Vulnerability Modeling and Monadic Dynamical Systems

Dissertation

zur Erlangung des Grades
eines Doktors der Naturwissenschaften (Dr. rer. nat.)
am Fachbereich Mathematik & Informatik
der Freien Universität Berlin

vorgelegt von
Cezar Ionescu

Berlin, November 2008

Erster Gutachter und Betreuer:

Prof. Dr. Rupert Klein, *Freie Universität Berlin, Institut für Mathematik*

Zweiter Gutachter:

Prof. Dr. Paul Flondor, *"Politechnica" University of Bucharest, Department of Mathematics*

Datum der Disputation: 09.02.2009

Contents

Acknowledgements

My first thanks go to Rupert Klein for undertaking to supervise and energetically seeing through the production of a somewhat unusual thesis, developed under unorthodox circumstances by a sometimes difficult student.

I would like to thank all my teachers: I wish I had been a better student. To Paul Flondor, in particular, I owe much, much more than excellent mathematical instruction.

I am grateful to Richard Klein for the effort he has put in introducing me to the concepts of vulnerability in the context of climate change research, and for the patience with which he allowed himself to be exposed to their translation in unfamiliar mathematical notation.

Many of the ideas in this work were presented in the Cartesian Seminar at PIK: thanks are due to all the participants to these discussions, particularly Nicola Botta, Carlo Jaeger, Jochen Hinkel and Rupert Klein.

Earlier drafts of this thesis have been carefully read by Daniel Lincke, Sarah Wolf, and Patrik Jansson.

Special thanks to Jochen Hinkel for shielding me from administrative duties while writing this thesis.

Thanks to Anne Biewald and Christian Elsner who helped with the surprisingly difficult task of writing a German summary. Of course, without Anne's generous support throughout the past four years, there would have probably been no thesis to summarize.

Chapter 1

Introduction

"Translations can sometimes create a sense of *explanation*", says Goldblatt in his account of categorial logic ([17]), and that is perhaps most true of translations in the language of mathematics. Writing a program, especially in a high-level declarative language such as Haskell, is a similar exercise, and it is the same resulting feeling of explanation which accounts for statements common in the software world, such as "you only understand it if you've implemented it". Moreover, the understanding achieved can be more easily shared with others by virtue of the fact that it is expressed in executable form. For the non-specialist, it is usually easier to play around with a program than with a theorem.

A recent example of using insights from functional programming to better understand a "real-world" domain is the work on financial contracts of Peyton-Jones and Eber described in [33]. Our work here aims to achieve similar results in the interdisciplinary world of Climate Impact Research, by focusing on the concept of vulnerability and the task of vulnerability assessment in a computational context.

1.1 Vulnerability and Haskell

In the past decade, the concept of "vulnerability" has played an important role in the fields such as climate change, food security or natural hazard studies. Vulnerability studies have often been successful in alerting policy-makers to precarious situations revealed by scientific analysis in these fields. The importance of the concept in the particular field of climate change is described, for example, as follows ([25]):

> . . . Studies based primarily on the output of climate models tend to be characterized by results with a high degree of uncertainty and large ranges, making it difficult to estimate levels of risk. In addition, the complexity of the climate, ecological, social and economic systems that researchers are modeling means that the validity of scenario results will inevitably be subject to ongoing criticism. . . . Such criticisms should not be interpreted as questioning the value of scenarios; indeed, there is no other tool for projecting future conditions. What they do, however, is emphasize the need for a strong foundation upon which scenarios can be applied, a foundation that provides a basis for managing risk despite uncertainties associated with future climate changes.
> *This foundation lies in the concept of vulnerability.*

Unfortunately, this foundation has problems of its own. The definitions of vulnerability differ across the fields mentioned, and even within any one field there seem to be a bewildering multitude of definitions to choose from (Thywissen, for example, summarizes thirty-five definitions of vulnerability in [36]!).

The definitions usually proposed are not mathematical ones, and leave quite a bit of room for interpretation. In the climate change community, a particularly influential conceptualization of vulnerability is the one proposed by the IPCC, which we describe in 3.3 together with three different studies which apply it to the task of assessing vulnerability. As will be seen, each of these studies uses its own "reading" of the IPCC definition, and while none of them can be considered "wrong", they are incompatible with one another.

Nevertheless, there is a tantalizing similarity underlying the definitions and the usage of "vulnerability", at least in those studies which are attempting to "project future conditions" by using computational tools, and which can thus be termed computational vulnerability assessment.

This is the same sort of similarity which emerges many times when writing computer programs: one finds oneself using the same computational patterns over and over again, with small yet important differences. Inventing the proper concepts and tools for describing these patterns as instances of the same structure has been one of the main driving forces of computer science. Structured programming, object-orientation, template programming, can all be seen as increasingly sophisticated solutions to the problem of *generic programming.*

Functional programming languages such as Haskell, with clean syntax, staying close to the original mathematical notation from which they borrow their expressive type systems and algebraic type classes, seem particularly promising as candidates for reliable generic implementations.

It seems therefore natural to try to apply the techniques used in writing generic programs to the problem of capturing the commonalities of the various definitions and usages of vulnerability, at least those that appear in computational vulnerability assessments. In doing so, we can hope to reap the usual benefits of generic programming: we obtain a more robust design, better code, and code reuse. Perhaps more significantly, we hope to gain a better understanding of the concept of vulnerability itself, and thus make an interdisciplinary contribution of some importance. Since this understanding is, in final instance, expressed in mathematical terms, it can be communicated unambiguously to the trained specialist. However, since it is also implemented in working code, we can hope to convey at least some of the consequences to the layman, by providing suitable examples.

This, then, is the idea of this work. The road to its realization is, however, quite long and not altogether straight. We have accordingly prepared a road-map for the convenience of our fellow travellers.

1.2 Road-map

1. Experience shows that Haskell syntax is easily picked up by the non-specialist or by programmers who have not seen Haskell code before, so they are *not* the intended readership for our next chapter, "Mathematical Preliminaries and Notation". It is also quite likely that the specialists in vulnerability assessment are going to want to skip the technically difficult details and concentrate more on the conceptual understanding of vulnerability emerging from Chapters 3 and 4. For this, an intuitive understanding of Haskell notation is entirely sufficient. Similarly, programmers interested in knowing about the design of software based on our model of vulnerability are going to want to read, besides the two chapters just mentioned, Chapter 7, which requires a working knowledge of Haskell (at the level of [1]), but they don't really need to read about coalgebras or the connections to category theory.

