (of a discrete deterministic dynamical system) through a data declaration

$$
\begin{aligned}
&\mathbf{data}\ \mathit{DetSys} : \mathit{Type} \to \mathit{Type}\ \mathbf{where}\\
&\quad \mathit{MkDetSys} : \{X : \mathit{Type}\} \to (X \to X) \to \mathit{DetSys}\ X
\end{aligned}
$$

A specific system on X is then declared to be a value of type $\mathit{DetSys}\ X$. The domain of $f : \mathit{DetSys}\ X$ is often called the *state space* of f:

$$
\begin{aligned}
&\mathit{StateSpace} : \{X : \mathit{Type}\} \to \mathit{DetSys}\ X \to \mathit{Type}\\
&\mathit{StateSpace} = \mathit{Domain}
\end{aligned}
$$

The most obvious operation that we can do with a system is to iterate it a certain number of steps. This is often called the *flow* of the system:

$$
\begin{aligned}
&\mathit{flow} : \{X : \mathit{Type}\} \to \mathbb{N} \to \mathit{DetSys}\ X \to \mathit{DetSys}\ X\\
&\mathit{flow}\ \ Z\ \ \ \ f\ x = x\\
&\mathit{flow}\ (S\ n)\ f\ x = \mathit{flow}\ n\ f\ (f\ x)
\end{aligned}
$$

Notice that $\mathit{flow}\ n\ f$ can also be defined in a point-free notation

$$
\begin{aligned}
&\mathit{flow} : \{X : \mathit{Type}\} \to \mathbb{N} \to \mathit{DetSys}\ X \to \mathit{DetSys}\ X\\
&\mathit{flow}\ \ Z\ \ \ \ f = \mathit{id}\\
&\mathit{flow}\ (S\ n)\ f = (\mathit{flow}\ n\ f) \circ f
\end{aligned}
$$

and that $\mathit{flow}\ n\ f$ has the same type as f. In physics, the standard notation for $\mathit{flow}\ n\ f$ is f^n.

Exercise 5.1. Encode the mathematical specification

$$\forall m, n \in \mathbb{N},\ \ f : DetSys\ X,\ \ x \in X,\ \ flow\ (m + n)\ f\ x = flow\ n\ f\ (flow\ m\ f\ x)$$

in Idris through the type of an *flowSpec* value.

Exercise 5.2. Implement *flowSpec* by pattern matching on m.

Another fundamental notion in dynamical systems theory is that of the *trajectory* (of a dynamical system) starting at a certain *initial* state:

$$\begin{aligned}
&trj : \{X : Type\} \to (n : \mathbb{N}) \to DetSys\ X \to X \to Vect\ (S\ n)\ X \\
&trj\ \ Z\ \ \ \ \ \ f\ x = x :: Nil \\
&trj\ (S\ n)\ f\ x = x :: trj\ n\ f\ (f\ x)
\end{aligned}$$

Exercise 5.3. *trj* fulfills a specification similar to *flowSpec*. Encode this specification in the type of a function *trjSpec* using only *flow*, $tail : Vect\ (S\ n)\ X \to Vect\ n\ X$ and vector concatenation.

Exercise 5.4. Implement *trjSpec* on the basis of

$$\begin{aligned}
postulate\ trjLemma1 : &\ \{X : Type\} \to (m : \mathbb{N}) \to (f : DetSys\ X) \to (x : X) \to \\
&\ head\ (trj\ m\ f\ x) = x
\end{aligned}$$

$$\begin{aligned}
postulate\ trjLemma2 : &\ \{X : Type\} \to (m : \mathbb{N}) \to (f : DetSys\ X) \to (x : X) \to \\
&\ tail\ (trj\ (S\ m)\ f\ x) = trj\ m\ f\ (f\ x)
\end{aligned}$$

and

$$\begin{aligned}
postulate\ headTailLemma : &\ \{n : \mathbb{N}\} \to \{A : Type\} \to \\
&\ (xs : Vect\ (S\ n)\ A) \to head\ xs :: tail\ xs = xs
\end{aligned}$$

Perhaps not surprisingly, the last element of the trajectory of length n of $f : DetSys\ X$ starting in x is just $flow\ n\ f\ x$:

$$
\begin{aligned}
\mathit{flowTrjLemma} \;:\; & \{X \;:\; \mathit{Type}\} \;\to \\
& (n \;:\; \mathbb{N}) \;\to\; (f \;:\; \mathit{DetSys}\; X) \;\to \\
& (x \;:\; X) \;\to\; \mathit{flow}\; n\; f\; x = \mathit{last}\; (\mathit{trj}\; n\; f\; x)
\end{aligned}
$$

Exercise 5.5. Implement *flowTrjLemma* on the basis of

$$
\begin{aligned}
\mathit{postulate}\; \mathit{lastLemma} \;:\; & \{A \;:\; \mathit{Type}\} \;\to\; \{n \;:\; \mathbb{N}\} \;\to \\
& (x \;:\; A) \;\to\; (xs \;:\; \mathit{Vect}\; (S\; n)\; A) \;\to\; \mathit{last}\; (x :: xs) = \mathit{last}\; xs
\end{aligned}
$$

5.2 Discrete non-deterministic dynamical systems

What if the outcome of a system is uncertain? In this case, for a given $x \;:\; X$ we can have more than one *possible* next state.

If we do not have any additional information, we say that the system is *non-deterministic*. In this case, we can represent all possible next states by a list:

$$
\begin{aligned}
& \mathit{NonDetSys} \;:\; \mathit{Type} \;\to\; \mathit{Type} \\
& \mathit{NonDetSys}\; X = X \;\to\; \mathit{List}\; X
\end{aligned}
$$

Lists are equipped with so-called *return*, *join* and *map* operations

$$
\begin{aligned}
& \mathit{retList} \;:\; \{A \;:\; \mathit{Type}\} \;\to\; A \;\to\; \mathit{List}\; A \\
& \mathit{retList}\; x = x :: \mathit{Nil}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{joinList} \;:\; \{A \;:\; \mathit{Type}\} \;\to\; \mathit{List}\; (\mathit{List}\; A) \;\to\; \mathit{List}\; A \\
& \mathit{joinList}\;\; \mathit{Nil} \qquad\;\; = \mathit{Nil} \\
& \mathit{joinList}\; (xs :: xss) = xs +\!\!+\; \mathit{joinList}\; xss
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{mapList} \;:\; \{A, B \;:\; \mathit{Type}\} \;\to\; (A \;\to\; B) \;\to\; \mathit{List}\; A \;\to\; \mathit{List}\; B \\
& \mathit{mapList}\; f\;\; \mathit{Nil} \qquad = \mathit{Nil} \\
& \mathit{mapList}\; f\; (x :: xs) = f\; x :: \mathit{mapList}\; f\; xs
\end{aligned}
$$

that fulfill certain *naturality* conditions. For instance, $\mathit{mapList}\; f\; (\mathit{retList}\; x) = \mathit{retList}\; (f\; x)$:

$$
\begin{aligned}
& *\mathit{Lecture5} > \mathit{retList}\; 3 \\
& [3] \;:\; \mathit{List}\; \mathit{Integer}
\end{aligned}
$$

and

$$
*\mathit{Lecture5} > \mathit{mapList}\; (2+)\; (\mathit{retList}\; 1) = \mathit{retList}\; (2 + 1)
$$

$$[3] = [3] \; : \; \mathit{Type}$$

and

$$* \mathit{Lecture5} > \mathit{joinList} \; [[1,5,4],[3,4,9]]$$
$$[1,5,4,3,4,9] \; : \; \mathit{List\ Integer}$$

Remark: Every deterministic system $f \; : \; X \; \to \; X$ can be represented by a non-deterministic system:

$$\mathit{embedDetIntoNonDet} \; : \{X \; : \; \mathit{Type}\} \; \to \; \mathit{DetSys}\ X \; \to \; \mathit{NonDetSys}\ X$$
$$\mathit{embedDetIntoNonDet}\ f = \mathit{retList} \circ f$$

Using $\mathit{mapList}$ and $\mathit{joinList}$ one can implement a function

$$\mathit{flowNonDetSys} \; : \{X \; : \; \mathit{Type}\} \; \to \; (m \; : \; \mathbb{N}) \; \to \; \mathit{NonDetSys}\ X \; \to \; \mathit{NonDetSys}\ X$$

that iterates a non-deterministic system:

$$\mathit{flowNonDetSys} \;\; Z \qquad f\ x = \mathit{retList}\ x$$
$$\mathit{flowNonDetSys}\ (S\ m)\ f\ x = \mathit{joinList}\ (\mathit{mapList}\ (\mathit{flowNonDetSys}\ m\ f)\ (f\ x))$$

Notice that, if we define

$$\mathit{bindList} \; : \{A, B \; : \; \mathit{Type}\} \; \to \; (A \; \to \; \mathit{List}\ B) \; \to \; \mathit{List}\ A \; \to \; \mathit{List}\ B$$
$$\mathit{bindList}\ f\ as = \mathit{joinList}\ (\mathit{mapList}\ f\ as)$$

the second clause of $\mathit{flowNonDetSys}$ can be written as

$$\mathit{flowNonDetSys}\ (S\ m)\ f\ x = \mathit{bindList}\ (\mathit{flowNonDetSys}\ m\ f)\ (f\ x)$$

A comparison with the flow of deterministic systems

$$\mathit{flow}\ (S\ m)\ f\ x = \mathit{flow}\ m\ f\ (f\ x) = (\mathit{flow}\ m\ f)\ (f\ x) = ((\mathit{flow}\ m\ f) \circ f)\ x$$

suggests that $\mathit{bindList}$ is a kind of evaluation. Consistently with this interpretation, one has

Lemma: $\forall m, n \in \mathbb{N},\ f \; : \; \mathit{NonDetSys}\ X, x \in X,\ \mathit{flow}'\ (m{+}n)\ f\ x = \mathit{bindList}\ (\mathit{flow}'\ n\ f)\ (\mathit{flow}'\ m\ f\ x)$ with $\mathit{flow}' = \mathit{flowNonDetSys}$.

We will prove the lemma in a more generic setup in lecture 6. Next, consider

$$\mathit{repr} \; : \{X \; : \; \mathit{Type}\} \; \to \; \mathit{NonDetSys}\ X \; \to \; \mathit{DetSys}\ (\mathit{List}\ X)$$
$$\mathit{repr} = \mathit{bindList}$$

The function associates to any non-deterministic system on an arbitrary type X, a deterministic system on $\mathit{List}\ X$. We say that $\mathit{repr}\ f$ is the deterministic representation of f. This terminology is justified by the result:

$$\mathit{reprLemma} \; : \; \{X \; : \; \mathit{Type}\} \; \to \; (n \; : \; \mathbb{N}) \; \to \; (f \; : \; \mathit{NonDetSys}\ X) \; \to$$
$$(xs \; : \; \mathit{List}\ X) \; \to \; \mathit{repr}\ (\mathit{flowNonDetSys}\ n\ f)\ xs = \mathit{flow}\ n\ (\mathit{repr}\ f)\ xs$$

Exercise 5.6. Implement *reprLemma* using:

$$
\begin{aligned}
\mathit{bindListPresExtEq} \;:\; & \{A, B : \mathit{Type}\} \to (f, g : A \to \mathit{List}\ B) \to \\
& ((a : A) \to f\ a = g\ a) \to \\
& ((as : \mathit{List}\ A) \to \mathit{bindList}\ f\ as = \mathit{bindList}\ g\ as)
\end{aligned}
$$

$$
\mathit{rightIdentityList} \;:\; \{A : \mathit{Type}\} \to (as : \mathit{List}\ A) \to \mathit{bindList}\ \mathit{retList}\ as = as
$$

$$
\begin{aligned}
\mathit{bindListAssociative} \;:\; & \{A, B, C : \mathit{Type}\} \to (f : A \to \mathit{List}\ B) \to (g : B \to \mathit{List}\ C) \to \\
& (as : \mathit{List}\ A) \to \\
& \mathit{bindList}\ (\mathit{bindList}\ g \circ f)\ as = \mathit{bindList}\ g\ (\mathit{bindList}\ f\ as)
\end{aligned}
$$

Exercise 5.7. Implement *bindListPresExtEq* and *rightIdentityList*. Start by implementing

$$
\begin{aligned}
\mathit{mapListPresExtEq} \;:\; & \{A, B : \mathit{Type}\} \to (f, g : A \to \mathit{List}\ B) \to \\
& ((a : A) \to f\ a = g\ a) \to \\
& ((as : \mathit{List}\ A) \to \mathit{mapList}\ f\ as = \mathit{mapList}\ g\ as)
\end{aligned}
$$

Exercise 5.8. The funcion *flowNonDetSys* produces a lot of duplicates. For instance, for

$$
\begin{aligned}
& f : \mathbb{N} \to \mathit{List}\ \mathbb{N} \\
& f\ Z \qquad\;\; = [Z, S\ Z] \\
& f\ (S\ m) = [m, S\ m, S\ (S\ m)]
\end{aligned}
$$

one obtaines

$$
\begin{aligned}
& *\mathit{Lecture5} > \mathit{flowNonDetSys}\ 3\ f\ Z \\
& [0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 1, 2, 3] : \mathit{List}\ \mathbb{N}
\end{aligned}
$$

The function *nub* eliminates list duplicates:

$$
\begin{aligned}
& *\mathit{Lecture5} > \mathit{nub}\ (\mathit{flowNonDetSys}\ 3\ f\ Z) \\
& [0, 1, 2, 3] : \mathit{List}\ \mathbb{N}
\end{aligned}
$$

Using *nub*, write a function *flowNonDetSys'* that produces no duplicates and runs faster (for a large enough number of iterations) than the original version.

In much the same way as we can iterate non-deterministic systems a fixed number of times, we can compute all the possible trajectories of fixed length that start at a given initial value:

$$\begin{aligned}
&\mathit{trjNonDetSys} : \{X : \mathit{Type}\} \to (n : \mathbb{N}) \to \mathit{NonDetSys}\ X \to X \to \mathit{List}\ (\mathit{Vect}\ (S\ n)\ X)\\
&\mathit{trjNonDetSys}\ \ Z\ \ \ \ f\ x = \mathit{mapList}\ (x{::})\ (\mathit{retList}\ \mathit{Nil})\\
&\mathit{trjNonDetSys}\ (S\ n)\ f\ x = \mathit{mapList}\ (x{::})\ (\mathit{bindList}\ (\mathit{trjNonDetSys}\ n\ f)\ (f\ x))
\end{aligned}$$

Exercise 5.9. $\mathit{trjNonDetSys}$ computes all the possible trajectories of a system starting from a given initial value. For instance

$$\begin{aligned}
&*\mathit{Lecture5} > \mathit{trjNonDetSys}\ 0\ f\ Z\\
&[[0]] : \mathit{List}\ (\mathit{Vect}\ 1\ \mathbb{N})
\end{aligned}$$

$$\begin{aligned}
&*\mathit{Lecture5} > \mathit{trjNonDetSys}\ 1\ f\ Z\\
&[[0,0],[0,1]] : \mathit{List}\ (\mathit{Vect}\ 2\ \mathbb{N})
\end{aligned}$$

$$\begin{aligned}
&*\mathit{Lecture5} > \mathit{trjNonDetSys}\ 2\ f\ Z\\
&[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[0,1,2]] : \mathit{List}\ (\mathit{Vect}\ 3\ \mathbb{N})
\end{aligned}$$

Explain the implementation of $\mathit{trjNonDetSys}$. What is the type of $x{::}$? What is the type of Nil on the RHS of $\mathit{trjNonDetSys}\ Z\ f\ x$? What are the types of $\mathit{trjNonDetSys}\ n\ f$, $f\ x$ and $\mathit{bindList}\ (\mathit{trjNonDetSys}\ n\ f)\ (f\ x)$?

5.3 Discrete stochastic dynamical systems

Sometimes we know enough about a system to be able to estimate its transition *probabilities*.

In this case, we say that the system is *stochastic*. Stochastic systems can be described by functions of type $X \to \mathit{Prob}\ X$. Here $\mathit{Prob}\ X$ represents the type of finite probability distributions on X:

$$\mathit{Prob} : \mathit{Type} \to \mathit{Type}$$

We are not going to *define* Prob in this lecture. Instead, we *specify* properties that finite probability distributions are required to fulfill.

Let us first recall the basic notions of elementary probability theory, that is, of probability theory for finite, non-empty sets.

In this context, *events* are subsets of a finite, non-empty set X: $\mathit{Event}\ X = \mathcal{P}\ X$. The set X represents the possible *outcomes* of a random process and a *probability* is a function of type $\mathit{Event}\ X \to \mathbb{R}$ that fulfilsl the axioms (Kolmogorov, 1933):

1. $\forall e \in \mathit{Event}\ X,\ \ P\ e \geqslant 0$.
2. $P\ \emptyset = 0$ and $P\ X = 1$.
3. $\forall e, e' \in \mathit{Event}\ X,\ \ e\ \cap\ e' = \emptyset \Rightarrow P\ (e\ \cup\ e') = P\ e + P\ e'$.

In elementary probability theory, a probability distribution on a finite, non-empty set X is a function $\pi\ :\ X\ \rightarrow\ \mathbb{R}$ such that $\sum_{x \in X} \pi\ x = 1$.

Thus, a probability distribution $\pi\ :\ X\ \rightarrow\ \mathbb{R}$ induces a probability function $P_\pi\ :\ \mathit{Event}\ X\ \rightarrow\ \mathbb{R}$ via $P_\pi\ e = \sum_{x \in e} \pi\ x$.

We can formalize this fragment of probability theory in Idris by representing probability distributions on values of a type X by values of type $\mathit{Prob}\ X$.

In this formalization, X does not need to be a finite type. But the set of values of type X whose probability is non-zero has to be finite. For $\mathit{pd}\ :\ \mathit{Prob}\ X$, we call this set the *support* of pd:

$$\mathit{supp}\ :\ \{A\ :\ \mathit{Type}\}\ \rightarrow\ \mathit{Prob}\ A\ \rightarrow\ \mathit{List}\ A$$

The probability associated with a probability distribution $\mathit{pd}\ :\ \mathit{Prob}\ X$ is then given by $\mathit{prob}\ \mathit{pd}$ with

$$\mathit{prob}\ :\ \{A\ :\ \mathit{Type}\}\ \rightarrow\ \mathit{Prob}\ A\ \rightarrow\ (A\ \rightarrow\ \mathit{Bool})\ \rightarrow\ \mathbb{R}$$

where $\mathit{prob}\ \mathit{pd}\ e$ represents the probability of the event e according to pd. As in the case of non-deterministic systems, we require $\mathit{Prob}\ X$ to be equipped with return, join and map operations:

$$\mathit{retProb}\ :\ \{A\ :\ \mathit{Type}\}\ \rightarrow\ A\ \rightarrow\ \mathit{Prob}\ A$$

$$\mathit{joinProb}\ :\ \{A\ :\ \mathit{Type}\}\ \rightarrow\ \mathit{Prob}\ (\mathit{Prob}\ A)\ \rightarrow\ \mathit{Prob}\ A$$

$$\mathit{mapProb}\ :\ \{A, B\ :\ \mathit{Type}\}\ \rightarrow\ (A\ \rightarrow\ B)\ \rightarrow\ \mathit{Prob}\ A\ \rightarrow\ \mathit{Prob}\ B$$

These have natural interpretations in probability theory. Thus, $\mathit{retProb}$ is the function that associates to any value x of an arbitrary type X the probability distribution concentrated on x:

$$\mathit{prob}\ (\mathit{retProb}\ x)\ e = 1\ \iff\ e\ x = \mathit{True}$$
$$\mathit{prob}\ (\mathit{retProb}\ x)\ e = 0\ \iff\ e\ x = \mathit{False}$$

$\mathit{joinProb}$ is the function that reduces probability distributions over probability distributions over X to probability distributions over X. It fulfills

$$\mathit{prob}\ (\mathit{joinProb}\ \mathit{pd2})\ e = \mathit{sum}\ [\mathit{prob}\ \mathit{pd2}\ (\mathit{is}\ \mathit{pd}) * \mathit{prob}\ \mathit{pd}\ e \mid \mathit{pd} \leftarrow \mathit{supp}\ \mathit{pd2}]$$

which can be interpreted as the "law of total probability": here $\mathit{is}\ \mathit{pd}$ is the characteristic function of pd

$$\begin{aligned}&\mathit{is} : \{A : \mathit{Type}\} \to \mathit{Eq}\ A \Rightarrow A \to A \to \mathit{Bool}\\&\mathit{is}\ a\ a' = a == a'\end{aligned}$$

and thus, for $x\ != y$, $\mathit{is}\ x$ and $\mathit{is}\ y$ are disjoint events. With $\mathit{retProb}$, $\mathit{joinProb}$, $\mathit{mapProb}$ and

$$\begin{aligned}&\mathit{StochSys} : \mathit{Type} \to \mathit{Type}\\&\mathit{StochSys}\ X = X \to \mathit{Prob}\ X\end{aligned}$$

we can implement a function

$$\mathit{flowStochSys} : \{X : \mathit{Type}\} \to (m : \mathbb{N}) \to \mathit{StochSys}\ X \to \mathit{StochSys}\ X$$

that computes all the states that can be obtained by iterating a stochastic system starting from an initial $x : X$. The implementation can be derived by copy & paste from $\mathit{flowNonDet}$:

$$\begin{aligned}&\mathit{flowStochSys}\ Z\qquad f\ x = \mathit{retProb}\ x\\&\mathit{flowStochSys}\ (S\ m)\ f\ x = \mathit{joinProb}\ (\mathit{mapProb}\ (\mathit{flowStochSys}\ m\ f)\ (f\ x))\end{aligned}$$

Similarly, one can implement bind, repr, $\mathit{reprLemma}$, trj, etc. for stochastic systems with obvious interpretations.

In the next lecture we will amalgamate the commonalities between deterministic, non-deterministic and stochastic systems in the notion of *monadic* dynamical systems.

Monadic dynamical systems allow one to account for different kinds of uncertainties in a simple and seamless way.

Solutions

Exercise 5.1:

$$\begin{aligned}\mathit{flowSpec} : \{X : \mathit{Type}\} \to (m : \mathbb{N}) \to (n : \mathbb{N}) \to (f : \mathit{DetSys}\ X) \to (x : X) \to\\ \mathit{flow}\ (m + n)\ f\ x = \mathit{flow}\ n\ f\ (\mathit{flow}\ m\ f\ x)\end{aligned}$$

Exercise 5.2:

$$\begin{aligned}\mathit{flowSpec}\ Z\ n\ f\ x =\ &(\mathit{flow}\ (Z + n)\ f\ x)\\ =\ &\{\mathit{Refl}\} =\\ &(\mathit{flow}\ n\ f\ x)\\ =\ &\{\mathit{Refl}\} =\\ &(\mathit{flow}\ n\ f\ (\mathit{flow}\ Z\ f\ x))\\ &\mathit{QED}\end{aligned}$$

$$
\begin{aligned}
\mathit{flowSpec}\ (S\ m)\ n\ f\ x = &\ (\mathit{flow}\ ((S\ m) + n)\ f\ x)\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ (\mathit{flow}\ (S\ (m + n))\ f\ x)\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ (\mathit{flow}\ (m + n)\ f\ (f\ x))\\
=&\ \{\ \mathit{flowSpec}\ m\ n\ f\ (f\ x)\ \} =\\
&\ (\mathit{flow}\ n\ f\ (\mathit{flow}\ m\ f\ (f\ x)))\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ (\mathit{flow}\ n\ f\ (\mathit{flow}\ (S\ m)\ f\ x))\\
&\ \mathit{QED}
\end{aligned}
$$

Exercise 5.3:

$$
\begin{aligned}
&\mathit{trjSpec} : \{X : \mathit{Type}\} \to (m : \mathbb{N}) \to (n : \mathbb{N}) \to (f : \mathit{DetSys}\ X) \to (x : X) \to\\
&\qquad \mathit{trj}\ (m + n)\ f\ x = \mathit{trj}\ m\ f\ x +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ (\mathit{flow}\ m\ f\ x))
\end{aligned}
$$

Exercise 5.4:

$$
\begin{aligned}
\mathit{trjSpec}\ Z\ n\ f\ x =&\ (\mathit{trj}\ (Z + n)\ f\ x)\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ (\mathit{trj}\ n\ f\ x)\\
=&\ \{\ \mathit{sym}\ (\mathit{headTailLemma}\ (\mathit{trj}\ n\ f\ x))\ \} =\\
&\ (\mathit{head}\ (\mathit{trj}\ n\ f\ x) :: \mathit{tail}\ (\mathit{trj}\ n\ f\ x))\\
=&\ \{\ \mathit{replace}\ \{P = \lambda X \Rightarrow \mathit{head}\ (\mathit{trj}\ n\ f\ x) :: \mathit{tail}\ (\mathit{trj}\ n\ f\ x)\\
&\qquad\qquad\qquad\qquad =\\
&\qquad\qquad\qquad\quad X :: \mathit{tail}\ (\mathit{trj}\ n\ f\ x)\}\\
&\qquad\qquad\qquad\quad (\mathit{trjLemma1}\ n\ f\ x)\ \mathit{Refl}\ \} =\\
&\ (x :: \mathit{tail}\ (\mathit{trj}\ n\ f\ x))\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ ((x :: \mathit{Nil}) +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ x))\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ (\mathit{trj}\ Z\ f\ x +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ (\mathit{flow}\ Z\ f\ x)))\\
&\ \mathit{QED}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{trjSpec}\ (S\ m)\ n\ f\ x =&\ (\mathit{trj}\ ((S\ m) + n)\ f\ x)\\
=&\ \{\ \mathit{Refl}\ \} =\\
&\ (\mathit{trj}\ (S\ (m + n))\ f\ x)
\end{aligned}
$$

$$
\begin{aligned}
&= \{\, \mathit{Refl} \,\} = \\
&\quad (x :: \mathit{trj}\ (m + n)\ f\ (f\ x)) \\
&= \{\, \mathit{replace}\ \{P = \lambda X \Rightarrow x :: \mathit{trj}\ (m + n)\ f\ (f\ x) = x :: X\,\} \\
&\quad (\mathit{trjSpec}\ m\ n\ f\ (f\ x))\ \mathit{Refl} \,\} = \\
&\quad (x :: (\mathit{trj}\ m\ f\ (f\ x) +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ (\mathit{flow}\ m\ f\ (f\ x))))) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad ((x :: \mathit{trj}\ m\ f\ (f\ x)) +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ (\mathit{flow}\ m\ f\ (f\ x)))) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{trj}\ (S\ m)\ f\ x +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ (\mathit{flow}\ m\ f\ (f\ x)))) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{trj}\ (S\ m)\ f\ x +\!\!+\ \mathit{tail}\ (\mathit{trj}\ n\ f\ (\mathit{flow}\ (S\ m)\ f\ x))) \\
&\mathit{QED}
\end{aligned}
$$

Exercise 5.5:

$$
\begin{aligned}
\mathit{flowTrjLemma}\ Z\ f\ x &= (\mathit{flow}\ Z\ f\ x) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (x) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{last}\ (\mathit{trj}\ Z\ f\ x)) \\
&\mathit{QED}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{flowTrjLemma}\ (S\ m)\ f\ x &= (\mathit{flow}\ (S\ m)\ f\ x) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad ((\mathit{flow}\ m\ f)\ (f\ x)) \\
&= \{\, \mathit{flowTrjLemma}\ m\ f\ (f\ x) \,\} = \\
&\quad (\mathit{last}\ (\mathit{trj}\ m\ f\ (f\ x))) \\
&= \{\, \mathit{sym}\ (\mathit{lastLemma}\ x\ (\mathit{trj}\ m\ f\ (f\ x))) \,\} = \\
&\quad (\mathit{last}\ (x :: \mathit{trj}\ m\ f\ (f\ x))) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{last}\ (\mathit{trj}\ (S\ m)\ f\ x)) \\
&\mathit{QED}
\end{aligned}
$$

Exercise 5.6:

$$
\begin{aligned}
\mathit{reprLemma}\ Z\ f\ xs &= (\mathit{repr}\ (\mathit{flowNonDetSys}\ Z\ f)\ xs) \\
&= \{\, \mathit{Refl} \,\} =
\end{aligned}
$$

$$
\begin{aligned}
&(\mathit{bindList}\ (\mathit{flowNonDetSys}\ Z\ f)\ \mathit{xs}) \\
&= \{\ \mathit{bindListPresExtEq}\ (\mathit{flowNonDetSys}\ Z\ f)\ \mathit{retList}\ (\lambda a \Rightarrow \mathit{Refl})\ \mathit{xs}\ \} = \\
&(\mathit{bindList}\ \mathit{retList}\ \mathit{xs}) \\
&= \{\ \mathit{rightIdentityList}\ \mathit{xs}\ \} = \\
&(\mathit{xs}) \\
&= \{\ \mathit{Refl}\ \} = \\
&(\mathit{flow}\ Z\ (\mathit{repr}\ f)\ \mathit{xs}) \\
&\mathit{QED}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reprLemma}\ (S\ m)\ f\ \mathit{xs} &= (\mathit{repr}\ (\mathit{flowNonDetSys}\ (S\ m)\ f)\ \mathit{xs}) \\
&= \{\ \mathit{Refl}\ \} = \\
&\quad (\mathit{bindList}\ (\mathit{flowNonDetSys}\ (S\ m)\ f)\ \mathit{xs}) \\
&= \{\ \mathit{bindListPresExtEq}\ (\mathit{flowNonDetSys}\ (S\ m)\ f) \\
&\qquad\qquad (\lambda x \Rightarrow \mathit{bindList}\ (\mathit{flowNonDetSys}\ m\ f)\ (f\ x)) \\
&\qquad\qquad (\lambda x \Rightarrow \mathit{Refl}) \\
&\qquad\qquad \mathit{xs}\ \} = \\
&\quad (\mathit{bindList}\ (\lambda x \Rightarrow \mathit{bindList}\ (\mathit{flowNonDetSys}\ m\ f)\ (f\ x))\ \mathit{xs}) \\
&= \{\ \mathit{bindListAssociative}\ f\ (\mathit{flowNonDetSys}\ m\ f)\ \mathit{xs}\ \} = \\
&\quad (\mathit{bindList}\ (\mathit{flowNonDetSys}\ m\ f)\ (\mathit{bindList}\ f\ \mathit{xs})) \\
&= \{\ \mathit{Refl}\ \} = \\
&\quad (\mathit{repr}\ (\mathit{flowNonDetSys}\ m\ f)\ (\mathit{bindList}\ f\ \mathit{xs})) \\
&= \{\ \mathit{reprLemma}\ m\ f\ (\mathit{bindList}\ f\ \mathit{xs})\ \} = \\
&\quad (\mathit{flow}\ m\ (\mathit{repr}\ f)\ (\mathit{bindList}\ f\ \mathit{xs})) \\
&= \{\ \mathit{Refl}\ \} = \\
&\quad (\mathit{flow}\ m\ (\mathit{repr}\ f)\ (\mathit{repr}\ f\ \mathit{xs})) \\
&= \{\ \mathit{Refl}\ \} = \\
&\quad (\mathit{flow}\ (S\ m)\ (\mathit{repr}\ f)\ \mathit{xs}) \\
&\mathit{QED}
\end{aligned}
$$

Exercise 5.7:

$$
\begin{aligned}
\mathit{mapListPresExtEq}\ :\ &\{A, B\ :\ \mathit{Type}\}\ \to\ (f, g\ :\ A\ \to\ \mathit{List}\ B)\ \to \\
&((a\ :\ A)\ \to\ f\ a = g\ a)\ \to \\
&((\mathit{as}\ :\ \mathit{List}\ A)\ \to\ \mathit{mapList}\ f\ \mathit{as} = \mathit{mapList}\ g\ \mathit{as})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{mapListPresExtEq}\ f\ g\ p\ \ \mathit{Nil} &= \mathit{Refl} \\
\mathit{mapListPresExtEq}\ f\ g\ p\ (a :: \mathit{as}) &= (\mathit{mapList}\ f\ (a :: \mathit{as})) \\
&= \{\ \mathit{Refl}\ \} =
\end{aligned}
$$

$$
\begin{aligned}
&(f\ a :: mapList\ f\ as)\\
&= \{\ cong\ \{f = \lambda\alpha \Rightarrow \alpha :: mapList\ f\ as\}\ (p\ a)\ \} =\\
&(g\ a :: mapList\ f\ as)\\
&= \{\ cong\ (mapListPresExtEq\ f\ g\ p\ as)\ \} =\\
&(g\ a :: mapList\ g\ as)\\
&= \{\ Refl\ \} =\\
&(mapList\ g\ (a :: as))\\
&QED
\end{aligned}
$$

$$
\begin{aligned}
bindListPresExtEq\ :\ &\{A, B\ :\ Type\}\ \rightarrow\ (f, g\ :\ A\ \rightarrow\ List\ B)\ \rightarrow\\
&((a\ :\ A)\ \rightarrow\ f\ a = g\ a)\ \rightarrow\\
&((as\ :\ List\ A)\ \rightarrow\ bindList\ f\ as = bindList\ g\ as)
\end{aligned}
$$

$$
\begin{aligned}
bindListPresExtEq\ f\ g\ p\ as = &\ (bindList\ f\ as)\\
&= \{\ Refl\ \} =\\
&(joinList\ (mapList\ f\ as))\\
&= \{\ cong\ (mapListPresExtEq\ f\ g\ p\ as)\ \} =\\
&(joinList\ (mapList\ g\ as))\\
&= \{\ Refl\ \} =\\
&(bindList\ g\ as)\\
&QED
\end{aligned}
$$

$$
rightIdentityList\ :\ \{A\ :\ Type\}\ \rightarrow\ (as\ :\ List\ A)\ \rightarrow\ bindList\ retList\ as = as
$$

$$
\begin{aligned}
&rightIdentityList\ \ Nil \qquad = Refl\\
&rightIdentityList\ (a :: as) = (bindList\ retList\ (a :: as))\\
&\qquad = \{\ Refl\ \} =\\
&\qquad (joinList\ (mapList\ retList\ (a :: as)))\\
&\qquad = \{\ Refl\ \} =\\
&\qquad (joinList\ (retList\ a :: mapList\ retList\ as))\\
&\qquad = \{\ Refl\ \} =\\
&\qquad (retList\ a +\!\!+\ joinList\ (mapList\ retList\ as))\\
&\qquad = \{\ Refl\ \} =\\
&\qquad (retList\ a +\!\!+\ bindList\ retList\ as)\\
&\qquad = \{\ cong\ (rightIdentityList\ as)\ \} =\\
&\qquad (retList\ a +\!\!+\ as)\\
&\qquad = \{\ Refl\ \} =\\
&\qquad ((a :: Nil) +\!\!+\ as)
\end{aligned}
$$

$$
\begin{aligned}
&= \{\, \mathit{Refl} \,\} = \\
&\quad (a :: (\mathit{Nil} +\!\!+ \mathit{as})) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (a :: \mathit{as}) \\
&\mathit{QED}
\end{aligned}
$$

Exercise 5.8:

$$
\begin{aligned}
&\mathit{flowNonDetSys}' : \{X : \mathit{Type}\} \to (\mathit{Eq}\ X) \Rightarrow (m : \mathbb{N}) \to \mathit{NonDetSys}\ X \to \mathit{NonDetSys}\ X \\
&\mathit{flowNonDetSys}'\ Z \qquad f\ x = \mathit{retList}\ x \\
&\mathit{flowNonDetSys}'\ (S\ m)\ f\ x = \mathit{nub}\ (\mathit{joinList}\ (\mathit{mapList}\ (\mathit{flowNonDetSys}'\ m\ f)\ (f\ x)))
\end{aligned}
$$

Lecture 6: Time-discrete monadic dynamical systems

6.1 Natural transformations, monads

- The implementations of *flow* and *trj* for *List* and *Prob* are, mutatis mutandis, the same.
- Any deterministic system can be represented by an equivalent non-deterministic or stochastic system and the other way round!

As it turns out, *Identity*, *List*, *Prob* are *monads*. In category theory, a monad is an *endo-functor* M on a category $\mathbb{C}$ together with two *natural transformations* η and μ such that

$$\begin{tikzcd} M\,X \arrow[r, "\eta\ (M\,X)"] \arrow[dr, "id"'] & M\,(M\,X) \arrow[d, "\mu\ X"] & M\,X \arrow[l, "M\,(\eta\,X)"'] \arrow[dl, "id"] \\ & M\,X & \end{tikzcd} \qquad \begin{tikzcd} M\,(M\,(M\,X)) \arrow[r, "M\,(\mu\,X)"] \arrow[d, "\mu\,(M\,X)"'] & M\,(M\,X) \arrow[d, "\mu\,X"] \\ M\,(M\,X) \arrow[r, "\mu\,X"'] & M\,X \end{tikzcd}$$

commute. A natural transformation γ between two functors F and G between categories $\mathbb{A}$ and $\mathbb{B}$, is a family of arrows in $\mathbb{B}$ indexed by objects in $\mathbb{A}$ such that $(G\ f) \circ (\eta\ X) = (\eta\ Y) \circ (F\ f)$ (left). For $\eta : X \rightarrow M\ X$ and $\mu : M\ (M\ X) \rightarrow M\ X$ this condition is captured in the middle and right diagrams:

$$\begin{tikzcd} F\,X \arrow[r, "F\,f"] \arrow[d, "\gamma\,X"'] & F\,Y \arrow[d, "\gamma\,Y"] \\ G\,X \arrow[r, "G\,f"'] & G\,Y \end{tikzcd} \qquad \begin{tikzcd} X \arrow[r, "f"] \arrow[d, "\eta\,X"'] & Y \arrow[d, "\eta\,Y"] \\ M\,X \arrow[r, "M\,f"'] & M\,Y \end{tikzcd} \qquad \begin{tikzcd} M\,(M\,X) \arrow[r, "M\,(M\,f)"] \arrow[d, "\mu\,X"'] & M\,(M\,Y) \arrow[d, "\mu\,Y"] \\ M\,X \arrow[r, "M\,f"'] & M\,Y \end{tikzcd}$$

Thus, a monad is a functor with the additional properties:

1. Naturality of η: $(M\ f) \circ (\eta\ X) = (\eta\ Y) \circ f$.
2. Naturality of μ: $(M\ f) \circ (\mu\ X) = (\mu\ Y) \circ (M\ (M\ f))$.
3. Triangle left: $(\mu\ X) \circ (\eta\ (M\ X)) = id$.
4. Triangle right: $(\mu\ X) \circ (M\ (\eta\ X)) = id$.
5. Square: $(\mu\ X) \circ (M\ (\mu\ X)) = (\mu\ X) \circ (\mu\ (M\ X))$.

6.2 Interfaces, implementations and generic programming

In Idris, the notions of functor and monad are encoded in a hierarchy of *interfaces*. Idris interfaces (in Haskell *type classes*) factor in the common features (*methods* and *axioms*) of a certain class of data types. For instance, the *Num* interface describes the common features of data types that implement basic numerical arithmetic:

```
interface Num ty where
  (+) : ty -> ty -> ty
  (*) : ty -> ty -> ty
```

$$\mathit{fromInteger} : \mathit{Integer} \to \mathit{ty}$$

A data type for which $(+)$, $(*)$ and $\mathit{fromInteger}$ can be defined, can be declared to be an *implementation* (or an *instance*) of Num. For example, $\mathbb{N}$, Int and Double are all implementation of Num.