   The audience for this chapter consists of computer scientists who want a refresher of these and similar subjects (monads, fixed points, fusion theorems), and, most particularly, of mathematicians who are unfamiliar with Haskell. The latter are most at disadvantage: Haskell notation has been designed to approximate usual mathematical notation, but the differences can be much more confusing to the trained mathematician than to the layman, for in mathematics, perhaps more than anywhere else, the saying of Flaubert holds: "le bon Dieu est dans le detail". Our preliminaries are designed to minimize this confusion, by presenting familiar subjects in Haskell clothing.

2. Chapter 3, “Definitions of vulnerability”, consists of a close reading of various definitions and examples of usage of vulnerability, in different contexts. We encounter the expected “family resemblances”, but also many puzzling differences, representing as many problems to be solved by any attempt at providing a synthesis of these definitions.

   We start by reading the natural language definition, as given by the Oxford Dictionary of English, considering it likely that the technical usage represents a refinement of the everyday one. We examine the examples given, noting the relative character of vulnerability, its negative connotations, and its potential aspect.

   The next section is devoted to the usage of vulnerability in Development Studies. We take advantage of the synthetical effort made by Calvo and Dercon in [4], which unifies a large number of disparate studies in the field, on the basis of an axiomatic approach. We particularly note the usage of “vulnerability *to*”: in the Development Studies, one commonly says, for example, “vulnerability to poverty”. Thus, one is vulnerable to potential harmful results. In the Climate Change community, one typically says “vulnerability to climate change”, emphasizing the *causes* of the harmful results.

   The subject of vulnerability in the context of climate change is treated next, first by examining the most influential definition in the field, the one proposed by the IPCC in its assessment report. Analysis of this definition also involves looking at the related concepts of “sensitivity” and “adaptive capacity”.

   The somewhat problematic character of the IPCC definition is illustrated in the study of three computational vulnerability assessments, which all attempt to operationalize this definition. The differences in interpretation are pointed out, while at the same time continuing the effort of uncovering the common structure of the various assessments.

3. Armed with the knowledge about the various facets of vulnerability obtained in the previous chapter, we move on to the task of formulating a unifying mathematical model. The fundamental idea expressed by this model is that vulnerability is a measure of possible future harm: the various studies we have seen differ as to how “possible future” is expressed (deterministically, non-deterministically, or stochastically), what counts as “harm” and how the various possible harms are to be measured.

   We return to the definitions and assessments of the previous chapter, and translate them in terms of our model. In so doing, we refine our model to represent related notions such as sensitivity or adaptive capacity. In the process, we gain confidence that the model is indeed capable of expressing all these variations. An interesting result is that the problem of "vulnerability *to*" uncovered in the previous chapter can be explained in terms of the model: thus, the model is consistent with common usage, and in a certain sense can be said to justify it.

The measure which is supposed to assess the possible future harms is perhaps the main source of difference of the different assessments. The mathematical formulation allows us to express a natural condition on this measure, which can be used to uncover possible inconsistencies.

The translations of various examples in a common framework is interesting in its own right, independent of the technical developments of the next chapters. It is an illustration of how a mathematical model can be used for the task of "meta-analysis", which is of importance in many projects involving vulnerability assessment.

4. The model of vulnerability as a measure of the harm or impacts suffered along possible future evolutions raises the question of how to compute these evolutions. In computational vulnerability assessment, this is achieved by using complex models representing the climate, ecological, social and economical systems. In the chapter "Dynamical Systems", we look for a generic way to compute these evolutions. The problem is that the various dynamical systems used are of different types: discrete and continuous, deterministic, stochastic, nondeterministic (scenarios), fuzzy, etc. and, importantly, combinations of all these. Is there a common structure to all these systems? We look at the definitions proposed in classical systems theory and computer science, and at several examples. Finally, and perhaps luckily, we discover the class of monadic dynamical systems which is indeed general enough to represent the models of interest in vulnerability assessment, and which offers a uniform way of computing their trajectories.

5. In Chapter 6, "Working with Monadic Systems", we build the infrastructure needed to use the generic trajectory computing functions defined in the previous chapter. We define functions for transforming systems with discrete or continuous input in monadic systems, and implement the means for combining various monadic systems in order to yield new monadic systems. In the end, we have the means for putting systems of various types in series, in parallel, or in a feedback loop, and the trajectories of the resulting systems are still calculated by the same generic functions. This algebra of monadic systems is achieved by using a surprisingly small number of combinators, and is the main technical achievement of the thesis.

6. Finally, we return to the subject of vulnerability. Using monadic dynamical systems, we are able to give a surprisingly simple and general implementation of the model of vulnerability developed in the first

part of the thesis. An important question we address regards the compatibility of the measures of vulnerability used in various studies. We would like to know what changes if we replace some models used in our assessment by others representing the same system, but having a different type, say stochastic instead of deterministic. What can be reused from the previous assessment? How can we compare the various assessments resulting from using different vulnerability measures? The answer to such questions is given in Section 7.2, where we define the conditions of compatibility of vulnerability measures, and show how to translate a measure defined for one type of system to another type.

Combining systems, defining vulnerability measures, translating from one type to another, are all complex tasks which leave a lot of room for error. Fortunately, we can formulate correctness conditions which can be automatically tested by software tools such as QuickCheck, and we devote Section 7.3 to showing how that is done.

We conclude by showing how the model can be used in an idealized example of a vulnerability assessment-like problem. This "toy example" exhibits many of the features of full-blown vulnerability assessments: possible evolutions given by combined systems of different types, conflicting definitions of "harm", partial ordering of vulnerability measurements. Our analysis of the compromises that need to be made between the conflicting goals, and the possible changes to the problem that could lead to better results, have many parallels in the "real-world" discussions on the impacts of climate change.

7. As usual, the end point of our journey turns out to be only the starting point for many other possible travels. In the last chapter, "Conclusions and Perspectives", we briefly review some of the most important landmarks seen on our way, and list some of the possible destinations for the future.

Chapter 2

Mathematical Preliminaries and Notation

2.1 Notation

Most of the mathematical concepts we shall use, such as function, strict order relation, monad and so on, are easily represented computationally in the Haskell programming language. We have therefore decided to adopt a Haskell notation throughout. This has the advantage that there is no need for a separate implementation of most concepts: the description we give *is* the implementation and the files comprising this document are at the same time the source code of our programs. The disadvantage is that Haskell differs at several points from standard mathematical notation. To alleviate the potential confusion caused by this, we present in this section a summary of the most important differences.

1. *Membership.* Haskell is a typed language, so in most cases the membership relation is replaced by the "of type" relation, for example $n \in \mathbb{N}$ is written $n :: \mathbb{N}$. We will represent subsets of values of a given type by lists, and use the standard Haskell *elem* function to express membership of a value to a given list. As is common usage, we have configured the typesetting program to print the standard membership symbol $\in$ for infix uses of *elem*. The names of types are capitalized, thus we write $a :: A$ for the membership of a to the type A. Haskell allows polymorphic type assignment, for example $[\,] :: [a]$ which can be read "for any type a, $[\,]$ denotes an element of type list of a, namely the empty list". As can be seen in this example, type variables are written in lowercase and are implicitely universally quantified. The Haskell type class system provides a way of restricting the domain of quantification to those types which have been declared to be members of a type class. For example

$$min :: Ord\ a \Rightarrow a \rightarrow a \rightarrow a$$

declares *min* to denote a value of type $a \rightarrow a \rightarrow a$ for any type a on which a total order has been defined (see below).

2. *Functions.* Functions $f : A \rightarrow B$ are represented in Haskell as $f :: A \rightarrow B$. Function application, $f(a)$ is denoted by juxtaposition: $f\ a$, the brackets being omitted.

3. *Order relations.* In Haskell, the types on which a *total* order has been defined are grouped together in a *type class Ord*. To add some type A to this group, one must provide an *instance declaration* defining (at least) the relation $\leqslant$:

```
instance Ord A where
  a1 ⩽ a2 = ...
```

   The Haskell compiler generates the other relations $<$, $\geqslant$ and $>$ under the assumption that the order is total.

   We will be interested however more in *partial* orders, for which we define a type class *PartialOrd*. In order to distinguish total orders from partial orders typographicall, we will use the "squarish" symbols $\sqsubseteq$ and $\sqsupseteq$ for partial orders.

4. *Functors.* Functors are represented in Haskell by type constructors, and are written in uppercase. For example, the functor $F\ A = 1 + A$ is called in Haskell *Maybe*, the result of applying it to an arbitrary datatype A is accordingly written *Maybe* A. Just as with arbitrary datatypes, variables of functor type are written in lowercase. Thus, $g :: Functor\ f \Rightarrow a \rightarrow f\ a$ is read "for any type a and functor f, g denotes a coalgebra of f with carrier a". The namespaces of values and types are disjoint, so in this example we can also write $f :: Functor\ f \Rightarrow a \rightarrow f\ a$.