For a specific data type, this is done by defining $(+)$, $(*)$ and $\mathit{fromInteger}$ for that type. For instance, for $\mathbb{N}$:

$$\begin{array}{l} \mathit{implementation}\ \mathit{Num}\ \mathbb{N}\ \textbf{where} \\ \quad (+) = \mathit{plus} \\ \quad (*) \ = \mathit{mult} \\ \quad \mathit{fromInteger} = \mathit{fromIntegerNat} \end{array}$$

where $\mathit{plus}, \mathit{mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\mathit{fromIntegerNat} : \mathit{Integer} \to \mathbb{N}$ have to be completely defined and in scope.

Another example of interfaces that we have already encountered in lecture 5 is Eq. This represents the class of types that can be compared for equality. Like Num, Eq is defined in the Idris prelude:

$$\begin{array}{l} \mathit{interface}\ \mathit{Eq}\ \mathit{ty}\ \textbf{where} \\ \quad (==) : \mathit{ty} \to \mathit{ty} \to \mathit{Bool} \\ \quad (\neq) : \mathit{ty} \to \mathit{ty} \to \mathit{Bool} \\ \quad x \neq y = \neg\,(x == y) \\ \quad x == y = \neg\,(x \neq y) \end{array}$$

Notice that Eq specifies *default methods* for $(==)$ and $(\neq)$. Implementations of Eq have to define one of them but they can also define both. Idris interfaces can be *refined*. For instance

$$\begin{array}{l} \mathit{interface}\ \mathit{Num}\ \mathit{ty} \Rightarrow \mathit{Neg}\ \mathit{ty}\ \textbf{where} \\ \quad \mathit{negate} : \mathit{ty} \to \mathit{ty} \\ \quad (-) \quad\ : \mathit{ty} \to \mathit{ty} \to \mathit{ty} \end{array}$$

requires implementations of Neg to implement the features of Num plus negate and $(-)$. Implementations can also be derived *generically* from other implementations. For instance

$$\begin{array}{l} \mathit{implementation}\ (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow \mathit{Eq}\ (a, b)\ \textbf{where} \\ \quad (==)\ (a, c)\ (b, d) = (a == b) \land (c == d) \end{array}$$

explains how values of type (a, b) (for arbitrary types a and b)can be compared for equality, provided that both values of type a and values of type b can be compared for equality.

Interfaces are a powerful mechanism for *generic programming*. One can define functions that work for all implementations of one or more interfaces. We have already seen an example with nub:

$$\begin{array}{l} *\,\mathit{Lecture6} > \ : t\ \mathit{nub} \\ \mathit{Prelude.List.nub} : \mathit{Eq}\ a \Rightarrow \mathit{List}\ a \to \mathit{List}\ a \end{array}$$

$$* \mathit{Lecture6} > \; : \; t \; \mathit{sum}$$
$$\mathit{sum} \; : \; \mathit{Foldable} \; t \Rightarrow \mathit{Num} \; a \Rightarrow t \; a \; \rightarrow \; a$$

6.3 Functor and monad interfaces

In category theory, a functor between categories $\mathbb{A}$ and $\mathbb{B}$, is a mapping of objects and arrow of $\mathbb{A}$ into objects and arrows of $\mathbb{B}$ that preserves identity and composition.

In discussing the notion of a monad, we have required M to be and endo-functor on a category $\mathbb{C}$ and used X, Y, $M\ X$, $M\ Y$ (amd $M\ (M\ X)$, $M\ (M\ (M\ Y))$, etc.) to denote objects in $\mathbb{C}$. Similarly, we have used $M\ f$, $M\ (\eta\ X)$, etc. to denote arrows, also in $\mathbb{C}$.

In Idris the notion of a monad is encoded in a hierarchy of interfaces. Both for historical reasons and for reasons that we do not have time to discuss in this course, this hierarchy is not as straightforward as the category-theoretical notion.

To understanding monadic dynamical system and, more generally, the computational theory of policy advice and verified, optimal decision making discussed in [3], it is not necessary to understand the details of this hierarchy.

However, it is useful to keep in mind the category-theoretical notions of functor and monad and be comfortable with the basic interfaces that encode these notions in Idris. These are, with some simplifications

$$
\begin{array}{l}
\mathit{interface}\ \mathit{Functor}\ (F\ :\ \mathit{Type}\ \rightarrow\ \mathit{Type})\ \textbf{where} \\
\quad \mathit{map}\ :\ (A\ \rightarrow\ B)\ \rightarrow\ F\ A\ \rightarrow\ F\ B
\end{array}
$$

$$
\begin{array}{l}
\mathit{interface}\ \mathit{Functor}\ M \Rightarrow \mathit{Monad}\ (M\ :\ \mathit{Type}\ \rightarrow\ \mathit{Type})\ \textbf{where} \\
\quad \mathit{pure}\ \ :\ A\ \rightarrow\ M\ A \\
\quad (\gg\!=)\ :\ M\ A\ \rightarrow\ (A\ \rightarrow\ M\ B)\ \rightarrow\ M\ B \\
\quad \mathit{join}\ \ \ :\ M\ (M\ A)\ \rightarrow\ M\ A
\end{array}
$$

Thus, in Idris, the arrow mapping part of a functor F is called *map* and the natural transformations η and μ associated with a monad M are called *pure* (or *return*) and *join*, respectively. The operation $(\gg\!=)$ is usually referred to as "bind" and can be derived from *join* and *map*.

Many Idris type constructors turn out to be monads. In particular, *List* is a monad and *Prob* is a monad with *map*, *pure*, $(\gg\!=)$ and *join* defined by *mapList*, *retList*, *bindList*, ..., *joinProb* as discussed in lecture 5.

Traditionally, the Idris *Functor* and *Monad* interfaces specify only methods, not properties. These are collected in suitable refinements of the base interfaces: *VeriFunctor* and *VeriMonad*. Thus, a *VeriFunctor* is a *Functor* whose *map* preserves identity, composition and extensional equality:

$$\begin{array}{l}
\textit{interface Functor } F \Rightarrow \textit{VeriFunctor } (F : \textit{Type} \to \textit{Type}) \textbf{ where} \\
\quad \textit{mapPresId} \quad : \textit{ExtEq } (\textit{map id}) \textit{ id} \\
\quad \textit{mapPresComp} : (f : A \to B) \to (g : B \to C) \to \textit{ExtEq } (\textit{map } (g \circ f)) (\textit{map } g \circ \textit{map } f) \\
\quad \textit{mapPresExtEq} : (f, g : A \to B) \to \textit{ExtEq } f\ g \to \textit{ExtEq } (\textit{map } f) (\textit{map } g)
\end{array}$$

In the above interface, $\textit{ExtEq}$ is a property of functions of the same type:

$$\begin{array}{l}
\textit{ExtEq} : (f, g : A \to B) \to \textit{Type} \\
\textit{ExtEq } f\ g = (a : A) \to f\ a = g\ a
\end{array}$$

Thus, $\textit{mapPresId}$ posits that $\textit{map id fa} = \textit{id fa} = \textit{fa}$ for arbitrary $\textit{fa}$.

Exercise 6.1. Implement $\textit{mapPresId}$ for $F = \textit{List}$, $\textit{map} = \textit{mapList}$ and $\textit{mapList}$ defined as in lecture 5.

Similarly, $\textit{mapPresComp}$ posits that for arbitrary types A, B and C, for every $f : A \to B$ and $g : B \to C$ and for every $\textit{fa}$ of suitable type

$$\textit{map } (g \circ f)\ \textit{fa} = (\textit{map } g \circ \textit{map } f)\ \textit{fa} = \textit{map } g\ (\textit{map } f\ \textit{fa})$$

Exercise 6.2. What is the type of $\textit{fa}$ in the above equation? What is the type of $\textit{map } f\ \textit{fa}$?

The last axiom of $\textit{VeriFunctor}$ requires $\textit{map}$ to preserve extensional equality. We will come back to this axiom later in this lecture. For the time being, we remark that all functors encountered so far fulfill this axiom.

Exercise 6.3. Implement $\textit{mapPresExtEq}$ for $F = \textit{List}$, $\textit{map} = \textit{mapList}$ and $\textit{mapList}$ defined as in lecture 5.

Consistently with the category-theoretical characterization of monads discussed above, a $\textit{VeriMonad}$ is then a $\textit{VeriFunctor}$ together with two natural transformations η and μ that fulfill the monadic axioms 1-5. In Idris, η is called $\textit{pure}$ (or $\textit{ret}$) and μ is called $\textit{join}$:

$$\begin{array}{l}
\textit{interface } (\textit{VeriFunctor } M, \textit{Monad } M) \Rightarrow \textit{VeriMonad } (M : \textit{Type} \to \textit{Type}) \textbf{ where} \\
\quad \textit{pureNatTrans} \quad : (f : A \to B) \to \textit{ExtEq } (\textit{map } f \circ \textit{pure}) (\textit{pure} \circ f) \\
\quad \textit{joinNatTrans} \quad : (f : A \to B) \to \textit{ExtEq } (\textit{map } f \circ \textit{join}) (\textit{join} \circ \textit{map } (\textit{map } f)) \\
\quad \textit{triangleLeft} \quad : \textit{ExtEq } (\textit{join} \circ \textit{pure}) \textit{ id} \\
\quad \textit{triangleRight} \quad : \textit{ExtEq } (\textit{join} \circ \textit{map pure}) \textit{ id}
\end{array}$$

$$\begin{aligned}
&\mathit{squareLemma} &&: \mathit{ExtEq}\ (\mathit{join} \circ \mathit{map}\ \mathit{join})\ (\mathit{join} \circ \mathit{join})\\
&\mathit{bindJoinMapSpec} &&: (f : A \to M\ B) \to \mathit{ExtEq}\ (\gg\!= f)\ (\mathit{join} \circ \mathit{map}\ f)
\end{aligned}$$

The last axiom of *VeriMonad* posits that $ma \gg\!= f = \mathit{join}\ (\mathit{map}\ f\ ma)$ for all ma and f of suitable types. The definition of *bindList* from lecture 5 (with flipped arguments) fulfills this axiom. The axioms of *VeriMonad* allow to derive a number of important, generic results. In the rest of this lecture, we will make use of the following ones:

$$\begin{aligned}
&|||\ \forall f, \forall g, (\forall a, f\ a = g\ a) \Rightarrow (\forall ma, ma \gg\!= f = ma \gg\!= g)\\
&\mathit{bindPresExtEq} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{A, B : \mathit{Type}\} \to (\mathit{VeriMonad}\ M) \Rightarrow\\
&\qquad (f, g : A \to M\ B) \to \mathit{ExtEq}\ f\ g \to \mathit{ExtEq}\ (\gg\!= f)\ (\gg\!= g)
\end{aligned}$$

$$\begin{aligned}
&|||\ \forall f, \forall a, (\mathit{pure}\ a) \gg\!= f = f\ a\\
&\mathit{leftIdentity} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{A, B : \mathit{Type}\} \to (\mathit{VeriMonad}\ M) \Rightarrow\\
&\qquad (f : A \to M\ B) \to \mathit{ExtEq}\ (\lambda a \Rightarrow (\mathit{pure}\ a) \gg\!= f)\ f
\end{aligned}$$

$$\begin{aligned}
&|||\ \forall ma, ma \gg\!= \mathit{pure} = ma\\
&\mathit{rightIdentity} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{A : \mathit{Type}\} \to (\mathit{VeriMonad}\ M) \Rightarrow\\
&\qquad \mathit{ExtEq}\ \{A = M\ A\}\ (\gg\!= \mathit{pure})\ \mathit{id}
\end{aligned}$$

$$\begin{aligned}
&|||\ \forall f, \forall g, \forall ma, (ma \gg\!= f) \gg\!= g = ma \gg\!= (\lambda a \Rightarrow (f\ a) \gg\!= g)\\
&\mathit{associativity} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{A, B, C : \mathit{Type}\} \to (\mathit{VeriMonad}\ M) \Rightarrow\\
&\qquad (f : A \to M\ B) \to (g : B \to M\ C) \to\\
&\qquad \mathit{ExtEq}\ (\lambda ma \Rightarrow (ma \gg\!= f) \gg\!= g)\ (\gg\!= (\lambda x \Rightarrow (f\ x) \gg\!= g))
\end{aligned}$$

$$\begin{aligned}
&|||\ \forall f, \forall g, \forall ma, \mathit{map}\ f\ (ma \gg\!= g) = ma \gg\!= \mathit{map}\ f \circ g\\
&\mathit{mapBindLemma} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{A, B, C : \mathit{Type}\} \to (\mathit{VeriMonad}\ M) \Rightarrow\\
&\qquad (f : B \to C) \to (g : A \to M\ B) \to\\
&\qquad \mathit{ExtEq}\ \{A = M\ A\}\ (\lambda ma \Rightarrow \mathit{map}\ f\ (ma \gg\!= g))\ (\gg\!= \mathit{map}\ f \circ g)
\end{aligned}$$

Implementations can be found in the *VeriMonad* component of [2].

The notion of monad is fundamental in computer science and its usage and applications are ubiquitous in functional programming languages. Among others, monads support the implementation of functional programs in an imperative style via the so-called *do* notation:

```
m_add : Maybe Int -> Maybe Int -> Maybe Int
m_add x y = do x' <- x          -- Extract value from x
               y' <- y          -- Extract value from y
               pure (x' + y')   -- Add them
```

The data type *Maybe* which allows to model partial functions in a controlled fashion

```
data Maybe : (a : Type) -> Type where
```

$$\begin{array}{lcl} \mathit{Nothing} & : & \mathit{Maybe}\ a \\ \mathit{Just} & : & (x\ :\ a)\ \to\ \mathit{Maybe}\ a \end{array}$$

is a monad, and ($\gg\!=$) for *Maybe* fulfills the specification

$$\begin{array}{l} \mathit{Nothing} \gg\!= f = \mathit{Nothing} \\ (\mathit{Just}\ x)\ \gg\!= f = f\ x \end{array}$$

and the **do** expression on the RHS of $m\_add\ x\ y =$ above is syntactic sugar for

$$\begin{array}{l} m\_add\ :\ \mathit{Maybe\ Int}\ \to\ \mathit{Maybe\ Int}\ \to\ \mathit{Maybe\ Int} \\ m\_add\ x\ y = x \gg\!= (\lambda x' \Rightarrow (y \gg\!= (\lambda y' \Rightarrow \mathit{pure}\ (x' + y')))) \end{array}$$

Exercise 6.4. What is the result of $m\_add\ (\mathit{Just}\ 2)\ \mathit{Nothing}$? Apply the definition of m_add to give an semi-formal proof of the result by equational reasoning.

The do-notation is also the basis for *list comprehension* as for instance in

$$\begin{array}{l} *\ \mathit{Lecture6} > [2 * i \mid i \leftarrow [3, 0, 1]] \\ [6, 0, 2]\ :\ \mathit{List\ Integer} \end{array}$$

More generally, monads are used to encapsulate various kinds of "computational effects" and thereby allow to model computations with side-effects in a purely functional setting. The idea to use monads for this purpose goes back to a seminal paper by Moggi [5] and was further popularized by Wadler [6].

6.4 Monadic systems

The notion od *monadic dynamical system* was originally introduced by C. Ionescu in [4].

In a nutshell, the idea is to account for different kinds of uncertainty in dynamical systems – deterministic, non-deterministic, stochastic, etc. as discussed in lecture 5 – in a seamless way.

This also makes it possible to prove important results (like for instance the representation theorems of lecture 5) for the general case and avoid tedious and error-prone repetitions. We follow essentially the pattern of definitions and proofs put forward in lecture 5.

6.4.1 Preliminaries

A discrete monadic dynamical system on a set X is a function of type $X\ \to\ M\ X$ where M is a monad:

$$
\begin{aligned}
&\mathit{MonadicSys} : (M : \mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
&\mathit{MonadicSys}\ M\ X = X \to M\ X
\end{aligned}
$$

The set X in $f : \mathit{MonadicSys}\ M\ X$ is often called the "state space" of the system f:

$$
\begin{aligned}
&\mathit{StateSpace} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \mathit{MonadicSys}\ M\ X \to \mathit{Type} \\
&\mathit{StateSpace} = \mathit{Domain}
\end{aligned}
$$

Every deterministic system on X can be represented by a monadic systems on X:

$$
\begin{aligned}
&\mathit{embed} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \mathit{Monad}\ M \Rightarrow \mathit{DetSys}\ X \to \mathit{MonadicSys}\ M\ X \\
&\mathit{embed}\ f = \mathit{pure} \circ f
\end{aligned}
$$

6.4.2 Flow

The flow of a monadic system f over t steps is another monadic system:

$$
\begin{aligned}
&\mathit{flow} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad \mathbb{N} \to \mathit{MonadicSys}\ M\ X \to \mathit{MonadicSys}\ M\ X \\
&\mathit{flow}\ \ Z\ \quad f\ x = \mathit{pure}\ x \\
&\mathit{flow}\ (S\ n)\ f\ x = f\ x \gg\!= \mathit{flow}\ n\ f
\end{aligned}
$$

Trivially, one has $\mathit{flow}\ Z\ f = \mathit{pure}$:

$$
\begin{aligned}
&\mathit{flowLemma1} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad\qquad (f : \mathit{MonadicSys}\ M\ X) \to \mathit{ExtEq}\ (\mathit{flow}\ Z\ f)\ \mathit{pure} \\
&\mathit{flowLemma1}\ f\ x = \mathit{Refl}
\end{aligned}
$$

and also $\mathit{flow}\ (t + t')\ f\ x = \mathit{flow}\ t\ f\ x \gg\!= \mathit{flow}\ t'\ f$:

$$
\begin{aligned}
&\mathit{flowLemma2} : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \{m, n : \mathbb{N}\} \to \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad\qquad (f : \mathit{MonadicSys}\ M\ X) \to \mathit{ExtEq}\ (\mathit{flow}\ (m + n)\ f)\ (\lambda x \Rightarrow \mathit{flow}\ m\ f\ x \gg\!= \mathit{flow}\ n\ f) \\
&\mathit{flowLemma2}\ \{m = Z\}\ \{n\}\ f\ x = \\
&\qquad (\mathit{flow}\ (Z + n)\ f\ x) \\
&\quad = \{\mathit{Refl}\} = \\
&\qquad (\mathit{flow}\ n\ f\ x) \\
&\quad = \{\mathit{sym}\ (\mathit{leftIdentity}\ (\mathit{flow}\ n\ f)\ x)\} = \\
&\qquad (\mathit{pure}\ x \gg\!= \mathit{flow}\ n\ f) \\
&\quad = \{\mathit{Refl}\} = \\
&\qquad (\mathit{flow}\ Z\ f\ x \gg\!= \mathit{flow}\ n\ f) \\
&\quad \mathit{QED}
\end{aligned}
$$

$$
\mathit{flowLemma2}\ \{m = S\ l\}\ \{n\}\ f\ x =
$$

$$
\begin{aligned}
&\quad (\mathit{flow}\ (S\ l + n)\ f\ x) \\
&= \{\mathit{Refl}\} = \\
&\quad (f\ x \gg\!= \mathit{flow}\ (l + n)\ f) \\
&= \{\mathit{bindPresExtEq}\ (\mathit{flow}\ (l + n)\ f)\ (\lambda y \Rightarrow \mathit{flow}\ l\ f\ y \gg\!= \mathit{flow}\ n\ f)\ (\mathit{flowLemma2}\ f)\ (f\ x)\} = \\
&\quad (f\ x \gg\!= (\lambda y \Rightarrow \mathit{flow}\ l\ f\ y \gg\!= \mathit{flow}\ n\ f)) \\
&= \{\mathit{sym}\ (\mathit{associativity}\ (\mathit{flow}\ l\ f)\ (\mathit{flow}\ n\ f)\ (f\ x))\} = \\
&\quad ((f\ x \gg\!= \mathit{flow}\ l\ f) \gg\!= \mathit{flow}\ n\ f) \\
&= \{\mathit{Refl}\} = \\
&\quad (\mathit{flow}\ (S\ l)\ f\ x \gg\!= \mathit{flow}\ n\ f) \\
&\mathit{QED}
\end{aligned}
$$

Every monadic system $f : \mathit{MonadicSys}\ M\ X$ can be represented by an equivalent deterministic systems on $M\ X$

$$
\begin{aligned}
&\mathit{repr}\ : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \mathit{VeriMonad}\ M \Rightarrow \mathit{MonadicSys}\ M\ X \to \mathit{DetSys}\ (M\ X) \\
&\mathit{repr}\ f\ xs = xs \gg\!= f
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{flowDetSys}\ : \{X : \mathit{Type}\} \to \mathbb{N} \to \mathit{DetSys}\ X \to \mathit{DetSys}\ X \\
&\mathit{flowDetSys} = \mathit{Lecture5.flow}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reprLemma}\ : \{M : \mathit{Type} \to \mathit{Type}\} \to \{X : \mathit{Type}\} \to \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad\qquad (n : \mathbb{N}) \to (f : \mathit{MonadicSys}\ M\ X) \to \\
&\qquad\qquad \mathit{ExtEq}\ \{A = M\ X\}\ (\gg\!= \mathit{flow}\ n\ f)\ (\mathit{flowDetSys}\ n\ (\mathit{repr}\ f)) \\
&\mathit{reprLemma}\ Z\ f\ mx = \\
&\qquad (mx \gg\!= \mathit{flow}\ Z\ f) \\
&\quad = \{\mathit{bindPresExtEq}\ (\mathit{flow}\ Z\ f)\ \mathit{pure}\ (\mathit{flowLemma1}\ f)\ mx\} = \\
&\qquad (mx \gg\!= \mathit{pure}) \\
&\quad = \{\mathit{rightIdentity}\ mx\} = \\
&\qquad (mx) \\
&\quad = \{\mathit{Refl}\} = \\
&\qquad (\mathit{flowDetSys}\ Z\ (\mathit{repr}\ f)\ mx) \\
&\quad \mathit{QED} \\
&\mathit{reprLemma}\ (S\ m)\ f\ xs = \\
&\qquad (xs \gg\!= \mathit{flow}\ (S\ m)\ f) \\
&\quad = \{\mathit{bindPresExtEq}\ (\mathit{flow}\ (S\ m)\ f)\ (\lambda x \Rightarrow f\ x \gg\!= \mathit{flow}\ m\ f)\ (\lambda x \Rightarrow \mathit{Refl})\ xs\} = \\
&\qquad (xs \gg\!= (\lambda x \Rightarrow f\ x \gg\!= \mathit{flow}\ m\ f)) \\
&\quad = \{\mathit{sym}\ (\mathit{associativity}\ f\ (\mathit{flow}\ m\ f)\ xs)\} = \\
&\qquad ((xs \gg\!= f) \gg\!= \mathit{flow}\ m\ f) \\
&\quad = \{\mathit{reprLemma}\ m\ f\ (xs \gg\!= f)\} = \\
&\qquad ((\mathit{flowDetSys}\ m\ (\mathit{repr}\ f))\ (xs \gg\!= f))
\end{aligned}
$$

$$= \{ \mathit{Refl} \} =$$
$$((\mathit{flowDetSys}\ m\ (\mathit{repr}\ f))\ ((\mathit{repr}\ f)\ \mathit{xs}))$$
$$= \{ \mathit{Refl} \} =$$
$$(\mathit{flowDetSys}\ (S\ m)\ (\mathit{repr}\ f)\ \mathit{xs})$$
$$\mathit{QED}$$

6.4.3 Trajectories

For a dynamical system $f\ :\ \mathit{MonadicSys}\ M\ X$, the trajectories of length $n\ :\ \mathbb{N}$ starting at $x\ :\ X$ are

$$\begin{aligned}
&\mathit{trj}\ :\ \{M\ :\ \mathit{Type}\ \to\ \mathit{Type}\}\ \to\ \{X\ :\ \mathit{Type}\}\ \to\ \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad (n\ :\ \mathbb{N})\ \to\ \mathit{MonadicSys}\ M\ X\ \to\ X\ \to\ M\ (\mathit{Vect}\ (S\ n)\ X) \\
&\mathit{trj}\ \ Z\ \ \ \ \ f\ x = \mathit{map}\ (x{::})\ (\mathit{pure}\ \mathit{Nil}) \\
&\mathit{trj}\ (S\ n)\ f\ x = \mathit{map}\ (x{::})\ ((f\ x) \gg\!= \mathit{trj}\ n\ f)
\end{aligned}$$

Remember that for deterministic systems the last state of the trajectory of length n starting in x is $\mathit{flow}\ n\ f\ x$.

In the general, monadic case $\mathit{trj}\ n\ f\ x$ is an M-structure of vectors. But mapping last on $\mathit{trj}\ n\ f\ x$ yields, again, $\mathit{flow}\ n\ f$:

$$\begin{aligned}
&\mathit{flowTrjLemma}\ :\ \{M\ :\ \mathit{Type}\ \to\ \mathit{Type}\}\ \to\ \{X\ :\ \mathit{Type}\}\ \to\ \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad (n\ :\ \mathbb{N})\ \to\ (f\ :\ \mathit{MonadicSys}\ M\ X)\ \to \\
&\qquad \mathit{ExtEq}\ (\mathit{flow}\ n\ f)\ (\mathit{map}\ \{a = \mathit{Vect}\ (S\ n)\ X\}\ \mathit{last} \circ (\mathit{trj}\ n\ f))
\end{aligned}$$

To prove this result, we first derive the auxiliary lemma

$$\begin{aligned}
&\mathit{mapLastLemma}\ :\ \{M\ :\ \mathit{Type}\ \to\ \mathit{Type}\}\ \to\ \{X\ :\ \mathit{Type}\}\ \to\ \{n\ :\ \mathbb{N}\}\ \to\ \mathit{VeriMonad}\ M \Rightarrow \\
&\qquad (x\ :\ X)\ \to\ \mathit{ExtEq}\ \{A = M\ (\mathit{Vect}\ (S\ n)\ X)\}\ (\mathit{map}\ \mathit{last} \circ \mathit{map}\ (x{::}))\ (\mathit{map}\ \mathit{last}) \\
&\mathit{mapLastLemma}\ \{X\}\ \{n\}\ x\ \mathit{mvs} = \\
&\qquad (\mathit{map}\ \mathit{last}\ (\mathit{map}\ (x{::})\ \mathit{mvs})) \\
&\quad = \{\mathit{sym}\ (\mathit{mapPresComp}\ \{A = \mathit{Vect}\ (S\ n)\ X\}\ (x{::})\ \mathit{last}\ \mathit{mvs})\} = \\
&\qquad (\mathit{map}\ (\mathit{last} \circ (x{::}))\ \mathit{mvs}) \\
&\quad = \{\mathit{mapPresExtEq}\ (\mathit{last} \circ (x{::}))\ \mathit{last}\ (\mathit{lastLemma}\ x)\ \mathit{mvs}\} = \\
&\qquad (\mathit{map}\ \mathit{last}\ \mathit{mvs}) \\
&\quad \mathit{QED}
\end{aligned}$$

And finally implement $\mathit{flowTrjLemma}\ n\ f$ by induction on n:

$$\begin{aligned}
&\mathit{flowTrjLemma}\ \{X\}\ Z\ f\ x = \\
&\qquad (\mathit{flow}\ Z\ f\ x) \\
&\quad = \{\mathit{Refl}\} =
\end{aligned}$$

$$
\begin{aligned}
&\quad (pure\ x) \\
&= \{ Refl \} = \\
&\quad (pure\ (last\ (x :: Nil))) \\
&= \{ sym\ (pureNatTrans\ last\ (x :: Nil)) \} = \\
&\quad (map\ last\ (pure\ (x :: Nil))) \\
&= \{ cong\ \{ f = map\ last \}\ (sym\ (pureNatTrans\ \{ A = Vect\ Z\ X \}\ (x ::)\ Nil)) \} = \\
&\quad (map\ last\ (map\ (x ::)\ (pure\ Nil))) \\
&= \{ Refl \} = \\
&\quad (map\ last\ (trj\ Z\ f\ x)) \\
&QED
\end{aligned}
$$

$$
\begin{aligned}
&flowTrjLemma\ (S\ m)\ f\ x \ = \\
&\quad (flow\ (S\ m)\ f\ x) \\
&= \{ Refl \} = \\
&\quad (f\ x \gg= flow\ m\ f) \\
&= \{ bindPresExtEq\ (flow\ m\ f)\ (map\ last \circ (trj\ m\ f))\ (flowTrjLemma\ m\ f)\ (f\ x) \} = \\
&\quad (f\ x \gg= map\ last \circ (trj\ m\ f)) \\
&= \{ sym\ (mapBindLemma\ last\ (trj\ m\ f)\ (f\ x)) \} = \\
&\quad (map\ last\ ((f\ x) \gg= trj\ m\ f)) \\
&= \{ sym\ (mapLastLemma\ x\ ((f\ x) \gg= trj\ m\ f)) \} = \\
&\quad (map\ last\ (map\ (x ::)\ ((f\ x) \gg= trj\ m\ f))) \\
&= \{ Refl \} = \\
&\quad (map\ last\ (trj\ (S\ m)\ f\ x)) \\
&QED
\end{aligned}
$$

6.5 Time dependent dynamical system

In many important applications, one has to deal with dynamical systems in which X can be different at different iteration steps.

For example, in lecture 1 we have sketched a sequential decision problem in which

- At the first decision step, the decision maker observes zero cumulated emissions, high current emissions, unavailable technologies and a good world.
- ... if the cumulated emissions increase beyond a critical threshold, the probability that the world becomes bad steeply increases.

This suggests that the set of values that cumulated emissions can take, i.e. the *type* of cumulated emissions might change as the system evolves.

For concreteness, assume that at each step the cumulated emissions can only increase by one. Let e denote the cumulated emissions. Then we have the following situation

$$at\ step\ 0, \mid e \in \{0\} \mid$$
$$at\ step\ 1, \mid e \in \{0, 1\} \mid$$
$$at\ step\ n, \mid e \in \{0 \mathinner{..} n\} \mid$$

if cumulated emissions are a component of a type X that represents the set of observable states of a system, then X will depend on an iteration counter. Formally

$$X : \mathbb{N} \to Type$$

A monadic dynamical system on X could then be specified in terms of a monad M

$$M : Type \to Type$$

and of a transition function $next$

$$next : (t : \mathbb{N}) \to X\ t \to M\ (X\ (S\ t))$$

Here the variable t denotes an iteration counter. In case $X\ t$ entails a notion of time or, in other words, if we have a function

$$time : \{t : \mathbb{N}\} \to X\ t \to \mathbb{N}$$

that associate a time to state values, we can formalize the idea that the system evolves forwards (in time) by requiring

$$nextMonInc : (t, t' : \mathbb{N}) \to (x : X\ t) \to (x' : X\ t') \to t\ `LTE`\ t' \to time\ x\ `LTE`\ time\ x'$$

Similarly we can require a system to evolve backwards in time.

6.6 Decision making under uncertainty

In decision making problems, one has to do with systems whose evolution depends both on the system's state and on the options available to the decision maker.

In control theory, the options are called controls. They typically depend on the systems's state that is, the options available to the decision maker can be fifferent in different states.

Example: A central bank can typically increase or decrease the interest rates. But the amount by which a central bank is able to do so, can depend on the current interest rates and perhaps on other economic observables like for instance growth and unemployment or other measures and indicators.

Example: A country might be able to increase or decrease the emissions of certain dangerous pollutants. But the options available to decision makers might depend on the availability or non-availability of effective filtering technologies, on the state of the economy or perhaps on the actual level of pollutant.

It is not difficult to modify the notion of a time-dependent monadic dynamical system to represent decision making problems. All we need to do is to introduce the notion of (possibly state-dependent) controls

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

The transition function of the system at step t will then depend both on the current state $x \ : \ X \ t$ and on the control $y \ : \ Y \ t \ x$ selected

$$X \ : \ \mathbb{N} \ \to \ \mathit{Type}$$

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

$$M \ : \ \mathit{Type} \ \to \ \mathit{Type}$$

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

In decision making under uncertainty, X, Y, M and next are typically given and the problem is that of finding sequences of controls such that the resulting trajectories fulfill certain conditions.

6.7 Coming up

In the next lecture we will start formalizing finite horizon sequential decision problems for the deterministic case. These problems are at the core of *dynamic programming* as originally proposed by Bellman in 1957 [1] and the first step towards optimal decision making under uncertainty.

References

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

[2] Nicola Botta. IdrisLibs. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`, 2016–2018.

[3] Nicola Botta, Patrik Jansson, and Cezar Ionescu. Contributions to a computational theory of policy advice and avoidability. *J. Funct. Program.*, (27, e23), 2017.

[4] Cezar Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems.* PhD thesis, Freie Universität Berlin, 2009.

[5] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[6] Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*, volume 118 of *NATO ASI Series*, pages 233–264. Springer, 1992.

Lecture 7: Deterministic sequential decision problems, naïve theory

We go back to the *deterministic case* and first build a theory of optimal decision making for deterministic sequential decision problems (SDP). In lecture 9 we will then generalize the theory to *monadic* SDPs.

7.1 States, controls and transition function

At the core of an SDP we have a dynamical system with control as discussed in lecture 5:

$$
\begin{aligned}
&X &&: (t : \mathbb{N}) \to \mathit{Type} \\
&Y &&: (t : \mathbb{N}) \to (x : X\ t) \to \mathit{Type} \\
&\mathit{next} &&: (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to X\ (S\ t)
\end{aligned}
$$

Remember that $X\ t$ represents the set of states a decision maker can observe at decision step t. For a given state $x : X\ t$, $Y\ t\ x$ are the controls (options, choices, etc.) available to the decision maker in x.

Remark: In order to specify an SDP, X, Y and next have to be defined.

Example: In an emission problem like the one discussed in the first lecture, $X\ t$ represents the cumulated emissions, the current emissions, the availability of technologies for reducing emissions and the "state of the world".

7.2 Reward functions

SDPs can be formulated is by introducing a *reward* function

$$
\begin{aligned}
&\mathit{Val} : \mathit{Type} \\
&\mathit{reward} : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to (x' : X\ (S\ t)) \to \mathit{Val}
\end{aligned}
$$

that associates a unique value to every transition. Specifically, $\mathit{reward}\ t\ x\ y\ x'$ represents the reward (payoff, etc.) that the decision maker associates to a transition from x to x' when the control y is selected.

Remark: In many SDPs, controls are associated with the consumption of resources that might be scarce: money, fuel, common goods, etc.

Remark: In shortest path and optimal routing problems, rewards are often zero everywhere and one for values of x' corresponding to the destination or goal.

Since the original work of Bellman [1], the above has turned out to be a useful approach for formulating and solving SDPs. The idea is that the decision maker seeks controls that maximize the sum of the rewards obtained in a finite number of steps. This implies that values of type Val have to be "addable"

$$(\oplus) : \mathit{Val} \to \mathit{Val} \to \mathit{Val}$$

Moreover, Val has to be equipped with a "zero"

$$\mathit{zero} : \mathit{Val}$$

and with a binary "comparison" relation

$$(\leqslant) : \mathit{Val} \to \mathit{Val} \to \mathit{Type}$$

Remark: In many SDPs, Val is $\mathbb{N}$ or $\mathbb{R}$ and $(\oplus)$ and zero are the standard addition and its neutral element.

7.3 Policies and policy sequences

Policies are functions that associate to every state $x : X\ t$ at decision step t a control in $Y\ t\ x$:

$$\begin{aligned}
&\mathit{Policy} : (t : \mathbb{N}) \to \mathit{Type}\\
&\mathit{Policy}\ t = (x : X\ t) \to Y\ t\ x
\end{aligned}$$

Policy sequences are literally sequences of policies:

$$\begin{aligned}
&\textbf{data}\ \mathit{PolicySeq} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{Type}\ \textbf{where}\\
&\quad \mathit{Nil} : \{t : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ Z\\
&\quad (::) : \{t, n : \mathbb{N}\} \to \mathit{Policy}\ t \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{PolicySeq}\ t\ (S\ n)
\end{aligned}$$

The Nil data constructor warrants that we can construct an empty policy sequence for every decision step; $(::)$ warrants that with a decision policy for step t and a policy sequence that supports n decision steps starting from states in $X\ (S\ t)$, we can construct a policy sequence that supports $S\ n$ decision steps starting from states in $X\ t$.

Exercise 7.1. Assume that $X\ t = R$ and $Y\ t\ x = S$ for all $t : \mathbb{N}$, $x : X\ t = R$. Formalize the notions of policy and policy sequence for this special case.

7.4 The value of policy sequences

Given a policy sequence for n decision steps, we can easily compute the value of taking n decisions according to that sequence in terms of the sum of the rewards obtained:

$$\begin{aligned}
&\mathit{val} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ n \to (x : X\ t) \to \mathit{Val}\\
&\mathit{val}\ \{t\}\ \mathit{Nil}\qquad x = \mathit{zero}\\
&\mathit{val}\ \{t\}\ (p :: ps)\ x = \textbf{let}\ y = p\ x\ \textbf{in}\\
&\qquad\qquad\qquad\quad \textbf{let}\ x' = \mathit{next}\ t\ x\ y\ \textbf{in}\\
&\qquad\qquad\qquad\quad \mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val}\ ps\ x'
\end{aligned}$$

7.5 Optimality of policy sequences

Remember that the decision maker seeks controls that maximize the sum of the rewards obtained in a finite number of decision steps.

Because val exactly computes this sum for arbitrary policy sequences, we can formalise what it means for one such sequence to be *optimal*:

$$OptPolicySeq : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to Type$$
$$OptPolicySeq\ \{t\}\ \{n\}\ ps = (ps' : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps'\ x \leqslant val\ ps\ x$$

Remark: This notion of optimality contains a quantification over all states $x : X\ t$. This implies that a policy sequence which is worse (in terms of val) than another sequence in a particular state cannot be optimal!

7.6 Optimal extensions of policy sequences

The computation of *optimal extensions* of policy sequences is the key for the computation of optimal policy sequences.

An extension of a policy sequence for making m decision steps starting from states at decision step $S\ t$ is a policy for taking decisions at step t.