   In Category Theory, the action of a functor F on an arbitrary function $g : A \rightarrow B$ is denoted $F\ g : F\ A \rightarrow F\ B$. Given a type constructor F, representing a functor, Haskell cannot automatically generate the extension of F to functions. This extension has to be explicitly provided by the user, by instanciating F as a member of the type class *Functor*. The name of the associated operation is *fmap*. Here is, for example, the function associated with the functor *Maybe*:

```
instance Functor Maybe where
  fmap :: (a → b) → Maybe a → Maybe b
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

The programmer is responsible for veryfing that the defining properties of a functorial operation are satisfied ($\mathit{fmap}\ \mathit{id} = \mathit{id}$ and $\mathit{fmap}\ (f \cdot g) = \mathit{fmap}\ f \cdot \mathit{fmap}\ g$)

5. *Natural transformations.* As shown in [37], Haskell functions of type $\mathit{alpha} :: F\ a \rightarrow G\ a$ correspond to natural transformations from the functor F to the functor G, that is, for any $x :: a \rightarrow b$ we have that

   $$\mathit{fmap}\ x \cdot \mathit{alpha} = \mathit{alpha} \cdot \mathit{fmap}\ x$$

   (the first occurence of $\mathit{fmap}\ x$ above has the type $G\ a \rightarrow G\ b$, the second $F\ a \rightarrow F\ b$).

6. *Cartesian products.* The cartesian product of two sets, $A \times B$ is written in Haskell (A, B). The projection functions are $\mathit{fst} :: (a, b) \rightarrow a$ and $\mathit{snd} :: (a, b) \rightarrow b$. Given $f :: a \rightarrow b$ and $g :: a \rightarrow c$, the function denoted usually by $< f, g >$ is written in Haskell $\mathit{pair}\ (f, g)$ with the definition

   $$\begin{aligned} &\mathit{pair} :: (a \rightarrow b, a \rightarrow c) \rightarrow a \rightarrow (b, c) \\ &\mathit{pair}\ (f, g)\ a = (f\ a, g\ a) \end{aligned}$$

   Doing things component-wise is written cross and defined as

   $$\begin{aligned} &\mathit{cross} :: (a \rightarrow b, c \rightarrow d) \rightarrow (a, c) \rightarrow (b, d) \\ &\mathit{cross}\ (f, g) = \mathit{pair}\ (f \cdot \mathit{fst}, g \cdot \mathit{snd}) \end{aligned}$$

   A number of properties hold for pair and cross, their usage is usually signalled by the phrase *pair calculus.* For example,

   $$\begin{aligned} &\mathit{pair}\ (f, g) \cdot h = \mathit{pair}\ (f \cdot h, g \cdot h) && \text{-- fusion} \\ &\mathit{cross}\ (f, g) \cdot \mathit{pair}\ (h, k) = \mathit{pair}\ (f \cdot h, g \cdot k) && \text{-- absorption} \end{aligned}$$

7. *Coproducts.* The coproduct (the disjoint sum) of two sets, $A + B$ is represented in Haskell by the datatype $\mathit{Either}\ A\ B$, defined by the data declaration **data** $\mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$. Left and Right, the two constructors, also play the role of the injections more commonly called inl and inr. Given $f :: a \rightarrow b$ and $g :: c \rightarrow b$, the dual construction to the pair above, denoted by $[f, g]$ in most mathematics texts, is called case and is defined by

   $$\begin{aligned} &\mathit{case} :: (a \rightarrow b, c \rightarrow b) \rightarrow \mathit{Either}\ a\ c \rightarrow b \\ &\mathit{case}\ (f, g)\ (\mathit{Left}\ a) = f\ a \\ &\mathit{case}\ (f, g)\ (\mathit{Right}\ c) = g\ c \end{aligned}$$

   In Haskell, case is provided by the standard prelude in curried form under the name either.

   A related function is the *conditional*:

$$\begin{array}{l}
\mathit{cond} \qquad\qquad :: (a \to \mathit{Bool}, a \to b, a \to b) \to a \to b \\
\mathit{cond}\ (p, f, g)\ a = \mathbf{if}\ p\ a\ \mathbf{then}\ f\ a\ \mathbf{else}\ g\ a
\end{array}$$

The conditional satisfies a number of properties, for example

$$\begin{array}{l}
\mathit{cond}\ (\mathit{const\ False}, f, g) = g \\
\mathit{cond}\ (\mathit{const\ True}, f, g) = f \\
\mathit{cond}\ (p, f, g) \cdot h = \mathit{cond}\ (p \cdot h, f \cdot h, g \cdot h)
\end{array}$$

2.2 Monads

An important structure in Category Theory in general, and in Computing Science in particular, is that of a *monad*.

Definition 1 (Monad) *A* monad *is a triple* (M, η, μ) *consisting of*

1. *an endo-functor* $M\ :\ \mathbf{C} \longrightarrow \mathbf{C}$
2. *a natural transformation* $\eta\ :\ Id \longrightarrow M$
3. *a natural transformation* $\mu\ :\ MM \longrightarrow M$

such that

$$\begin{array}{lllll}
\mu \cdot \eta M & = & \mu \cdot M\eta & = & id \\
\mu \cdot \mu M & = & \mu \cdot M\mu & &
\end{array}$$

Haskell provides a *Monad* class, which, given the above, has a somewhat surprising definition. Given that, as explained above, in Haskell functors are represented by instances of the class type *Functor* and natural transformations are expressed by polymorphic functions, one would expect something like the following:

$$\begin{array}{l}
\mathbf{class}\ (\mathit{Functor}\ m) \Rightarrow \mathit{Monad}\ m\ \mathbf{where} \\
\quad \eta :: x \to m\ x \\
\quad \mu :: m\ (m\ x) \to m\ x
\end{array}$$

satisfying the axioms of functoriality, naturality and the monad equations:

$$\begin{array}{ll}
\mathit{fmap\ id} & = id \\
\mathit{fmap}\ (f \cdot g) & = \mathit{fmap}\ f \cdot \mathit{fmap}\ g \\
\mathit{fmap}\ f \cdot \eta & = \eta \cdot f \\
\mathit{fmap}\ f \cdot \mu & = \mu \cdot (\mathit{fmap}\ (\mathit{fmap}\ f)) \\
\mu \cdot \eta & = id \\
\mu \cdot \mathit{fmap}\ \eta & = id \\
\mu \cdot \mathit{fmap}\ \mu & = \mu \cdot \mu
\end{array}$$

We could call this definition “classical”.

The choice made by the Haskell implementors is different:

$$
\begin{array}{l}
\textbf{class } \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: x \to m\ x \\
\quad (\rhd) :: m\ x \to (x \to m\ y) \to m\ y
\end{array}
$$

which are subject to the following three requirements:

$$
\begin{array}{l}
(\mathit{return}\ x) \rhd f = f\ x \\
mx \rhd \mathit{return} = mx \\
(mx \rhd f) \rhd g = mx \rhd h \\
\quad \textbf{where} \\
\quad h\ x = (f\ x) \rhd g
\end{array}
$$

The operator $\rhd$ is called *the monadic bind operator*.

The two definitions are equivalent, in the sense that if one has *fmap*, η and μ as in the "classical" version, one can define

$$
\begin{array}{l}
\mathit{return} = \eta \\
mx \rhd f = \mu\ ((\mathit{fmap}\ f)\ mx)
\end{array}
$$

and the resulting *return* and $\rhd$ statisfy the three requirements from the Haskell version. Perhaps more surprisingly, *return* and $\rhd$ can also be used to define appropriate *fmap*, η and μ, which verify all the *seven* axioms, by

$$
\begin{array}{l}
\mathit{fmap}\ f = (\rhd(\mathit{return} \cdot f)) \\
\eta = \mathit{return} \\
\mu = (\rhd \mathit{id})
\end{array}
$$

This is the reason for the Haskell definition: it requires the programmer to check fewer axioms.

In many cases, it is more natural to use $\lhd$ which is just the flipped version of $\rhd$ and is defined automatically by Haskell from it. Some properties of the bind operator are best shown by using the definition of $\lhd$ in terms of μ and *fmap*: one which we will have the occasion to use in a later chapter is:

Proposition 1 (*fmap* and $\lhd$) *We have*

$$(f \cdot g) \lhd mx = f \lhd (\mathit{fmap}\ g\ mx)$$

Proof.

$$
\begin{array}{ll}
 & (f \cdot g) \lhd mx \\
= & \{ \text{ Definition of } \lhd \text{ in terms of } \mu \ \} \\
 & (\mu \cdot \mathit{fmap}\ (f \cdot g))\ mx \\
= & \{ \ \mathit{fmap} \text{ functor, composition } \} \\
 & (\mu \cdot \mathit{fmap}\ f)\ (\mathit{fmap}\ g\ mx) \\
= & \{ \text{ Definition of } \lhd \text{ in terms of } \mu \ \} \\
 & f \lhd (\mathit{fmap}\ g\ mx)
\end{array}
$$

□

2.2.1 The Kleisli construction: monads and monoids

There exists yet a third, equivalent, definition of a monad class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{unit} :: x \to m\ x \\
\quad (\diamond) :: (y \to m\ z) \to (x \to m\ y) \to (x \to m\ z)
\end{array}
$$

such that

$$
\begin{array}{l}
\mathit{unit} \diamond f = f \\
f \diamond \mathit{unit} = f \\
(f \diamond g) \diamond h = f \diamond (g \diamond h)
\end{array}
$$

The operator $\diamond$ is called *Kleisli composition*, and the categorial construction which has inspired this Haskell definition is called *the Kleisli construction.*

We can obtain the elements of the Haskell definition in terms of the Kleisli one by

$$
\begin{array}{l}
\mathit{return} = \mathit{unit} \\
(\rhd f) = f \diamond \mathit{id}
\end{array}
$$

and we can go the other way around by

$$
\begin{array}{l}
\mathit{unit}\ = \mathit{return} \\
g \diamond f = (\rhd g) \cdot f
\end{array}
$$

The axioms of the Haskell definition are as many as those of the Kleisli one, and they are arguably somewhat more awkward. However, the monadic bind operator has proven more useful in practice than the Kleisli composition operator, and has been given precedence.

For us, it is important to note that, if M is a monad, then the set $\mathit{Hom}\ (A, M\ A)$ of arrows from A to $M\ A$ is a monoid with respect to the Kleisli composition. The axioms given above state the associativity of the operator, and the fact that *unit* is indeed appropriately named.

2.3 Examples

In this section, we present a number of monads, which we shall use extensively in the sequel.

2.3.1 Identity.

The identity functor is represented in Haskell as

$$\textbf{newtype}\ \mathit{Id}\ a = \mathit{Id}\ a$$

The new type *Id a* is isomorphic to a: in one direction, the isomorphism is given by the constructor *Id*, in the other by its inverse

$$\begin{array}{ll} \mathit{unwrapId} & :: \mathit{Id}\ a \to a \\ \mathit{unwrapId}\ (\mathit{Id}\ a) & = a \end{array}$$

Unfortunately, we cannot make *Id a* inherit the properties of a: in particular, we have to define

$$\begin{array}{l} \textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Id}\ a)\ \textbf{where} \\ \quad \mathit{Id}\ x \equiv \mathit{Id}\ y = x \equiv y \end{array}$$

The identity functor is a monad:

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathit{Id}\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{Id}\ a) = \mathit{Id}\ (f\ a) \end{array}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Monad}\ \mathit{Id}\ \textbf{where} \\ \quad \mathit{return}\ a = \mathit{Id}\ a \\ \quad (\mathit{Id}\ a) \rhd f = f\ a \end{array}$$

The *return* of the *Id* monad is (isomorphic to) the identity function, and the ($\rhd$) operator is (isomorphic to) functional application.

2.3.2 Sets and lists.

The powerset functor, P, is a monad:

$$\begin{array}{l} \textbf{instance}\ \mathit{Monad}\ P\ \textbf{where} \\ \quad \begin{array}{ll} \mathit{return}\ a & = \{\, a \,\} \\ \mathit{as} \rhd f & = \cup \{ f\ a \mid a \in \mathit{as} \} \end{array} \end{array}$$

However, this is not a Haskell code fragment! Implementing the powerset functor is not a trivial exercise. The solution adopted in Haskell, for instance, restricts the domain of P to *ordered sets*, for efficiency reasons. Unfortunately, one loses thus the ability to declare P as an instance of the *Monad* class. It is tempting to introduce P using characteristic functions as a way of representing sets:

$$\textbf{newtype}\ P\ a = P\ (a \to \mathit{Bool})$$

but we still lose the ability to make P an instance of the monad class: the $\cup$ operation is not, in general, implementable.

The solution we have adopted is to use lists as representations of sets. The datatype of lists is predefined in Haskell as an instance of the *Monad* class, with

```
instance Monad [] where
  return a = [a]
  as ▷ f   = concat [f a | a ← as]
```

The main difference between lists and sets is, of course, that lists may contain duplicates. The Haskell function *nub* removes duplicates and can be used to "regularize" the representation of sets of types which are instances of the typeclass *Eq*. Another difference is that equality of sets does not imply the equality of their list based representations (which depends on the order of appearance of elements in the lists). In the sequel, we shall not have to use the equality test $\equiv$ for sets, and therefore we sidestep this difficulty.

2.3.3 Simple probability distributions

A simple probability distribution is one with a finite (or at most countable) support, so what we want is something like the following:

```
type Supp a = [a]
```

to represent the support, and

```
type Prob = Double

newtype SimpleProb a = SP (a → Prob, Supp a)
```

to represent the distribution. Unfortunately, we can not declare this new type to be an instance of the monad class, since we can only define the *return* and ($\triangleright$) operations on types which are instances of *Eq*. (For a similar reason, the Haskell implementation of the powerset functor, *Set*, cannot be declared to be an instance of the *Monad* type class.)

Therefore, we shall adopt the representation used by Erwig in [12]:

```
newtype SimpleProb a = SP [(a, Prob)] deriving Show

unwrapSP :: SimpleProb a → [(a, Prob)]
unwrapSP (SP ds) = ds
```

Values *sp* of type *SimpleProb a* are required to satisfy

```
sum (map snd (unwrapSP sp)) = 1.0
```

The support of a simple probability distribution *ds* is the set of values in the list *map fst ds*, but now we need to ensure that there are no values associated to zero probability:

```
supp :: Eq a ⇒ SimpleProb a → [a]
supp (SP ds) = nub (map fst (filter notz ds))
  where
  notz (x, p) = p ≢ 0
```

The main difference between the list-based representation and the functional one is that we can now have several values associated with the same element, and values associated to zero probability (which are therefore outside the support). This makes it necessary to introduce a function that normalizes the representation:

```
normalize :: Eq a ⇒ SimpleProb a → SimpleProb a
normalize (SP ds) = SP (map (pair (id, f)) (supp (SP ds)))
  where
  f a = sum [p | (x, p) ← ds, x ≡ a]
```

We can now define *SimpleProb* as an instance of the typeclasses *Eq* and *Monad*.

```
instance Eq a ⇒ Eq (SimpleProb a) where
  sp1 ≡ sp2 = and (map (∈ ps2) ps1)
    where
    ps1 = unwrapSP (normalize sp1)
    ps2 = unwrapSP (normalize sp2)

instance Monad SimpleProb where
  return a = SP [(a, 1.0)]
  SP (ds1) ▷ f = SP (concat (map g ds1))
    where
    g (a, p) = map h (unwrapSP (f a))
      where
      h (x, p′) = (x, p′ ∗ p)
```

The monadic bind operator expresses the conditional probabilities of the elements in the target of f, depending on the distribution on the elements in the source of f. For finite, identical source and target, f can be represented as a stochastic matrix, and the bind operator gives the transition of the associated Markov chain.

2.3.4 Simple fuzzy sets

Fuzzy sets are generalisations of characteristic functions, taking values in $[0, 1]$ instead of $\{0, 1\}$. A fuzzy subset of some set a is therefore a function $f :: a \to [0, 1]$.

As in the case of probability distributions, we are actually going to implement and use *simple fuzzy sets*, fuzzy sets with finite or countable support. The definitions parallel those given above for *SimpleProb*.