A policy p is an optimal extension of a policy sequence ps if there is no better way than $p :: ps$ to make $S\ m$ decision steps starting from step t:

$$OptExt : \{t, m : \mathbb{N}\} \to PolicySeq\ (S\ t)\ m \to Policy\ t \to Type$$
$$OptExt\ \{t\}\ ps\ p = (p' : Policy\ t) \to (x : X\ t) \to val\ (p' :: ps)\ x \leqslant val\ (p :: ps)\ x$$

The idea behind the notion of optimal extension is that if p is an optimal extension of ps and ps is optimal, then $p :: ps$ is optimal. This is *Bellman's principle of optimality.*

7.7 Bellman's principle

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

If $\leqslant$ is reflexive and transitive and $\oplus$ is monotonic with respect to $\leqslant$,

$$lteRefl : \{a : Val\} \to a \leqslant a$$

$$lteTrans : \{a, b, c : Val\} \to a \leqslant b \to b \leqslant c \to a \leqslant c$$

$$plusMon : \{a, b, c, d : Val\} \to a \leqslant b \to c \leqslant d \to (a \oplus c) \leqslant (b \oplus d)$$

then proving Bellman's principle is straightforward:

$$
\begin{aligned}
&\mathit{Bellman}\ \{t\}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep} = \mathit{opps}\ \textbf{where}\\
&\quad \mathit{opps}\ (p' :: \mathit{ps}')\ x =\\
&\qquad \textbf{let}\ y' = p'\ x\ \textbf{in}\\
&\qquad \textbf{let}\ x' = \mathit{next}\ t\ x\ y'\ \textbf{in}\\
&\qquad \textbf{let}\ s_1 = \mathit{plusMon}\ \mathit{lteRefl}\ (\mathit{ops}\ \mathit{ps}'\ x')\ \textbf{in} \quad \text{--}\ \mathit{val}\ (p' :: \mathit{ps}')\ x \leqslant \mathit{val}\ (p' :: \mathit{ps})\ x\\
&\qquad \textbf{let}\ s_2 = \mathit{oep}\ p'\ x\ \textbf{in} \quad \text{--}\ \mathit{val}\ (p' :: \mathit{ps})\ x \leqslant \mathit{val}\ (p :: \mathit{ps})\ x\\
&\qquad \mathit{lteTrans}\ s_1\ s_2
\end{aligned}
$$

7.8 Generic verified backwards induction

From the reflexivity of $\leqslant$ it follows that empty policy sequences are optimal:

$$
\begin{aligned}
&\mathit{nilOptPolicySeq} : \mathit{OptPolicySeq}\ \mathit{Nil}\\
&\mathit{nilOptPolicySeq}\ \mathit{Nil}\ x = \mathit{lteRefl}
\end{aligned}
$$

Thus, assuming that we have a method for computing optimal extensions of arbitrary policy sequences:

$$
\mathit{optExt} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{Policy}\ t
$$

$$
\mathit{optExtSpec} : \{t, n : \mathbb{N}\} \to (\mathit{ps} : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})
$$

it is easy to implement a generic backwards induction for computing optimal policies for arbitrary decision problems:

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

and to prove that bi is correct:

$$
\begin{aligned}
&\mathit{biLemma} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)\\
&\mathit{biLemma}\ t\ \ Z \quad\ = \mathit{nilOptPolicySeq}\\
&\mathit{biLemma}\ t\ (S\ n) = \textbf{let}\ \mathit{ps} \ \ = \mathit{bi}\ (S\ t)\ n\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \ \textbf{let}\ \mathit{ops} = \mathit{biLemma}\ (S\ t)\ n\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \ \textbf{let}\ p \quad = \mathit{optExt}\ \mathit{ps}\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \ \textbf{let}\ \mathit{oep} = \mathit{optExtSpec}\ \mathit{ps}\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \ \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}
\end{aligned}
$$

7.9 Naive theory, wrap up

The theory is applied in three steps:

- First, specify a concrete SDP by implementing X, Y, $next$, Val, $reward$, $\oplus$, $zero$, $\leqslant$ and $optExt$.
- Then, apply $bi\ t\ n$ and compute an optimal policy sequence $[p_t \mathinner{..} p_{t+n-1}]$ for $n > 0$ decision steps starting from step t.
- For an initial observation $x_t : X\ t$, compute the n optimal controls:

$$
\begin{array}{llll}
y_t & = p_t\ x_t, & x_{t+1} = next\ \ t & x_t\ \ y_t \\
y_{t+1} & = p_{t+1}\ x_{t+1}, & x_{t+2} = next\ (t+1) & x_{t+1}\ \ y_{t+1} \\
\ldots & & & \\
y_{t+n-1} & = p_{t+n-1}\ x_{t+n-1}, & x_{t+n} = next\ (t+n-1) & x_{t+n-1}\ \ y_{t+n-1}
\end{array}
$$

- Bonus: If $\leqslant$ is reflexive and transitive, $\oplus$ is monotonic with respect to $\leqslant$ and $optExt$ fulfills $optExtSpec$ then $y_t, y_{t+1} \mathinner{..} y_{t+n-1}$ are verified optimal decisions.

7.10 The bad news

The naive theory is simple and straightforward but has a major flaw.

What if $Y\ t\ x_t$ is empty? In this case, we cannot compute a $y_t : Y\ t\ x_t$ and thus a next state! There is so far nothing in our formulation that prevents the set of controls associated with a certain state to be empty.

Conversely, there is nothing in the computation of optimal policies that prevents a policy to select a control that through $next$ leads to a state whose set of controls is empty.

As a result, we might not be able to "solve" even very simple decision problems. This is made evident in the following example. Let

$$
\begin{array}{l}
head\ : \{t, n\ :\ \mathbb{N}\}\ \to\ PolicySeq\ t\ (S\ n)\ \to\ Policy\ t \\
head\ (p :: ps) = p
\end{array}
$$

$$
\begin{array}{l}
tail\ : \{t, n\ :\ \mathbb{N}\}\ \to\ PolicySeq\ t\ (S\ n)\ \to\ PolicySeq\ (S\ t)\ n \\
tail\ (p :: ps) = ps
\end{array}
$$

and

```
data GoodOrBad = Good | Bad

implementation Show GoodOrBad where
  show Good = "Good"
```

$$\mathit{show}\ \mathit{Bad}\ = \texttt{"Bad"}$$

$$\textbf{data}\ \mathit{UpOrDown} = \mathit{Up} \mid \mathit{Down}$$

Consider the decision problem

$$X\ t = \mathit{GoodOrBad}$$

$$\begin{aligned} &Y\ t\ \mathit{Good} = \mathit{UpOrDown} \\ &Y\ t\ \mathit{Bad}\ \ = \mathit{Void} \end{aligned}$$

In Good there are two options: Up and Down. But in Bad there are no controls to select. We can define a transition function

$$\begin{aligned} &\mathit{next}\ t\ \mathit{Good}\ \mathit{Up} && = && \mathit{Good} \\ &\mathit{next}\ t\ \mathit{Good}\ \mathit{Down} && = && \mathit{Bad} \\ &\mathit{next}\ t\ \mathit{Bad}\ \ \ v && \mathit{impossible} \end{aligned}$$

but we will not be able to apply next for the Bad case unless we manage to construct a value $v\ :\ \mathit{Void}$. This is impossible. Still, we can complete the specification of the problem:

$$\begin{aligned} &\mathit{Val} = \mathbb{N} \\ &(\oplus) = (+) \\ &\mathit{zero} = Z \\ &(\leqslant) = \mathit{Prelude}.\mathbb{N}.\mathit{LTE} \\ &\mathit{reward}\ t\ \mathit{Good}\ \mathit{Up}\ x' && = && 1 \\ &\mathit{reward}\ t\ \mathit{Good}\ \mathit{Down}\ x' && = && 3 \\ &\mathit{reward}\ t\ \mathit{Bad}\ v\ x' && \mathit{impossible} \end{aligned}$$

Notice that, again, we will not be able to compute an argument $v\ :\ \mathit{Void}$ to apply reward. This also implies that we cannot give a complete implementation of optExt. We can compute a best control for Good:

$$\begin{aligned} &\mathit{optExt}\ \{t\}\ \mathit{ps}\ \mathit{Good} = \\ &\quad \textbf{let}\ x'\mathit{Up}\quad\ \ = \mathit{next}\ t\ \mathit{Good}\ \mathit{Up}\ \textbf{in} \\ &\quad \textbf{let}\ x'\mathit{Down}\ = \mathit{next}\ t\ \mathit{Good}\ \mathit{Down}\ \textbf{in} \\ &\quad \textbf{let}\ \mathit{valUp}\quad\ = \mathit{reward}\ t\ \mathit{Good}\ \mathit{Up}\quad\ x'\mathit{Up}\quad\ \oplus \mathit{val}\ \mathit{ps}\ x'\mathit{Up}\ \textbf{in} \\ &\quad \textbf{let}\ \mathit{valDown} = \mathit{reward}\ t\ \mathit{Good}\ \mathit{Down}\ x'\mathit{Down} \oplus \mathit{val}\ \mathit{ps}\ x'\mathit{Down}\ \textbf{in} \\ &\quad \textbf{if}\ \mathit{valUp} \geqslant \mathit{valDown}\ \textbf{then}\ \mathit{Up}\ \textbf{else}\ \mathit{Down} \end{aligned}$$

But not for Bad:

$$\mathit{optExt}\ \{t\}\ \mathit{ps}\ \mathit{Bad} =\ ?\mathit{whatNow}$$

In spite of this, we can try to compute a policy sequence for two decision steps and see what happens:

$$
\begin{aligned}
&\mathit{computation} \;:\; \mathit{IO}\;() \\
&\mathit{computation} = \textbf{let}\; \mathit{ps} \;=\; \mathit{bi}\;0\;2\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; x_0 \;=\; \mathit{Good}\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; p_0 \;=\; \mathit{head}\;\mathit{ps}\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; \mathit{y0} \;=\; p_0\;x_0\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; x_1 \;=\; \mathit{next}\;Z\;x_0\;\mathit{y0}\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; p_1 \;=\; \mathit{head}\;(\mathit{tail}\;\mathit{ps})\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; y_1 \;=\; p_1\;x_1\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{let}\; x_2 \;=\; \mathit{next}\;1\;x_1\;y_1\;\textbf{in} \\
&\qquad\qquad\qquad\;\; \textbf{do}\; \mathit{putStrLn}\;(\texttt{"x0 = "} +\!\!+\; \mathit{show}\;x_0) \\
&\qquad\qquad\qquad\qquad\; \mathit{putStrLn}\;(\texttt{"x1 = "} +\!\!+\; \mathit{show}\;x_1) \\
&\qquad\qquad\qquad\qquad\; \mathit{putStrLn}\;(\texttt{"x2 = "} +\!\!+\; \mathit{show}\;x_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{main} \;:\; \mathit{IO}\;() \\
&\mathit{main} = \mathit{computation}
\end{aligned}
$$

Exercise 7.2. Do you expect this program to terminate? If so, what do you expect to be the result of the computation?

7.11 Wrap-up, outlook

- If the control space for one or more states is empty, the theory becomes problematic. We can proceed in two ways:
- Require $Y\;t\;x$ to be non-empty for all $t : \mathbb{N}$, $x : X\;t$.
- Accept that $Y\;t\;x$ might be empty and be more careful in the definition of the domain and of the codomain of policies.
- The second approach is the one adopted in [4], [3] and [2]. We discuss it in the next lecture.

Additional remarks: trajectories and consistency of val

We want to show that $\mathit{val}\;\mathit{ps}\;x$ does indeed compute the sum of the rewards obtained along the trajectory that is obtained under the policy sequence ps when starting in x. To this end, we start by defining sequences of state-control pairs

$$
\begin{aligned}
&\textbf{data}\; \mathit{StateCtrlSeq} \;:\; (t \;:\; \mathbb{N}) \;\to\; (n \;:\; \mathbb{N}) \;\to\; \mathit{Type}\;\textbf{where} \\
&\quad \mathit{Last} \;:\; \{t \;:\; \mathbb{N}\} \;\to\; (x \;:\; X\;t) \;\to\; \mathit{StateCtrlSeq}\;t\;(S\;Z)
\end{aligned}
$$

$$(:::) \ : \{t, n : \mathbb{N}\} \to \Sigma\,(X\ t)\,(Y\ t) \to StateCtrlSeq\ (S\ t)\ n \to StateCtrlSeq\ t\ (S\ n)$$

and a function $sumR$ that computes the sum of the rewards of a state-control sequence:

$$\begin{aligned}
&head' : \{t, n : \mathbb{N}\} \to StateCtrlSeq\ t\ (S\ n) \to X\ t\\
&head'\ (Last\ x) = x\\
&head'\ (MkSigma\ x\ y ::: xys) = x
\end{aligned}$$

$$\begin{aligned}
&sumR : \{t, n : \mathbb{N}\} \to StateCtrlSeq\ t\ n \to Val\\
&sumR\ \{t\}\ \{n = S\ \ Z\}\ \ \ \ \ (Last\ x) \qquad\qquad\quad = zero\\
&sumR\ \{t\}\ \{n = S\ (S\ m)\}\ (MkSigma\ x\ y ::: xys) = reward\ t\ x\ y\ (head'\ xys) \oplus sumR\ xys
\end{aligned}$$

Next, we implement a function that computes the trajectory that is obtained under a policy sequence ps when starting in x:

$$\begin{aligned}
&trj : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to StateCtrlSeq\ t\ (S\ n)\\
&trj\ \{t\}\ Nil\ x \qquad\ = Last\ x\\
&trj\ \{t\}\ (p :: ps)\ x = \mathbf{let}\ y\ \ = p\ x\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad \mathbf{let}\ x' = next\ t\ x\ y\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad (MkSigma\ x\ y) ::: trj\ ps\ x'
\end{aligned}$$

Finally, we compute the measure of the sum of the rewards obtained along the trajectory that is obtained under the policy sequence ps when starting in x

$$\begin{aligned}
&val' : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to Val\\
&val'\ ps\ x = sumR\ (trj\ ps\ x)
\end{aligned}$$

Now we can formulate the property that $val\ ps$ does indeed compute the measure of the sum of the rewards obtained along the trajectories obtained under the policy sequence ps:

$$valVal'Th : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps\ x = val'\ ps\ x$$

With

$$\begin{aligned}
&head'Lemma : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to head'\ (trj\ ps\ x) = x\\
&head'Lemma\ \ Nil \qquad x = Refl\\
&head'Lemma\ (p :: ps)\ x = Refl
\end{aligned}$$

we can prove the val-val' theorem by induction on ps:

$$\begin{aligned}
&valVal'Th : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps\ x = val'\ ps\ x\\
&valVal'Th \qquad\quad Nil\ x \qquad = Refl\\
&valVal'Th\ \{t\}\ (p :: ps)\ x =\\
&\quad \mathbf{let}\ y = p\ x\ \mathbf{in}\\
&\quad \mathbf{let}\ x' = next\ t\ x\ y\ \mathbf{in}\\
&\qquad (val\ (p :: ps)\ x)
\end{aligned}$$

$$
\begin{aligned}
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val}\ \mathit{ps}\ x') \\
&= \{\, \mathit{cong}\ (\mathit{valVal'Th}\ \mathit{ps}\ x') \,\} = \\
&\quad (\mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val'}\ \mathit{ps}\ x') \\
&= \{\, \mathit{cong}\ \{f = \lambda\alpha \Rightarrow \mathit{reward}\ t\ x\ y\ \alpha \oplus \mathit{val'}\ \mathit{ps}\ x'\}\ (\mathit{sym}\ (\mathit{head'Lemma}\ \mathit{ps}\ x')) \,\} = \\
&\quad (\mathit{reward}\ t\ x\ y\ (\mathit{head'}\ (\mathit{trj}\ \mathit{ps}\ x')) \oplus \mathit{val'}\ \mathit{ps}\ x') \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{sumR}\ ((\mathit{MkSigma}\ x\ y) ::: \mathit{trj}\ \mathit{ps}\ x')) \\
&= \{\, \mathit{Refl} \,\} = \\
&\quad (\mathit{val'}\ (p :: \mathit{ps})\ x) \\
&\mathit{QED}
\end{aligned}
$$

Solutions

Exercise 7.1:

$$
\begin{aligned}
&\mathit{Policy}\ :\ \mathit{Type} \\
&\mathit{Policy} = R\ \to\ S
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PolicySeq}\ :\ \mathbb{N}\ \to\ \mathit{Type} \\
&\mathit{PolicySeq}\ n = \mathit{Vect}\ n\ \mathit{Policy}
\end{aligned}
$$

References

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

[2] Nicola Botta. IdrisLibs. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`, 2016–2018.

[3] Nicola Botta, Patrik Jansson, and Cezar Ionescu. Contributions to a computational theory of policy advice and avoidability. *J. Funct. Program.*, (27, e23), 2017.

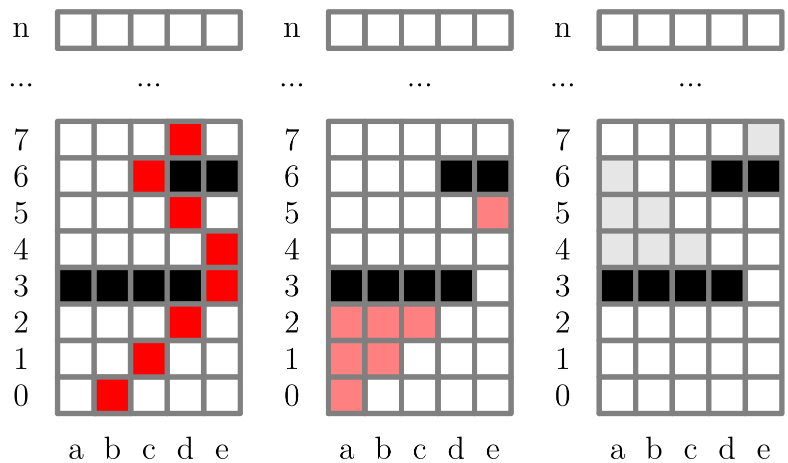
[4] Nicola Botta, Patrik Jansson, Cezar Ionescu, David R. Christiansen, and Edwin Brady. Sequential decision problems, dependent types and generic solutions. *Logical Methods in Computer Science*, 13(1), 2017.

Lecture 8: Viability and reachability

Objectives of this lecture

- Get acquainted with the notions of *viability* and *reachability*
- Use these notions to revisit and improve the notion of policy sequence
- Adapt the notion of optimality and the generic backward induction to the improved theory
- Revisit the problematic example from the last lecture within the new setting

Consider an SDP as in the following sketch



Here, the state space at decision steps 0, 1, 2, 4, 5 and for $t \geqslant 7$ consists of the cells a, b, c, d and e.

But at decision steps 3 and 6, the black cells do not belong to the state space space: $X\ 3 = \{\, e\,\}$ and $X\ 6 = \{\, a, b, c\,\}$.

Further, the controls available to the decision maker only support moving to a adjacent cells.

For example, at decision step 0, the decision maker can move from cell a to cell a or b; from b, she can move to a, b or c. And so on.

The set of controls in cells a, b and c at decision step 2 is empty: $Y\ 2\ a = Y\ 2\ b = Y\ 2\ c = \{\,\}$. The controls in d and e admit transitions to the only cell contained in $X\ 3$, namely e.

Similarly, the set of controls for cell e at decision step 5 is empty and from d one can only move to c.

On the left of the figure you can see a trajectory compatible with these assumptions in red.

In the middle of the figure, the cells from which less than three decision steps can be done are flagged in light red: in particular, no steps can be done starting from cells a, b and c at $t = 2$ and from cell e at $t = 5$. Only one step can be done starting from cells a and b at $t = 1$ and two steps can be done from cell a at $t = 0$.

On the right of the figure, the cells that cannot be reached no matter what the initial cell at step 0 is and which controls are selected are greyed.

8.1 Viability

In lecture 7 we have specified an SDP in terms of

$$
\begin{array}{ll}
X & : (t : \mathbb{N}) \to \mathit{Type} \\
Y & : (t : \mathbb{N}) \to (x : X\ t) \to \mathit{Type} \\
\mathit{next} & : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to X\ (S\ t) \\
\mathit{Val} & : \mathit{Type} \\
\mathit{reward} & : (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to (x' : X\ (S\ t)) \to \mathit{Val} \\
(\oplus) & : \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\
\mathit{zero} & : \mathit{Val} \\
(\leqslant) & : \mathit{Val} \to \mathit{Val} \to \mathit{Type} \\
\mathit{lteRefl} & : \{a : \mathit{Val}\} \to a \leqslant a \\
\mathit{lteTrans} & : \{a, b, c : \mathit{Val}\} \to a \leqslant b \to b \leqslant c \to a \leqslant c \\
\mathit{plusMon} & : \{a, b, c, d : \mathit{Val}\} \to a \leqslant b \to c \leqslant d \to (a \oplus c) \leqslant (b \oplus d)
\end{array}
$$

The picture makes very clear that, if we allow $Y\ t\ x$ to be empty for certain states $x : X\ t$, we need to be careful in the definition of the domain and of the codomain of policy functions.

For instance cell a cannot be in the domain of a policy for taking decisions at step 0 that is the head of a policy sequence of length greater than or equal to three!

Conversely, a policy that supports taking more than 2 decision steps cannot select a control in cell b (c) at step 0 that leads to cell a (b) at step one!

We can account for these constraints in a simple and general way by introducing the notion of *viability*.

Informally, a state $x : X\ t$ is viable for n steps if one can make n decision steps starting from x. We can formalize this idea in terms of a $\mathit{Viable}\ n\ x$ type:

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

We know from lecture 7 that we have to refine the notion of policy. At the same time, we do not want to impose unnecessary restrictions on the range of SDPs to which the theory can be applied.

This suggests that we should probably avoid *defining* *Viable* and instead specify minimal properties that problem-specific implementations need to fulfil.

One property of *Viable* is rather obvious: every state should be viable for zero steps. Thus

$$viableSpec0 \ : \{t \ : \ \mathbb{N}\} \ \rightarrow \ (x \ : \ X \ t) \ \rightarrow \ Viable \ Z \ x$$

We want to formalize the intuition that if one can take $n+1$ decision steps starting from a state x, then x must admit a control that leads to a next state from which at least n further steps can be done. That is, to a next state that is viable for n steps:

$$\begin{aligned} viableSpec1 \ : \ & \{t \ : \ \mathbb{N}\} \ \rightarrow \ \{n \ : \ \mathbb{N}\} \ \rightarrow \ (x \ : \ X \ t) \ \rightarrow \\ & Viable \ (S \ n) \ x \ \rightarrow \ Exists \ (\lambda y \Rightarrow Viable \ n \ (next \ t \ x \ y)) \end{aligned}$$

The third and last requirement that we impose is the converse: if a state admits a control which is good for n steps, that state is viable for $n+1$ steps:

$$\begin{aligned} viableSpec2 \ : \ & \{t \ : \ \mathbb{N}\} \ \rightarrow \ \{n \ : \ \mathbb{N}\} \ \rightarrow \ (x \ : \ X \ t) \ \rightarrow \\ & Exists \ (\lambda y \Rightarrow Viable \ n \ (next \ t \ x \ y)) \ \rightarrow \ Viable \ (S \ n) \ x \end{aligned}$$

Exercise 8.1. Give a generic implementation of *Viable* and prove that it fulfills *ViableSpec0*, *ViableSpec1* and *ViableSpec2*.

Exercise 8.1:

$$\begin{aligned} & Viable \ \{t\} \ \ Z \ \ \ \ \ \ x = Unit \\ & Viable \ \{t\} \ (S \ n) \ x = Exists \ (\lambda y \Rightarrow Viable \ n \ (next \ t \ x \ y)) \end{aligned}$$

$$viableSpec0 \ \{t\} \ x = ()$$

$$viableSpec1 \ \{t\} \ \{n\} \ x \ (Evidence \ y \ gy) = Evidence \ y \ gy$$

$$viableSpec2 \ \{t\} \ \{n\} \ x \ (Evidence \ y \ gy) = Evidence \ y \ gy$$

8.2 Reachability

In the SDP sketched in the figure, the gray cells on the right cannot be reached from any initial cell, no matter which controls are selected.

Including these states in the domain of policies is not logically problematic but potentially very inefficient.

In concrete SDPs, it is not uncommon that a large number of states in $X\ t$ are actually unreachable for large t.

Computing optimal controls for these states would be an unnecessary waste of resources.

We can avoid this by restricting the domain of policies of type $\mathit{Policy}\ t$ to values in $X\ t$ that are actually reachable.

To this end, we need to formalize the notion of reachability. We proceed in the same way as for viability. Instead of *defining* the notion of reachability, we specify it.

Client applications will be able to take advantage of the knowledge about the specific SDP at stake to provide efficient implementations of $\mathit{Reachable}$:

$$\mathit{Reachable} : \{t' : \mathbb{N}\} \to X\ t' \to \mathit{Type}$$

$$\mathit{reachableSpec0} : (x : X\ Z) \to \mathit{Reachable}\ x$$

$$\mathit{reachableSpec1} : \{t : \mathbb{N}\} \to (x : X\ t) \to \mathit{Reachable}\ x \to (y : Y\ t\ x) \to$$

$$\mathit{Reachable}\ (\mathit{next}\ t\ x\ y)$$

The type of $\mathit{reachableSpec0}$ encodes the idea that every initial state is reachable.

The type of $\mathit{reachableSpec1}$ formalizes the idea that if $x : X\ t$ is reachable, every $y : Y\ t\ x$ implies that $\mathit{next}\ z\ x\ y$ is also reachable.

We also want to encode the idea that if $x' : X\ (t+1)$ is reachable, then there must exist a state $x : X\ t$ that is reachable and a control $y : Y\ t\ x$ that allows a transition from x to x'.

Exercise 8.2. Encode this idea in the type of a $\mathit{reachableSpec2}$ value. Suggestion: first, formalize what it means for a state $x : X\ t$ to be a predecessor of a state $x' : X\ (S\ t)$ by implementing

$$\mathit{Pred} : \{t : \mathbb{N}\} \to X\ t \to X\ (S\ t) \to \mathit{Type}$$

Then refine this notion: define what it means for a state $x : X\ t$ to be a reachable predecessor of a state $x' : X\ (S\ t)$ by implementing

$$\mathit{ReachablePred} : \{t : \mathbb{N}\} \to X\ t \to X\ (S\ t) \to \mathit{Type}$$

Finally, give the type of $\mathit{reachableSpec2}$.

Exercise 8.2:

$$\mathit{Pred}\ \{t\}\ x\ x' = \mathit{Exists}\ (\lambda y \Rightarrow x' = \mathit{next}\ t\ x\ y)$$

$$ReachablePred\ x\ x' = (Reachable\ x, x\ `Pred`\ x')$$

$$reachableSpec2 : \{t : \mathbb{N}\} \to (x' : X\ (S\ t)) \to Reachable\ x' \to Exists\ (\lambda x \Rightarrow x\ `ReachablePred`\ x')$$

Exercise 8.3. Give a generic default implementation of $Reachable$ and prove that it fulfills $reachableSpec0$, $reachableSpec1$ and $reachable2$.

Exercise 8.3:

$$Reachable\ \{t' = Z\}\quad x' = Unit$$
$$Reachable\ \{t' = S\ t\}\ x' = Exists\ (\lambda x \Rightarrow ReachablePred\ x\ x')$$

$$reachableSpec0\ x = ()$$

$$reachableSpec1\ x\ rx\ y = Evidence\ x\ (rx, Evidence\ y\ Refl)$$

$$reachableSpec2\ \{t\}\ x'\ rx' = rx'$$

8.3 Policies and policy sequences revisited

We are now ready to refine the notion of policy to avoid the difficulties discussed in lecture 7. Recall that there we defined

$$Policy : (t : \mathbb{N}) \to Type$$
$$Policy\ t = (x : X\ t) \to Y\ t\ x$$

and then realized that a policy which does not support n decision steps cannot be the first element of a policy sequence of length n.

We encode the idea that a policy at decision step t might only support a finite number n of decision steps with an additional parameter:

$$Policy : (t : \mathbb{N}) \to (n : \mathbb{N}) \to Type$$

For $n = Z$ (zero decison steps) we do not actually need any rule and we can define $Policy\ t\ Z$ to be the sigleton type:

$$Policy\ t\ Z = Unit$$

For $n = S\ m$ we want to express the idea that a value of type $Policy\ t\ n$ is a decision rule which associates to each state in $x : X\ t$ that is reachable and viable for n steps a good control in $Y\ t\ x$.

A good control in $Y\ t\ x$ is just a $y : Y\ t\ x$ paired with a proof that y is good:

$$GoodY : (t : \mathbb{N}) \to (x : X\ t) \to (m : \mathbb{N}) \to Type$$
$$GoodY\ t\ x\ m = \Sigma\ (Y\ t\ x)\ (\lambda y \Rightarrow Viable\ m\ (next\ t\ x\ y))$$

With a notion of good controls in place, we can define what a policy that supports $n = S\ m$ decision steps is. Wrapping up:

$$Policy\ t\ \ Z \quad\quad = Unit$$
$$Policy\ t\ (S\ m) = (x : X\ t) \to Reachable\ x \to Viable\ (S\ m)\ x \to GoodY\ t\ x\ m$$

Policy sequences are, as in lecture 7, sequences of policies:

$$\textbf{data}\ PolicySeq : (t : \mathbb{N}) \to (n : \mathbb{N}) \to Type\ \textbf{where}$$
$$\quad Nil : \{t : \mathbb{N}\} \to PolicySeq\ t\ Z$$
$$\quad (::) : \{t, n : \mathbb{N}\} \to Policy\ t\ (S\ n) \to PolicySeq\ (S\ t)\ n \to PolicySeq\ t\ (S\ n)$$

Remark: Notice the $t\ (S\ n)$, $(S\ t)\ n$, $t\ (S\ n)$ pattern in the $Cons$ constructor of policy sequences.

8.4 The value of policy sequences

The computation of val is essentially as in lecture 7

$$val : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to (x : X\ t) \to Val$$
$$val\ \{t\}\ Nil \quad\quad x = zero$$
$$val\ \{t\}\ (p :: ps)\ x = \textbf{let}\ y = p\ x\ \textbf{in}$$
$$\quad\quad\quad\quad \textbf{let}\ x' = next\ t\ x\ y\ \textbf{in}$$
$$\quad\quad\quad\quad reward\ t\ x\ y\ x' \oplus val\ ps\ x'$$

but there is a twist. The policy p can only be applied to states in $X\ t$ that are reachable and viable and computes not just a control but a *good* control!

This is crucial because, in order to compute the value of the tail ps in $x' = next\ t\ x\ y$, we have to provide evidence that x' is reachable and viable!

The definition of val accounts for the fact that we have carefully restricted both the domain and the codomain of policies.

$$val : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to (x : X\ t) \to Reachable\ x \to Viable\ n\ x \to Val$$
$$val\ \{t\}\ Nil \quad\quad x\ rx\ vx = zero$$
$$val\ \{t\}\ (p :: ps)\ x\ rx\ vx = \textbf{let}\ gy\ = p\ x\ rx\ vx\ \textbf{in}$$

$$
\begin{array}{ll}
\textbf{let } y \;\; = \mathit{outl}\ gy \textbf{ in} & \\
\textbf{let } x' \;= \mathit{next}\ t\ x\ y \textbf{ in} & \\
\textbf{let } rx' = \mathit{reachableSpec1}\ x\ rx\ y \textbf{ in} & \text{-- ?hole1} \\
\textbf{let } vx' = \mathit{outr}\ gy \textbf{ in} & \text{-- ?hole2} \\
\mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val}\ ps\ x'\ rx'\ vx' &
\end{array}
$$

In the definition of val, we have used the helper function outl. outl and its counterpart outr are just the projections for existential types: Σ, Exists, etc.

$$
\begin{array}{l}
\mathit{outl} \;:\; \{A \,:\, \mathit{Type}\} \to \{P \,:\, A \to \mathit{Type}\} \to \Sigma\ A\ P \to A \\
\mathit{outl}\ (\mathit{MkSigma}\ a\ \_) = a
\end{array}
$$

$$
\begin{array}{l}
\mathit{outr} \;:\; \{A \,:\, \mathit{Type}\} \to \{P \,:\, A \to \mathit{Type}\} \to (s \,:\, \Sigma\ A\ P) \to P\ (\mathit{outl}\ s) \\
\mathit{outr}\ (\mathit{MkSigma}\ \_\ p) = p
\end{array}
$$

Exercise 8.4. In the definition of val we have two holes left: $\mathit{hole1}$ and $\mathit{hole2}$. Fill in these holes and complete the implementation of val. Suggestion: recall the specification of $\mathit{Reachable}$ and the definition of good controls.

8.5 Optimality, optimal extensions, Bellman's principle

The notions of optimality of policy sequences, of optimal extension of policy sequences and Bellman's principle are, mutatis mutandis, the same as in lecture 7:

$$
\begin{array}{l}
\mathit{OptPolicySeq} \;:\; \{t, n \,:\, \mathbb{N}\} \to \mathit{PolicySeq}\ t\ n \to \mathit{Type} \\
\mathit{OptPolicySeq}\ \{t\}\ \{n\}\ ps = (ps' \,:\, \mathit{PolicySeq}\ t\ n) \to \\
\qquad\qquad\qquad\qquad (x \,:\, X\ t) \to (rx \,:\, \mathit{Reachable}\ x) \to (vx \,:\, \mathit{Viable}\ n\ x) \to \\
\qquad\qquad\qquad\qquad \mathit{val}\ ps'\ x\ rx\ vx \leqslant \mathit{val}\ ps\ x\ rx\ vx
\end{array}
$$

$$
\begin{array}{l}
\mathit{OptExt} \;:\; \{t, m \,:\, \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ m \to \mathit{Policy}\ t\ (S\ m) \to \mathit{Type} \\
\mathit{OptExt}\ \{t\}\ \{m\}\ ps\ p = (p' \,:\, \mathit{Policy}\ t\ (S\ m)) \to \\
\qquad\qquad\qquad\qquad (x \,:\, X\ t) \to (rx \,:\, \mathit{Reachable}\ x) \to (vx \,:\, \mathit{Viable}\ (S\ m)\ x) \to \\
\qquad\qquad\qquad\qquad \mathit{val}\ (p' :: ps)\ x\ rx\ vx \leqslant \mathit{val}\ (p :: ps)\ x\ rx\ vx
\end{array}
$$

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

Proving Bellman's principle carries over from lecture 7:

$$
\begin{aligned}
&\mathit{Bellman}\ \{t\}\ \{m\}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep} = \mathit{opps}\ \textbf{where}\\
&\quad \mathit{opps}\ (p' :: \mathit{ps}')\ x\ \mathit{rx}\ \mathit{vx} =\\
&\qquad \textbf{let}\ \mathit{gy}' = p'\ x\ \mathit{rx}\ \mathit{vx}\ \textbf{in}\\
&\qquad \textbf{let}\ y' = \ \mathit{outl}\ \mathit{gy}'\ \textbf{in}\\
&\qquad \textbf{let}\ x' = \ \mathit{next}\ t\ x\ y'\ \textbf{in}\\
&\qquad \textbf{let}\ \mathit{rx}' = \mathit{reachableSpec1}\ x\ \mathit{rx}\ y'\ \textbf{in}\\
&\qquad \textbf{let}\ \mathit{vx}' = \mathit{outr}\ \mathit{gy}'\ \textbf{in}\\
&\qquad \textbf{let}\ s_1\ = \mathit{plusMon}\ \mathit{lteRefl}\ (\mathit{ops}\ \mathit{ps}'\ x'\ \mathit{rx}'\ \mathit{vx}')\ \textbf{in}\\
&\qquad \textbf{let}\ s_2\ = \mathit{oep}\ p'\ x\ \mathit{rx}\ \mathit{vx}\ \textbf{in}\\
&\qquad \mathit{lteTrans}\ s_1\ s_2
\end{aligned}
$$

8.6 Generic verified backwards induction

Apart from the additional index in the type of policies, this part of the theory is unchanged from lecture 7:

$$
\begin{aligned}
&\mathit{nilOptPolicySeq}\ :\ \mathit{OptPolicySeq}\ \mathit{Nil}\\
&\mathit{nilOptPolicySeq}\ \mathit{Nil}\ x\ \mathit{rx}\ \mathit{vx} = \mathit{lteRefl}
\end{aligned}
$$

$$
\mathit{optExt}\ :\ \{t, n\ :\ \mathbb{N}\}\ \to\ \mathit{PolicySeq}\ (S\ t)\ n\ \to\ \mathit{Policy}\ t\ (S\ n)
$$

$$
\mathit{optExtSpec}\ :\ \{t, n\ :\ \mathbb{N}\}\ \to\ (\mathit{ps}\ :\ \mathit{PolicySeq}\ (S\ t)\ n)\ \to\ \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})
$$

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

$$
\begin{aligned}
&\mathit{biLemma}\ :(t\ :\ \mathbb{N})\ \to\ (n\ :\ \mathbb{N})\ \to\ \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)\\
&\mathit{biLemma}\ t\ \ Z\ \ \ \ \ = \mathit{nilOptPolicySeq}\\
&\mathit{biLemma}\ t\ (S\ n) = \textbf{let}\ \mathit{ps}\ \ \ = \mathit{bi}\ (S\ t)\ n\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \textbf{let}\ \mathit{ops} = \mathit{biLemma}\ (S\ t)\ n\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \textbf{let}\ p\ \ \ \ = \mathit{optExt}\ \mathit{ps}\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \textbf{let}\ \mathit{oep} = \mathit{optExtSpec}\ \mathit{ps}\ \textbf{in}\\
&\qquad\qquad\qquad\quad\ \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}
\end{aligned}
$$

8.7 What have we gained?

Let's consider again the example from lecture 7:

$$
\begin{aligned}
&head \;:\; \{t, n \;:\; \mathbb{N}\} \;\to\; PolicySeq\ t\ (S\ n) \;\to\; Policy\ t\ (S\ n)\\
&head\ (p :: ps) = p\\[1em]
&tail \;:\; \{t, n \;:\; \mathbb{N}\} \;\to\; PolicySeq\ t\ (S\ n) \;\to\; PolicySeq\ (S\ t)\ n\\
&tail\ (p :: ps) = ps\\[1em]
&\mathbf{data}\ GoodOrBad = Good \mid Bad\\[1em]
&implementation\ Show\ GoodOrBad\ \mathbf{where}\\
&\quad show\ Good = \texttt{"Good"}\\
&\quad show\ Bad\ \ = \texttt{"Bad"}\\[1em]
&\mathbf{data}\ UpOrDown = Up \mid Down\\[1em]
&X\ \ t = GoodOrBad\\[1em]
&Y\ \ t\ Good = UpOrDown\\
&Y\ \ t\ Bad\ \ \ = Void\\[1em]
&next\ t\ Good\ Up \qquad = \quad Good\\
&next\ t\ Good\ Down \quad = \quad Bad\\
&next\ t\ Bad\ \ \ v \qquad impossible\\[1em]
&Val = \mathbb{N}\\
&(\oplus) = (+)\\
&zero = Z\\
&(\leqslant) = Prelude.\mathbb{N}.LTE\\
&reward\ t\ Good\ Up\ x' \qquad = \quad 1\\
&reward\ t\ Good\ Down\ x' \quad = \quad 3\\
&reward\ t\ Bad\ v\ x' \qquad impossible
\end{aligned}
$$

Notice that, as in lecture 7, we will not be able to compute an argument $v \;:\; Void$ to apply $reward$.

Remember that policies are now parameterized on two natural numbers: t and n. The second one characterizes how many decision steps the policy does support.

In implementing $optExt$, we have to distinguish between policy extensions of policy sequences of length zero and policy extensions of sequences of length greater or equal to one.

In the first case we can select both *Up* and *Down* because, even though the second control implies a transition to *Bad* (remember that the control set of *Bad* is void!), no further decision steps are required from there:

$$
\begin{aligned}
& optExt\ \{t\}\ \{n = Z\}\ ps\ Good\ rGood\ vGood = \\
& \quad \textbf{let}\ x1' \qquad = next\ t\ Good\ Up\ \textbf{in} \\
& \quad \textbf{let}\ rx1' \qquad = reachableSpec1\ \ Good\ rGood\ Up\ \textbf{in} \\
& \quad \textbf{let}\ vx1' \qquad = viableSpec0\ \{t = S\ t\}\ x1'\ \textbf{in} \\
& \quad \textbf{let}\ x2' \qquad = next\ t\ Good\ Down\ \textbf{in} \\
& \quad \textbf{let}\ rx2' \qquad = reachableSpec1\ \ Good\ rGood\ Down\ \textbf{in} \\
& \quad \textbf{let}\ vx2' \qquad = viableSpec0\ \{t = S\ t\}\ x2'\ \textbf{in} \\
& \quad \textbf{let}\ valUp \quad\ = reward\ t\ Good\ Up \quad\ \ (next\ t\ Good\ Up \quad\ ) \oplus val\ ps\ x1'\ rx1'\ vx1'\ \textbf{in} \\
& \quad \textbf{let}\ valDown = reward\ t\ Good\ Down\ (next\ t\ Good\ Down) \oplus val\ ps\ x2'\ rx2'\ vx2'\ \textbf{in} \\
& \quad \textbf{if}\ valUp \geqslant valDown\ \textbf{then}\ (MkSigma\ Up\ ())\ \textbf{else}\ (MkSigma\ Down\ ()) \\
& optExt\ \{t\}\ \{n = Z\}\ ps\ Bad\ rBad\ (Evidence\ v\ \_) = absurd\ v
\end{aligned}
$$

Exercise 8.5. Notice that, in contrast to lecture 7, we can now give a complete implementation of the (absurd) *Bad* case. Do you see why v is absurd?