```
type UI = Double  -- unit interval

newtype SimpleFuzzy a = SF [(a, UI)] deriving Show

unwrapSF :: SimpleFuzzy a → [(a, UI)]
unwrapSF (SF fs) = fs

fsupp :: Eq a ⇒ SimpleFuzzy a → [a]
fsupp (SF fs) = nub (map fst (filter noz fs))
          where
          noz (a, x) = x ≢ 0

fnormalize :: Eq a ⇒ SimpleFuzzy a → SimpleFuzzy a
fnormalize (SF fs) = SF (map (pair (id, f)) (fsupp (SF fs)))
  where
  f a = maximum [x | (b, x) ← fs, b ≡ a]

instance Eq a ⇒ Eq (SimpleFuzzy a) where
  $sf_1$ ≡ $sf_2$ = length fs1 ≡ length fs2 ∧
    and (map (∈ fs1) fs2)
    where
    fs1 = unwrapSF (fnormalize $sf_1$)
    fs2 = unwrapSF (fnormalize $sf_2$)

instance Monad SimpleFuzzy where
  return a = SF [(a, 1.0)]
  SF (fs1) ⊳ f = SF (concat (map g fs1))
    where
    g (a, x) = map h (unwrapSF (f a))
      where
      h (b, x′) = (b, min x x′)
```

The monadic bind operator of the fuzzy set monad is similar to the one of the simple probability monad, in that it represents the fuzzy membership function of the results of f given a fuzzy uncertainty in the argument of f.

2.4 Recursive datatypes, algebras, coalgebras.

Haskell allows (in fact, encourages) the definition of recursive datatypes such as

```
data ℕ = Zero | Succ ℕ
```

the natural numbers, or

```
data List1 a = Wrap a | Cons (a, List1 a)
```

the datatype of non-empty lists. The most popular such datatype is that of lists, predefined in the standard prelude, which uses a more convenient notation:

```
data [a] = [] | a : [a]
```

The semantics of these datatype declarations are given by the fixed points of the functors read from the right-hand sides as, respectively: $FNat\ X = 1 + X$, $FList1\ A\ X = A + A \times X$, $FList\ A\ X = 1 + A \times X$. The last two functors are parametrized on the type A.

In Haskell, these functors are represented by the declarations

```
newtype FNat x = N  (Either () x)
newtype FList1 a x = L1 (Either a (a, x))
newtype FList a x = L (Either () (a, x))
```

(The last declaration corresponds to a curried version of the constructors of $[a]$, and therefore is only isomorphic to the functor underlying $[a]$).

together with instance declarations:

```
instance Functor FNat where
  fmap f (N (Left ()))  = N (Left ())
  fmap f (N (Right x)) = N (Right (f x))


instance Functor (FList1 a) where
  fmap f (L1 (Left a))        = L1 (Left a)
  fmap f (L1 (Right (a, x))) = L1 (Right (a, f x))


instance Functor (FList a) where
  fmap f (L (Left ()))        = L (Left ())
  fmap f (L (Right (a, x))) = L (Right (a, f x))
```

The expression "the fixed point" is justified in the category *CPO* (pointed complete partial orders) which underlies much of the semantics of Haskell, or in *Rel* (the category of relations). There, these functors have only one fixed point. In *Set*, the situation is somewhat more complicated: the functors we consider have a least fixed point and a different greatest fixed point. Moreover, there exist functors which have a fixed point in *CPO*, but not a greatest fixed point in *Set* (for example, $F\ X = A \times X$).

If F is a functor, then a function of type $F\ X \to X$ is called an *F-algebra*, or, if there is no danger of confusion, just *algebra*, and X is called the *carrier* of the F-algebra. The **data** declarations above introduce the following algebras:

$$\begin{array}{ll}
[\mathit{Zero}, \mathit{Succ}] & :: \mathit{FNat}\ \mathbb{N} \to \mathbb{N} \\
[\mathit{Wrap}, \mathit{Cons}] & :: (\mathit{FList1}\ a)\ (\mathit{List1}\ a) \to \mathit{List1}\ a \\
[[\,], (:)] & :: (\mathit{FList}\ a)\ [a] \to [a]
\end{array}$$

A morphism between two F-algebras $f :: F\ X \to X$ and $g :: F\ Y \to Y$ is a function $h :: X \to Y$ such that

$$h \cdot f = g \cdot \mathit{fmap}\ h$$

If T is the least fixed point of a functor F, then the F-algebra that witnesses the isomorphism $\mathit{alpha} :: F\ T \to T$ has the following property: for any other F-algebra $f :: F\ X \to X$ there exists a unique function $h :: T \to X$ such that

$$h \cdot \mathit{alpha} = f \cdot \mathit{fmap}\ h$$

That is, there exists a unique morphism from *alpha* to any other F-algebra: *alpha* is an initial object in the category which has as objects F-algebras and as arrows the morphisms between them. Because of this, *alpha* is called an initial algebra. The algebras introduced by the data declarations above witness unique fixed points in *CPO*, therefore also least fixed points, and are therefore all initial.

The initiality of these algebras is what guarantees that the common *fold* functions actually exist. These are functions which construct the unique morphism given an arbitrary algebra. For example:

$$\begin{array}{ll}
\mathit{foldn} & :: (\mathit{FNat}\ x \to x) \to \mathbb{N} \to x \\
\mathit{foldn}\ f\ \mathit{Zero} & = f\ (N\ (\mathit{Left}\ ())) \\
\mathit{foldn}\ f\ (\mathit{Succ}\ n) & = f\ (N\ (\mathit{Right}\ (\mathit{foldn}\ f\ n)))
\end{array}$$

$$\begin{array}{ll}
\mathit{foldl1} & :: (\mathit{FList1}\ a\ x \to x) \to \mathit{List1}\ a \to x \\
\mathit{foldl1}\ f\ (\mathit{Wrap}\ a) & = f\ (\mathit{L1}\ (\mathit{Left}\ a)) \\
\mathit{foldl1}\ f\ (\mathit{Cons}\ (a, as)) & = f\ (\mathit{L1}\ (\mathit{Right}\ (a, \mathit{foldl1}\ f\ as)))
\end{array}$$

$$\begin{array}{ll}
\mathit{foldl} & :: (\mathit{FList}\ a\ x \to x) \to [a] \to x \\
\mathit{foldl}\ f\ [\,] & = f\ (L\ (\mathit{Left}\ ())) \\
\mathit{foldl}\ f\ (a : as) & = f\ (L\ (\mathit{Right}\ (a, \mathit{foldl}\ f\ as)))
\end{array}$$

Examples of *fold* functions are ubiquitous in programming: the classic introduction to functional programming [3] and its successor [1] contain hundreds of examples. For instance:

```
plus m = foldn f
  where
  f (N (Left ())) = m
  f (N (Right x)) = Succ x
```

```
max1 = foldl1 f
        where
        f (L1 (Left a)) = a
        f (L1 (Right (a, x))) = max a x
```

```
sum = foldl f
       where
       f (L (Left ())) = 0
       f (L (Right (a, x))) = a + x
```

The uniformity of these *fold* functions virtually guarantees that their creation can be described generically. Indeed, the fixed point of a functor can be introduced by the declaration:

```
newtype (Functor f) ⇒ FixP f = In (f (FixP f))
```

The initial algebra associated with *FixP f* is *In*. Thus, the following isomorphisms hold:

$$\begin{aligned} \mathbb{N} &\equiv \mathit{FixP}\ \mathit{FNat} \\ \mathit{List1}\ a &\equiv \mathit{FixP}\ (\mathit{FList1}\ a) \\ [a] &\equiv \mathit{FixP}\ (\mathit{Flist}\ a) \end{aligned}$$

The initiality of *In* implies that

$$\mathit{fold}\ f \cdot \mathit{In} \equiv f \cdot \mathit{fmap}\ (\mathit{fold}\ f)$$

for any algebra f. This is not a computational characterization, but we can use the inverse of the isomorphism *In*:

```
out :: (Functor f) ⇒ FixP f → f (FixP f)
out (In x) = x
```

and define the *fold* function generically:

```
fold :: (Functor f) ⇒ (f a → a) → FixP f → a
fold f = f · fmap (fold f) · out
```

We have:

```
fplus m = fold f
  where
  f (N (Left ())) = m
  f (N (Right n)) = In (Right (In (Right n)))
```

```
fmax1 = fold f
  where
  f (L1 (Left a)) = a
  f (L1 (Right (a, x))) = max a x
```

```
fsum = fold f
      where
      f (L (Left ())) = 0
      f (L (Right (a, x))) = a + x
```

An initial algebra such as *In* is always an isomorphism, but in *CPO* and *Rel*, though not in *Set*, the inverse of *In* is a *final coalgebra.* Coalgebras are the duals of algebras: thus, arrows $f :: X \rightarrow F\ X$ for some functor F. A morphism from $f :: X \rightarrow F\ X$ to $g :: Y \rightarrow F\ Y$ is a function $h :: X \rightarrow Y$ such that:

$$g \cdot h = \mathit{fmap}\ h \cdot f$$

A coalgebra is *final* if it is a terminal object in the category which has as objects coalgebras of F and as arrows morphisms between them. Thus, the finality of *out* implies that for any $g :: x \rightarrow f\ x$ there exists a unique function $h :: x \rightarrow \mathit{FixP}\ f$ such that:

$$\mathit{out} \cdot h = \mathit{fmap}\ h \cdot g$$

As in the case of initial algebras, this characterization can be turned into a computationally adequate definition by using the fact that the inverse of *out* is *In*:

$$h = \mathit{In} \cdot \mathit{fmap}\ h \cdot g$$

The pattern of constructing, for any coalgebra, the corresponding function h is called *unfold* and is defined by:

```
unfold   :: (Functor f) ⇒ (a → f a) → a → FixP f
unfold f = In · fmap (unfold f) · f
```

unfold f is characterized by the *universal property*:

$$\phi = \mathit{In} \cdot \mathit{fmap}\ \phi \cdot f \equiv \phi = \mathit{unfold}\ f$$

As a simple example, we present two programs related to the Collatz conjecture. This conjecture states that the following process terminates for any given argument n:

```
coll :: Integer → Integer
coll n = if n ≡ 1 then 1 else coll m
  where
  m = next n
next n = if even n then n `div` 2 else (3 ∗ n + 1)
```

In other words, $\forall n,\ \mathit{coll}\ n = 1$.

Then, counting the number of steps the argument requires in order to reach 1 is given by:

```
countN :: Integer → FixP FNat
countN = unfold f
           where
           f n = if n ≡ 1 then (N (Left ()))
             else (N (Right (next n)))
```

and collecting the trajectory required to reach 1 is given by:

```
trjL1 :: Integer → FixP (FList1 Integer)
trjL1 = unfold f
         where
         f n = if n ≡ 1 then (L1 (Left 1))
           else (L1 (Right (n, next n)))
```

Probably the best way to understand these programs is to rewrite them in terms of the original data declarations:

```
count   :: Integer → ℕ
count n = if n ≡ 1 then Zero
            else Succ (count (next n))