In the second case, we *know* that only one control supports a transition to a next state from which further decision steps are doable. Thus, we just pick up this control:

$$
\begin{aligned}
& optExt\ \{t\}\ \{n = S\ m\}\ ps\ Good\ rGood\ vGood = \\
& \quad \textbf{let}\ ey = viableSpec1\ \{t = t\}\ \{n = S\ m\}\ Good\ vGood\ \textbf{in} \\
& \quad MkSigma\ (getWitness\ ey)\ (getProof\ \ ey) \\
& optExt\ \{t\}\ \{n = S\ m\}\ ps\ Bad\ rBad\ (Evidence\ v\ \_) = absurd\ v
\end{aligned}
$$

We can now implement the same computation as in lecture 7. In contrast to lecture 7, however we have to provide (compute, construct) evidences that our initial state $x_0 = Good$ is reachable and viable for two steps!

$$
computation\ :\ IO\ ()
$$

$$
\begin{aligned}
computation = \ & \textbf{let}\ ps \quad = bi\ 0\ 2\ \textbf{in} \\
& \textbf{let}\ x_0 \quad = Good\ \textbf{in} \\
& \textbf{let}\ rx0 = ()\ \textbf{in} \\
& \textbf{let}\ vx0 = Evidence\ Up\ (Evidence\ Up\ ())\ \textbf{in} \\
& \textbf{let}\ p_0 \quad = head\ ps\ \textbf{in} \\
& \textbf{let}\ gy0 = p_0\ x_0\ rx0\ vx0\ \textbf{in} \\
& \textbf{let}\ y0 \quad = outl\ gy0\ \textbf{in}
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{let}\ x_1 \ \ = \mathit{next}\ Z\ x_0\ \mathit{y0}\ \textbf{in}\\
\textbf{let}\ \mathit{rx1} = \mathit{reachableSpec1}\ \{t = 0\}\ x_0\ \mathit{rx0}\ \mathit{y0}\ \textbf{in}\\
\textbf{let}\ \mathit{vx1} = \mathit{outr}\ \mathit{gy0}\ \textbf{in}\\
\textbf{let}\ p_1 \ \ = \mathit{head}\ (\mathit{tail}\ \mathit{ps})\ \textbf{in}\\
\textbf{let}\ \mathit{gy1} = p_1\ x_1\ \mathit{rx1}\ \mathit{vx1}\ \textbf{in}\\
\textbf{let}\ y_1 \ \ = \mathit{outl}\ \mathit{gy1}\ \textbf{in}\\
\textbf{let}\ x_2 \ \ = \mathit{next}\ 1\ x_1\ y_1\ \textbf{in}\\
\textbf{do}\ \mathit{putStrLn}\ (\texttt{"x0 = "} +\!\!+\ \mathit{show}\ x_0)\\
\quad\ \ \mathit{putStrLn}\ (\texttt{"x1 = "} +\!\!+\ \mathit{show}\ x_1)\\
\quad\ \ \mathit{putStrLn}\ (\texttt{"x2 = "} +\!\!+\ \mathit{show}\ x_2)
\end{array}
$$

Exercise 8.6. Comment the single steps of the implementation of *computation*.

$$
\begin{array}{l}
\mathit{main}\ :\ \mathit{IO}\ ()\\
\mathit{main} = \mathit{computation}
\end{array}
$$

Exercise 8.7. Do you expect this program to terminate? If so, what do you expect to be the result of the computation? Run *main* from the terminal. Do you obtain the expected result?

8.8 Wrap-up, outlook

- We have managed to fix a deficiency of the naive theory from lecture 7.
- The new theory requires stronger guarantees from the initial states.
- Attempts at computing n optimal decisions starting from states that support less than n decision steps are detected at compile time.
- We still face a major problem, however: fulfilling *optExtSpec* for problem-specific implementations of *optExt* like the ones discussed here and in lecture 7 is difficult and time consuming!
- What we need is a *generic* implementation of *optExt*.
- And, in order to apply the theory to decision problems under uncertainty and imperfect information, we need to extend it to *monadic* SDPs.

Solutions

Exercise 8.4:

$$rx' = reachableSpec1\ x\ rx\ y$$

$$vx' = outr\ gy$$

Lecture 9: Generic optimal extensions, viability and reachability tests

9.1 Wrap up lecture 8, part 1

In lecture 8 we have demonstrated that if we specify a SDP in terms of

$$\begin{aligned}
&X &&: (t : \mathbb{N}) \to \mathit{Type} \\
&Y &&: (t : \mathbb{N}) \to (x : X\ t) \to \mathit{Type} \\
&\mathit{next} &&: (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to X\ (S\ t) \\
&\mathit{Val} &&: \mathit{Type} \\
&\mathit{reward} &&: (t : \mathbb{N}) \to (x : X\ t) \to (y : Y\ t\ x) \to (x' : X\ (S\ t)) \to \mathit{Val} \\
&(\oplus) &&: \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\
&\mathit{zero} &&: \mathit{Val} \\
&(\leqslant) &&: \mathit{Val} \to \mathit{Val} \to \mathit{Type}
\end{aligned}$$

such that

$$\begin{aligned}
&\mathit{lteRefl} &&: \{a : \mathit{Val}\} \to a \leqslant a \\
&\mathit{lteTrans} &&: \{a, b, c : \mathit{Val}\} \to a \leqslant b \to b \leqslant c \to a \leqslant c \\
&\mathit{plusMon} &&: \{a, b, c, d : \mathit{Val}\} \to a \leqslant b \to c \leqslant d \to (a \oplus c) \leqslant (b \oplus d)
\end{aligned}$$

and if we can define

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

$$\mathit{Reachable} : \{t' : \mathbb{N}\} \to X\ t' \to \mathit{Type}$$

such that

$$\mathit{viableSpec0} : \{t : \mathbb{N}\} \to (x : X\ t) \to \mathit{Viable}\ Z\ x$$

$$\begin{aligned}
\mathit{viableSpec1} : \{t : \mathbb{N}\} \to \{n : \mathbb{N}\} \to (x : X\ t) \to \\
\mathit{Viable}\ (S\ n)\ x \to \mathit{Exists}\ (\lambda y \Rightarrow \mathit{Viable}\ n\ (\mathit{next}\ t\ x\ y))
\end{aligned}$$

$$\begin{aligned}
\mathit{viableSpec2} : \{t : \mathbb{N}\} \to \{n : \mathbb{N}\} \to (x : X\ t) \to \\
\mathit{Exists}\ (\lambda y \Rightarrow \mathit{Viable}\ n\ (\mathit{next}\ t\ x\ y)) \to \mathit{Viable}\ (S\ n)\ x
\end{aligned}$$

and

$$\mathit{reachableSpec0} : (x : X\ Z) \to \mathit{Reachable}\ x$$

$$\mathit{reachableSpec1} : \{t : \mathbb{N}\} \to (x : X\ t) \to \mathit{Reachable}\ x \to (y : Y\ t\ x) \to \mathit{Reachable}\ (\mathit{next}\ t\ x\ y)$$

$$\begin{aligned}
&\mathit{Pred} : \{t : \mathbb{N}\} \to X\ t \to X\ (S\ t) \to \mathit{Type} \\
&\mathit{Pred}\ \{t\}\ x\ x' = \mathit{Exists}\ (\lambda y \Rightarrow x' = \mathit{next}\ t\ x\ y)
\end{aligned}$$

$$\mathit{ReachablePred} : \{t : \mathbb{N}\} \to X\ t \to X\ (S\ t) \to \mathit{Type}$$
$$\mathit{ReachablePred}\ x\ x' = (\mathit{Reachable}\ x, x\ \text{`}\mathit{Pred}\text{`}\ x')$$

$$\mathit{reachableSpec2} : \{t : \mathbb{N}\} \to (x' : X\ (S\ t)) \to \mathit{Reachable}\ x' \to \mathit{Exists}\ (\lambda x \Rightarrow x\ \text{`}\mathit{ReachablePred}\text{`}\ x')$$

hold, then, if we can implement a function that computes optimal extensions of arbitrary policy sequences

$$\mathit{optExt} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{Policy}\ t\ (S\ n)$$

$$\mathit{optExtSpec} : \{t, n : \mathbb{N}\} \to (ps : \mathit{PolicySeq}\ (S\ t)\ n) \to \mathit{OptExt}\ ps\ (\mathit{optExt}\ ps)$$

we can implement a generic backwards induction

$$\mathit{bi} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{PolicySeq}\ t\ n$$

that is correct by construction, i.e. for which we can prove

$$\mathit{biLemma} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n)$$

We have derived this result in a context. This is given by the notions:

$$\mathit{GoodY} : (t : \mathbb{N}) \to (x : X\ t) \to (m : \mathbb{N}) \to \mathit{Type}$$
$$\mathit{GoodY}\ t\ x\ m = \Sigma\ (Y\ t\ x)\ (\lambda y \Rightarrow \mathit{Viable}\ m\ (\mathit{next}\ t\ x\ y))$$

$$\begin{aligned}
&\mathit{Policy} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{Type}\\
&\mathit{Policy}\ t\ Z \quad\ \ = \mathit{Unit}\\
&\mathit{Policy}\ t\ (S\ m) = (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ (S\ m)\ x \to \mathit{GoodY}\ t\ x\ m
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ \mathit{PolicySeq} : (t : \mathbb{N}) \to (n : \mathbb{N}) \to \mathit{Type}\ \textbf{where}\\
&\quad \mathit{Nil} : \{t : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ Z\\
&\quad (::) : \{t, n : \mathbb{N}\} \to \mathit{Policy}\ t\ (S\ n) \to \mathit{PolicySeq}\ (S\ t)\ n \to \mathit{PolicySeq}\ t\ (S\ n)
\end{aligned}$$

$$\mathit{val} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ n \to (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ n\ x \to \mathit{Val}$$

$$\begin{aligned}
&\mathit{val}\ \{t\}\ \mathit{Nil} \qquad\ x\ rx\ vx = \mathit{zero}\\
&\mathit{val}\ \{t\}\ (p :: ps)\ x\ rx\ vx = \textbf{let}\ gy \ \ = p\ x\ rx\ vx\ \textbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{let}\ y \ \ \ = \mathit{outl}\ gy\ \textbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{let}\ x' \ \ = \mathit{next}\ t\ x\ y\ \textbf{in}\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{let}\ rx' = \mathit{reachableSpec1}\ x\ rx\ y\ \textbf{in} \quad \text{-- ?hole1}\\
&\qquad\qquad\qquad\qquad\qquad\ \ \textbf{let}\ vx' = \mathit{outr}\ gy\ \textbf{in} \qquad\qquad\quad\ \text{-- ?hole2}\\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val}\ ps\ x'\ rx'\ vx'
\end{aligned}$$

$$
\begin{aligned}
&\mathit{OptPolicySeq} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ n \to \mathit{Type} \\
&\mathit{OptPolicySeq}\ \{t\}\ \{n\}\ ps = (ps' : \mathit{PolicySeq}\ t\ n) \to \\
&\qquad (x : X\ t) \to (rx : \mathit{Reachable}\ x) \to (vx : \mathit{Viable}\ n\ x) \to \\
&\qquad \mathit{val}\ ps'\ x\ rx\ vx \leqslant \mathit{val}\ ps\ x\ rx\ vx
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{OptExt} : \{t, m : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ m \to \mathit{Policy}\ t\ (S\ m) \to \mathit{Type} \\
&\mathit{OptExt}\ \{t\}\ \{m\}\ ps\ p = (p' : \mathit{Policy}\ t\ (S\ m)) \to \\
&\qquad (x : X\ t) \to (rx : \mathit{Reachable}\ x) \to (vx : \mathit{Viable}\ (S\ m)\ x) \to \\
&\qquad \mathit{val}\ (p' :: ps)\ x\ rx\ vx \leqslant \mathit{val}\ (p :: ps)\ x\ rx\ vx
\end{aligned}
$$

9.2 Wrap up lecture 8, part 2

In lecture 8, we have also shown that it is not difficult to give generic and correct implementations of *Viable* and *Reachable*:

$$
\begin{aligned}
&\mathit{Viable}\ \{t\}\ Z\ \quad x = \mathit{Unit} \\
&\mathit{Viable}\ \{t\}\ (S\ n)\ x = \mathit{Exists}\ (\lambda y \Rightarrow \mathit{Viable}\ n\ (\mathit{next}\ t\ x\ y))
\end{aligned}
$$

$$\mathit{viableSpec0}\ \{t\}\ x = ()$$

$$\mathit{viableSpec1}\ \{t\}\ \{n\}\ x\ (\mathit{Evidence}\ y\ gy) = \mathit{Evidence}\ y\ gy$$

$$\mathit{viableSpec2}\ \{t\}\ \{n\}\ x\ (\mathit{Evidence}\ y\ gy) = \mathit{Evidence}\ y\ gy$$

$$
\begin{aligned}
&\mathit{Reachable}\ \{t' = Z\}\ \quad x' = \mathit{Unit} \\
&\mathit{Reachable}\ \{t' = S\ t\}\ x' = \mathit{Exists}\ (\lambda x \Rightarrow \mathit{ReachablePred}\ x\ x')
\end{aligned}
$$

$$\mathit{reachableSpec0}\ x = ()$$

$$\mathit{reachableSpec1}\ x\ rx\ y = \mathit{Evidence}\ x\ (rx, \mathit{Evidence}\ y\ \mathit{Refl})$$

$$\mathit{reachableSpec2}\ \{t\}\ x'\ rx' = rx'$$

However, we have implemented *optExt* only for a specific (and quite simple) SDP and we have argued that showing that this implementation is correct (by implementing *optExtSpec*) would not be trivial.

The first objective of this lecture is to derive a generic, correct implementation of *optExt*.

To this end, we start by reminding ourselves of what it means to compute optimal extensions of policy sequences.

9.3 Generic, correct optimal extensions

Consider again the specification

$$optExt : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t\ (S\ n)$$

$$optExtSpec : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to OptExt\ ps\ (optExt\ ps)$$

The signature of $optExt$ tells us that, for arbitrary $t, n : \mathbb{N}$ and $ps : PolicySeq\ (S\ t)\ n$, $p = optExt\ ps$ has to be a policy for selecting controls at decision step t and that the controls selected by p have to support n further decision steps. Thus, because of the definition of $Policy$

$$\begin{aligned}
&Policy : (t : \mathbb{N}) \to (n : \mathbb{N}) \to Type\\
&Policy\ t\ \ Z \quad\ \ = Unit\\
&Policy\ t\ (S\ m) = (x : X\ t) \to Reachable\ x \to Viable\ (S\ m)\ x \to GoodY\ t\ x\ m
\end{aligned}$$

we also know that p has to associate a *good* control to every state $x : X\ t$ which is reachable and viable for $n+1$ steps. This is a control $y : Y\ t\ x$ paired with a proof that $next\ t\ x\ y$ is viable n steps:

$$\begin{aligned}
&GoodY : (t : \mathbb{N}) \to (x : X\ t) \to (m : \mathbb{N}) \to Type\\
&GoodY\ t\ x\ m = \Sigma\ (Y\ t\ x)\ (\lambda y \Rightarrow Viable\ m\ (next\ t\ x\ y))
\end{aligned}$$

Exercise 9.1. We know for sure that at least one such control exists. Do you see why?

Thus, if $Y\ t\ x$ happens to contain only one control ($Y\ t\ x$ is a singleton type), we can define

$$p\ x\ r\ v = gy$$

where, neglecting the differences between $Exists$ and Σ, gy is just $viableSpec1\ x\ v$! What if $Y\ t\ x$ contains more than one control?

In this case, we truly have to make a choice. The second part of the specification of $optExt$

$$optExtSpec : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to OptExt\ ps\ (optExt\ ps)$$

tells us that $p = optExt\ ps$ has to be an optimal extension of ps. The definitions of $OptExt$

$$\begin{aligned}
&OptExt : \{t, m : \mathbb{N}\} \to PolicySeq\ (S\ t)\ m \to Policy\ t\ (S\ m) \to Type\\
&OptExt\ \{t\}\ \{m\}\ ps\ p = (p' : Policy\ t\ (S\ m)) \to\\
&\qquad (x : X\ t) \to (rx : Reachable\ x) \to (vx : Viable\ (S\ m)\ x) \to\\
&\qquad val\ (p' :: ps)\ x\ rx\ vx \leqslant val\ (p :: ps)\ x\ rx\ vx
\end{aligned}$$

and of val

$$
\begin{aligned}
& \mathit{val} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ n \to (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ n\ x \to \mathit{Val} \\
& \mathit{val}\ \{t\}\ \mathit{Nil}\qquad x\ rx\ vx = \mathit{zero} \\
& \mathit{val}\ \{t\}\ (p :: ps)\ x\ rx\ vx = \mathbf{let}\ gy\ = p\ x\ rx\ vx\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ y\ = \mathit{outl}\ gy\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ x'\ = \mathit{next}\ t\ x\ y\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ rx' = \mathit{reachableSpec1}\ x\ rx\ y\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ vx' = \mathit{outr}\ gy\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val}\ ps\ x'\ rx'\ vx'
\end{aligned}
$$

suggest that $p\ x\ r\ v$ has to be *a* pair consisting of a control $y : Y\ t\ x$ and of a proof that $x' = \mathit{next}\ t\ x\ y$ is viable m steps such that

Condition: y maximises the sum of the current reward, $\mathit{reward}\ t\ x\ y\ x'$, and of the value $\mathit{val}\ ps\ x'\ rx'\ vx'$ of taking m further decision steps with ps.

We can see that this intuition is correct in three steps. First, we rewrite val in terms of a helper function cval:

$$
\begin{aligned}
& \mathit{mutual} \\
& \quad \mathit{cval} : \{t, m : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ m \to \\
& \qquad\quad (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ (S\ m)\ x \to \mathit{GoodY}\ t\ x\ m \to \mathit{Val} \\
& \quad \mathit{cval}\ \{t\}\ ps\ x\ rx\ vx\ gy = \mathbf{let}\ y\ = \mathit{outl}\ gy\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ x'\ = \mathit{next}\ t\ x\ y\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ rx' = \mathit{reachableSpec1}\ x\ rx\ y\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ vx' = \mathit{outr}\ gy\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathit{reward}\ t\ x\ y\ x' \oplus \mathit{val}\ ps\ x'\ rx'\ vx' \\
& \quad \mathit{val} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ t\ n \to (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ n\ x \to \mathit{Val} \\
& \quad \mathit{val}\ \{t\}\ \mathit{Nil}\qquad x\ rx\ vx = \mathit{zero} \\
& \quad \mathit{val}\ \{t\}\ (p :: ps)\ x\ rx\ vx = \mathbf{let}\ gy = p\ x\ rx\ vx\ \mathbf{in} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \mathit{cval}\ ps\ x\ rx\ vx\ gy
\end{aligned}
$$

The interpretation of $\mathit{cval}\ ps\ x\ rx\ vx\ gy$ is clear: it is the value (as always, in terms of sum of rewards) of selecting the (good) control gy at decision step t and then making m further decision steps according to the policy sequence ps.

Second, we assume that we can implement functions

$$
\begin{aligned}
& \mathit{cvalargmax} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \\
& \qquad\qquad (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ (S\ n)\ x \to \mathit{GoodY}\ t\ x\ n
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{cvalmax} : \{t, n : \mathbb{N}\} \to \mathit{PolicySeq}\ (S\ t)\ n \to \\
& \qquad\quad (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ (S\ n)\ x \to \mathit{Val}
\end{aligned}
$$

that fulfills the specification

$$\begin{aligned}
&cvalargmaxSpec : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to \\
&\qquad (x : X\ t) \to (rx : Reachable\ x) \to (vx : Viable\ (S\ n)\ x) \to \\
&\qquad cvalmax\ ps\ x\ rx\ vx = cval\ ps\ x\ rx\ vx\ (cvalargmax\ ps\ x\ rx\ vx)
\end{aligned}$$

$$\begin{aligned}
&cvalmaxSpec : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to \\
&\qquad (x : X\ t) \to (rx : Reachable\ x) \to (vx : Viable\ (S\ n)\ x) \to \\
&\qquad (y : GoodY\ t\ x\ n) \to (cval\ ps\ x\ rx\ vx\ y) \leqslant (cvalmax\ ps\ x\ rx\ vx)
\end{aligned}$$

In other words, $cvalargmax$ computes *a* good control that maximizes $cval$. The first specification ensures that $cvalmax$ is indeed the value of $cval$ for that control. The second specification ensures that no value of $cval$ is better than $cvalmax$.

Third, we show that under these assumption we can derive a correct, generic implementation of $optExt$ that is, an imlementation that fulfils

$$optExt : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t\ (S\ n)$$

$$optExtSpec : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to OptExt\ ps\ (optExt\ ps)$$

with

$$\begin{aligned}
&OptExt : \{t, m : \mathbb{N}\} \to PolicySeq\ (S\ t)\ m \to Policy\ t\ (S\ m) \to Type \\
&OptExt\ \{t\}\ \{m\}\ ps\ p = (p' : Policy\ t\ (S\ m)) \to \\
&\qquad (x : X\ t) \to (rx : Reachable\ x) \to (vx : Viable\ (S\ m)\ x) \to \\
&\qquad val\ (p' :: ps)\ x\ rx\ vx \leqslant val\ (p :: ps)\ x\ rx\ vx
\end{aligned}$$

This is done as follows:

$$\begin{aligned}
&optExt : \{t, n : \mathbb{N}\} \to PolicySeq\ (S\ t)\ n \to Policy\ t\ (S\ n) \\
&optExt\ \{t\}\ \{n\}\ ps = p\ \mathbf{where} \\
&\quad p \qquad\quad : Policy\ t\ (S\ n) \\
&\quad p\ x\ rx\ vx = cvalargmax\ ps\ x\ rx\ vx
\end{aligned}$$

$$\begin{aligned}
&optExtLemma : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ (S\ t)\ n) \to OptExt\ ps\ (optExt\ ps) \\
&optExtLemma\ \{t\}\ \{n\}\ ps\ p'\ x\ rx\ vx = s_4\ \mathbf{where} \\
&\quad p \quad : Policy\ t\ (S\ n) \\
&\quad p \quad = optExt\ ps \\
&\quad gy \ \ : GoodY\ t\ x\ n \\
&\quad gy \ \ = p\ x\ rx\ vx \\
&\quad y \quad : Y\ t\ x \\
&\quad y \quad = outl\ gy \\
&\quad gy' : GoodY\ t\ x\ n \\
&\quad gy' = p'\ x\ rx\ vx
\end{aligned}$$

$$
\begin{aligned}
&y' \;\; : Y\ t\ x \\
&y' \;\; = outl\ gy' \\
&s_1 \;\; : cval\ ps\ x\ rx\ vx\ gy' \leqslant cvalmax\ ps\ x\ rx\ vx \\
&s_1 \;\; = cvalmaxSpec\ ps\ x\ rx\ vx\ gy' \\
&s_2 \;\; : cval\ ps\ x\ rx\ vx\ gy' \leqslant cval\ ps\ x\ rx\ vx\ (cvalargmax\ ps\ x\ rx\ vx) \\
&s_2 \;\; = replace\ \{P = \lambda z \Rightarrow (cval\ ps\ x\ rx\ vx\ gy' \leqslant z)\}\ (cvalargmaxSpec\ ps\ x\ rx\ vx)\ s_1 \\
&\quad \text{-- the next steps are for the (sort of) human reader} \\
&s_3 \;\; : cval\ ps\ x\ rx\ vx\ gy' \leqslant cval\ ps\ x\ rx\ vx\ gy \\
&s_3 \;\; = s_2 \\
&s_4 \;\; : val\ (p' :: ps)\ x\ rx\ vx \leqslant val\ (p :: ps)\ x\ rx\ vx \\
&s_4 \;\; = s_3
\end{aligned}
$$

This completes the derivation of a generic, correct implementation of *optExt* but raises one important question.

Question: Can we compute good controls that maximize *cval* that is, provide correct implementations of *cvalargmax* and *cvalmax*? Under which conditions?

Exercise 9.2. Answer the above questions. What does it mean for *cvalargmax* and *cvalmax* to be correct? Could one establish the correctness of a given implementation by means of tests?

There is no provably correct generic method for solving arbitrary optimization problems.

But it is easy to find a best (good) control for a given (reachable and viable) $x : X\ t$ when set of controls $Y\ t\ x$ is *finite*.

The case in which a decision maker has to select one of a finite set of options is particularly important in practice.

Formalizing the notion of finiteness for a type and deriving correct implementations of *cvalargmax* and *cvalmax* for the finite case would go beyond the scope of these lectures.

But IdrisLibs [1] provides default implementations for this important case. To take advantage of these implementations, practitioners only need to provide a proof that $Y\ t\ x$ is finite for every $x : X\ t$ at every decision step t.

9.4 Viability and reachability decision procedures

In the example discussed at the end of lecture 8, we have computed two decision steps for a simple SDP. The computation entailed, among others, steps like

$$
\begin{aligned}
computation = &\ \mathbf{let}\ ps \;\; = bi\ 0\ 2\ \mathbf{in} \\
&\ \mathbf{let}\ x_0 \;\; = Good\ \mathbf{in} \\
&\ \mathbf{let}\ rx0 = ()\ \mathbf{in}
\end{aligned}
$$

$$\begin{aligned}
&\textbf{let } vx0 = Evidence\ Up\ (Evidence\ Up\ ())\ \textbf{in} \\
&\ldots \\
&\textbf{do } putStrLn\ (\texttt{"x0 = "} +\!\!+\ show\ x_0) \\
&\quad\ \ putStrLn\ (\texttt{"x1 = "} +\!\!+\ show\ x_1) \\
&\quad\ \ putStrLn\ (\texttt{"x2 = "} +\!\!+\ show\ x_2)
\end{aligned}$$

In these steps, we have taken advantage of our understanding of the specific problem (and of the fact that the problem is very simple) to compute an evidence $vx0$ that the initial state $x_0 = Good$ is viable for two decision steps.

In the computation, a proof that x_0 is viable for two decision steps is mandatory to apply the first policy of ps to x_0 and thus compute an optimal control for the first decision step.

In general and for more realistic problems, proving the viability of an initial state for a sufficiently large number of decision steps might be difficult or simply impossible.

In the specific example, we would not have been able to construct any evidence of viability for more than zero decision steps if we had chosen $x_0 = Bad$.

In realistic applications (for instance, in a tool that supports the computation of optimal policies during negotiations on matter of GHG emissions) a decision maker might want to compute optimal policies for different initial states.

These observations suggest that, in order to apply the theory to realistic SDPs, it would be useful to have decision procedures for $Viable$ and $Reachable$.

Question: What is a decision procedure?

A decision procedure for a property $P : A \to Type$ of values of type A is a function that associates to every $a : A$ either a value of type $P\ a$ or a value of type $Not\ (P\ a)$.

A property $P : A \to Type$ which has a decision procedure is called *decidable*. The Idris prelude defines a data type

$$\begin{aligned}
&\textbf{data } Dec : Type \to Type\ \textbf{where} \\
&\quad Yes : (prf : prop) \to Dec\ prop \\
&\quad No\ \ : (contra : prop \to Void) \to Dec\ prop
\end{aligned}$$

to characterize decidable properties. If $P : Type$ is decidable, we can easily implement a decision procedure for any value of type P:

$$\begin{aligned}
&dec : \{P : Type\} \to Dec\ P \to Either\ P\ (Not\ P) \\
&dec\ (Yes\ prf) = Left\ prf \\
&dec\ (No\ contra) = Right\ contra
\end{aligned}$$

We can use Dec to formalize the notion of decidability for viability:

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

and take advantage of $decidableViable$ to implement a viability test inside $computation$:

$$computation =$$

$$
\begin{array}{l}
\mathbf{do}\ x_0 \leftarrow \mathit{pure}\ \mathit{Good} \\
\quad \mathit{putStrLn}\ (\texttt{"x0 = "} + \mathit{show}\ x_0) \\
\quad \mathbf{case}\ (\mathit{decidableViable}\ \{t = Z\}\ 2\ x_0)\ \mathbf{of} \\
\qquad (\mathit{Yes}\ \mathit{prf}) \quad \Rightarrow \mathbf{let}\ \mathit{ps} \quad = \mathit{bi}\ 0\ 2\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ \mathit{rx0} = ()\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ \mathit{vx0} = \mathit{prf}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad \mathbf{do}\ \mathit{putStrLn}\ (\texttt{"x0 = "} + \mathit{show}\ x_0) \\
\qquad\qquad\qquad\qquad\quad \mathit{putStrLn}\ (\texttt{"x1 = "} + \mathit{show}\ x_1) \\
\qquad\qquad\qquad\qquad\quad \mathit{putStrLn}\ (\texttt{"x2 = "} + \mathit{show}\ x_2) \\
\qquad (\mathit{No}\ \mathit{contra}) \Rightarrow \mathbf{do}\ \mathit{putStrLn}\ (\texttt{"x0 not viable for 2 decision steps"})
\end{array}
$$

A viability test is a necessary condition for computing optimal policy sequences of a length n for initial states that are selected at run time and that may or may not be actually viable for n steps.

9.5 Wrap-up, outlook

- We have obtained a *generic* implementation of optExt.
- We have seen how to take advantage of viability decision procedures for run-time tests.
- In the next lecture we will extend the theory to *monadic* SDPs.

Solutions

Exercise 9.1:

It is because of

$$
\begin{array}{l}
\mathit{viableSpec1} : \{t : \mathbb{N}\} \to \{n : \mathbb{N}\} \to (x : X\ t) \to \\
\qquad\qquad\quad \mathit{Viable}\ (S\ n)\ x \to \mathit{Exists}\ (\lambda y \Rightarrow \mathit{Viable}\ n\ (\mathit{next}\ t\ x\ y))
\end{array}
$$

Since

$$
\mathit{Policy}\ t\ (S\ n) = (x : X\ t) \to \mathit{Reachable}\ x \to \mathit{Viable}\ (S\ n)\ x \to \mathit{GoodY}\ t\ x\ n
$$

we know that x is viable $S\ n$ steps (we have a value of type $\mathit{Viable}\ (S\ n)\ x$) every time we have to compute a good control for that x.

References

[1] Nicola Botta. IdrisLibs. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`, 2016–2018.

Lecture 10: Extending the theory to monadic SDPs

In this lecture we extend the naive theory of deterministic sequential decision problems of lecture 7 to *monadic* SDPs. As a first step, we account for a problem's uncertainties. As we have seen in lecture 6, this can be done in terms of a type constructor M which has the structure of a *monad*:

$$M \ : \ Type \ \to \ Type$$

10.1 States, controls, transition and reward function

We formalize the notions of state space, control space, transition function and reward function as usual

$$\begin{array}{ll}
X & : (t \ : \ \mathbb{N}) \ \to \ Type \\
Y & : (t \ : \ \mathbb{N}) \ \to \ (x \ : \ X \ t) \ \to \ Type \\
next & : (t \ : \ \mathbb{N}) \ \to \ (x \ : \ X \ t) \ \to \ (y \ : \ Y \ t \ x) \ \to \ M \ (X \ (S \ t)) \\
Val & : Type \\
reward & : (t \ : \ \mathbb{N}) \ \to \ (x \ : \ X \ t) \ \to \ (y \ : \ Y \ t \ x) \ \to \ (x' \ : \ X \ (S \ t)) \ \to \ Val \\
(\oplus) & : Val \ \to \ Val \ \to \ Val \\
zero & : Val \\
(\leqslant) & : Val \ \to \ Val \ \to \ Type
\end{array}$$

10.2 Uncertainty measure

In the deterministic case $M \ X = X$ and the above functions completely define a sequential decision problem.

But when a decision step has an uncertain outcome, uncertainties about "next" states naturally yield uncertainties about rewards. In these cases, the decision maker faces a number of possible rewards (one for each possible next state) and has to explain how to measure such chances. In stochastic decision problems, possible next states (and, therefore possible rewards) are labeled with probabilities. In these cases, possible rewards are often measured in terms of their expected value. Here, again, we follow the approach proposed by Ionescu in [2] and introduce a measure

$$meas \ : \ M \ Val \ \to \ Val$$

10.3 Basic requirements

The basic requirements for implementing a verified form of backwards induction are, as in the deterministic case

$$\begin{array}{ll}
map & : \{A, B \ : \ Type\} \ \to \ (A \ \to \ B) \ \to \ M \ A \ \to \ M \ B \\
lteRefl & : \{a \ : \ Val\} \ \to \ a \leqslant a \\
lteTrans & : \{a, b, c \ : \ Val\} \ \to \ a \leqslant b \ \to \ b \leqslant c \ \to \ a \leqslant c
\end{array}$$

$$plusMon : \{a, b, c, d : Val\} \to a \leqslant b \to c \leqslant d \to (a \oplus c) \leqslant (b \oplus d)$$

Additionally, as shown in [2], $meas$ has to fulfill a monotonicity condition:

$$\begin{aligned} measMon : \{A : Type\} \to (f, g : A \to Val) \to ((a : A) \to (f\ a) \leqslant (g\ a)) \to \\ (ma : M\ A) \to meas\ (map\ f\ ma) \leqslant meas\ (map\ g\ ma) \end{aligned}$$

Under exact arithmetic, the expected value measure does fulfill $measMon$, as one would expect.

It is useful to introduce a binary operator that extends $(\oplus)$ to generic functions of codomain Val:

$$\begin{aligned} &(\bigoplus) : \{A : Type\} \to (A \to Val) \to (A \to Val) \to A \to Val \\ &f \bigoplus g = \lambda a \Rightarrow f\ a \oplus g\ a \end{aligned}$$

10.4 Policies and policy sequences

With these premises, the naive theory from lecture 7 extends very straightforwardly to the general, monadic case. The notions of policy and policy sequence are exactly the same:

$$\begin{aligned} &Policy : (t : \mathbb{N}) \to Type \\ &Policy\ t = (x : X\ t) \to Y\ t\ x \end{aligned}$$

$$\begin{aligned} &\mathbf{data}\ PolicySeq : (t : \mathbb{N}) \to (n : \mathbb{N}) \to Type\ \mathbf{where} \\ &\quad Nil : \{t : \mathbb{N}\} \to PolicySeq\ t\ Z \\ &\quad (::) : \{t, n : \mathbb{N}\} \to Policy\ t \to PolicySeq\ (S\ t)\ n \to PolicySeq\ t\ (S\ n) \end{aligned}$$

10.5 Value function

The definition of the value function is a natural extension of the deterministic definition from lecture 7. This was

$$\begin{aligned} &val : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to (x : X\ t) \to Val \\ &val\ \{t\}\ Nil \quad\ x = zero \\ &val\ \{t\}\ (p :: ps)\ x = \mathbf{let}\ y = p\ x\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad \mathbf{let}\ x' = next\ t\ x\ y\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad reward\ t\ x\ y\ x' \oplus val\ ps\ x' \end{aligned}$$

In the monadic case, $next\ t\ x\ y$ yields an M-structure (a list, a probability distribution, etc.) of values of type $X\ (S\ t)$.

As anticipated above, applying $reward\ t\ x\ y \bigoplus val\ ps$ to these values yields an M-structure of Val values. This uncertainty over possible rewards is then measured with $meas$:

$$val : \{t, n : \mathbb{N}\} \to PolicySeq\ t\ n \to (x : X\ t) \to Val$$

$$
\begin{array}{l}
val\ \{t\}\ \ Nil\qquad\ \ x = zero \\
val\ \{t\}\ (p :: ps)\ x = \mathbf{let}\ y = p\ x\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\ \ \mathbf{let}\ mx' = next\ t\ x\ y\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\ \ meas\ (map\ (reward\ t\ x\ y \bigoplus val\ ps)\ mx')
\end{array}
$$

Exercise 10.1. What are the types of mx', $reward\ t\ x\ y \bigoplus val\ ps$ and $map\ (reward\ t\ x\ y \bigoplus val\ ps)\ mx'$ in the definition of val?

We will come back to the definition of val later in this lecture.

10.6 Optimality notions and Bellman's principle

The notions of optimal policy sequence, optimal extension and Bellman's principle of optimality are exactly the same as in the deterministic case:

$$
\begin{array}{l}
OptPolicySeq\ :\ \{t, n\ :\ \mathbb{N}\}\ \to\ PolicySeq\ t\ n\ \to\ Type \\
OptPolicySeq\ \{t\}\ \{n\}\ ps = (ps'\ :\ PolicySeq\ t\ n)\ \to\ (x\ :\ X\ t)\ \to\ val\ ps'\ x \leqslant val\ ps\ x
\end{array}
$$

$$
\begin{array}{l}
OptExt\ :\ \{t, m\ :\ \mathbb{N}\}\ \to\ PolicySeq\ (S\ t)\ m\ \to\ Policy\ t\ \to\ Type \\
OptExt\ \{t\}\ ps\ p = (p'\ :\ Policy\ t)\ \to\ (x\ :\ X\ t)\ \to\ val\ (p' :: ps)\ x \leqslant val\ (p :: ps)\ x
\end{array}
$$

$$
\begin{array}{l}
Bellman\ :\ \{t, m\ :\ \mathbb{N}\}\ \to \\
\qquad\qquad (ps\ :\ PolicySeq\ (S\ t)\ m)\ \to\ OptPolicySeq\ ps\ \to \\
\qquad\qquad (p\ \ :\ Policy\ t)\qquad\qquad\quad \to\ OptExt\ ps\ p\ \to \\
\qquad\qquad OptPolicySeq\ (p :: ps)
\end{array}
$$

The implementation of $Bellman$ now crucially relies on the monotonicity of $meas$:

$$
\begin{array}{lll}
\multicolumn{3}{l}{Bellman\ \{t\}\ ps\ ops\ p\ oep\ (p' :: ps')\ x =} \\
\quad \mathbf{let}\ y' = & p'\ x\ \mathbf{in} & \\
\quad \mathbf{let}\ mx' = & next\ t\ x\ y'\ \mathbf{in} & \\
\quad \mathbf{let}\ f' = & reward\ t\ x\ y' \bigoplus val\ ps'\ \mathbf{in} & \\
\quad \mathbf{let}\ f\ = & reward\ t\ x\ y' \bigoplus val\ ps\ \mathbf{in} & \\
\quad \mathbf{let}\ s_0 = & \lambda x' \Rightarrow plusMon\ lteRefl\ (ops\ ps'\ x')\ \mathbf{in} & \text{-- ?} \\
\quad \mathbf{let}\ s_1 = & measMon\ f'\ f\ s_0\ mx'\ \mathbf{in} & \text{-- } val\ (p' :: ps')\ x \leqslant val\ (p' :: ps)\ x \\
\quad \mathbf{let}\ s_2 = & oep\ p'\ x\ \mathbf{in} & \text{-- } val\ (p' :: ps)\ x \leqslant val\ (p :: ps)\ x \\
\quad lteTrans\ s_1\ s_2 & &
\end{array}
$$

Exercise 10.2. What is the type of s_0 in the definition of *Bellman*?

10.7 Verified backwards induction

This fragment of the theory is exactly as in the deterministic case:

$$
\begin{aligned}
&\mathit{nilOptPolicySeq} \;:\; \mathit{OptPolicySeq}\ \mathit{Nil} \\
&\mathit{nilOptPolicySeq}\ \mathit{Nil}\ x = \mathit{lteRefl}
\end{aligned}
$$

$$\mathit{optExt} \;:\; \{t, n \;:\; \mathbb{N}\} \;\to\; \mathit{PolicySeq}\ (S\ t)\ n \;\to\; \mathit{Policy}\ t$$

$$\mathit{optExtSpec} \;:\; \{t, n \;:\; \mathbb{N}\} \;\to\; (\mathit{ps} \;:\; \mathit{PolicySeq}\ (S\ t)\ n) \;\to\; \mathit{OptExt}\ \mathit{ps}\ (\mathit{optExt}\ \mathit{ps})$$

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

$$
\begin{aligned}
&\mathit{biLemma} \;:\; (t \;:\; \mathbb{N}) \;\to\; (n \;:\; \mathbb{N}) \;\to\; \mathit{OptPolicySeq}\ (\mathit{bi}\ t\ n) \\
&\mathit{biLemma}\ t\ \ Z \qquad = \mathit{nilOptPolicySeq} \\
&\mathit{biLemma}\ t\ (S\ n) = \textbf{let}\ \mathit{ps}\ \ = \mathit{bi}\ (S\ t)\ n\ \textbf{in} \\
&\qquad\qquad\qquad\qquad \textbf{let}\ \mathit{ops} = \mathit{biLemma}\ (S\ t)\ n\ \textbf{in} \\
&\qquad\qquad\qquad\qquad \textbf{let}\ p\ \ \ = \mathit{optExt}\ \mathit{ps}\ \textbf{in} \\
&\qquad\qquad\qquad\qquad \textbf{let}\ \mathit{oep} = \mathit{optExtSpec}\ \mathit{ps}\ \textbf{in} \\
&\qquad\qquad\qquad\qquad \mathit{Bellman}\ \mathit{ps}\ \mathit{ops}\ p\ \mathit{oep}
\end{aligned}
$$

10.8 Naive monadic theory, wrap up

This completes the extension of the naive theory from lecture 7 to the monadic case. Putting together

- Viability and reachability constraints,
- Generic verified optimal extension of arbitrary policy sequences and
- General, monadic SDPs,

is not completely trivial. The full theory (monadic, with viability and reachability constraints and generic optimal extensions) is implemented in "IdrisLibs/SequentialDecisionProblems/FullTheory.lidr" [1].

In the remainder of this lecture, we will discuss an important question related to the interpretation of the value function in the monadic case.

In lecture 11, we dissect an application of the full theory to a climate emission problem.

10.9 The val-val' equivalence in the monadic case

We have argued that, for $ps : PolicySeq\ t\ n$ and $x : X\ t$, $val\ ps\ x$ represents the $meas$-measure (for instance, the expected value) of the sum of the rewards along the trajectories that are obtained under ps when starting in x.

In lecture 7, we have shown that, for the deterministic case, this is indeed the case, i.e.

$$valVal'Th : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps\ x = val'\ ps\ x$$

holds. We now want to derive the same result for the general, monadic case. As in the deterministic case, we start by defining sequences of state-control pairs

$$\begin{array}{l} \mathbf{data}\ StateCtrlSeq : (t : \mathbb{N}) \to (n : \mathbb{N}) \to Type\ \mathbf{where} \\ \quad Last : \{t : \mathbb{N}\} \to (x : X\ t) \to StateCtrlSeq\ t\ (S\ Z) \\ \quad (:::) : \{t, n : \mathbb{N}\} \to \\ \qquad\qquad \Sigma\ (X\ t)\ (Y\ t) \to StateCtrlSeq\ (S\ t)\ (S\ n) \to StateCtrlSeq\ t\ (S\ (S\ n)) \end{array}$$

and a function $sumR$ that computes the sum of the rewards of a state-control sequence:

$$\begin{array}{l} head : \{t, n : \mathbb{N}\} \to StateCtrlSeq\ t\ (S\ n) \to X\ t \\ head\ (Last\ x) \qquad\qquad\qquad = x \\ head\ (MkSigma\ x\ y ::: xys) = x \end{array}$$

$$\begin{array}{l} sumR : \{t, n : \mathbb{N}\} \to StateCtrlSeq\ t\ n \to Val \\ sumR\ \{t\}\ (Last\ x) \qquad\qquad\qquad = zero \\ sumR\ \{t\}\ (MkSigma\ x\ y ::: xys) = reward\ t\ x\ y\ (head\ xys) \oplus sumR\ xys \end{array}$$

Next, we implement a function which computes all the trajectories that are obtained under a policy sequence ps when starting in x. For this, M has to be equipped with monadic operations

$$pure : \{A : Type\} \to A \to M\ A$$

$$(\gg\!=) : \{A, B : Type\} \to M\ A \to (A \to M\ B) \to M\ B$$

$$join : \{A : Type\} \to M\ (M\ A) \to M\ A$$

As usual, we require the operations to fulfill the functor and monad specification from lecture 6:

$$mapPresId : ExtEq\ (map\ id)\ id$$

$$\begin{array}{l} mapPresComp : \{A, B, C : Type\} \to \\ \qquad\qquad (f : A \to B) \to (g : B \to C) \to ExtEq\ (map\ (g \circ f))\ (map\ g \circ map\ f) \end{array}$$

$$mapPresExtEq : \{A, B : Type\} \to (f, g : A \to B) \to ExtEq\ f\ g \to ExtEq\ (map\ f)\ (map\ g)$$

$$pureNatTrans : \{A, B : Type\} \to (f : A \to B) \to ExtEq\ (map\ f \circ pure)\ (pure \circ f)$$

$$joinNatTrans : \{A, B : Type\} \to (f : A \to B) \to ExtEq\ (map\ f \circ join)\ (join \circ map\ (map\ f))$$

$$triangleLeft : ExtEq\ (join \circ pure)\ id$$

$$triangleRight : ExtEq\ (join \circ map\ pure)\ id$$

$$squareLemma : ExtEq\ (join \circ map\ join)\ (join \circ join)$$

$$bindJoinMapSpec : \{A, B : Type\} \to (f : A \to M\ B) \to ExtEq\ (\gg\!= f)\ (join \circ map\ f)$$

With $pure$ and $\gg\!=$, we can express the computation of the trajectories as

$$\begin{aligned}
&trj : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to M\ (StateCtrlSeq\ t\ (S\ n))\\
&trj\ \{t\}\ Nil \quad\ \ x = pure\ (Last\ x)\\
&trj\ \{t\}\ (p :: ps)\ x = \mathbf{let}\ y = p\ x\ \mathbf{in}\\
&\qquad\qquad\qquad\quad\ \mathbf{let}\ mx' = next\ t\ x\ y\ \mathbf{in}\\
&\qquad\qquad\qquad\quad\ map\ ((MkSigma\ x\ y){:::})\ (mx' \gg\!= trj\ ps)
\end{aligned}$$

and compute the measure of the sum of the rewards obtained along the trajectories that are obtained under the policy sequence ps when starting in x

$$\begin{aligned}
&val' : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to Val\\
&val'\ ps\ x = meas\ (map\ sumR\ (trj\ ps\ x))
\end{aligned}$$

Exercise 10.3. What is the type of $map\ sumR\ (trj\ ps\ x)$ in the definition of $val'\ ps\ x$? How does the size of $map\ sumR\ (trj\ ps\ x)$ depend on the length of ps? $val'\ ps\ x$ applies $meas$ only once. How does the number of applications of $meas$ depend on the length of ps in $val\ ps\ x$?

Now we can formulate the property that $val\ ps$ does indeed compute the measure of the sum of the rewards obtained along the trajectories obtained under the policy sequence ps:

$$valVal'Th : \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to val\ ps\ x = val'\ ps\ x$$

As it turns out, the measure function $meas$ has to fulfill three natural conditions for the val-val' theorem to hold. These are

$$\begin{aligned}
&measPlusLemma : \{A : Type\} \to (f, g : A \to Val) \to (ma : M\ A) \to\\
&\qquad\qquad meas\ (map\ (f \bigoplus g)\ ma) = meas\ (map\ f\ ma) \oplus meas\ (map\ g\ ma)
\end{aligned}$$

$$measJoinLemma : (vss : M\ (M\ Val)) \to meas\ (join\ vss) = meas\ (map\ meas\ vss)$$

$$\begin{aligned} measConstLemma :\ & \{A : Type\} \to (v : Val) \to (ma : M\ A) \to \\ & meas\ (map\ (const\ v)\ ma) = v \end{aligned}$$

Exercise 10.4. Describe the meaning of *measPlusLemma*, *measJoinLemma* and *measConstLemma* in words. Rewrite the types of the measure lemmas using the property *ExtEq*.

In order to prove $valVal'\,Th$, we first put forward a few auxiliary lemmas:

$$\begin{aligned} mapConstLemma :\ & \{A, B, C : Type\} \to (c : C) \to (ma : M\ A) \to (f : A \to B) \to \\ & map\ (const\ c)\ ma = map\ (const\ c)\ (map\ f\ ma) \end{aligned}$$

$$measRetLemma : (v : Val) \to meas\ (pure\ v) = v$$

$$\begin{aligned} mapJoinLemma :\ & \{A, B, C : Type\} \to \\ & (f : B \to C) \to (g : A \to M\ B) \to (ma : M\ A) \to \\ & map\ f\ (join\ (map\ g\ ma)) = join\ (map\ (map\ f \circ g)\ ma) \end{aligned}$$

$$\begin{aligned} mapHeadLemma :\ & \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to (x : X\ t) \to \\ & map\ head\ (trj\ ps\ x) = map\ (const\ x)\ (trj\ ps\ x) \end{aligned}$$

$$\begin{aligned} measMapHeadLemma :\ & \{t, n : \mathbb{N}\} \to (ps : PolicySeq\ t\ n) \to \\ & (f : X\ t \to Val) \to (x : X\ t) \to \\ & (meas \circ (map\ (f \circ head) \circ (trj\ ps)))\ x = f\ x \end{aligned}$$

$$\begin{aligned} measMapHeadLemma' :\ & \{t, n : \mathbb{N}\} \to \\ & (ps : PolicySeq\ t\ n) \to (mx : M\ (X\ t)) \to (f : X\ t \to Val) \to \\ & meas\ (map\ (f \circ head)\ (mx \gg= trj\ ps)) = meas\ (map\ f\ mx) \end{aligned}$$

$$\begin{aligned} sumRLemma :\ & \{t, m : \mathbb{N}\} \to (x : X\ t) \to (y : Y\ t\ x) \to \\ & (xyss : M\ (StateCtrlSeq\ (S\ t)\ (S\ m))) \to \\ & map\ sumR\ (map\ ((MkSigma\ x\ y)::)\ xyss) \\ & = \\ & map\ (((reward\ t\ x\ y) \circ head) \bigoplus sumR)\ xyss \end{aligned}$$

We prove these lemmas in section 10.10 below. With their help, we can prove the equivalence of val and val' by induction on ps:

$$valVal'Th \;:\; \{t, n \;:\; \mathbb{N}\} \;\to\; (ps \;:\; PolicySeq\ t\ n) \;\to\; (x \;:\; X\ t) \;\to\; val\ ps\ x = val'\ ps\ x$$

$$\begin{aligned}
valVal'Th\ Nil\ x = \;& (val\ Nil\ x) \\
=\;& \{Refl\} = \\
& (zero) \\
=\;& \{sym\ (measRetLemma\ zero)\} = \\
& (meas\ (pure\ zero)) \\
=\;& \{Refl\} = \\
& (meas\ (pure\ (sumR\ (Last\ x)))) \\
=\;& \{cong\ (sym\ (pureNatTrans\ sumR\ (Last\ x)))\} = \\
& (meas\ (map\ sumR\ (pure\ (Last\ x)))) \\
=\;& \{Refl\} = \\
& (meas\ (map\ sumR\ (trj\ Nil\ x))) \\
=\;& \{Refl\} = \\
& (val'\ Nil\ x) \\
& QED
\end{aligned}$$

$valVal'Th\ \{t\}\ \{n = S\ m\}\ (p :: ps)\ x =$

$$\begin{aligned}
&\textbf{let}\ y \quad = p\ x\ \textbf{in} \\
&\textbf{let}\ mx' = next\ t\ x\ y\ \textbf{in} \\
&\textbf{let}\ r \quad = reward\ t\ x\ y\ \textbf{in} \\
&\textbf{let}\ h \quad = trj\ ps\ \textbf{in} \\
&\textbf{let}\ lhs \;= meas\ (map\ r\ mx')\ \textbf{in} \\
&\textbf{let}\ lhs' = meas\ (map\ (r \circ head)\ (mx' \gg\!= h))\ \textbf{in} \\
&\textbf{let}\ rhs \;= meas\ (map\ meas\ (map\ (map\ sumR)\ (map\ h\ mx')))\ \textbf{in} \\
&\quad (val\ (p :: ps)\ x) \\
&= \{Refl\} = \\
&\quad (meas\ (map\ (r \bigoplus val\ ps)\ mx')) \\
&= \{measPlusLemma\ r\ (val\ ps)\ mx'\} = \\
&\quad (meas\ (map\ r\ mx') \oplus meas\ (map\ (val\ ps)\ mx')) \\
&\quad \text{-- } val\ ps\ x = val'\ ps\ x \Rightarrow map\ (val\ ps)\ mx' = map\ (val'\ ps)\ mx' \\
&= \{cong\ \{f = \lambda\alpha \Rightarrow lhs \oplus meas\ \alpha\} \\
&\qquad (mapPresExtEq\ (val\ ps)\ (val'\ ps)\ (valVal'Th\ ps)\ mx')\} = \\
&\quad (lhs \oplus meas\ (map\ (val'\ ps)\ mx')) \\
&\quad \text{-- val' ps x = ((meas . map sumR) . h) x =¿} \\
&\quad \text{-- map (val' ps) mx' = map ((meas . map sumR) . h) mx'} \\
&= \{cong\ \{f = \lambda\alpha \Rightarrow lhs \oplus meas\ \alpha\}
\end{aligned}$$

$$
\begin{aligned}
&\qquad (mapPresExtEq\ (val'\ ps)\\
&\qquad\quad ((meas \circ map\ \{A = StateCtrlSeq\ (S\ t)\ (S\ m)\}\ sumR) \circ h)\\
&\qquad\qquad (\lambda x \Rightarrow Refl)\ mx')\} =\\
&\quad (lhs \oplus meas\ (map\ ((meas \circ map\ sumR) \circ h)\ mx'))\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow lhs \oplus meas\ \alpha\}\\
&\qquad (mapPresComp\ h\ (meas \circ map\ \{A = StateCtrlSeq\ (S\ t)\ (S\ m)\}\ sumR)\ mx')\} =\\
&\quad (lhs \oplus meas\ (map\ (meas \circ (map\ sumR))\ (map\ h\ mx')))\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow lhs \oplus meas\ \alpha\}\\
&\qquad (mapPresComp\ (map\ \{A = StateCtrlSeq\ (S\ t)\ (S\ m)\}\ sumR)\ meas\ (map\ h\ mx'))\} =\\
&\quad (lhs \oplus meas\ (map\ meas\ (map\ (map\ sumR)\ (map\ h\ mx'))))\\
&= \{ Refl \} =\\
&\quad (meas\ (map\ r\ mx') \oplus rhs)\\
&\quad \text{-- } measMapHeadLemma':\ meas\ (map\ (f \circ head)\ (xs \gg= trj\ ps)) = meas\ (map\ f\ xs)\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow \alpha \oplus rhs\}\\
&\qquad (sym\ (measMapHeadLemma'\ ps\ mx'\ r))\} =\\
&\quad (meas\ (map\ (r \circ head)\ (mx' \gg= h)) \oplus rhs)\\
&= \{ Refl \} =\\
&\quad (meas\ (map\ (r \circ head)\ (mx' \gg= h)) \oplus meas\ (map\ meas\ (map\ (map\ sumR)\ (map\ h\ mx'))))\\
&\quad \text{-- } measJoinLemma:\ meas\ (join\ vss) = meas\ (map\ meas\ vss)\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow lhs' \oplus \alpha\}\\
&\qquad (sym\ (measJoinLemma\ (map\ (map\ sumR)\ (map\ h\ mx'))))\} =\\
&\quad (lhs' \oplus meas\ (join\ (map\ (map\ sumR)\ (map\ h\ mx'))))\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow lhs' \oplus meas\ (join\ \alpha)\}\\
&\qquad (sym\ (mapPresComp\ h\ (map\ \{A = StateCtrlSeq\ (S\ t)\ (S\ m)\}\ sumR)\ mx'))\} =\\
&\quad (lhs' \oplus meas\ (join\ \{A = Val\}\ (map\ (map\ sumR \circ h)\ mx')))\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow lhs' \oplus meas\ \alpha\}\\
&\qquad (sym\ (mapJoinLemma\ sumR\ h\ mx'))\} =\\
&\quad (lhs' \oplus meas\ (map\ sumR\ (join\ (map\ h\ mx'))))\\
&= \{ cong\ \{f = \lambda\alpha \Rightarrow lhs' \oplus meas\ (map\ sumR\ \alpha)\}\\
&\qquad (sym\ (bindJoinMapSpec\ \{B = StateCtrlSeq\ (S\ t)\ (S\ m)\}\ h\ mx'))\} =\\
&\quad (lhs' \oplus meas\ (map\ sumR\ (mx' \gg= h)))\\
&= \{ Refl \} =\\
&\quad (meas\ (map\ (r \circ head)\ (mx' \gg= h)) \oplus meas\ (map\ sumR\ (mx' \gg= h)))\\
&= \{ sym\ (measPlusLemma\ (r \circ head)\ sumR\ (mx' \gg= h))\} =\\
&\quad (meas\ (map\ ((r \circ head) \bigoplus sumR)\ (mx' \gg= h)))\\
&\quad \text{-- } sumRLemma:\ map\ sumR\ (map\ ((MkSigma\ x\ y)::)\ xyss) = map\ ((r \circ head) \bigoplus sumR)\ xyss\\
&= \{ cong\ (sym\ (sumRLemma\ \{m = m\}\ x\ y\ (mx' \gg= h)))\} =\\
&\quad (meas\ (map\ sumR\ (map\ \{A = StateCtrlSeq\ (S\ t)\ (S\ m)\}\ ((MkSigma\ x\ y)::)\\
&\qquad (mx' \gg= trj\ ps))))\\
&= \{ Refl \} =
\end{aligned}
$$

$$
\begin{aligned}
&\quad (meas\ (map\ sumR\ (trj\ (p :: ps)\ x)))\\
&= \{\ Refl\ \} =\\
&\quad (val'\ (p :: ps)\ x)\\
&QED
\end{aligned}
$$

10.10 Auxiliary results

$$
\begin{aligned}
mapConstLemma\ c\ ma\ f = \quad & (map\ (const\ c)\ ma)\\
&= \{\ Refl\ \} =\\
&\quad (map\ ((const\ c) \circ f)\ ma)\\
&= \{\ mapPresComp\ f\ (const\ c)\ ma\ \} =\\
&\quad (map\ (const\ c)\ (map\ f\ ma))\\
&QED
\end{aligned}
$$

$$
\begin{aligned}
measRetLemma\ v = \quad & (meas\ (pure\ v))\\
&= \{\ cong\ (sym\ (pureNatTrans\ (const\ v)\ v))\ \} =\\
&\quad (meas\ (map\ (const\ v)\ (pure\ v)))\\
&= \{\ measConstLemma\ v\ (pure\ v)\ \} =\\
&\quad (v)\\
&QED
\end{aligned}
$$

$$
\begin{aligned}
mapJoinLemma\ f\ g\ ma = \quad & (map\ f\ (join\ (map\ g\ ma)))\\
&= \{\ joinNatTrans\ f\ (map\ g\ ma)\ \} =\\
&\quad (join\ (map\ (map\ f)\ (map\ g\ ma)))\\
&= \{\ Refl\ \} =\\
&\quad (join\ ((map\ (map\ f) \circ (map\ g))\ ma))\\
&= \{\ cong\ (sym\ (mapPresComp\ g\ (map\ f)\ ma))\ \} =\\
&\quad (join\ (map\ (map\ f \circ g)\ ma))\\
&QED
\end{aligned}
$$

$$
\begin{aligned}
&mapHeadLemma\ \{\ t\ \}\ \{\ n = Z\ \}\ Nil\ x\\
&\quad = \quad (map\ head\ (trj\ Nil\ x))\\
&\quad = \{\ Refl\ \} =\\
&\qquad (map\ head\ (pure\ (Last\ x)))\\
&\quad = \{\ pureNatTrans\ head\ (Last\ x)\ \} =\\
&\qquad (pure\ (head\ (Last\ x)))\\
&\quad = \{\ Refl\ \} =\\
&\qquad (pure\ x)
\end{aligned}
$$

$$
\begin{aligned}
&= \{\, sym\ (pureNatTrans\ \{A = StateCtrlSeq\ t\ (S\ Z)\}\ (const\ x)\ (Last\ x))\,\} = \\
&\quad (map\ (const\ x)\ (pure\ (Last\ x))) \\
&= \{\, Refl\,\} = \\
&\quad (map\ (const\ x)\ (trj\ Nil\ x)) \\
&QED
\end{aligned}
$$

$$
\begin{aligned}
&mapHeadLemma\ \{t\}\ \{n = S\ m\}\ (p :: ps)\ x \\
&\quad = \ \ \mathbf{let}\ y \quad\ = p\ x\ \mathbf{in} \\
&\qquad\ \ \mathbf{let}\ xy \quad = MkSigma\ x\ y\ \mathbf{in} \\
&\qquad\ \ \mathbf{let}\ mx' \ = next\ t\ x\ y\ \mathbf{in} \\
&\qquad\ \ \mathbf{let}\ xyss = (mx' \ggg trj\ ps)\ \mathbf{in} \\
&\qquad\ \ (map\ head\ (trj\ (p :: ps)\ x)) \\
&\quad = \{\, Refl\,\} = \\
&\qquad (map\ head\ (map\ \{B = StateCtrlSeq\ t\ (S\ (S\ m))\}\ (xy:::)\ xyss)) \\
&\quad = \{\, sym\ (mapPresComp\ \{B = StateCtrlSeq\ t\ (S\ (S\ m))\}\ (xy:::)\ head\ xyss)\,\} = \\
&\qquad (map\ (head \circ (xy:::))\ xyss) \\
&\quad = \{\, Refl\,\} = \\
&\qquad (map\ (const\ x)\ xyss) \\
&\quad = \{\, mapConstLemma\ \{B = StateCtrlSeq\ t\ (S\ (S\ m))\}\ x\ xyss\ (xy:::)\,\} = \\
&\qquad (map\ (const\ x)\ (map\ \{B = StateCtrlSeq\ t\ (S\ (S\ m))\}\ (xy:::)\ xyss)) \\
&\quad = \{\, Refl\,\} = \\
&\qquad (map\ (const\ x)\ (trj\ (p :: ps)\ x)) \\
&\quad QED
\end{aligned}
$$

$$
\begin{aligned}
&measMapHeadLemma\ \{t\}\ \{n\}\ ps\ f\ x \\
&\quad = \ \ ((meas \circ (map\ (f \circ head) \circ (trj\ ps)))\ x) \\
&\quad = \{\, Refl\,\} = \\
&\qquad (meas\ (map\ (f \circ head)\ (trj\ ps\ x))) \\
&\quad = \{\, cong\ (mapPresComp\ head\ f\ (trj\ ps\ x))\,\} = \\
&\qquad (meas\ (map\ f\ (map\ head\ (trj\ ps\ x)))) \\
&\quad = \{\, cong\ \{f = \lambda\alpha \Rightarrow meas\ (map\ f\ \alpha)\}\ (mapHeadLemma\ ps\ x)\,\} = \\
&\qquad (meas\ (map\ f\ (map\ (const\ x)\ (trj\ ps\ x)))) \\
&\quad = \{\, cong\ (sym\ (mapPresComp\ \{A = StateCtrlSeq\ t\ (S\ n)\}\ (const\ x)\ f\ (trj\ ps\ x)))\,\} = \\
&\qquad (meas\ (map\ (f \circ (const\ x))\ (trj\ ps\ x))) \\
&\quad = \{\, Refl\,\} = \\
&\qquad (meas\ (map\ (const\ (f\ x))\ (trj\ ps\ x))) \\
&\quad = \{\, measConstLemma\ (f\ x)\ (trj\ ps\ x)\,\} = \\
&\qquad (f\ x) \\
&\quad QED
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{measMapHeadLemma'}\ \{t\}\ \{n\}\ \mathit{ps}\ \mathit{mx}\ f \\
& \quad = \ \ \textbf{let}\ g = \mathit{trj}\ \mathit{ps}\ \textbf{in} \\
& \qquad (\mathit{meas}\ (\mathit{map}\ (f \circ \mathit{head})\ (\mathit{mx} \ggg g))) \\
& \quad = \{\mathit{cong}\ \{f = \lambda\alpha \Rightarrow \mathit{meas}\ (\mathit{map}\ (f \circ \mathit{head})\ \alpha)\}\ (\mathit{bindJoinMapSpec}\ \{B = \mathit{StateCtrlSeq}\ t\ (S\ n)\}\ g\ \mathit{mx})\} = \\
& \qquad (\mathit{meas}\ (\mathit{map}\ (f \circ \mathit{head})\ (\mathit{join}\ (\mathit{map}\ g\ \mathit{mx})))) \\
& \quad = \{\mathit{cong}\ (\mathit{mapJoinLemma}\ (f \circ \mathit{head})\ g\ \mathit{mx})\} = \\
& \qquad (\mathit{meas}\ (\mathit{join}\ (\mathit{map}\ (\mathit{map}\ (f \circ \mathit{head}) \circ g)\ \mathit{mx}))) \\
& \quad = \{\mathit{measJoinLemma}\ (\mathit{map}\ (\mathit{map}\ (f \circ \mathit{head}) \circ g)\ \mathit{mx})\} = \\
& \qquad (\mathit{meas}\ (\mathit{map}\ \mathit{meas}\ (\mathit{map}\ (\mathit{map}\ (f \circ \mathit{head}) \circ g)\ \mathit{mx}))) \\
& \quad = \{\mathit{cong}\ (\mathit{sym}\ (\mathit{mapPresComp}\ (\mathit{map}\ (f \circ \mathit{head}) \circ g)\ \mathit{meas}\ \mathit{mx}))\} = \\
& \qquad (\mathit{meas}\ (\mathit{map}\ (\mathit{meas} \circ (\mathit{map}\ (f \circ \mathit{head}) \circ g))\ \mathit{mx})) \\
& \quad = \{\mathit{cong}\ (\mathit{mapPresExtEq}\ (\mathit{meas} \circ (\mathit{map}\ (f \circ \mathit{head}) \circ (\mathit{trj}\ \mathit{ps})))\ f\ (\mathit{measMapHeadLemma}\ \mathit{ps}\ f)\ \mathit{mx})\} = \\
& \qquad (\mathit{meas}\ (\mathit{map}\ f\ \mathit{mx})) \\
& \quad \mathit{QED}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{sumRLemma}\ \{t\}\ \{m\}\ x\ y\ \mathit{xyss} \\
& \quad = \ \ (\mathit{map}\ \mathit{sumR}\ (\mathit{map}\ \{A = \mathit{StateCtrlSeq}\ (S\ t)\ (S\ m)\}\ ((\mathit{MkSigma}\ x\ y){:::})\ \mathit{xyss})) \\
& \quad = \{\mathit{sym}\ (\mathit{mapPresComp}\ \{A = \mathit{StateCtrlSeq}\ (S\ t)\ (S\ m)\}\ ((\mathit{MkSigma}\ x\ y){:::})\ \mathit{sumR}\ \mathit{xyss})\} = \\
& \qquad (\mathit{map}\ \{A = \mathit{StateCtrlSeq}\ (S\ t)\ (S\ m)\}\ (\mathit{sumR} \circ ((\mathit{MkSigma}\ x\ y){:::}))\ \mathit{xyss}) \\
& \quad = \{\mathit{Refl}\} = \\
& \qquad (\mathit{map}\ (((\mathit{reward}\ t\ x\ y) \circ \mathit{head}) \bigoplus \mathit{sumR})\ \mathit{xyss}) \\
& \quad \mathit{QED}
\end{aligned}
$$

References

[1] Nicola Botta. IdrisLibs. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`, 2016–2018.

[2] Cezar Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin, 2009.

Lecture 11: Specifying an emission problem

In this lecture we learn how to specify and solve the GHG emission problem sketched in lecture 1 using the *SequentialDecisionProblems* components of *IdrisLibs*, see [2].

The idea is that "best" decisions on levels of greenhouse gases (GHG) emissions (that is, how much GHG shall be allowed to be emitted in a given time period) are affected by three major sources of uncertainty:

1. uncertainty about the (typically negative) effects of high GHG concentrations in the atmosphere,
2. uncertainty about the availability of effective (cheap, efficient) technologies for reducing GHG emissions,
3. uncertainty about the capability of actually implementing a decision on a given GHG emission level.

We study the effects of these uncertainties on optimal sequences of emission policies.

We design an emission game that accounts for all three sources of uncertainty and yet is simple enough to support investigating the logical consequences of different assumptions through comparisons and parametric studies.

For a more comprehensive discussion of this approach and of the emission game, see [1].

11.1 Controls

We consider a game in which, at each decision step, the decision maker can select between low and high GHG emissions

$$Y\ t\ x = \mathit{LowHigh}$$

Low emissions, if implemented, increase the cumulated GHG emissions less than high emissions.

11.2 States

At each decision step, the decision maker has to choose an option on the basis of four data: the cumulated GHG emissions, the current emission level (low or high), the availability of effective technologies for reducing GHG emissions and the state of the world. Effective technologies for reducing GHG emissions can be either available or unavailable. The state of the world can be either good or bad:

$$\begin{aligned}
&\mathit{CumulatedEmissions} : (t : \mathbb{N}) \to \mathit{Type}\\
&\mathit{CumulatedEmissions}\ t = \mathit{Fin}\ (S\ t)
\end{aligned}$$

$$X\ t = (\mathit{CumulatedEmissions}\ t, \mathit{LowHigh}, \mathit{AvailableUnavailable}, \mathit{GoodBad})$$

The idea is that the game starts with zero cumulated emissions, high emission levels, unavailable GHG technologies and with the world in a good state.

In these conditions, the probability to enter the bad state is low. But if the cumulated emissions increase beyond a fixed critical threshold, the probability that the state of the world turns bad increases. If the world is the bad state, there is no chance to come back to the good state.

Similarly, the probability that effective technologies for reducing GHG emissions become available increases after a fixed number of decision steps. Once available, effective technologies stay available for ever.

The capability of actually implementing a decision on a given GHG emission level in general depends on many factors. In our simplified setup, we just investigate the effect of inertia: implementing low emissions is easier when low emission policies are already in place than when the current emission policies are high emission policies. Similarly, implementing high emission policies is easier under high emissions policies than under low emissions policies.

11.3 Transition function

The critical cumulated emissions threshold:

$$crE \; : \; Double$$
$$crE = 4.0$$

The critical number of decision steps:

$$crN \; : \; \mathbb{N}$$
$$crN = 2$$

The probability of staying in a good world when the cumulated emissions are $\leqslant$ the critical threshold crE:

$$pS1 \; : \; NonNegDouble$$
$$pS1 = cast \; 0.9$$

The probability of staying in a good world when the cumulated emissions are $\geqslant$ the critical threshold crE:

$$pS2 \; : \; NonNegDouble$$
$$pS2 = cast \; 0.1$$

$$check01 \; : \; pS2 \leqslant pS1 \quad \text{-- semantic check}$$
$$check01 = MkLTE \; Oh$$

The probability of effective technologies for reducing GHG emissions becoming available when the number of decision steps is below crN:

$$pA1 \; : \; NonNegDouble$$
$$pA1 = cast \; 0.1$$

The probability of effective technologies for reducing GHG emissions becoming available when the number of decision steps is above crN:

$pA2 \;:\; NonNegDouble$
$pA2 = cast\ 0.9$

$check02 \;:\; pA1 \leqslant pA2$ -- semantic check
$check02 = MkLTE\ Oh$

The probability of being able to implement low emission policies when the current emissions are low and low emissions are selected:

$pLL \;:\; NonNegDouble$
$pLL = cast\ 0.9$

The probability of being able to implement low emission policies when the current emissions are high and low emissions are selected:

$pLH \;:\; NonNegDouble$
$pLH = cast\ 0.7$

$check03 \;:\; pLH \leqslant pLL$ -- semantic check
$check03 = MkLTE\ Oh$

The probability of being able to implement high emission policies when the current emissions are low and high emissions are selected;

$pHL \;:\; NonNegDouble$
$pHL = cast\ 0.7$

The probability of being able to implement high emission policies when the current emissions are high and high emissions are selected:

$pHH \;:\; NonNegDouble$
$pHH = cast\ 0.9$

$check04 \;:\; pHL \leqslant pHH$ -- semantic check
$check04 = MkLTE\ Oh$

Low emissions leave the cumulated emissions unchanged, high emissions increase the cumulated emissions by one:

The transition function:

The transition function: **high** emissions

The transition function: **high** emissions, **unavailable** GHG technologies

The transition function: **high** emissions, **unavailable** GHG technologies, **good** world

using implementation NumNonNegDouble

```
SequentialDecisionProblems.CoreTheory.nexts t (e, H, U, G) L =
  let ttres = mkSimpleProb   -- case t ⩽ crN ∧ fromFin e ⩽ crE
    [((weaken e,  L, U, G),       pLH  * (1 − pA1) *       pS1),
     ((FS e,      H, U, G), (1 − pLH) * (1 − pA1) *       pS1),
     ((weaken e,  L, A, G),       pLH  *      pA1  *       pS1),
     ((FS e,      H, A, G), (1 − pLH) *      pA1  *       pS1),
     ((weaken e,  L, U, B),       pLH  * (1 − pA1) * (1 − pS1)),
     ((FS e,      H, U, B), (1 − pLH) * (1 − pA1) * (1 − pS1)),
     ((weaken e,  L, A, B),       pLH  *      pA1  * (1 − pS1)),
     ((FS e,      H, A, B), (1 − pLH) *      pA1  * (1 − pS1))] in
  let tfres = mkSimpleProb   -- case t ⩽ crN ∧ fromFin e > crE
    [((weaken e,  L, U, G),       pLH  * (1 − pA1) *       pS2),
     ((FS e,      H, U, G), (1 − pLH) * (1 − pA1) *       pS2),
     ((weaken e,  L, A, G),       pLH  *      pA1  *       pS2),
     ((FS e,      H, A, G), (1 − pLH) *      pA1  *       pS2),
     ((weaken e,  L, U, B),       pLH  * (1 − pA1) * (1 − pS2)),
     ((FS e,      H, U, B), (1 − pLH) * (1 − pA1) * (1 − pS2)),
     ((weaken e,  L, A, B),       pLH  *      pA1  * (1 − pS2)),
     ((FS e,      H, A, B), (1 − pLH) *      pA1  * (1 − pS2))] in
  let ftres = mkSimpleProb   -- case t > crN ∧ fromFin e ⩽ crE
    [((weaken e,  L, U, G),       pLH  * (1 − pA2) *       pS1),
     ((FS e,      H, U, G), (1 − pLH) * (1 − pA2) *       pS1),
     ((weaken e,  L, A, G),       pLH  *      pA2  *       pS1),
     ((FS e,      H, A, G), (1 − pLH) *      pA2  *       pS1),
     ((weaken e,  L, U, B),       pLH  * (1 − pA2) * (1 − pS1)),
     ((FS e,      H, U, B), (1 − pLH) * (1 − pA2) * (1 − pS1)),
     ((weaken e,  L, A, B),       pLH  *      pA2  * (1 − pS1)),
     ((FS e,      H, A, B), (1 − pLH) *      pA2  * (1 − pS1))] in
  let ffres = mkSimpleProb   -- case t > crN ∧ fromFin e > crE
    [((weaken e,  L, U, G),       pLH  * (1 − pA2) *       pS2),
     ((FS e,      H, U, G), (1 − pLH) * (1 − pA2) *       pS2),
     ((weaken e,  L, A, G),       pLH  *      pA2  *       pS2),
     ((FS e,      H, A, G), (1 − pLH) *      pA2  *       pS2),
     ((weaken e,  L, U, B),       pLH  * (1 − pA2) * (1 − pS2)),
     ((FS e,      H, U, B), (1 − pLH) * (1 − pA2) * (1 − pS2)),
     ((weaken e,  L, A, B),       pLH  *      pA2  * (1 − pS2)),
     ((FS e,      H, A, B), (1 − pLH) *      pA2  * (1 − pS2))] in
  case (t ⩽ crN) of
```

```
  True  ⇒ case (fromFin e ≤ crE) of
            True  ⇒ trim ttres
            False ⇒ trim tfres
  False ⇒ case (fromFin e ≤ crE) of
            True  ⇒ trim ftres
            False ⇒ trim ffres
SequentialDecisionProblems.CoreTheory.nexts t (e, H, U, G) H =
  let ttres = mkSimpleProb
    [((weaken e,  L, U, G), (1 - pHH) * (1 - pA1) *       pS1),
     ((FS e,      H, U, G),      pHH  * (1 - pA1) *       pS1),
     ((weaken e,  L, A, G), (1 - pHH) *      pA1  *       pS1),
     ((FS e,      H, A, G),      pHH  *      pA1  *       pS1),
     ((weaken e,  L, U, B), (1 - pHH) * (1 - pA1) * (1 - pS1)),
     ((FS e,      H, U, B),      pHH  * (1 - pA1) * (1 - pS1)),
     ((weaken e,  L, A, B), (1 - pHH) *      pA1  * (1 - pS1)),
     ((FS e,      H, A, B),      pHH  *      pA1  * (1 - pS1))] in
  let tfres = mkSimpleProb
    [((weaken e,  L, U, G), (1 - pHH) * (1 - pA1) *       pS2),
     ((FS e,      H, U, G),      pHH  * (1 - pA1) *       pS2),
     ((weaken e,  L, A, G), (1 - pHH) *      pA1  *       pS2),
     ((FS e,      H, A, G),      pHH  *      pA1  *       pS2),
     ((weaken e,  L, U, B), (1 - pHH) * (1 - pA1) * (1 - pS2)),
     ((FS e,      H, U, B),      pHH  * (1 - pA1) * (1 - pS2)),
     ((weaken e,  L, A, B), (1 - pHH) *      pA1  * (1 - pS2)),
     ((FS e,      H, A, B),      pHH  *      pA1  * (1 - pS2))] in
  let ftres = mkSimpleProb
    [((weaken e,  L, U, G), (1 - pHH) * (1 - pA2) *       pS1),
     ((FS e,      H, U, G),      pHH  * (1 - pA2) *       pS1),
     ((weaken e,  L, A, G), (1 - pHH) *      pA2  *       pS1),
     ((FS e,      H, A, G),      pHH  *      pA2  *       pS1),
     ((weaken e,  L, U, B), (1 - pHH) * (1 - pA2) * (1 - pS1)),
     ((FS e,      H, U, B),      pHH  * (1 - pA2) * (1 - pS1)),
     ((weaken e,  L, A, B), (1 - pHH) *      pA2  * (1 - pS1)),
     ((FS e,      H, A, B),      pHH  *      pA2  * (1 - pS1))] in
  let ffres = mkSimpleProb
    [((weaken e,  L, U, G), (1 - pHH) * (1 - pA2) *       pS2),
     ((FS e,      H, U, G),      pHH  * (1 - pA2) *       pS2),
     ((weaken e,  L, A, G), (1 - pHH) *      pA2  *       pS2),
```

$$
\begin{array}{l}
\quad ((FS\ e, \quad H, A, G), \quad pHH \ * \quad pA2 \ * \quad pS2),\\
\quad ((weaken\ e, \quad L, U, B), \ (1 - pHH) * (1 - pA2) * (1 - pS2)),\\
\quad ((FS\ e, \quad H, U, B), \quad pHH \ * (1 - pA2) * (1 - pS2)),\\
\quad ((weaken\ e, \quad L, A, B), \ (1 - pHH) * \quad pA2 \ * (1 - pS2)),\\
\quad ((FS\ e, \quad H, A, B), \quad pHH \ * \quad pA2 \ * (1 - pS2))] \ \mathbf{in}\\
\mathbf{case}\ (t \leqslant crN)\ \mathbf{of}\\
True \ \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of}\\
\qquad True \ \Rightarrow trim\ ttres\\
\qquad False \Rightarrow trim\ tfres\\
False \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of}\\
\qquad True \ \Rightarrow trim\ ftres\\
\qquad False \Rightarrow trim\ ffres
\end{array}
$$

The transition function: **high** emissions, **unavailable** GHG technologies, **bad** world

$$
\begin{array}{l}
SequentialDecisionProblems.CoreTheory.nexts\ t\ (e, H, U, B)\ L =\\
\quad \mathbf{let}\ ttres = mkSimpleProb\\
\qquad [((weaken\ e, \quad L, U, B), \quad pLH \ * (1 - pA1)),\\
\qquad ((FS\ e, \quad H, U, B), (1 - pLH) * (1 - pA1)),\\
\qquad ((weaken\ e, \quad L, A, B), \quad pLH \ * \quad pA1),\\
\qquad ((FS\ e, \quad H, A, B), (1 - pLH) * \quad pA1)] \ \mathbf{in}\\
\quad \mathbf{let}\ tfres = mkSimpleProb\\
\qquad [((weaken\ e, \quad L, U, B), \quad pLH \ * (1 - pA1)),\\
\qquad ((FS\ e, \quad H, U, B), (1 - pLH) * (1 - pA1)),\\
\qquad ((weaken\ e, \quad L, A, B), \quad pLH \ * \quad pA1),\\
\qquad ((FS\ e, \quad H, A, B), (1 - pLH) * \quad pA1)] \ \mathbf{in}\\
\quad \mathbf{let}\ ftres = mkSimpleProb\\
\qquad [((weaken\ e, \quad L, U, B), \quad pLH \ * (1 - pA2)),\\
\qquad ((FS\ e, \quad H, U, B), (1 - pLH) * (1 - pA2)),\\
\qquad ((weaken\ e, \quad L, A, B), \quad pLH \ * \quad pA2),\\
\qquad ((FS\ e, \quad H, A, B), (1 - pLH) * \quad pA2)] \ \mathbf{in}\\
\quad \mathbf{let}\ ffres = mkSimpleProb\\
\qquad [((weaken\ e, \quad L, U, B), \quad pLH \ * (1 - pA2)),\\
\qquad ((FS\ e, \quad H, U, B), (1 - pLH) * (1 - pA2)),\\
\qquad ((weaken\ e, \quad L, A, B), \quad pLH \ * \quad pA2),\\
\qquad ((FS\ e, \quad H, A, B), (1 - pLH) * \quad pA2)] \ \mathbf{in}\\
\quad \mathbf{case}\ (t \leqslant crN)\ \mathbf{of}\\
\quad True \ \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of}\\
\qquad\quad True \ \Rightarrow trim\ ttres\\
\qquad\quad False \Rightarrow trim\ tfres
\end{array}
$$

$$
\begin{array}{l}
\quad \mathit{False} \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \Rightarrow \mathit{trim}\ \mathit{ftres} \\
\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{ffres} \\
\mathit{SequentialDecisionProblems.CoreTheory.nexts}\ t\ (e, H, U, B)\ H = \\
\quad \textbf{let}\ \mathit{ttres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, U, B),\ (1 - \mathit{pHH}) * (1 - \mathit{pA1})), \\
\qquad ((\mathit{FS}\ e, \quad H, U, B),\ \mathit{pHH} * (1 - \mathit{pA1})), \\
\qquad ((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH}) * \mathit{pA1}), \\
\qquad ((\mathit{FS}\ e, \quad H, A, B),\ \mathit{pHH} * \mathit{pA1})]\ \textbf{in} \\
\quad \textbf{let}\ \mathit{tfres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, U, B),\ (1 - \mathit{pHH}) * (1 - \mathit{pA1})), \\
\qquad ((\mathit{FS}\ e, \quad H, U, B),\ \mathit{pHH} * (1 - \mathit{pA1})), \\
\qquad ((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH}) * \mathit{pA1}), \\
\qquad ((\mathit{FS}\ e, \quad H, A, B),\ \mathit{pHH} * \mathit{pA1})]\ \textbf{in} \\
\quad \textbf{let}\ \mathit{ftres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, U, B),\ (1 - \mathit{pHH}) * (1 - \mathit{pA2})), \\
\qquad ((\mathit{FS}\ e, \quad H, U, B),\ \mathit{pHH} * (1 - \mathit{pA2})), \\
\qquad ((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH}) * \mathit{pA2}), \\
\qquad ((\mathit{FS}\ e, \quad H, A, B),\ \mathit{pHH} * \mathit{pA2})]\ \textbf{in} \\
\quad \textbf{let}\ \mathit{ffres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, U, B),\ (1 - \mathit{pHH}) * (1 - \mathit{pA2})), \\
\qquad ((\mathit{FS}\ e, \quad H, U, B),\ \mathit{pHH} * (1 - \mathit{pA2})), \\
\qquad ((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH}) * \mathit{pA2}), \\
\qquad ((\mathit{FS}\ e, \quad H, A, B),\ \mathit{pHH} * \mathit{pA2})]\ \textbf{in} \\
\quad \textbf{case}\ (t \leqslant \mathit{crN})\ \textbf{of} \\
\quad \mathit{True}\ \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \Rightarrow \mathit{trim}\ \mathit{ttres} \\
\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{tfres} \\
\quad \mathit{False} \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \Rightarrow \mathit{trim}\ \mathit{ftres} \\
\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{ffres}
\end{array}
$$

The transition function: **high** emissions, **available** GHG technologies

The transition function: **high** emissions, **available** GHG technologies, **good** world

$$
\begin{array}{l}
\mathit{SequentialDecisionProblems.CoreTheory.nexts}\ t\ (e, H, A, G)\ L = \\
\quad \textbf{let}\ \mathit{ttres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, A, G),\ \mathit{pLH} * \mathit{pS1}), \\
\qquad ((\mathit{FS}\ e, \quad H, A, G),\ (1 - \mathit{pLH}) * \mathit{pS1}), \\
\qquad ((\mathit{weaken}\ e, \quad L, A, B),\ \mathit{pLH} * (1 - \mathit{pS1})),
\end{array}
$$

$$
\begin{array}{l}
\quad ((FS\ e, \quad H, A, B), (1 - pLH) * (1 - pS1))]\ \mathbf{in} \\
\mathbf{let}\ tfres = mkSimpleProb \\
\quad [((weaken\ e, \quad L, A, G), \quad pLH \ * \quad pS2), \\
\quad ((FS\ e, \quad H, A, G), (1 - pLH) * \quad pS2), \\
\quad ((weaken\ e, \quad L, A, B), \quad pLH \ * (1 - pS2)), \\
\quad ((FS\ e, \quad H, A, B), (1 - pLH) * (1 - pS2))]\ \mathbf{in} \\
\mathbf{let}\ ftres = mkSimpleProb \\
\quad [((weaken\ e, \quad L, A, G), \quad pLH \ * \quad pS1), \\
\quad ((FS\ e, \quad H, A, G), (1 - pLH) * \quad pS1), \\
\quad ((weaken\ e, \quad L, A, B), \quad pLH \ * (1 - pS1)), \\
\quad ((FS\ e, \quad H, A, B), (1 - pLH) * (1 - pS1))]\ \mathbf{in} \\
\mathbf{let}\ ffres = mkSimpleProb \\
\quad [((weaken\ e, \quad L, A, G), \quad pLH \ * \quad pS2), \\
\quad ((FS\ e, \quad H, A, G), (1 - pLH) * \quad pS2), \\
\quad ((weaken\ e, \quad L, A, B), \quad pLH \ * (1 - pS2)), \\
\quad ((FS\ e, \quad H, A, B), (1 - pLH) * (1 - pS2))]\ \mathbf{in} \\
\mathbf{case}\ (t \leqslant crN)\ \mathbf{of} \\
True \ \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of} \\
\qquad\qquad True \ \Rightarrow trim\ ttres \\
\qquad\qquad False \Rightarrow trim\ tfres \\
False \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of} \\
\qquad\qquad True \ \Rightarrow trim\ ftres \\
\qquad\qquad False \Rightarrow trim\ ffres
\end{array}
$$

$$
\begin{array}{l}
SequentialDecisionProblems.CoreTheory.nexts\ t\ (e, H, A, G)\ H = \\
\quad \mathbf{let}\ ttres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, A, G), \ (1 - pHH) * \quad pS1), \\
\quad\quad ((FS\ e, \quad H, A, G), \quad pHH \ * \quad pS1), \\
\quad\quad ((weaken\ e, \quad L, A, B), \ (1 - pHH) * (1 - pS1)), \\
\quad\quad ((FS\ e, \quad H, A, B), \quad pHH \ * (1 - pS1))]\ \mathbf{in} \\
\quad \mathbf{let}\ tfres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, A, G), \ (1 - pHH) * \quad pS2), \\
\quad\quad ((FS\ e, \quad H, A, G), \quad pHH \ * \quad pS2), \\
\quad\quad ((weaken\ e, \quad L, A, B), \ (1 - pHH) * (1 - pS2)), \\
\quad\quad ((FS\ e, \quad H, A, B), \quad pHH \ * (1 - pS2))]\ \mathbf{in} \\
\quad \mathbf{let}\ ftres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, A, G), \ (1 - pHH) * \quad pS1), \\
\quad\quad ((FS\ e, \quad H, A, G), \quad pHH \ * \quad pS1), \\
\quad\quad ((weaken\ e, \quad L, A, B), \ (1 - pHH) * (1 - pS1)),
\end{array}
$$

```
    ((FS e,       H, A, B),      pHH  * (1 - pS1))] in
  let ffres = mkSimpleProb
   [((weaken e,   L, A, G), (1 - pHH) *      pS2),
    ((FS e,       H, A, G),      pHH  *      pS2),
    ((weaken e,   L, A, B), (1 - pHH) * (1 - pS2)),
    ((FS e,       H, A, B),      pHH  * (1 - pS2))] in
  case (t <= crN) of
  True  => case (fromFin e <= crE) of
             True  => trim ttres
             False => trim tfres
  False => case (fromFin e <= crE) of
             True  => trim ftres
             False => trim ffres
```

The transition function: **high** emissions, **available** GHG technologies, **bad** world

```
SequentialDecisionProblems.CoreTheory.nexts t (e, H, A, B) L =
  let ttres = mkSimpleProb
   [((weaken e,   L, A, B),      pLH),
    ((FS e,       H, A, B), (1 - pLH))] in
  let tfres = mkSimpleProb
   [((weaken e,   L, A, B),      pLH),
    ((FS e,       H, A, B), (1 - pLH))] in
  let ftres = mkSimpleProb
   [((weaken e,   L, A, B),      pLH),
    ((FS e,       H, A, B), (1 - pLH))] in
  let ffres = mkSimpleProb
   [((weaken e,   L, A, B),      pLH),
    ((FS e,       H, A, B), (1 - pLH))] in
  case (t <= crN) of
  True  => case (fromFin e <= crE) of
             True  => trim ttres
             False => trim tfres
  False => case (fromFin e <= crE) of
             True  => trim ftres
             False => trim ffres
SequentialDecisionProblems.CoreTheory.nexts t (e, H, A, B) H =
  let ttres = mkSimpleProb
   [((weaken e,   L, A, B), (1 - pHH)),
    ((FS e,       H, A, B),      pHH) ] in
```

$$
\begin{array}{l}
\textbf{let } \mathit{tfres} = \mathit{mkSimpleProb} \\
\quad [((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH})), \\
\quad\ ((\mathit{FS}\ e, \qquad H, A, B), \qquad \mathit{pHH})\ ]\ \textbf{in} \\
\textbf{let } \mathit{ftres} = \mathit{mkSimpleProb} \\
\quad [((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH})), \\
\quad\ ((\mathit{FS}\ e, \qquad H, A, B), \qquad \mathit{pHH})\ ]\ \textbf{in} \\
\textbf{let } \mathit{ffres} = \mathit{mkSimpleProb} \\
\quad [((\mathit{weaken}\ e, \quad L, A, B),\ (1 - \mathit{pHH})), \\
\quad\ ((\mathit{FS}\ e, \qquad H, A, B), \qquad \mathit{pHH})\ ]\ \textbf{in} \\
\textbf{case}\ (t \leqslant \mathit{crN})\ \textbf{of} \\
\mathit{True}\ \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \Rightarrow \mathit{trim}\ \mathit{ttres} \\
\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{tfres} \\
\mathit{False} \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \Rightarrow \mathit{trim}\ \mathit{ftres} \\
\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{ffres}
\end{array}
$$

The transition function: **low** emissions

The transition function: **low** emissions, **unavailable** GHG technologies

The transition function: **low** emissions, **unavailable** GHG technologies, **good** world

$$
\begin{array}{l}
\mathit{SequentialDecisionProblems.CoreTheory.nexts}\ t\ (e, L, U, G)\ L = \\
\quad \textbf{let } \mathit{ttres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, U, G), \qquad \mathit{pLL} \ * (1 - \mathit{pA1}) * \qquad \mathit{pS1}), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, U, G), (1 - \mathit{pLL}) * (1 - \mathit{pA1}) * \qquad \mathit{pS1}), \\
\qquad\ ((\mathit{weaken}\ e, \quad L, A, G), \qquad \mathit{pLL} \ * \qquad \mathit{pA1} \ * \qquad \mathit{pS1}), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, A, G), (1 - \mathit{pLL}) * \qquad \mathit{pA1} \ * \qquad \mathit{pS1}), \\
\qquad\ ((\mathit{weaken}\ e, \quad L, U, B), \qquad \mathit{pLL} \ * (1 - \mathit{pA1}) * (1 - \mathit{pS1})), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, U, B), (1 - \mathit{pLL}) * (1 - \mathit{pA1}) * (1 - \mathit{pS1})), \\
\qquad\ ((\mathit{weaken}\ e, \quad L, A, B), \qquad \mathit{pLL} \ * \qquad \mathit{pA1} \ * (1 - \mathit{pS1})), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, A, B), (1 - \mathit{pLL}) * \qquad \mathit{pA1} \ * (1 - \mathit{pS1}))]\ \textbf{in} \\
\quad \textbf{let } \mathit{tfres} = \mathit{mkSimpleProb} \\
\qquad [((\mathit{weaken}\ e, \quad L, U, G), \qquad \mathit{pLL} \ * (1 - \mathit{pA1}) * \qquad \mathit{pS2}), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, U, G), (1 - \mathit{pLL}) * (1 - \mathit{pA1}) * \qquad \mathit{pS2}), \\
\qquad\ ((\mathit{weaken}\ e, \quad L, A, G), \qquad \mathit{pLL} \ * \qquad \mathit{pA1} \ * \qquad \mathit{pS2}), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, A, G), (1 - \mathit{pLL}) * \qquad \mathit{pA1} \ * \qquad \mathit{pS2}), \\
\qquad\ ((\mathit{weaken}\ e, \quad L, U, B), \qquad \mathit{pLL} \ * (1 - \mathit{pA1}) * (1 - \mathit{pS2})), \\
\qquad\ ((\mathit{FS}\ e, \qquad H, U, B), (1 - \mathit{pLL}) * (1 - \mathit{pA1}) * (1 - \mathit{pS2})), \\
\qquad\ ((\mathit{weaken}\ e, \quad L, A, B), \qquad \mathit{pLL} \ * \qquad \mathit{pA1} \ * (1 - \mathit{pS2})),
\end{array}
$$

```
      ((FS e,     H, A, B), (1 - pLL) *       pA1  * (1 - pS2))] in
  let ftres = mkSimpleProb
     [((weaken e, L, U, G),      pLL  * (1 - pA2) *      pS1),
      ((FS e,     H, U, G), (1 - pLL) * (1 - pA2) *      pS1),
      ((weaken e, L, A, G),      pLL  *       pA2  *      pS1),
      ((FS e,     H, A, G), (1 - pLL) *       pA2  *      pS1),
      ((weaken e, L, U, B),      pLL  * (1 - pA2) * (1 - pS1)),
      ((FS e,     H, U, B), (1 - pLL) * (1 - pA2) * (1 - pS1)),
      ((weaken e, L, A, B),      pLL  *       pA2  * (1 - pS1)),
      ((FS e,     H, A, B), (1 - pLL) *       pA2  * (1 - pS1))] in
  let ffres = mkSimpleProb
     [((weaken e, L, U, G),      pLL  * (1 - pA2) *      pS2),
      ((FS e,     H, U, G), (1 - pLL) * (1 - pA2) *      pS2),
      ((weaken e, L, A, G),      pLL  *       pA2  *      pS2),
      ((FS e,     H, A, G), (1 - pLL) *       pA2  *      pS2),
      ((weaken e, L, U, B),      pLL  * (1 - pA2) * (1 - pS2)),
      ((FS e,     H, U, B), (1 - pLL) * (1 - pA2) * (1 - pS2)),
      ((weaken e, L, A, B),      pLL  *       pA2  * (1 - pS2)),
      ((FS e,     H, A, B), (1 - pLL) *       pA2  * (1 - pS2))] in
  case (t <= crN) of
  True  => case (fromFin e <= crE) of
           True  => trim ttres
           False => trim tfres
  False => case (fromFin e <= crE) of
           True  => trim ftres
           False => trim ffres
SequentialDecisionProblems.CoreTheory.nexts t (e, L, U, G) H =
  let ttres = mkSimpleProb
     [((weaken e, L, U, G), (1 - pHL) * (1 - pA1) *      pS1),
      ((FS e,     H, U, G),      pHL  * (1 - pA1) *      pS1),
      ((weaken e, L, A, G), (1 - pHL) *       pA1  *      pS1),
      ((FS e,     H, A, G),      pHL  *       pA1  *      pS1),
      ((weaken e, L, U, B), (1 - pHL) * (1 - pA1) * (1 - pS1)),
      ((FS e,     H, U, B),      pHL  * (1 - pA1) * (1 - pS1)),
      ((weaken e, L, A, B), (1 - pHL) *       pA1  * (1 - pS1)),
      ((FS e,     H, A, B),      pHL  *       pA1  * (1 - pS1))] in
  let tfres = mkSimpleProb
     [((weaken e, L, U, G), (1 - pHL) * (1 - pA1) *      pS2),
```

$$
\begin{array}{llll}
((FS\ e, & H, U, G), & pHL \ast (1 - pA1) \ast & pS2), \\
((weaken\ e, & L, A, G), & (1 - pHL) \ast \quad pA1 \ast & pS2), \\
((FS\ e, & H, A, G), & pHL \ast \quad pA1 \ast & pS2), \\
((weaken\ e, & L, U, B), & (1 - pHL) \ast (1 - pA1) \ast & (1 - pS2)), \\
((FS\ e, & H, U, B), & pHL \ast (1 - pA1) \ast & (1 - pS2)), \\
((weaken\ e, & L, A, B), & (1 - pHL) \ast \quad pA1 \ast & (1 - pS2)), \\
((FS\ e, & H, A, B), & pHL \ast \quad pA1 \ast & (1 - pS2))]\ \mathbf{in}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{let}\ ftres = mkSimpleProb \\
\begin{array}{llll}
[((weaken\ e, & L, U, G), & (1 - pHL) \ast (1 - pA2) \ast & pS1), \\
((FS\ e, & H, U, G), & pHL \ast (1 - pA2) \ast & pS1), \\
((weaken\ e, & L, A, G), & (1 - pHL) \ast \quad pA2 \ast & pS1), \\
((FS\ e, & H, A, G), & pHL \ast \quad pA2 \ast & pS1), \\
((weaken\ e, & L, U, B), & (1 - pHL) \ast (1 - pA2) \ast & (1 - pS1)), \\
((FS\ e, & H, U, B), & pHL \ast (1 - pA2) \ast & (1 - pS1)), \\
((weaken\ e, & L, A, B), & (1 - pHL) \ast \quad pA2 \ast & (1 - pS1)), \\
((FS\ e, & H, A, B), & pHL \ast \quad pA2 \ast & (1 - pS1))]\ \mathbf{in}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{let}\ ffres = mkSimpleProb \\
\begin{array}{llll}
[((weaken\ e, & L, U, G), & (1 - pHL) \ast (1 - pA2) \ast & pS2), \\
((FS\ e, & H, U, G), & pHL \ast (1 - pA2) \ast & pS2), \\
((weaken\ e, & L, A, G), & (1 - pHL) \ast \quad pA2 \ast & pS2), \\
((FS\ e, & H, A, G), & pHL \ast \quad pA2 \ast & pS2), \\
((weaken\ e, & L, U, B), & (1 - pHL) \ast (1 - pA2) \ast & (1 - pS2)), \\
((FS\ e, & H, U, B), & pHL \ast (1 - pA2) \ast & (1 - pS2)), \\
((weaken\ e, & L, A, B), & (1 - pHL) \ast \quad pA2 \ast & (1 - pS2)), \\
((FS\ e, & H, A, B), & pHL \ast \quad pA2 \ast & (1 - pS2))]\ \mathbf{in}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{case}\ (t \leqslant crN)\ \mathbf{of} \\
True \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of} \\
\qquad True \Rightarrow trim\ ttres \\
\qquad False \Rightarrow trim\ tfres \\
False \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of} \\
\qquad True \Rightarrow trim\ ftres \\
\qquad False \Rightarrow trim\ ffres
\end{array}
$$

The transition function: **low** emissions, **unavailable** GHG technologies, **bad** world

$$
\begin{array}{l}
SequentialDecisionProblems.CoreTheory.nexts\ t\ (e, L, U, B)\ L = \\
\mathbf{let}\ ttres = mkSimpleProb \\
\begin{array}{lll}
[((weaken\ e, & L, U, B), & pLL \ast (1 - pA1)), \\
((FS\ e, & H, U, B), & (1 - pLL) \ast (1 - pA1)), \\
((weaken\ e, & L, A, B), & pLL \ast \quad pA1),
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\quad\quad ((FS\ e, \quad H, A, B), (1 - pLL) * \quad pA1)]\ \mathbf{in} \\
\quad \mathbf{let}\ tfres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, U, B), \quad pLL \ * (1 - pA1)), \\
\quad\quad ((FS\ e, \quad H, U, B), (1 - pLL) * (1 - pA1)), \\
\quad\quad ((weaken\ e, \quad L, A, B), \quad pLL \ * \quad pA1), \\
\quad\quad ((FS\ e, \quad H, A, B), (1 - pLL) * \quad pA1)]\ \mathbf{in} \\
\quad \mathbf{let}\ ftres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, U, B), \quad pLL \ * (1 - pA2)), \\
\quad\quad ((FS\ e, \quad H, U, B), (1 - pLL) * (1 - pA2)), \\
\quad\quad ((weaken\ e, \quad L, A, B), \quad pLL \ * \quad pA2), \\
\quad\quad ((FS\ e, \quad H, A, B), (1 - pLL) * \quad pA2)]\ \mathbf{in} \\
\quad \mathbf{let}\ ffres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, U, B), \quad pLL \ * (1 - pA2)), \\
\quad\quad ((FS\ e, \quad H, U, B), (1 - pLL) * (1 - pA2)), \\
\quad\quad ((weaken\ e, \quad L, A, B), \quad pLL \ * \quad pA2), \\
\quad\quad ((FS\ e, \quad H, A, B), (1 - pLL) * \quad pA2)]\ \mathbf{in} \\
\quad \mathbf{case}\ (t \leqslant crN)\ \mathbf{of} \\
\quad True \ \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of} \\
\quad\quad\quad\quad True \ \Rightarrow trim\ ttres \\
\quad\quad\quad\quad False \Rightarrow trim\ tfres \\
\quad False \Rightarrow \mathbf{case}\ (fromFin\ e \leqslant crE)\ \mathbf{of} \\
\quad\quad\quad\quad True \ \Rightarrow trim\ ftres \\
\quad\quad\quad\quad False \Rightarrow trim\ ffres \\
SequentialDecisionProblems.CoreTheory.nexts\ t\ (e, L, U, B)\ H = \\
\quad \mathbf{let}\ ttres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, U, B), (1 - pHL) * (1 - pA1)), \\
\quad\quad ((FS\ e, \quad H, U, B), \quad pHL \ * (1 - pA1)), \\
\quad\quad ((weaken\ e, \quad L, A, B), (1 - pHL) * \quad pA1), \\
\quad\quad ((FS\ e, \quad H, A, B), \quad pHL \ * \quad pA1)]\ \mathbf{in} \\
\quad \mathbf{let}\ tfres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, U, B), (1 - pHL) * (1 - pA1)), \\
\quad\quad ((FS\ e, \quad H, U, B), \quad pHL \ * (1 - pA1)), \\
\quad\quad ((weaken\ e, \quad L, A, B), (1 - pHL) * \quad pA1), \\
\quad\quad ((FS\ e, \quad H, A, B), \quad pHL \ * \quad pA1)]\ \mathbf{in} \\
\quad \mathbf{let}\ ftres = mkSimpleProb \\
\quad\quad [((weaken\ e, \quad L, U, B), (1 - pHL) * (1 - pA2)), \\
\quad\quad ((FS\ e, \quad H, U, B), \quad pHL \ * (1 - pA2)), \\
\quad\quad ((weaken\ e, \quad L, A, B), (1 - pHL) * \quad pA2),
\end{array}
$$

$$
\begin{array}{l}
\quad ((FS\ e, \quad H, A, B), \quad pHL\ * \quad pA2)]\ \textbf{in} \\
\textbf{let}\ \mathit{ffres} = mkSimpleProb \\
\quad [((weaken\ e, \quad L, U, B),\ (1 - pHL) * (1 - pA2)), \\
\quad ((FS\ e, \quad H, U, B), \quad pHL\ * (1 - pA2)), \\
\quad ((weaken\ e, \quad L, A, B),\ (1 - pHL) * \quad pA2), \\
\quad ((FS\ e, \quad H, A, B), \quad pHL\ * \quad pA2)]\ \textbf{in} \\
\textbf{case}\ (t \leqslant crN)\ \textbf{of} \\
True\ \Rightarrow \textbf{case}\ (fromFin\ e \leqslant crE)\ \textbf{of} \\
\qquad True\ \Rightarrow trim\ ttres \\
\qquad False \Rightarrow trim\ tfres \\
False \Rightarrow \textbf{case}\ (fromFin\ e \leqslant crE)\ \textbf{of} \\
\qquad True\ \Rightarrow trim\ ftres \\
\qquad False \Rightarrow trim\ \mathit{ffres}
\end{array}
$$

The transition function: **low** emissions, **available** GHG technologies

The transition function: **low** emissions, **available** GHG technologies, **good** world

$$
\begin{array}{l}
SequentialDecisionProblems.CoreTheory.nexts\ t\ (e, L, A, G)\ L = \\
\quad \textbf{let}\ ttres = mkSimpleProb \\
\qquad [((weaken\ e, \quad L, A, G), \quad pLL\ * \quad pS1), \\
\qquad ((FS\ e, \quad H, A, G), (1 - pLL) * \quad pS1), \\
\qquad ((weaken\ e, \quad L, A, B), \quad pLL\ * (1 - pS1)), \\
\qquad ((FS\ e, \quad H, A, B), (1 - pLL) * (1 - pS1))]\ \textbf{in} \\
\quad \textbf{let}\ tfres = mkSimpleProb \\
\qquad [((weaken\ e, \quad L, A, G), \quad pLL\ * \quad pS2), \\
\qquad ((FS\ e, \quad H, A, G), (1 - pLL) * \quad pS2), \\
\qquad ((weaken\ e, \quad L, A, B), \quad pLL\ * (1 - pS2)), \\
\qquad ((FS\ e, \quad H, A, B), (1 - pLL) * (1 - pS2))]\ \textbf{in} \\
\quad \textbf{let}\ ftres = mkSimpleProb \\
\qquad [((weaken\ e, \quad L, A, G), \quad pLL\ * \quad pS1), \\
\qquad ((FS\ e, \quad H, A, G), (1 - pLL) * \quad pS1), \\
\qquad ((weaken\ e, \quad L, A, B), \quad pLL\ * (1 - pS1)), \\
\qquad ((FS\ e, \quad H, A, B), (1 - pLL) * (1 - pS1))]\ \textbf{in} \\
\quad \textbf{let}\ \mathit{ffres} = mkSimpleProb \\
\qquad [((weaken\ e, \quad L, A, G), \quad pLL\ * \quad pS2), \\
\qquad ((FS\ e, \quad H, A, G), (1 - pLL) * \quad pS2), \\
\qquad ((weaken\ e, \quad L, A, B), \quad pLL\ * (1 - pS2)), \\
\qquad ((FS\ e, \quad H, A, B), (1 - pLL) * (1 - pS2))]\ \textbf{in} \\
\quad \textbf{case}\ (t \leqslant crN)\ \textbf{of} \\
\quad True\ \Rightarrow \textbf{case}\ (fromFin\ e \leqslant crE)\ \textbf{of}
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad \mathit{True} \Rightarrow \mathit{trim}\ \mathit{ttres}\\
&\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{tfres}\\
&\quad \mathit{False} \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of}\\
&\qquad\qquad \mathit{True} \Rightarrow \mathit{trim}\ \mathit{ftres}\\
&\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{ffres}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{SequentialDecisionProblems.CoreTheory.nexts}\ t\ (e, L, A, G)\ H =\\
&\quad \textbf{let}\ \mathit{ttres} = \mathit{mkSimpleProb}\\
&\qquad [((\mathit{weaken}\ e, \ L, A, G),\ (1 - \mathit{pHL}) * \mathit{pS1}),\\
&\qquad ((\mathit{FS}\ e, \ H, A, G),\ \mathit{pHL} * \mathit{pS1}),\\
&\qquad ((\mathit{weaken}\ e, \ L, A, B),\ (1 - \mathit{pHL}) * (1 - \mathit{pS1})),\\
&\qquad ((\mathit{FS}\ e, \ H, A, B),\ \mathit{pHL} * (1 - \mathit{pS1}))]\ \textbf{in}\\
&\quad \textbf{let}\ \mathit{tfres} = \mathit{mkSimpleProb}\\
&\qquad [((\mathit{weaken}\ e, \ L, A, G),\ (1 - \mathit{pHL}) * \mathit{pS2}),\\
&\qquad ((\mathit{FS}\ e, \ H, A, G),\ \mathit{pHL} * \mathit{pS2}),\\
&\qquad ((\mathit{weaken}\ e, \ L, A, B),\ (1 - \mathit{pHL}) * (1 - \mathit{pS2})),\\
&\qquad ((\mathit{FS}\ e, \ H, A, B),\ \mathit{pHL} * (1 - \mathit{pS2}))]\ \textbf{in}\\
&\quad \textbf{let}\ \mathit{ftres} = \mathit{mkSimpleProb}\\
&\qquad [((\mathit{weaken}\ e, \ L, A, G),\ (1 - \mathit{pHL}) * \mathit{pS1}),\\
&\qquad ((\mathit{FS}\ e, \ H, A, G),\ \mathit{pHL} * \mathit{pS1}),\\
&\qquad ((\mathit{weaken}\ e, \ L, A, B),\ (1 - \mathit{pHL}) * (1 - \mathit{pS1})),\\
&\qquad ((\mathit{FS}\ e, \ H, A, B),\ \mathit{pHL} * (1 - \mathit{pS1}))]\ \textbf{in}\\
&\quad \textbf{let}\ \mathit{ffres} = \mathit{mkSimpleProb}\\
&\qquad [((\mathit{weaken}\ e, \ L, A, G),\ (1 - \mathit{pHL}) * \mathit{pS2}),\\
&\qquad ((\mathit{FS}\ e, \ H, A, G),\ \mathit{pHL} * \mathit{pS2}),\\
&\qquad ((\mathit{weaken}\ e, \ L, A, B),\ (1 - \mathit{pHL}) * (1 - \mathit{pS2})),\\
&\qquad ((\mathit{FS}\ e, \ H, A, B),\ \mathit{pHL} * (1 - \mathit{pS2}))]\ \textbf{in}\\
&\quad \textbf{case}\ (t \leqslant \mathit{crN})\ \textbf{of}\\
&\quad \mathit{True} \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of}\\
&\qquad\qquad \mathit{True} \Rightarrow \mathit{trim}\ \mathit{ttres}\\
&\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{tfres}\\
&\quad \mathit{False} \Rightarrow \textbf{case}\ (\mathit{fromFin}\ e \leqslant \mathit{crE})\ \textbf{of}\\
&\qquad\qquad \mathit{True} \Rightarrow \mathit{trim}\ \mathit{ftres}\\
&\qquad\qquad \mathit{False} \Rightarrow \mathit{trim}\ \mathit{ffres}
\end{aligned}
$$

The transition function: **low** emissions, **available** GHG technologies, **bad** world

$$
\begin{aligned}
&\mathit{SequentialDecisionProblems.CoreTheory.nexts}\ t\ (e, L, A, B)\ L =\\
&\quad \textbf{let}\ \mathit{ttres} = \mathit{mkSimpleProb}\\
&\qquad [((\mathit{weaken}\ e, \ L, A, B),\ \mathit{pLL}),\\
&\qquad ((\mathit{FS}\ e, \ H, A, B), (1 - \mathit{pLL}))]\ \textbf{in}
\end{aligned}
$$

```
  let tfres = mkSimpleProb
    [((weaken e,    L, A, B),         pLL),
     ((FS e,        H, A, B), (1 − pLL))] in
  let ftres = mkSimpleProb
    [((weaken e,    L, A, B),         pLL),
     ((FS e,        H, A, B), (1 − pLL))] in
  let ffres = mkSimpleProb
    [((weaken e,    L, A, B),         pLL),
     ((FS e,        H, A, B), (1 − pLL))] in
  case (t ⩽ crN) of
  True  ⇒ case (fromFin e ⩽ crE) of
             True  ⇒ trim ttres
             False ⇒ trim tfres
  False ⇒ case (fromFin e ⩽ crE) of
             True  ⇒ trim ftres
             False ⇒ trim ffres
SequentialDecisionProblems.CoreTheory.nexts t (e, L, A, B) H =
  let ttres = mkSimpleProb
    [((weaken e,    L, A, B), (1 −  pHL)),
     ((FS e,        H, A, B),       pHL) ] in
  let tfres = mkSimpleProb
    [((weaken e,    L, A, B), (1 −  pHL)),
     ((FS e,        H, A, B),       pHL) ] in
  let ftres = mkSimpleProb
    [((weaken e,    L, A, B), (1 −  pHL)),
     ((FS e,        H, A, B),       pHL) ] in
  let ffres = mkSimpleProb
    [((weaken e,    L, A, B), (1 −  pHL)),
     ((FS e,        H, A, B),       pHL) ] in
  case (t ⩽ crN) of
  True  ⇒ case (fromFin e ⩽ crE) of
             True  ⇒ trim ttres
             False ⇒ trim tfres
  False ⇒ case (fromFin e ⩽ crE) of
             True  ⇒ trim ftres
             False ⇒ trim ffres
```

11.4 Val and LTE:

Values of type Val are just non-negative double precision floating point numbers, addition, zero and $(\leqslant)$ are defined accordingly:

$$Val = NonNegDouble.NonNegDouble$$

$$plus = NonNegDouble.Operations.plus$$

$$zero = fromInteger@\{NumNonNegDouble\}\ 0$$

$$(\leqslant) = NonNegDouble.Predicates.LTE$$

$$reflexiveLTE = NonNegDouble.LTEProperties.reflexiveLTE$$

$$transitiveLTE = NonNegDouble.LTEProperties.transitiveLTE$$

$$monotonePlusLTE = NonNegDouble.LTEProperties.monotonePlusLTE$$

$$totalPreorderLTE = NonNegDouble.LTEProperties.totalPreorderLTE$$

11.5 Reward function

The idea is that being in a good world yields one unit of benefits per step and being in a bad world yield less benefits. These are defined by the ratio $badOverGood$.

The ratio between the benefits in a bad world and the benefits in a good world:

$$badOverGood : NonNegDouble$$
$$badOverGood = cast\ 0.89$$

$$check05 : badOverGood \leqslant 1 \quad \text{-- semantic check}$$
$$check05 = MkLTE\ Oh$$

Emitting GHGs also brings benefits. These are a fraction of the step benefits in a good world and low emissions bring less benefits than high emissions:

The ratio between low emissions benefits and step benefits in a good world, when effective technologies for reducing GHG emissions are unavailable:

$$lowOverGoodUnavailable : NonNegDouble$$
$$lowOverGoodUnavailable = cast\ 0.1$$

$$check06 \ : \ lowOverGoodUnavailable \leqslant 1 \quad \text{-- semantic check}$$
$$check06 = MkLTE \ Oh$$

The ratio between low emissions benefits and step benefits in a good world, when effective technologies for reducing GHG emissions are available:

$$lowOverGoodAvailable \ : \ NonNegDouble$$
$$lowOverGoodAvailable = cast \ 0.2$$

$$check07 \ : \ lowOverGoodAvailable \leqslant 1 \quad \text{-- semantic check}$$
$$check07 = MkLTE \ Oh$$
$$check08 \ : \ lowOverGoodUnavailable \leqslant lowOverGoodAvailable \quad \text{-- semantic check}$$
$$check08 = MkLTE \ Oh$$

The ratio between high emissions benefits and step benefits in a good world:

$$highOverGood \ : \ NonNegDouble$$
$$highOverGood = cast \ 0.3$$

$$check09 \ : \ highOverGood \leqslant 1 \quad \text{-- semantic check}$$
$$check09 = MkLTE \ Oh$$

$$check10 \ : \ lowOverGoodAvailable \leqslant highOverGood \quad \text{-- semantic check}$$
$$check10 = MkLTE \ Oh$$

The rewards only depend on the next state, not on the current state or on the selected control:

$$\begin{aligned}
&using \ implementation \ NumNonNegDouble \\
&\quad reward \ t \ x \ y \ (e, H, U, G) = 1 \qquad\qquad\quad\;\, + 1 * highOverGood \\
&\quad reward \ t \ x \ y \ (e, H, U, B) = 1 * badOverGood + 1 * highOverGood \\
&\quad reward \ t \ x \ y \ (e, H, A, G) = 1 \qquad\qquad\quad\;\, + 1 * highOverGood \\
&\quad reward \ t \ x \ y \ (e, H, A, B) = 1 * badOverGood + 1 * highOverGood \\
&\quad reward \ t \ x \ y \ (e, L, U, G) = 1 \qquad\qquad\quad\;\, + 1 * lowOverGoodUnavailable \\
&\quad reward \ t \ x \ y \ (e, L, U, B) = 1 * badOverGood + 1 * lowOverGoodUnavailable \\
&\quad reward \ t \ x \ y \ (e, L, A, G) = 1 \qquad\qquad\quad\;\, + 1 * lowOverGoodAvailable \\
&\quad reward \ t \ x \ y \ (e, L, A, B) = 1 * badOverGood + 1 * lowOverGoodAvailable
\end{aligned}$$

11.6 Completing the specification

In order to apply the verified, generic backwards induction algorithm of *CoreTheory* to compute optimal policies for our problem, we have to explain how the decision maker accounts for uncertainties on rewards induced by uncertainties in the transition function. We assume that the decision maker measures uncertain rewards by their expected value:

$$meas = expectedValue$$

$$measMon = monotoneExpectedValue$$

Further on, we have to implement the notions of viability and reachability. We start by positing that all states are viable for any number of steps (remember $Viable : (n : \mathbb{N}) \to X\ t \to Type$):

$$Viable\ n\ x = Unit$$

From this definition, it trivially follows that all elements of an arbitrary list of states are viable for an arbitrary number of steps:

$$\begin{aligned}
&viableLemma : \{t, n : \mathbb{N}\} \to (xs : List\ (State\ t)) \to All\ (Viable\ n)\ xs \\
&viableLemma\ \ Nil \qquad = Nil \\
&viableLemma\ (x :: xs) = () :: (viableLemma\ xs)
\end{aligned}$$

This fact and the (less trivial) result that simple probability distributions are never empty, see $nonEmptyLemma$ in $MonadicProperties$ in $SimpleProb$, allows us to show that the above definition of $Viable$ fulfills $viableSpec1$ (remember that $viableSpec1$ is of type $(x : X\ t) \to Viable\ (S\ n)\ x \to GoodCtrl\ t\ x$):

$$\begin{aligned}
&viableSpec1\ \{t\}\ \{n\}\ s\ v = \\
&\quad MkSigma\ H\ (ne, av)\ \mathbf{where} \\
&\qquad ne : NotEmpty\ (nexts\ t\ s\ H) \\
&\qquad ne = nonEmptyLemma\ (nexts\ t\ s\ H) \\
&\qquad av : All\ (Viable\ n)\ (nexts\ t\ s\ H) \\
&\qquad av = viableLemma\ (support\ (nexts\ t\ s\ H))
\end{aligned}$$

Because we have taken $Viable\ n\ x$ to be the singleton type, $Viable$ is finite and decidable:

-- SequentialDecisionProblems.Utils.finiteViable n x = finiteUnit

$$decidableViable\ n\ x = decidableUnit$$

For reachability, we proceed in a similar way. We say that all states are reachable

$$Reachable\ x' = Unit$$

which immediately implies (remember that $reachableSpec1$ is of type $(x : X\ t) \to Reachable\ x \to (y : Y\ t\ x) \to All\ Reachable\ (nexts\ t\ x\ y)$):

$$\begin{aligned}
&reachableSpec1\ \{t\}\ x\ r\ y = all\ (nexts\ t\ x\ y)\ \mathbf{where} \\
&\quad all : (sp : SimpleProb\ (State\ (S\ t))) \to All\ Reachable\ sp \\
&\quad all\ sp = all'\ (support\ sp)\ \mathbf{where} \\
&\qquad all' : (xs : List\ (State\ (S\ t))) \to Data.List.Quantifiers.All\ Reachable\ xs \\
&\qquad all'\ \ Nil \qquad = Nil \\
&\qquad all'\ (x :: xs) = () :: (all'\ xs)
\end{aligned}$$

and decidability of *Reachable*:

$$decidableReachable\ x = decidableUnit$$

Finally, we have to show that controls are finite (remember that *finiteCtrl* is of type $(x : X\ t) \to Finite\ (Y\ t\ x)$):

$$finiteCtrl\ t = finiteLowHigh$$

and, in order to use the fast, tail-recursive tabulated version of backwards induction, that states are finite:

$$finiteState\ t = finiteTuple4\ finiteFin\ finiteLowHigh\ finiteAvailableUnavailable\ finiteGoodBad$$

11.7 Optimal policies and possible state-control sequences

We can now apply the results of the *CoreTheory* and of the *FullTheory* from *SequentialDecisionProblems* to compute verified optimal policies, possible state-control sequences, etc. We want to be able to show the outcome of the decision process. This requires implementing functions to print states and controls:

```
showState {t} (e, H, U, G) = "(" ++ show (finToNat e) ++ ",H,U,G)"
showState {t} (e, H, U, B) = "(" ++ show (finToNat e) ++ ",H,U,B)"
showState {t} (e, H, A, G) = "(" ++ show (finToNat e) ++ ",H,A,G)"
showState {t} (e, H, A, B) = "(" ++ show (finToNat e) ++ ",H,A,B)"
showState {t} (e, L, U, G) = "(" ++ show (finToNat e) ++ ",L,U,G)"
showState {t} (e, L, U, B) = "(" ++ show (finToNat e) ++ ",L,U,B)"
showState {t} (e, L, A, G) = "(" ++ show (finToNat e) ++ ",L,A,G)"
showState {t} (e, L, A, B) = "(" ++ show (finToNat e) ++ ",L,A,B)"

showCtrl {t} {x} L = "L"
showCtrl {t} {x} H = "H"
```

With these in place, we can implement a program that reads the number of decision steps from the command line, computes a verified optimal policy sequence and outputs some statistics of possible trajectories and expected sum of rewards.

```
using implementation ShowNonNegDouble
  partial
  computation : {[STDIO]} Eff ()
  computation =
    do putStr ("enter number of steps:\n")
       nSteps ← getNat
       putStrLn "nSteps (number of decision steps):"
```

```
putStrLn ("    " ++ show nSteps)
putStrLn "computing optimal policies ..."
ps ← pure (tabTailRecursiveBackwardsInduction Z nSteps)
putStrLn "computing possible state-control sequences ..."
mxys ← pure (possibleStateCtrlSeqs (FZ, H, U, G) () () ps)
putStrLn "pairing possible state-control sequences with their values ..."
mxysv ← pure (possibleStateCtrlSeqsRewards' mxys)
putStrLn "computing (naively) the number of possible state-control sequences ..."
n ← pure (length (toList mxysv))
putStrLn "number of possible state-control sequences:"
putStrLn ("    " ++ show n)
putStrLn "computing (naively) the most probable state-control sequence ..."
xysv ← pure (naiveMostProbableProb mxysv)
putStrLn "most probable state-control sequence and its probability:"
putStrLn ("    " ++ show xysv)
putStrLn "sorting (naively) the possible state-control sequences ..."
xysvs ← pure (naiveSortToList mxysv)
putStrLn "most probable state-control sequences (first 3) and their probabilities:"
putStrLn (showlong (take 3 xysvs))
putStrLn "measure of possible rewards:"
putStrLn ("    " ++ show (meas (SequentialDecisionProblems.CoreTheory.fmap snd mxysv)))
putStrLn "done!"
```

For a more comprehensive implementation, see *EmissionsGame2* in *SequentialDecisionProblems.applications*.

Exercise 11.1. Compile this program with "make Lecture11.exe" from the command line. Run the program for 0, 1, 2, 4, 8 and 9 decision steps and annotate the run time. Put forward an hypothesis about the run time complexity in the number of decision steps. Check your hypothesis.

```
partial
main : IO ()
main = run computation
```

References

[1] N. Botta, P. Jansson, and C. Ionescu. The impact of uncertainty on optimal emission policies. *Earth System Dynamics*, 9(2):525–542, 2018.

[2] Nicola Botta. IdrisLibs. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`, 2016–2018.

Extra-Lecture 1: Logic and Proofs-as-Programs

POTSDAM INSTITUTE FOR
CLIMATE IMPACT RESEARCH

LOGIC AND PROOFS-AS-PROGRAMS

MARCH 2020

Leibniz Association

TIPPING POINTS IN THE EARTH SYSTEM

Horizon 2020
Programme

Plan for the lecture

- Historical sketch and general overview
- Brief introduction to formal logic
  - Proposistional Logic
  - Predicate Logic
- Gentzen systems and proofs-as-programs
- Wrap Up

2

Plan

Historical sketch and general overview

Brief introduction to formal logic
- Propositional Logic
- Predicate Logic

Gentzen systems and proofs-as-programs

Wrap Up

P I K

3

Leibniz's dream

Leibniz, early 18th century:

- **"calculus ratiocinator"**
  a framework for universal logical calculation
- **"characteristica universalis"**
  a universal language to be used within the above framework

Not in his lifetime, but...

- Considered as early **anticipation of modern logic** emerging in the middle of the 19th century (Boole, Peirce, De Morgan...)
- Frege, 1879: seeks to **implement Leibniz's vision** in his seminal "Begriffsschrift" ($\sim$ "conceptual language") in which he presents the **propositional calculus** and **quantification theory**
- Cantor, 1874: proposes (naïve) **set theory** as foundation of mathematics

P I K

4

"Foundational crisis of mathematics"

Beginning of the 20th century marked by controversy about the foundations of mathematics.

- Math had become more and more abstract at the end of 19th century leading to serious controversy
- Necessity for **verified foundation**
- Frege's system and Cantor's set theory turned out to be **inconsistent**
- Example: **Russell's paradox –** in Cantor's "naïve" set theory with unrestricted comprehension axiom one can form the set

$$A = \{x \mid \neg(x \in x)\}$$

"the set of all sets that do not contain themselves"
But what about A? Does it contain itself or not?

P I K

5

Dealing with paradoxes

Several proposals to overcome flaws in attempts for formal foundations of mathematics:

- Hilbert: **Proof theory** and search for consistency proof ("Hilbert's program")
- Russell: **Ramified type theory**
- Zermelo: **Axiomatic set theory**

Another, more radical response: Brouwer questions validity of classical axioms like the **tertium non datur**, and initiates **intuitionistic** logic/ **constructive** mathematics
where for a logical proposition P

proof of *P* = *construction of a witness* for *P*

P I K

6

Timeline of modern type theories

Logic and **computation**

- Natural Deduction — Gentzen, 1935
- λ-calculus (logical paradox amounting to non-terminating computation) — Church, 1930s
- Simple Theory of Types — Church, 1940
- Formulas as types ("Curry-Howard-correspondence") — Howard, 1969
- 2nd Order λ-calculus — Girard/Reynolds, 1971/74
- Dependent Type Theory — Martin-Löf, 1970s
- Univalent Type Theory/ Homotopy Type Theory — Voevodsky/ Awody&Warren, $\sim$ 2006

7

Curry-Howard-correspondence

- Correspondence between Gentzen's **Intuitionistic Natural Deduction** ($\mathcal{NJ}$) and Church's **simply-typed** λ **calculus** (STL)
- Three levels of correspondence:

propositions	as	types
proofs	as	programs
simplification of proofs	as	evaluation

- Correspondence also with **Cartesian Closed Categories** — Lambek, 1980
- Similar correspondences for many logics; setting in this talk: **Higher Order Intuitionistic Logic** $\simeq$ **Dependent Type Theory**

8

More correspondences

Curry-Howard-Lambek-Correspondence **extends in several ways**:

- **Other logics**: E.g. Linear Logic, Linear λ calculus and Monoidal Closed Categories
- **Higher Order Logic**: Martin-Löf Type Theory/ modern implementations of Dependent Type Theory build on the correspondence and extend it to higher order logic with inductive and coinductive definition and reasoning principles (where recursive programs correspond to proofs by induction)
- **Abstract homotopy theory**: Equality proofs in Intensional Martin-Löf Type Theory correspond to paths in topological spaces

P I K

9

Programming languages based on type theory

- Provide a **framework for both programming and mathematical specification**: at the same time "theorem prover" and "programming language", they are powerful tools for **program verification**
- Via the **propositions-as-types** and **proofs-as-programs** correspondence, the data types amount to mathematical statements and programs to their proofs
- The rigorous setting reduces the "semantic gap" between mathematical model and implementation and allows to **ensure meta-theoretical properties of all well-typed programs**
- Examples: Coq, Agda, Idris, Lean

P I K

10

Mathematics: Computer-verified proofs

Examples of large-scale formalization projects which have been carried out within implementations of (different) type theories:

- **The four color theorem**, in Coq Gonthier et al., 2005
- **The Feit-Thomson ("odd order") theorem**, in Coq Gonthier et al., 2012
- **The Kepler conjecture**, in Isabelle/HOL Hales et al., 2014

Broader formalization projects:

- **Univalent Foundations** – formalizatiion of various branches of mathematics initiated by Voevodsky and based on his Univalence Axiom, in Coq Ahrens et al.
- Mathematical library with ambition to **formalize state of the art number theory**, in Lean Buzzard et al.

P I K

11

Plan

Historical sketch and general overview

Brief introduction to formal logic
 Proposistional Logic
 Predicate Logic

Gentzen systems and proofs-as-programs

Wrap Up

P I K

12

Formal logic

- You already have a basic hands-on idea of logical statements from the main lectures
- We first take a look at a standard, set-theory based introduction to propositional logic as can be found in textbooks such as [18]
- We then take a look at how a similar definition might look in Idris, following the "DSL of math" approach by Ionescu and Jansson [11]

P I K

13

Plan

Historical sketch and general overview

Brief introduction to formal logic

Propositional Logic

Predicate Logic

Gentzen systems and proofs-as-programs

Wrap Up

P I K

14

The language of propositional logic I

Definition

The language of propositional logic has an alphabet consisting of

- proposition symbols: $A, B, C, \ldots$
- connectives: $\top, \bot, \Rightarrow, \wedge, \vee, \neg$
- auxiliary symbols: (,)

The proposition symbols, $\top$ and $\bot$ stand for indecomposable propositions that are usually called atoms or atomic propositions.

The propositional connectives

The connectives go by the following names and sometimes by some other choice of notation, here are some examples:

conjunction	$\wedge$	*AND*	&&
disjunction	$\vee$	*OR*	$\|\|$
implication	$\Rightarrow$	*IMPLIES*	$\supset$
negation	$\neg$	*NEG*	$\sim$
truth	$\top$	*TRUE*	1
falsity	$\bot$	*FALSE*	0

The language of propositional logic II

We have to specify which strings over the alphabet for propositional logic are what we consider well-formed propositions:

Definition

The set *PROP* of propositions is the smallest set X with the properties

(i) $A \in X$ *for all* proposition symbols A,

(ii) $\top \in X$,

(iii) $\bot \in X$,

(iv) $\varphi, \psi \in X$ *implies* $(\varphi \Rightarrow \psi) \in X$

(v) $\varphi, \psi \in X$ *implies* $(\varphi \wedge \psi) \in X, (\varphi \vee \psi) \in X, (\varphi \Rightarrow \psi) \in X$

(vi) $\varphi, \psi \in X$ *implies* $(\varphi \vee \psi) \in X$,

(vii) $\varphi \in X$ *implies* $(\neg\varphi) \in X$

P I K

17

Examples and convention

Example

The following are examples of well-formed formulas:

$$(A \Rightarrow (B \Rightarrow C))$$

$$(A \vee \bot)$$

$$(\neg(A \wedge B))$$

Exercise: Give examples of well-formed propositions involving the other connectives.

Convention: In order to avoid writing too many brackets, one usually defines precedence rules for the connectives and allows to omit brackets if there is no ambiguity.

(Standard: $\wedge$ binds tighter than $\vee$ which binds tighter than $\Rightarrow$; $\Rightarrow$ associates to the right)

P I K

18

Proving properties of propositions

To prove properties of propositions one reasons inductively, proceeding in analogy to the definition of propositions:

Start with the atoms (base cases) and then deal with the composite propositions (step cases).

This proof method is justified by the following theorem, which provides the induction principle for *PROP*:

(next slide)

P I K

19

Induction principle

Theorem

Let P be a property, then $P(\varphi)$ holds for all $\varphi \in PROP$ if

(i) $P(A)$, *for all proposition symbols A,*

(ii) $P(\top)$,

(iii) $P(\bot)$,

(iv) $P(\varphi)$ *and* $P(\psi)$ *implies* $P((\varphi \Rightarrow \psi))$,

(v) $P(\varphi)$ *and* $P(\psi)$ *implies* $P((\varphi \wedge \psi))$,

(vi) $P(\varphi)$ *and* $P(\psi)$ *implies* $P((\varphi \vee \psi))$,

(vii) $P(\varphi)$ *implies* $P((\neg\varphi))$.

P I K

20

Proof of the induction principle

Proof.

Let $X = \{\varphi \in PROP \mid P(\varphi)\}$, then X satisfies the conditions $(i) - (vii)$ in the definition of the set $PROP$ above.

As by definition $PROP$ is the smallest set satisfying these conditions, it must hold that $PROP \subseteq X$, and thus for all $\varphi \in PROP, P(\varphi)$ holds. □

P I K

21

Proving properties of propositions

We already have seen a proof by induction in the example for equational reasoning in section 2.5 of the regular lectures.

Recall that in this example we worked with the recursive definition of the exponentiation function:

$$\forall x \in \mathbb{R}, x^0 = 1 \qquad \text{and}$$

$$\forall x \in \mathbb{R}, m \in \mathbb{N}, x^{1+m} = x * x^m$$

An analogous principle of definition exists as well for the syntax of propositional logic:

(next slide)

P I K

22

Definition by recursion

Theorem

Let mappings $H_{\Rightarrow}, H_{\wedge}, H_{\vee} : X^2 \to X$, $H_{\neg} : X \to X$ and $H_{\top}, H_{\bot} : X$ be given and let H_{ps} be a mapping from the set of proposition symbols into X, then there exists exactly one mapping $F : PROP \to X$ such that

$$\begin{cases} F(A) & = H_{ps}(A) \quad \text{for } A \text{ proposition symbol} \\ F(\varphi \Rightarrow \psi) & = H_{\Rightarrow}(F(\varphi), F(\psi)) \\ F(\varphi \wedge \psi) & = H_{\wedge}(F(\varphi), F(\psi)) \\ F(\varphi \vee \psi) & = H_{\vee}(F(\varphi), F(\psi)) \\ F(\neg\varphi) & = H_{\neg}(F(\varphi)) \end{cases}$$

(In general we would still have to prove the existence of a unique function satisfying the above equations.)

P I K

23

Example for definition by recursion

As an example, we might define the number of brackets $b(\varphi)$ of a proposition φ, $b : PROP \to \mathbb{N}$:

Example

$$\begin{cases} b(A) & = 0 \quad \text{for } A \text{ proposition symbol} \\ b(\top) & = 0 \\ b(\bot) & = 0 \\ b(\varphi \Rightarrow \psi) & = b(\varphi) + b(\psi) + 2 \\ b(\varphi \wedge \psi) & = b(\varphi) + b(\psi) + 2 \\ b(\varphi \vee \psi) & = b(\varphi) + b(\psi) + 2 \\ b(\neg\varphi) & = b(\varphi) + 1 \end{cases}$$

P I K

24

A first look at semantics

- Having defined the syntax of propositional logic and we now ask about its *meaning*, that is: its semantics
- The probably best-known semantics is the classical boolean semantics, the so-called truth-table semantics
- In this interpretation, the propositional variables can take values in $\{0, 1\}$ and the logical connectives are interpreted as boolean functions

P I K

25

Classical logic: truth-table-semantics I

As the name suggests, the interpretation is usually given by tables:

A	B	$A \Rightarrow B$	$A \vee B$	$A \wedge B$	$\neg A$
0	0	1	0	0	1
0	1	1	1	0	1
1	0	0	1	0	0
1	1	1	1	1	0

$\perp$ is interpreted as constant 0 and $\top$ as constant 1.

P I K

26

Classical logic: truth-table-semantics II

As only the observable results of applying these functions are relevant, operators are easily interdefinable.

Compare e.g. the following table with the one on the last slide:

A	B	$\neg A \vee B$	$\neg A \wedge \neg B$	$\neg A \vee \neg B$	$A \Rightarrow 0$
0	0	1	0	0	1
0	1	1	1	0	1
1	0	0	1	0	0
1	1	1	1	1	0

P I K

27

Classical logic: truth-table-semantics III

To be more formal, we can make use of the principle of definition by recursion, as in the earlier example.

Let a be a mapping from the set of proposition symbols to $\{0, 1\}$. Then we can define an interpretation or evaluation function $e : PROP \to \{0, 1\}$ by:

$$\begin{cases} e(A) & = a(A) \quad \text{for } A \text{ proposition symbol} \\ e(\top) & = 1 \\ e(\bot) & = 0 \\ e(\varphi \Rightarrow \psi) & = max((1 - e(\varphi), e(\psi)) \\ e(\varphi \wedge \psi) & = min(e(\varphi), e(\psi)) \\ e(\varphi \vee \psi) & = max(e(\varphi), e(\psi)) \\ e(\neg\varphi) & = 1 - e(\varphi) \end{cases}$$

P I K

28

Object- and meta-language I

- All the definitions we have seen so far use a meta-language (e.g. *the smallest set X with the properties..*,$\in$, *implies*, . . .) to make definitions and reason about an object language
- The meta language in this case is a semi-formal set theory while the object language is the language of propositional logic, a newly defined syntax
- *Proof by induction* and *definition by recursion* play a crucial role

P I K

29

Object and meta language II

- When we use an expressive type theory like the one underlying Idris as formal meta-language to make definitions as the above, meta-theoretic statements are now part of the programming language
- Syntaxes as the language of propositional logic can be defined as inductive data types (using the "data" keyword in Idris)
- Moreover, the underlying type theory provides us with the principle of definition by recursion "for free": In functional programming it is known as pattern matching and is the standard way to define functions out of inductive data types
- **As an aside:** in Idris, we neither have to bother with defining an alphabet and well-formed strings over it separately: values of inductive types are syntax trees, the alphabet is given implicitly with the definition

P I K

30

A DSL of propositional logic – Syntax

Thus, we can define the syntax of propositional logic as abstract data type in Idris. This might be seen as implementing a domain-specific language of propositional logic:

```
> data PropSyntax : Type where
>   PropAtom     : String -> PropSyntax
>   PropFalse    : PropSyntax
>   PropTrue     : PropSyntax
>   PropNot      : PropSyntax -> PropSyntax
>   PropAnd      : PropSyntax -> PropSyntax -> PropSyntax
>   PropOr       : PropSyntax -> PropSyntax -> PropSyntax
>   PropImplies  : PropSyntax -> PropSyntax -> PropSyntax
```

P I K

31

A DSL of propositional logic – Syntax

Example

```
> pr1 : PropSyntax
> pr1 = PropOr (PropAtom "A") (PropNot (PropAtom "A"))
>
> pr2 : PropSyntax
> pr2 = PropImplies (PropAtom "A") (PropAtom "B")
>
> pr3 : PropSyntax
> pr3 = PropImplies (PropAnd (PropAtom "A") (PropAtom "B"))
>       (PropAnd (PropAtom "B") (PropAtom "A"))
```

P I K

32

A DSL of propositional logic – Syntax

We can define the above truth table semantics by pattern matching: an evaluation function `evalPC : PropSyntax -> Bool` using Idris' data type for booleans (with constructors `True` and `False`) and Boolean functions `not`, `&&`, || from the Idris standard library.

```
> evalAt : String -> Bool
>
> evalPC : PropSyntax -> Bool
> evalPC PropFalse = False
> evalPC PropTrue = True
> evalPC (PropNot     x  ) = not (evalPC x)
> evalPC (PropAnd     x y) = (evalPC x) && (evalPC y)
> evalPC (PropOr      x y) = (evalPC x) || (evalPC y)
> evalPC (PropImplies x y) = (not (evalPC x)) || (evalPC y)
> evalPC (PropAtom    s  ) = evalAt s
```

P I K

33

Denotational vs. operational semantics

- In terms of domain-specific languages, `evalPC` is a translation from a syntactic to a semantic domain: `PropSyntax` here is the syntactic domain (*abstract syntax*), `Bool` is the semantic domain
- Truth table semantics is a denotational semantics: What is important is just the *result* of evaluating a proposition – its *denotation*
- However, there is a more dymnamical way to look at semantics: Instead of asking just about the truth value of a proposition, one might ask about its proofs. This leads to the constructive or intuitionistic perception of logic (⇝ see the historical sketch)
- Classical and intuitionistic logic share the same syntax (although negation is not taken as primitive in the intuitionistic case) - but differ in their conception of what is an acceptable proof
- This is reflected in the interpretation of the logical operators

P I K

34

Brouwer-Heyting-Kolmogorov interpretation

Definition

A primitive object p as proof of an atomic proposition A is given by a construction

$$p \text{ proves } A$$

The meaning of complex proofs is inductively defined by:

- There is a *unique* construction of $\top$.
- There is *no* construction of $\bot$.
- A proof of $P \Rightarrow Q$ is a construction f which *transforms* a proof Φ of P into a proof $f(\Phi)$ of Q.
- A proof of $P_0 \wedge P_1$ is a pair $\langle \Phi_0, \Phi_1 \rangle$ consisting of a proof Φ_0 of P_0 *and* a proof Φ_1 of P_1.
- A proof of a disjunction $P_0 \vee P_1$ is a pair $\langle b, \Phi \rangle$ where $b = 0$ and Φ is a proof of P_0 *or* $b = 1$ and Φ is a proof of P_1.

P I K

35

Construction

- Note that in the BHK semantics it is not further specified what exactly *is* a construction of an atomic proposition – it is assumed to be given by a context
- In arithmetics the proof of a formula $n = m$ might e.g. be a calculation that reduces both sides of the equality to the same number
- However, this underspecification of the notion of "construction" leads to different possible interpretations depending on the chosen underlying definition
- One possibility to concretely define what is meant by "construction" amounts to the proofs-as-programs-correspondence between intuitionistic logic and the simply-typed λ calculus

P I K

36

Plan

Historical sketch and general overview

Brief introduction to formal logic
- Propositional Logic
- Predicate Logic

Gentzen systems and proofs-as-programs

Wrap Up

A glimpse of predicate logic

- We will just try to give a very rough idea of predicate logic
- In first order logic (FOL), formulas may contain terms, e.g. $\forall n.Sn > n$
- Terms are either variables (like n) or function (or more generally relation) symbols (like S, $>$) applied to a suitable number of terms.
- The function symbols often formalize a specific domain, e.g. number theory.
- First order quantification is over term variables of a specific domain (encoded as strings in the example below), and *not over propositions* – this would require higher order quantification

Informally: meaning of the quantifiers

Now, what is the semantics of the quantifiers? What is a proof of

$$\forall n.\exists m.m > n?$$

- The universal quantifier $\forall$ can be seen as *generalization of the conjunction* $\wedge$:
$$\forall x.Px \simeq Px_1 \wedge Px_2 \wedge \ldots$$
- This is ok in case there are only finitely many such x, but would need an infinite number of proofs in the infinite case!
- This problem is usually solved by introducing an arbitrary fresh variable x' and proving $P\,x'$ without knowing anything about x
- Thus, a proof of $\forall x.Px$ can be seen as a function p from terms to proofs such that px is a proof of Px for each term x
- Similarly, $\exists x.Px$ can be seen as potentially *infinite version of disjunction* $\vee$:
$$\exists x.Px \simeq Px_1 \vee Px_s \vee \ldots$$

P I K

39

A DSL of a first order language – Syntax

In Idris, we might define a first order syntax with function symbols $+$ and $*$ as follows:

```
> data Num : Type where
>   FOLPlus : Num -> Num -> Num
>   FOLMult : Num -> Num -> Num
>   FOLVar  : String -> Num

> data FOL : Type where
>   FOLFalse   : FOL
>   FOLTrue    : FOL
>   FOLNot     : FOL -> FOL
>   FOLAnd     : FOL -> FOL -> FOL
>   FOLOr      : FOL -> FOL -> FOL
>   FOLImplies : FOL -> FOL -> FOL
>   FOLEq      : Num -> Num -> FOL
>   FOLForall  : String -> FOL -> FOL
>   FOLExists  : String -> FOL -> FOL
```

P I K

40

BHK interpretation for predicate logic

The BHK-semantics for intuitionistic propositional logic *extends* to intuitionistic predicate logic. One assumes that individual variables range over some *basic* domain D (which means that no further proof for a fact $d \in D$ is required).

- For atomic predicates $p(x)$, there is for every $d \in D$ for which $P(d)$ has a proof a construction $p(d)$.
- A proof of $\forall x.P(x)$ is a construction transforming any $d \in D$ into a proof $p(d)$ of $P(d)$.
- A proof p of $\exists x.P(x)$ is a pair $\langle d, q \rangle$ where q is a proof of $P(d)$ (that is, free occurrences of the variable x in $P(x)$ are replaced by d).

P I K

41

Plan

Historical sketch and general overview

Brief introduction to formal logic
  Propositional Logic
  Predicate Logic

Gentzen systems and proofs-as-programs

Wrap Up

P I K

42

Curry-Howard

- As put mentioned in the historical sketch, the so-called Curry-Howard-correspondence amounts to a stuctural correspondence between proofs in Gentzen's natural deduction proof system $\mathcal{NJ}$ and typing derivations in Church's simply-typed λ calculus STL
- As such it forms the core of which Martin-Löf type theory is a vast extension
- We will look at at the $\Rightarrow, \wedge$-fragment of $\mathcal{NJ}$ and the corresponding fragment of STL and then see a small example in order to get a more concrete idea of the correspondence

P I K

43

Natural Deduction

Hypothesis: A

Introduction:

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-}I$$

$$\frac{\begin{array}{c}[A]^x \\ \vdots \\ B\end{array}}{A \Rightarrow B}\ \Rightarrow\text{-}I_x$$

Elimination:

$$\frac{A \wedge B}{A}\ \wedge\text{-}E_1 \qquad \frac{A \wedge B}{B}\ \wedge\text{-}E_2$$

$$\frac{A \Rightarrow B \qquad A}{B}\ \Rightarrow\text{-}E$$

P I K

44

λ-calculus with products

The λ-calculus can be seen as a minimal functional programming language.

Its syntax in Backus-Naur-form:

$$s, t ::== x \mid \lambda x.t \mid (s\ t) \mid \langle s, t\rangle \mid \mathtt{pr}_1\ t \mid \mathtt{pr}_2\ t$$

(Church first defined the λ-calculus without types; he introduced types to the language to avoid "paradoxical" terms.)

45

Simply-typed λ-calculus

Hypothesis: $x : A$

Introduction:

$$\frac{s : A \qquad t : B}{\langle s, t\rangle : A \times B}\ \times\text{-}I$$

$$\frac{\begin{array}{c}[x : A]^x \\ \vdots \\ t : B\end{array}}{\lambda x.t : A \to B}\ \to\text{-}I_x$$

Elimination:

$$\frac{t : A \times B}{\mathtt{pr}_1\ t : A}\ \times\text{-}E_1 \qquad \frac{t : A \times B}{\mathtt{pr}_2\ t : B}\ \times\text{-}E_2$$

$$\frac{s : A \to B \qquad t : A}{(s\ t) : B}\ \to\text{-}E$$

46

Example: proof

The $\mathcal{NJ}$ inference rules can now be combined to build proofs, e.g.:

$$\dfrac{\dfrac{\dfrac{[B \wedge A]^z}{A}\wedge\text{-}E_2 \qquad \dfrac{[B \wedge A]^z}{B}\wedge\text{-}E_1}{A \wedge B}\wedge\text{-}I}{(B \wedge A) \Rightarrow (A \wedge B)}\Rightarrow\text{-}I^z$$

Exchanging $\wedge$ against $\times$ and $\Rightarrow$ against $\rightarrow$ and annotating the proof tree with λ-terms . . .

Example: program

. . . we get the following typing derivation for the program $\lambda z.\, \langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z\rangle$:

$$\dfrac{\dfrac{\dfrac{z : [B \times A]^z}{\mathtt{pr}_2\, z : A}\times\text{-}E_2 \qquad \dfrac{[z : B \times A]^z}{\mathtt{pr}_1\, z : B}\times\text{-}E_2}{\langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z\rangle : A \times B}\times\text{-}I}{\lambda z.\, \langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z\rangle : (B \times A) \rightarrow (A \times B)}\rightarrow\text{-}I^z$$

By erasing the term annotations and exchanging again the connectives we would get back the original $\mathcal{NJ}$ proof.

Curry-Howard continued

- We have just seen two levels of the correspondence for the $\wedge, \Rightarrow / \times, \rightarrow$ fragments:
  - formulas and types
  - proof rules and typing rules
- But the correspondence extends to a third level:
  - proof simplification and program evaluation
- Let's have a look at the simplification rules and then another example . . .

P I K

49

Simplification of proofs

Proofs can be simplified, if they contain an application of an elimination rule for a connective after the application of an introduction rule for the same connective. Such an occurrence is called a cut.

$$\dfrac{\dfrac{\begin{array}{c}[A]^x\\ \vdots\\ B\end{array}}{A \Rightarrow B}\Rightarrow\text{-}I_x \qquad \begin{array}{c}\vdots\\ A\end{array}}{B}\Rightarrow\text{-}E \qquad\mapsto\qquad \begin{array}{c}\vdots\\ A\\ \vdots\\ B\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ A\end{array} \quad \begin{array}{c}\vdots\\ B\end{array}}{A \wedge B}\wedge\text{-}I}{A}\wedge\text{-}E_1 \quad\mapsto\quad \begin{array}{c}\vdots\\ A\end{array} \qquad\qquad \dfrac{\dfrac{\begin{array}{c}\vdots\\ A\end{array} \quad \begin{array}{c}\vdots\\ B\end{array}}{A \wedge B}\wedge\text{-}I}{B}\wedge\text{-}E_2 \quad\mapsto\quad \begin{array}{c}\vdots\\ B\end{array}$$

P I K

50

Simply-typed λ-calculus: Reduction

For the λ-calculus, evaluation of programs is defined by reduction rules for terms s, t:

$$((\lambda x.t)\ s) \mapsto t\{x := s\}$$

$$\mathtt{pr}_1\ \langle s, t\rangle \mapsto s$$

$$\mathtt{pr}_2\ \langle s, t\rangle \mapsto t$$

Expressions which can be reduced consist of a constructor term (*here*: lambda, pair) followed by a destructor term (*here*: application, projection).
These expressions are called redexes.
(More precisely, this is just one form of reduction, usually called β-reduction. But we do not look at other forms of reduction in this lecture.)

51

Example: Proof simplification

Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{[B \wedge A]^z}{A}\wedge\text{-}E_2 \qquad \dfrac{[B \wedge A]^z}{B}\wedge\text{-}E_1}{A \wedge B}\wedge\text{-}I}{(B \wedge A) \Rightarrow (A \wedge B)}\Rightarrow\text{-}I^z \qquad \dfrac{B \qquad A}{B \wedge A}\wedge\text{-}I}{A \wedge B}\Rightarrow\text{-}E$$

It is somewhat uneconomic and one would like to eliminate unnecessary detours from proofs to obtain proofs in a normal form which does not contain any cuts.

52

Example: Proof simplification

In fact, there is a systematic procedure to obtain such normal form proofs in $\mathcal{NJ}$. (It is called cut elimination.)
In our example, we can simplify as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{[B\wedge A]^z}{A}\,\wedge\text{-}E_2 \qquad \dfrac{[B\wedge A]^z}{B}\,\wedge\text{-}E_1}{A\wedge B}\,\wedge\text{-}I}{(B\wedge A)\Rightarrow(A\wedge B)}\,\Rightarrow\text{-}I^z \qquad \dfrac{B \qquad A}{B\wedge A}\,\wedge\text{-}I}{A\wedge B}\,\Rightarrow\text{-}E$$

simplifies to

$$\dfrac{\dfrac{\dfrac{B \qquad A}{B\wedge A}\,\wedge\text{-}I}{A}\,\wedge\text{-}E_2 \qquad \dfrac{\dfrac{B \qquad A}{B\wedge A}\,\wedge\text{-}I}{B}\,\wedge\text{-}E_1}{A\wedge B}\,\wedge\text{-}I$$

53

Example: Proof simplification

... which further simplifies:

$$\dfrac{\dfrac{\dfrac{B \qquad A}{B\wedge A}\,\wedge\text{-}I}{A}\,\wedge\text{-}E_2 \qquad \dfrac{\dfrac{B \qquad A}{B\wedge A}\,\wedge\text{-}I}{B}\,\wedge\text{-}E_1}{A\wedge B}\,\wedge\text{-}I$$

simplifies to

$$\dfrac{A \qquad B}{A\wedge B}\,\wedge\text{-}I$$

This proof cannot be made any shorter, it is cut-free.

54

Example: Proof simplification vs. program evaluation

Now we can again transform the original proof tree into a typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{z : [B \times A]^z}{\mathtt{pr}_2\, z : A}\,\times\text{-}E_2 \qquad \dfrac{[z : B \times A]^z}{\mathtt{pr}_1\, z : B}\,\times\text{-}E_1}{\langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z \rangle : A \times B}\,\times\text{-}I}{\lambda z.\, \langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z \rangle : (B \times A) \to (A \times B)}\,\to\text{-}I^z \qquad \dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,\times\text{-}I}{(\lambda z.\, \langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z \rangle)\, \langle y, x \rangle : A \times B}\,\to\text{-}E$$

It looks equally uneconomic... but this time simplification amounts to evaluation of λ-terms!

55

Example: Program evaluation

In STL the evaluation to a canonical form is called normalization and evaluation goes by the name reduction.

$$\dfrac{\dfrac{\dfrac{\dfrac{z : [B \times A]^z}{\mathtt{pr}_2\, z : A}\,\times\text{-}E_2 \qquad \dfrac{[z : B \times A]^z}{\mathtt{pr}_1\, z : B}\,\times\text{-}E_1}{\langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z \rangle : A \times B}\,\times\text{-}I}{\lambda z.\, \langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z \rangle : (B \times A) \to (A \times B)}\,\to\text{-}I^z \qquad \dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,\times\text{-}I}{(\lambda z.\, \langle \mathtt{pr}_2\, z, \mathtt{pr}_1\, z \rangle)\, \langle y, x \rangle : A \times B}\,\to\text{-}E$$

reduces to

$$\dfrac{\dfrac{\dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,\times\text{-}I}{\mathtt{pr}_2\, \langle y, x \rangle : A}\,\times\text{-}E_2 \qquad \dfrac{\dfrac{y : B \qquad x : A}{\langle y, x \rangle : B \times A}\,\times\text{-}I}{\mathtt{pr}_1\, \langle y, x \rangle : B}\,\times\text{-}E_1}{\langle \mathtt{pr}_2\, \langle y, x \rangle, \mathtt{pr}_1\, \langle y, x \rangle \rangle : A \times B}\,\times\text{-}I$$

56

Example: Proof simplification

... which further simplifies:

$$\dfrac{\dfrac{\dfrac{y : B \qquad x : A}{\langle y, x\rangle : B \times A}\times\text{-}I}{\mathtt{pr}_2\ \langle y, x\rangle : A}\times\text{-}E_2 \qquad \dfrac{\dfrac{y : B \qquad x : A}{\langle y, x\rangle : B \times A}\times\text{-}I}{\mathtt{pr}_1\ \langle y, x\rangle : B}\times\text{-}E_1}{\langle \mathtt{pr}_2\ \langle y, x\rangle, \mathtt{pr}_1\ \langle y, x\rangle\rangle : A \times B}\times\text{-}I$$

reduces to

$$\dfrac{x : A \qquad y : B}{\langle x, y\rangle : A \times B}\times\text{-}I$$

This program cannot be reduced any further, it is in normal form.

P I K

57

Plan

Historical sketch and general overview

Brief introduction to formal logic
- Proposistional Logic
- Predicate Logic

Gentzen systems and proofs-as-programs

Wrap Up

P I K

58

What have we done in this lecture?

- We have first sketched the historical roots of type theory and described the idea of proofs-as-programs without going into details
- We have then taken a brief look at a traditional approach to formal logic and contrasted it with a "domain-specific-language of logic" perspective; we have thereby seen that a language like Idris is not only based in logic, but also provides the tools that are required of a meta-language to implement and reason about different kinds of logic
- We have learned about two important notions of semantics, namely truth table semantics for classical logic and the Brouwer-Heyting-Kolmogorov interpretation for intuitionistic logic
- We have taken a look at the three levels of the Curry-Howard-correspondence restricted to the implication-conjunction fragment of propositional intuitionistic logic

P I K

59

Bibliography I

[1] Edwin Brady. *Type-Driven Development in Idris*. Manning Publications Co., 2017.

[2] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings CPP 2020*, pages 299–312. ACM, 2020.

[3] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 2:33, 346–366, 1932.

[4] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[5] Leonardo Mendonça de Moura et al. The lean theorem prover (system description). In *Automated Deduction - CADE-25 - Proceedings*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015.

[6] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935.

[7] Jean-Yves Girard. Une extension de interpretation de Gödel a analyse, et son application a elimination des coupures dans analyse et la theorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. Elsevier, 1971.

P I K

60

Bibliography II

[8] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In *ASCM 2007. Revised and Invited Papers*, volume 5081 of *LNCS*, page 333. Springer, 2007.

[9] Georges Gonthier et al. A machine-checked proof of the odd order theorem. In *ITP 2013. Proceedings*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[10] Thomas C. Hales et al. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[11] Patrik Jansson and Cezar Ionescu. Domain Specific Languages of Mathematics (DSLsofMath), 2014-2020.

[12] Joachim Lambek. Deductive systems and categories III. Cartesian closed categories, intuitionist propositional calculus, and combinatory logic. In *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 57–82. Springer, 1972.

[13] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[14] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[15] John C Reynolds. Towards a theory of type structure. 2012. Originally 1974.

[16] The Coq Development Team. The coq proof assistant, version 8.11.0, 2020.

P I K

61

Bibliography III

[17] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[18] Dirk van Dalen. *Logic and Structure*. Springer, 5th edition, 2004.

[19] Jan van Heijenoort, editor. *From Frege to Gödel: a Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, 1967. Reprinted 1971, 1976.

[20] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath – a computer-checked library of univalent mathematics. Available at `https://github.com/UniMath/UniMath`, 2017.

[21] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015.

[22] Ernst Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English translation, "Investigations in the foundations of set theory" in [19], pages 199–215.

P I K

62

Extra-Lecture 2: Functors and monads in category theory

POTSDAM INSTITUTE FOR CLIMATE IMPACT RESEARCH

CATEGORY THEORY

SOME BASICS FOR MONADIC DYNAMICAL SYSTEMS

MARCH 2020

Leibniz Association

TiPES TIPPING POINTS IN THE EARTH SYSTEM

Horizon 2020 Programme

Motivation: Monadic Dynamical Systems

The central notion underlying the theory of sequential decision problems under uncertainty [1, 2] is that of a monadic dynamical system [3].
The essential observation leading to this notion: Various types of uncertainties can be modeled uniformly using the concept of monad from category theory (non-determinism, probabilities, . . .)

- Monads were popularized in *Haskell* for encapsulating side effects [4, 5], e.g. exceptions, state, partiality, input/output . . .
- From the programming perspective, the approach amounts to implementing a small number of operators, usually called map, pure/return, join and bind
- In *Idris* [6] we can also prove that these operations fulfill the necessary axioms

BUT . . .

2

Motivation: Monadic Dynamical Systems

. . . to properly define **what a monad is**, we need to know some category-theoretical notions like

category, *functor*, *natural transformation*, . . .

To give a basic idea of category theory and the notions coming up in the context of monadic dynamical systems is the purpose of this lecture.

Plan

Introduction to basic concepts and terminology

Monads and adjunctions

Example: Deterministic dynamical systems

Plan

Introduction to basic concepts and terminology

Monads and adjunctions

Example: Deterministic dynamical systems

PIK

5

Concrete Categories

A somewhat informal definition:

Definition

A concrete category $\mathcal{C}$ is a collection of two kinds of entities, called objects and morphisms. The former are sets which are endowed with some kind of structure, and the latter are mappings between these sets which preserve that structure.

Among the morphisms, there is attached to each object X the identity mapping $\mathrm{id}_X : X \longrightarrow_{\mathcal{C}} X$ s.t. $\mathrm{id}_X(x) = x$ for all $x \in X$.

Morphisms $f : Y \longrightarrow_{\mathcal{C}} Y$ and $g : X \longrightarrow_{\mathcal{C}} Y$ may be composed to produce a morphism $f \circ_{\mathcal{C}} g : X \longrightarrow_{\mathcal{C}} Z$ such that $(f \circ_{\mathcal{C}} g)\, x = f(g(x))$ for all $x \in X$.

PIK

6

Concrete Categories - Examples

Example

- SET: the category of sets and total functions
- MON: with monoids (i.e. semigroups with a neutral element) as objects and homomorphisms as mappings which preserve the semigroup operation and the neutral element
- PRE: preordered sets (i.e. sets having a reflexive and transitive relation on them) as objects and monotone mappings which preserve this relation as morphisms
- TOP: topological spaces as objects and continuous functions as morphisms

PIK 7

(Directed) Metagraphs

Definition

A metagraph G consists of two classes: a class MOR_G of arrows (=oriented edges) and a class OBJ_G of objects (=nodes/vertices) and two mappings $\text{MOR}_G \rightrightarrows \text{OBJ}_G$ called **dom** (source) and **cod** (target).

$f : X \longrightarrow_G Y$ is then a notation for $(\mathbf{dom}\, f = X \;\wedge\; \mathbf{cod}\, f = Y)$.

A metagraph is said to be small if the classes of objects and morphisms are sets and then called (directed multi-) graph.

Example

GRPH, the category of small directed graphs, is another concrete category. Which property would you expect to be required of its morphisms?

PIK 8

Deductive systems and categories

Definition

A deductive system is a metagraph in which to each object X there is associated an identity arrow $\mathrm{id}_X : X. \longrightarrow_{\mathcal{C}} X$, and to each pair of arrows $f : Y \longrightarrow_{\mathcal{C}} Z$ and $g : X \longrightarrow_{\mathcal{C}} Y$ there is associated an arrow $f \circ_{\mathcal{C}} g : X \longrightarrow_{\mathcal{C}} Z$, the composition of f with g.

Definition

A category is a deductive system $\mathcal{C}$ in which the following equations hold for all $f : Y \longrightarrow_{\mathcal{C}} Z, g : X \longrightarrow_{\mathcal{C}} Y$ and $h : W \longrightarrow_{\mathcal{C}} X$:

$$\mathrm{id}_Z \circ_{\mathcal{C}} f = f = f \circ_{\mathcal{C}} \mathrm{id}_Y \tag{1}$$

$$(f \circ_{\mathcal{C}} g) \circ_{\mathcal{C}} h = f \circ_{\mathcal{C}} (g \circ_{\mathcal{C}} h) \tag{2}$$

A category is called small, if its underlying metagraph is.

P I K

9

GRPH

Definition

The category GRPH of small directed graphs is defined as follows:

Objects: directed graphs $G, G', G'', \ldots$

Morphisms: $f : G \longrightarrow_{\mathrm{GRPH}} G'$ are pairs $\langle f_M, f_O \rangle$ of morphisms $f_M : \mathrm{MOR}_G \longrightarrow_{\mathrm{SET}} \mathrm{MOR}_{G'}$ and $f_O : \mathrm{OBJ}_G \longrightarrow_{\mathrm{SET}} \mathrm{OBJ}_{G'}$ s.t. for $e : v \longrightarrow_G v'$ one has $f_M\, e : f_O\, v \longrightarrow_{G'} f_O\, v'$

Identity: $\mathrm{id}_G := \langle \mathrm{id}_{\mathrm{MOR}_G}, \mathrm{id}_{\mathrm{OBJ}_G} \rangle$

Composition: For $\langle f_M, f_O \rangle : G' \longrightarrow_{\mathrm{GRPH}} G''$ and $\langle g_M, g_O \rangle : G \longrightarrow_{\mathrm{GRPH}} G'$

$$\langle f_M, f_O \rangle \circ_{\mathrm{GRPH}} \langle g_M, g_O \rangle := \langle f_M \circ_{\mathrm{SET}} g_M, f_O \circ_{\mathrm{SET}} g_O \rangle$$

Exercise: Check that the identity and associativity laws hold!

P I K

10

Functors

To define a category CAT of small categories, we need to define an appropriate notion of morphism between categories.

We have already seen that the morphisms of a concrete category are supposed to preserve the structure of its objects. The additional structure in the case of categories amounts to its underlying graph structure, identity and composition.

Definition

A functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ is a morphism of metagraphs $\langle F_M, F_O \rangle$ (i.e. every arrow $f : X \longrightarrow_{\mathcal{C}} Y$ is mapped to an arrow $F_M\, f : F_O\, X \longrightarrow_{\mathcal{D}} F_O\, Y$), such that

$$F_M\, \mathrm{id}_X = \mathrm{id}_{F_O\, X} \tag{3}$$

$$F_M\, (f \circ_{\mathcal{C}} g) = F_M\, f \circ_{\mathcal{D}} F_M\, g \tag{4}$$

P I K

11

On functors and notation

When defining a functor F, we always have to define its object **and** its morphism part, F_O and F_M, respectively.

However, to avoid clutter, it is usual to **omit indices** if they are clear from the context.

This is why usually, **both** object and morphism parts of a functor F are simply denoted by F.

In **programming**, F_O and F_M however have to be defined separately and so this kind of overloading is not common. Instead the object part is usually given the name of the functor, e.g. *List*, and the morphism part is called *map* (or *fmap*).

This is the way you will see functors handled in Idris.

P I K

12

Functors : List Example

Example

$\mathit{List} : \textsc{Set} \longrightarrow \textsc{Set}$ is a functor $\langle \mathit{List}_M, \mathit{List}_O \rangle$, where List_O maps every set A to the underlying set of the **free monoid** $\langle \mathrm{List}\, A, [\,], ++ \rangle$ on this set and List_M amounts to the familiar *map* function.

- for $A \in \textsc{Obj}_{\textsc{Set}}$, $\mathit{List}_O\, A := \mathrm{List}\, A$
- for $f : A \longrightarrow_{\textsc{Set}} B$, $\mathit{List}_M\, f := \mathrm{mapList}\, f : \mathrm{List}\, A \longrightarrow_{\textsc{Set}} \mathrm{List}\, B$
- for all $A \in \textsc{Obj}_{\textsc{Set}}$, $\mathrm{mapList}\ \mathrm{id}_A = \mathrm{id}_{\mathrm{List}\, A}$
- for all $f : B \longrightarrow_{\textsc{Set}} C, g : A \longrightarrow_{\textsc{Set}} B$, $\mathrm{mapList}\,(f \circ_{\textsc{Set}} g) = (\mathrm{mapList}\, f) \circ_{\textsc{Set}} (\mathrm{mapList}\, g)$

The free monoid of **strings over an alphabet** A with **concatenation** as binary operation and the **empty word** as neutral element in a mathematical context is often denoted by $\underline{A^*} = \langle A^*, \epsilon, \circ \rangle$.

P I K

13

$\textsc{Cat}$: The category of small categories

With every category being a special directed graph, the category of small (!) categories inherits its identities and composition from $\textsc{Grph}$ (thus object and morphism maps are set functions).

Definition

The category $\textsc{Cat}$ of small categories is defined as follows:

Objects: small categories $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D} \ldots$

Morphisms: functors $F : \mathcal{C} \longrightarrow_{\textsc{Cat}} \mathcal{D}$

Identity: for all categories $\mathcal{C}$, the identity functor $\mathrm{Id}_{\mathcal{C}}$ which maps objects, resp. morphisms to themselves

Composition: For functors $F = \langle F_M, F_O \rangle : \mathcal{B} \longrightarrow_{\textsc{Cat}} \mathcal{C}$ and $G = \langle G_M, G_O \rangle : \mathcal{A} \longrightarrow_{\textsc{Cat}} \mathcal{B}$

$$F \circ_{\textsc{Cat}} G := F \circ_{\textsc{Grph}} G = \langle F_M \circ_{\textsc{Set}} G_M, F_O \circ_{\textsc{Set}} G_O \rangle$$

P I K

14

Large and small categories, Categorification

We have seen that instances of specific mathematical structures can be considered as objects of specific concrete categories (like MON, PRE, TOP) with structure preserving maps as morphisms. However, many of these are large categories, i.e. their classes of objects and morphisms are not sets.

But it is possible, to consider the *objects* of such categories *themselves* as small categories. This is called categorification.

The structure preserving maps between these objects then amount to functors between the corresponding small categories.

P I K

15

Categories : Categorification examples

- Every **set** can be considered as a discrete category with its elements as objects and no morphisms except the identities for each object.
- Any **monoid** $\underline{M} = \langle M, e, \otimes \rangle$ can be seen as a category with one object, the elements of M as morphisms, e as the identity and $\otimes$ as composition. That the axioms for identity and composition are fulfilled then follows from the corresponding properties of e and $\otimes$.
- Any **preordered set** $\underline{P} = \langle P, \leq \rangle$ can be seen as a category with the elements of P as objects and exactly one morphism between two objects X and Y iff $X \leq Y$. That the axioms are fulfilled then follows from reflexivity and transitivity of the preorder.

P I K

16

Categories as monoids

We have seen how to consider a monoid as small category.
But we also have:

A category $\mathcal{C}$ with one object $\bullet$ is a monoid, taking its morphisms as elements, composition as binary operation and the identity morphism as neutral element.

$$\underline{C} = \langle \mathrm{MOR}_{\mathcal{C}}, \mathrm{id}_{\bullet}, \circ_{\mathcal{C}} \rangle$$

P I K

17

Some more categories

Let $\mathcal{C}$ and $\mathcal{D}$ be arbitrary categories.

- The dual or opposite category $\mathcal{C}^{op}$ has the same objects as $\mathcal{C}$, but its morphisms are reversed, i.e. the domain and target maps are interchanged.
  A functor from $\mathcal{C}^{op}$ to $\mathcal{D}$ is often called a contravariant functor from $\mathcal{C}$ to $\mathcal{D}$.
- The product category $\mathcal{C} \times \mathcal{D}$ has as objects pairs of objects (C, D) with $C \in \mathrm{OBJ}_{\mathcal{C}}$ and $D \in \mathrm{OBJ}_{\mathcal{D}}$ and as morphisms pairs $(f, g) : (C, D) \longrightarrow_{\mathcal{C}\times\mathcal{D}} (C', D')$ with $f : C \longrightarrow_{\mathcal{C}} C'$ and $g : D \longrightarrow_{\mathcal{D}} D'$. Composition and identities are defined componentwise.
- The terminal category $\mathbf{1}$ which has just one object $\star$ and one morphism, namely $\mathrm{id}_{\star}$.

P I K

18

The Hom-functor

Definition

If $X, Y \in \textsc{Obj}_{\mathcal{C}}$, then $\textsc{Hom}_{\mathcal{C}}(X, Y)$ denotes the class of all morphisms $X \longrightarrow_{\mathcal{C}} Y$.

$\mathcal{C}$ is said to be locally small if $\textsc{Hom}_{\mathcal{C}}(X, Y)$ is a set for all pairs of objects $X, Y \in \textsc{Obj}_{\mathcal{C}}$.

If $\mathcal{C}$ is locally small, there exists a functor $\mathrm{Hom}_{\mathcal{C}} : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \textsc{Set}$ with $\mathrm{Hom}_{\mathcal{C}} = \langle (\mathrm{Hom}_{\mathcal{C}})_M, (\mathrm{Hom}_{\mathcal{C}})_O \rangle$:

- For $(X, Y) \in \textsc{Obj}_{\mathcal{C}^{op} \times \mathcal{C}}$, $(\mathrm{Hom}_{\mathcal{C}})_O(X, Y) := \textsc{Hom}_{\mathcal{C}}(X, Y)$
- For $(g, h) : (X', Y) \longrightarrow_{\mathcal{C}^{op} \times \mathcal{C}} (X, Y')$, $\mathrm{Hom}_{\mathcal{C}}(g, h) := \lambda f.h \circ_{\mathcal{C}} f \circ_{\mathcal{C}} g : \textsc{Hom}_{\mathcal{C}}(X, Y) \longrightarrow_{\textsc{Set}} \textsc{Hom}_{\mathcal{C}}(X', Y')$

P I K

19

Alternative definition of categories

Categories can equivalently be defined in terms of Hom-sets. This definition is usually adapted for implementation in type theory, replacing "SET" with "TYPE".

Definition

A small category is given by the following data:

- a set of objects $\textsc{Obj}_{\mathcal{C}}$,
- a function which assigns to each ordered pair $\langle X, Y \rangle$ of objects a set $\textsc{Hom}_{\mathcal{C}}(X, Y)$,
- for each $X \in \textsc{Obj}_{\mathcal{C}}$, a morphism $\mathrm{id}_X \in \textsc{Hom}_{\mathcal{C}}(X, X)$
- for each ordered triple $\langle X, Y, Z \rangle$ of objects, a function $\textsc{Hom}_{\mathcal{C}}(Y, Z) \times \textsc{Hom}_{\mathcal{C}}(X, Y) \longrightarrow_{\textsc{Set}} \textsc{Hom}_{\mathcal{C}}(X, Z)$ called composition and written $\circ_{\mathcal{C}}$.

where identity and composition fulfill the identity and associativity axioms as in the prior definition and every morphism is required to have a unique domain and codomain.

P I K

20

Natural Transformations

If we wish to consider categories of functors, again we first have to define the appropriate notion of structure-preserving morphism.

Definition

Given functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$, a natural transformation $\tau : F \longrightarrow G$ is a family of arrows $\tau_X : F\,X \longrightarrow_{\mathcal{D}} G\,X$ with one arrow for every $X \in \mathrm{O{\scriptsize BJ}}_{\mathcal{C}}$ such that

$$G\,f \circ_{\mathcal{D}} \tau_X = \tau_Y \circ_{\mathcal{D}} F\,f$$

for all $f : X \longrightarrow_{\mathcal{C}} Y$. Usually this condition is expressed by requiring that the square on the right *commutes*:

$$\begin{tikzcd} X \arrow[d, "f"] \\ Y \end{tikzcd} \qquad \begin{tikzcd} F\,X \arrow[r, "\tau_X"] \arrow[d, "F\,f"'] & G\,X \arrow[d, "G\,f"] \\ F\,Y \arrow[r, "\tau_Y"'] & G\,Y \end{tikzcd}$$

Composition of natural transformations

There are two ways in which natural transformations can be composed:

Definition

Given functors $F, G, H : \mathcal{C} \longrightarrow \mathcal{D}$ and natural transformations $\sigma : G \longrightarrow H, \tau : F \longrightarrow G$,
the vertical composition $\sigma \cdot \tau$ is defined by, for all $X \in \mathrm{O{\scriptsize BJ}}_{\mathcal{C}}$:

$$(\sigma \cdot \tau)_X := \sigma_X \circ_{\mathcal{D}} \tau_X$$

Definition

Given functors $F, F' : \mathcal{B} \longrightarrow \mathcal{C},\ G, G' : \mathcal{A} \longrightarrow \mathcal{B}$ and natural transformations $\sigma : F \longrightarrow F', \tau : G \longrightarrow G'$,
the horizontal composition $\sigma \circ \tau$ is defined by, for all $X \in \mathrm{O{\scriptsize BJ}}_{\mathcal{A}}$:

$$(\sigma \circ \tau)_X := F'\,\tau_X \circ_{\mathcal{C}} \sigma_{G\,X} = \sigma_{G'\,X} \circ_{\mathcal{C}} F\,\tau_X : (F \circ G)\,X \longrightarrow_{\mathcal{C}} (F' \circ G')X$$

Interchange law

Consider the following situation with F, F', G, G', H, H' functors and $\sigma, \sigma', \tau, \tau'$ natural transformations:

$$\begin{tikzcd}
\mathcal{A} \arrow[r, "G"] \arrow[r, bend left=60, "F"] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, shift left=3ex, "\Downarrow \sigma"] \arrow[r, phantom, shift right=3ex, "\Downarrow \tau"] & \mathcal{B} \arrow[r, "G'"] \arrow[r, bend left=60, "F'"] \arrow[r, bend right=60, "H'"'] \arrow[r, phantom, shift left=3ex, "\Downarrow \sigma'"] \arrow[r, phantom, shift right=3ex, "\Downarrow \tau'"] & \mathcal{C}
\end{tikzcd}$$

We can either first compose the natural transformations vertically and then vertically, or vice versa, giving respectively

$$(\tau' \cdot \sigma') \circ (\tau \cdot \sigma) \quad \text{or} \quad (\tau' \circ \tau) \cdot (\sigma' \circ \sigma)$$

The interchange law tells us that fortunately both are equal.
Exercise: Prove this by unfolding the definitions!

Functor categories

Given categories $\mathcal{C}$ and $\mathcal{D}$, we can now define the category of functors from $\mathcal{C}$ to $\mathcal{D}$:

Definition

The functor category $\mathcal{D}^{\mathcal{C}}$ is defined as follows:

Objects: functors $\mathcal{C} \longrightarrow \mathcal{D}$

Morphisms: natural transformations $\tau : F \longrightarrow_{\mathcal{D}^{\mathcal{C}}} G$

Identity: the identity natural transformation defined by $(\mathrm{id}_F)_X := \mathrm{id}_{FX}$ for all $X \in \mathrm{OBJ}_{\mathcal{C}}$

Composition: for natural transformations $\tau : G \longrightarrow_{\mathcal{D}^{\mathcal{C}}} H$ and $\sigma : F \longrightarrow_{\mathcal{D}^{\mathcal{C}}} G$, for all $X \in \mathrm{OBJ}_{\mathcal{C}}$

$$(\tau \circ_{\mathcal{D}^{\mathcal{C}}} \sigma)_X := (\tau \cdot \sigma)_X = \tau_X \circ_{\mathcal{D}} \sigma_X : FX \longrightarrow_{\mathcal{D}} HX$$

Again: Notation

We have already seen that the notation for the object and the morphism part of functors are overloaded in category-theoretical standard notation.

There are some more conventions, that are important to know and which we will also use in the following:

- The **composition of functors** $F \circ G$ is often simply written by juxtaposition FG.
- The **horizontal composition** $\tau \circ \sigma$ of natural transformations τ and σ is also usually written by juxtaposition $\tau\sigma$.
- Composing a functor F horizontally with some natural transformation τ, is called whiskering (on the left: $F\,\tau$, on the right: $\tau\,F$). It can be seen as special case of horizontal composition of natural transformations, if we consider F also as notation for the identity natural transformation id_F – even more overloading!

P I K

25

Examples: Structures as functors

We cannot only consider certain mathematical objects as categories, but also as functors:

- A **set** can be seen as a functor from a discrete category with one object to $\textsc{Set}$
- A **small graph** can be seen as a functor from the small category $(\cdot \rightrightarrows \cdot)$ to $\textsc{Set}$
- Considering a monoid $\underline{M} = \langle M, e, \otimes \rangle$ as category $\mathcal{M}$ with one object, an M**-set** can be seen as a functor from $\mathcal{M}$ to $\textsc{Set}$ (an M-set is a set A on which M *acts*, i.e. equipped with an action $\alpha : M \times A \longrightarrow_{\textsc{Set}} A$ such that , $\alpha\,\langle e, a \rangle = a$ and $\alpha\,\langle m \otimes m', a \rangle = \alpha\,\langle m, \alpha\,\langle m', a \rangle \rangle$ for all $a \in A$ and $m, m' \in M$)

The structure preserving morphisms between these objects then amount to natural transformations.

P I K

26

Plan

Introduction to basic concepts and terminology

Monads and adjunctions

Example: Deterministic dynamical systems

27

Monads Overview

- Generalization of **closure operators** on partially ordered sets
- Generalization of **monoids**
  ("monoid in the category of endofunctors")
- Monad algebras as generalization of **group actions**
- Computer science (Moggi's observation [4]):
  Computational effects (i.e. program behavior that in some sense differs from than the behavior of a pure total function) have the structure of monads

28

Monads

Definition

A monad $\underline{T} = \langle T, \eta, \mu \rangle$ is an endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$ with natural transformations $\mu : T \circ T \longrightarrow T$ and $\eta : \mathrm{Id}_{\mathcal{C}} \longrightarrow T$ such that the following diagrams commute:

$$
\begin{tikzcd}
(T\,(T\,(T\,X))) \arrow[r, "T\,\mu_X"] \arrow[d, "\mu_{T\,X}"'] & (T\,(T\,X)) \arrow[d, "\mu_X"] \\
(T\,(T\,X)) \arrow[r, "\mu_X"'] & T\,X
\end{tikzcd}
\qquad
\begin{tikzcd}
T\,X \arrow[r, "\eta_{T\,X}"] \arrow[dr, equal, "\mathrm{id}_{T\,X}"'] & (T\,(T\,X)) \arrow[d, "\mu_X"] & T\,X \arrow[l, "T\,\eta_X"'] \arrow[dl, equal, "\mathrm{id}_{T\,X}"] \\
& T\,X &
\end{tikzcd}
$$

Monads II

Spelled out, the conditions for an endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$ to be a monad amount to the following equations for all $X, Y \in \mathrm{OBJ}_{\mathcal{C}}$ and $f : X \longrightarrow_{\mathcal{C}} Y$:

(1) $T\,f \circ_{\mathcal{C}} \eta_X = \eta_Y \circ_{\mathcal{C}} f$ (naturality of η)

(2) $T\,f \circ_{\mathcal{C}} \mu_X = \mu_Y \circ_{\mathcal{C}} T\,(T\,f)$ (naturality of μ)

(3) $\mu_X \circ_{\mathcal{C}} \mu_{T\,X} = \mu_X \circ_{\mathcal{C}} T\,\mu_X$ (associativity)

(4) $\mu_X \circ_{\mathcal{C}} \eta_{T\,X} = \mathrm{id}_{T\,X}$ (left neutrality)

(5) $\mu_X \circ_{\mathcal{C}} T\,\eta_X = \mathrm{id}_{T\,X}$ (right neutrality)

Adjunctions

Definition

For categories $\mathcal{C}$ and $\mathcal{D}$, functors $F : \mathcal{C} \longrightarrow \mathcal{D}$, $G : \mathcal{D} \longrightarrow \mathcal{C}$ and natural transformations $\eta : \mathrm{Id}_{\mathcal{C}} \longrightarrow GF$ and $\varepsilon : FG \longrightarrow \mathrm{Id}_{\mathcal{D}}$, F and G are called adjoint functors, if the following two triangles commute for all $X \in \textsc{Obj}_{\mathcal{C}}$, $Y \in \textsc{Obj}_{\mathcal{D}}$:

$$
\begin{tikzcd}
G\,Y \arrow[r, "\eta_{G\,Y}"] \arrow[dr, equal, "\mathrm{id}_{G\,Y}"'] & (G\,(F\,(G\,Y))) \arrow[d, "G\,\varepsilon_Y"] \\
& G\,Y
\end{tikzcd}
\qquad
\begin{tikzcd}
(F\,(G\,(F\,X))) \arrow[r, "\varepsilon_{FX}"] & F\,X \\
F\,X \arrow[u, "F\,\eta_X"] \arrow[ur, equal, "\mathrm{id}_{F\,X}"']
\end{tikzcd}
$$

We then say, that $\langle F, G, \eta, \varepsilon \rangle : \mathcal{C} \rightharpoonup \mathcal{D}$ is an adjunction $F \dashv G$.

Adjunctions II

Every adjunction $\langle F, G, \eta, \varepsilon \rangle : \mathcal{C} \rightharpoonup \mathcal{D}$ can equivalently be characterized by a triple $\langle F, G, \varphi \rangle$, where φ is a bijection

$$\varphi_{X,Y} : \textsc{Hom}_{\mathcal{D}}(F\,X, Y) \cong \textsc{Hom}_{\mathcal{C}}(X, G\,Y)$$

which is natural in $X \in \textsc{Obj}_{\mathcal{C}}$ and $Y \in \textsc{Obj}_{\mathcal{D}}$.

We can obtain φ and φ^{-1} by defining,
for all $f : F\,X \longrightarrow_{\mathcal{D}} Y$ and $g : X \longrightarrow_{\mathcal{C}} G\,Y$:

$$\varphi\, f := G\,f \circ_{\mathcal{C}} \eta_X \quad \text{and} \quad \varphi^{-1}\, g := \varepsilon_Y \circ_{\mathcal{D}} F\,g$$

Conversely, we can construct η and ε by defining,
for all $X \in \textsc{Obj}_{\mathcal{C}}$, $Y \in \textsc{Obj}_{\mathcal{D}}$:

$$\eta_X = \varphi\, \mathrm{id}_{F\,X} \quad \text{and} \quad \varepsilon_Y = \varphi^{-1}\, \mathrm{id}_{G\,X}$$

Adjunctions and (co)monads

Every adjunction $\langle F, G, \eta, \varepsilon \rangle : \mathcal{C} \rightharpoonup \mathcal{D}$ **induces** a monad $\underline{GF} = \langle GF, \eta, G\varepsilon F \rangle$ on $\mathcal{C}$ (and a *comonad* $\underline{FG} = \langle FG, \varepsilon, F\eta G \rangle$ on $\mathcal{D}$).

We also have: Every monad (and comonad) can **canonically** be **decomposed** into two adjunctions.

The Kleisli decomposition of a monad T into an adjunction involves a category that is equivalent to the category of **free algebras** of the monad. It is the "smallest" (*initial*) adjunction that induces T.

The Eilenberg-Moore decomposition of a monad T into an adjunction involves the category of **all algebras** for this monad. It is the "largest" (*terminal*) adjunction that induces T.

P I K

33

Kleisli categories

Definition

Given a monad $\underline{T} = \langle T, \eta, \mu \rangle$ in a category $\mathcal{C}$, one can form its Kleisli category $\mathcal{C}_T$ as follows:

Objects: for every $X \in \mathrm{OBJ}_{\mathcal{C}}$, an object X_T

Morphisms: for every $f : X \longrightarrow_{\mathcal{C}} T\,Y$, a morphism $f_T : X_T \longrightarrow_{\mathcal{C}_T} Y_T$

Identity: for every object x_T,

$$\mathrm{id}_{x_T} := (\eta_X)_T : X_T \longrightarrow_{\mathcal{C}_T} X_T$$

(i.e. the morphism obtained from $\eta_X : X \longrightarrow_{\mathcal{C}} T\,X$)

Composition: for all $f_T : Y_T \longrightarrow_{\mathcal{C}_T} Z_T, g_T : X_T \longrightarrow_{\mathcal{C}_T} Y_T$,

$$f_T \circ_{\mathcal{C}_T} g_T := (\mu_X \circ_{\mathcal{C}} T\,f \circ_{\mathcal{C}} g)_T : X_T \longrightarrow_{\mathcal{C}_T} Z_T$$

P I K

34

Kleisli adjunction

We can now construct an adjunction $\langle F_T, G_T, \eta_T, \varepsilon_T \rangle : \mathcal{C} \rightharpoonup \mathcal{C}_T$.

Definition

Let functors $F_T : \mathcal{C} \longrightarrow \mathcal{C}_T, G_T : \mathcal{C}_T \longrightarrow \mathcal{C}$ be defined by,
for all $X, Y \in \mathrm{OBJ}_{\mathcal{C}}, X_T, Y_T \in \mathrm{OBJ}_{\mathcal{C}_T}, f : X \longrightarrow_{\mathcal{C}} Y, f_T : X_T \longrightarrow_{\mathcal{C}_T} Y_T$:

$$\begin{array}{lcl} (F_T)_O\, X & := & X_T \\ (F_T)_M\, f & := & (\eta_X \circ_{\mathcal{C}} f)_T \\ & & \\ (G_T)_O\, X_T & := & T\, X \\ (G_T)_M\, f_T & := & (\mu_X \circ_{\mathcal{C}} T\, f)_T \end{array}$$

Define moreover $\eta_T := \eta$ and
$(\varepsilon_T)_{X_T} := (\mathrm{id}_{TX})_T : F_T\, G_T\, X_T \longrightarrow_{\mathcal{C}_T} X_T$ for all $X_T \in \mathrm{OBJ}_{\mathcal{C}_T}$.

The monad $\underline{G_T F_T}$ induced by this adjunction is again $\underline{T}$.

Eilenberg-Moore categories

Definition

Given a monad $\underline{T} = \langle T, \eta, \mu \rangle$ in a category $\mathcal{C}$, one can form its Eilenberg-Moore category $\mathcal{C}^T$ (= category of T-algebras) as follows:

Objects: T-algebras $\langle X, h \rangle$ with $X \in \mathrm{OBJ}_{\mathcal{C}}$ and $h : T\, X \longrightarrow_{\mathcal{C}} X$, s.t.

$$h \circ_{\mathcal{C}} \mu_X = h \circ_{\mathcal{C}} T\, h \quad \text{and} \quad h \circ_{\mathcal{C}} \eta_X = \mathrm{id}_X$$

Morphisms: $\overline{f} : \langle X, h \rangle \longrightarrow_{\mathcal{C}^T} \langle Y, k \rangle$ are arrows $f : X \longrightarrow_{\mathcal{C}} Y$ with

$$f \circ_{\mathcal{C}} h = k \circ_{\mathcal{C}} T\, f$$

Identity: for all $X \in \mathrm{OBJ}_{\mathcal{C}}$, $id_{\langle X, h \rangle} := \overline{id_X}$

Composition: for all $\overline{f} : \langle Y, k \rangle \longrightarrow_{\mathcal{C}^T} \langle Z, l \rangle$, $\overline{g} : \langle X, h \rangle \longrightarrow_{\mathcal{C}^T} \langle Y, k \rangle$,

$$\overline{f} \circ_{\mathcal{C}^T} \overline{g} := \overline{f \circ_{\mathcal{C}} g} : \langle X, h \rangle \longrightarrow_{\mathcal{C}^T} \langle Z, l \rangle$$

Eilenberg-Moore adjunction

We can now construct an adjunction $\langle F^T, G^T, \eta^T, \varepsilon^T \rangle : \mathcal{C} \rightharpoonup \mathcal{C}^T$.

Definition

Let functors $F^T : \mathcal{C} \longrightarrow \mathcal{C}^T$, $G^T : \mathcal{C}^T \longrightarrow \mathcal{C}$ be defined by,
for all $X, Y \in \mathrm{OBJ}_{\mathcal{C}}$, $\langle X, h \rangle, \langle Y, k \rangle \in \mathrm{OBJ}_{\mathcal{C}^T}$,
$f : X \longrightarrow_{\mathcal{C}} Y$, $\bar{f} : \langle X, h \rangle \longrightarrow_{\mathcal{C}^T} \langle X, k \rangle$:

$$
\begin{array}{lcl}
(F^T)_O\, X & := & \langle T\,X, \mu_X \rangle \\
(F^T)_M\, f & := & T\,f \\
(G^T)_O\, \langle X, h \rangle & := & X \\
(G^T)_M\, \bar{f} & := & f
\end{array}
$$

Define moreover $\eta^T := \eta$ and
$\varepsilon^T \langle X, h \rangle := \bar{h} : F^T G^T \langle X, h \rangle \longrightarrow_{\mathcal{C}^T} \langle X, h \rangle$ for all $\langle X, h \rangle \in \mathrm{OBJ}_{\mathcal{C}^T}$.

The monad $\underline{G^T F^T}$ induced by this adjunction is again $\underline{T}$.

37

Adjunctions: Examples

The most common examples of adjoint functors follow one of these two “patterns”:

- Free $\dashv$ Forget *(typical for algebraic structures)*
- Colimit type $\dashv$ Diagonal $\dashv$ Limit type *(typical in logic)*

Example

The **free monoid functor** $\mathrm{Free}_{Mon} : \mathrm{SET} \longrightarrow \mathrm{MON}$ which maps every set to the free monoid generated by this set, has a right adjoint $\mathrm{Forget}_{Mon} : \mathrm{MON} \longrightarrow \mathrm{SET}$ which sends every monoid to its underlying set.
The monad on SET induced by this adjunction is the List-monad.

38

Plan

Introduction to basic concepts and terminology

Monads and adjunctions

Example: Deterministic dynamical systems

P I K

39

A categorical perspective on dynamical systems

Similar to the approach in the regular lectures, one can consider a **deterministic dynamical system** as an object of some category (e.g. SET) which is equipped with an endomorphism – to be thought of as a state space equipped with a transition function.

Definition

Let $\mathcal{C}$ be a category. Then we can construct the following category $\mathcal{C}^{\circlearrowleft}$ of $\mathcal{C}$-objects equipped with an endomorphism:

Objects: $\langle X, \alpha \rangle$ with $X \in \mathrm{OBJ}_{\mathcal{C}}$ and $\alpha : X \longrightarrow_{\mathcal{C}} X$,

Morphisms: $\overline{f} : \langle X, \alpha \rangle \longrightarrow_{\mathcal{C}^{\circlearrowleft}} \langle Y, \beta \rangle$ with $f : X \longrightarrow_{\mathcal{C}} Y$ and

$$f \circ_{\mathcal{C}} \alpha = \beta \circ_{\mathcal{C}} f$$

Identity: $\mathrm{id}_{\langle X, \alpha \rangle} := \overline{\mathrm{id}_X}$

Composition: for $\overline{f} : \langle Y, \beta \rangle \longrightarrow_{\mathcal{C}^{\circlearrowleft}} \langle Z, \gamma \rangle, \overline{g} : \langle X, \alpha \rangle \longrightarrow_{\mathcal{C}^{\circlearrowleft}} \langle Y, \beta \rangle$,

$$\overline{f} \circ_{\mathcal{C}^{\circlearrowleft}} \overline{g} := \overline{f \circ_{\mathcal{C}} g}$$

P I K

40

To wrap up: Some Popular Slogans

(from Lambek/Scott [7]):

"Many objects of interest to mathematicians . . . "

- "... congregate in concrete categories."
- "... are themselves small categories."
- "... may be viewed as functors from small categories to SET"

PIK

41

References and Conventions

- We are following the textbooks by Lambek and Scott [7] and Mac Lane [8].
- Categorical perspective on deterministic dynamical systems from Lawvere and Schanuel [9].
- We give both the axiomatic and the hom-based definition of categories and functors.
- Arrows in specific categories are annotated with the name of the category.
- Arrows which make sense in more than one ambient category are not annotated.
- We sometimes use λ-notation to write anonymous functions.
- The presentation glosses over foundational and equality issues (for a discussion see e.g. the introduction of [10]).

PIK

42

Bibliography I

[1] Nicola Botta, Patrik Jansson, and Cezar Ionescu. Contributions to a computational theory of policy advice and avoidability. *J. Funct. Program.*, 27:e23, 2017.

[2] Nicola Botta. IdrisLibs. `https://gitlab.pik-potsdam.de/botta/IdrisLibs`, 2016–2020.

[3] Cezar Ionescu. *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin, 2009.

[4] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23, 1989.

[5] Philip Wadler. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*, pages 233–264, 1992.

P I K

43

Bibliography II

[6] Edwin Brady. *Type-Driven Development in Idris*. Manning Publications Co., 2017.

[7] J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 2nd edition, 1994.

[8] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 2nd edition, 1998.

[9] F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics*. Cambridge University Press, 2nd edition, 2008.

[10] Benedikt Ahrens, Chris Kapulkin, and Michael Shulman. Univalent categories and the rezk completion. *Math. Struct. Comp. Sci.*, 25:1010–1039, January 2015.

P I K

44