trj   :: Integer → List1 Integer
trj n = if n ≡ 1 then (Wrap 1)
          else Cons (n, trj (next n))
```

The generic definitions are perhaps more of interest in formulating, proving and applying properties pertaining to all recursive datatypes, allowing, among others, to implement compiler optimization techniques (fusion, deforestation, etc.), or specify properties of dynamical systems and abstract data types as in Chapter 5, than for writing programs that are meant to be understandable and maintainable. The fusion theorem for *unfold*, for example, is:

Theorem 1 (Unfold fusion) *Let* $f :: x \to F\ x$, $h :: y \to F\ y$ *and* $g :: y \to x$. *Then*

$$\mathit{unfold}\ f \cdot g = \mathit{unfold}\ h \Leftarrow f \cdot g = \mathit{fmap}\ g \cdot h$$

The proof is a simple application of the unicity of *unfold*:

$$\begin{array}{ll} & \mathit{In} \cdot \mathit{fmap}\ (\mathit{unfold}\ f \cdot g) \cdot h \\ = & \{\ \text{fmap functor}\ \} \\ & \mathit{In} \cdot \mathit{fmap}\ (\mathit{unfold}\ f) \cdot \mathit{fmap}\ g \cdot h \\ = & \{\ \text{hypothesis}\ \} \\ & \mathit{In} \cdot \mathit{fmap}\ (\mathit{unfold}\ f) \cdot f \cdot g \\ = & \{\ \text{definition of } \mathit{unfold}\ f\ \} \\ & \mathit{unfold}\ f \cdot g \end{array}$$

Therefore, by the universal property of of *unfold*, $\mathit{unfold}\ f \cdot g = \mathit{unfold}\ h$.

Since lists in Haskell are not given by *FixP* definitions, but introduced by more convenient special syntax, the *unfold* function operating on them is written explicitely.

$$\begin{array}{ll} \mathit{unfoldl} & :: (x \to \mathit{Bool}) \to (x \to a) \to (x \to x) \to \\ & \quad x \to [a] \\ \mathit{unfoldl}\ p\ f\ g\ x = & \mathbf{if}\ p\ x\ \mathbf{then}\ [\,] \\ & \mathbf{else}\ (f\ x) : \mathit{unfoldl}\ p\ f\ g\ (g\ x) \end{array}$$

and the fusion theorem translates to

$$\begin{array}{l} \quad \mathit{unfoldl}\ p\ f\ g \cdot h = \mathit{unfoldl}\ p'\ f'\ g' \\ \Leftarrow \\ \quad p' = p \cdot h \\ \quad f' = f \cdot h \\ \quad h \cdot g' = g \cdot h \end{array}$$

We shall use this result in Chapter 5.

Chapter 3

Definitions of Vulnerability

In this chapter, we review some representative definitions of vulnerability. We start with the ordinary language definition, and then move on to the more technical ones which appear in the literature on Global Environmental Change.

3.1 The Oxford Dictionary of English definition

The latest edition of the Oxford Dictionary of English gives the following definition for "vulnerable" [35]:
vulnerable (adj.):

1. exposed to the possibility of being attacked or harmed, either physically or emotionally: *we were in a vulnerable position* | *small fish are* vulnerable to *predators*

2. Bridge (of a partnership) liable to higher penalties, either by convention or through having won one game towards a rubber.

Vulnerability according to the ODE is thus the condition of being vulnerable, and "vulnerable" is an adjective, a property that is predicated of something. This something is, in the context of the definition, the entity exposed to the possibility of harm. In the first example sentence, it is the position, in the second example sentence it is the small fish, and in the context of the game of Bridge, it is the partnership. The first of these is somewhat surprising: one would expect "vulnerable" to be predicated of the subject of the sentence, "we", rather than of the position. However, this is an example of a transferred epithet (a *hypallage*), and the sentence can be interpreted as "we were in a position in which we were vulnerable".

The second example sentence introduces an *adjective complement*: the idea of vulnerability *to* something. The small fish are exposed to the possibility of being harmed or attacked *by the predators*. Here, "vulnerable" becomes a binary predicate, since it is relative not just to the entity exposed to the possibility of harm, but also to the cause of that harm. This is the typical usage of vulnerable in the context of Climate Change studies: "vulnerable *to* climatic change".

As a final remark, we note the potential aspect of vulnerability. The entity that is said to be vulnerable is not "exposed to harm", but "exposed to *the possibility* of harm".

3.2 Vulnerability in the context of poverty analysis

In Development studies, the term "vulnerability" has gained prominence after being used in the 2000/1 World Development Report, where it was defined as "a measure of resilience against a shock – the likelihood that a shock will result in a decline in well-being" ([39], p.139). Vulnerability is here no longer a boolean predicate, but a measure which in general is going to take values other than *True* or *False*.

We can interpret this definition as a specialisation of the ordinary language usage, in the following way. The general idea of "being attacked or harmed" is replaced by the more specialised "suffers a decline in well-being". The "likelihood of ..." refines perhaps the idea of "exposure to the possibility of ...", suggesting that zero likelihood represents impossibility, and that harm might be more or less possible.

Many other conceptualisations have been proposed in order to "operationalize" the World Bank definition, or to account for aspects that were felt lacking, such as taking into account the magnitude of the decline in well-being, not just the likelihood of that decline. In [4], Calvo and Dercon propose the following definition of vulnerability *to poverty* as a synthesis of these various efforts:

> vulnerability is the magnitude of the threat of future poverty
> where we have

1. The "magnitude of the threat" combines the likelihood of suffering poverty in the future, as well as the severity of the poverty in that case.

2. Vulnerability is "an ex-ante statement about future poverty", that is, it is a statement about an uncertain future.

3. This definition is meant to apply only to a particular situation: "[...] we are referring to vulnerability *to poverty.* Individuals face several other threats such as illness, or crime, or loneliness. Yet we focus on poverty in particular, as this was also the focus other authors arguably

> had in mind when using the term 'vulnerability'. We thus understand expressions such as 'vulnerability to an epidemic' as a shortcut to 'vulnerability to poverty due to an epidemic'. "

The last remark is interesting in the light of the second example sentence in the Oxford Dictionary: "small fish are vulnerable to predators". If we take some notion of "harm" that befalls the fish as analogous to poverty, then Calvo and Dercon seem to suggest that this sentence is an abbreviation of "small fish are vulnerable to harm due to predators". In the Oxford Dictionary, "vulnerable to" was relative to the factors that induced the potential harm, here, "vulnerable to" is relative to the potential harm induced by the given factors.

Calvo and Dercon propose a set of requirements on a measure of vulnerability, which further elucidate their definition:

1. Vulnerability measures a set of outcomes across possible states of the world. These states of the world are assumed to be in finite number and come with associated probabilities.
2. Poverty is defined in terms of a threshold, which has the same value in all states of the world.
3. The states of the world in which the outcomes are above the threshold do not enter in the vulnerability measurement (this is called the "axiom of focus").
4. Monotonicity requirements: the likelier the outcomes below the threshold, and the greater their distance to the threshold, the greater the vulnerability.

A measure of vulnerability to poverty which satisfies these requirements has then the form:

$$V = sum\ [p\ i * v\ (x\ i) \mid i \leftarrow [1\,..\,n]]$$

where

$n :: \mathbb{N}$	-- the number of possible states of the world
$p :: \mathbb{N} \to [0,1]$	-- p i is the probability of state i
$v :: \mathbb{R} \to \mathbb{R}$	-- a monotonically decreasing and convex function
$x\ i = (y\ i)\ /\ z$	-- relative distance to threshold of outcome i
$y :: \mathbb{N} \to \mathbb{R}$	-- y i is the outcome in state i if below the
	-- threshold, 0 otherwise
$z :: \mathbb{R}$	-- the threshold

This measure generalises many of those proposed in the literature on Development Studies.

3.3 The IPCC definition of vulnerability

Within the Climate Change research community, the most influential definition of "vulnerability" is given by the Intergovernmental Panel of Climate Change in its assessment reports. The most recent of them, the Fourth Assessment Report [32], contains the following:

> *vulnerability*: the degree to which a system is susceptible to and unable to cope with, adverse effects of climate change, including climate variability and extremes. Vulnerability is a function of the character, magnitude and rate of climate variation to which a system is exposed, its sensitivity, and its adaptive capacity.

As in the previous section, vulnerability is not a boolean predicate, but one admitting degrees: "the degree to which ...". We can interpret the IPCC definition as a specialisation of the ODE one: there is a notion of "harm", phrased as the occurence of "adverse effects" with which the system is "unable to cope". The potentiality aspect is expressed by "susceptible" (versus, for example, "affected"). The role of climate change ("including climate variability and extremes") is that of an adjective complement, similar to the "predators" to which the small fish were said to be vulnerable. Here, the notion is of vulnerability *to climate change*. This is in contrast to the vulnerability *to poverty* defined in the previous section: climate change is the cause of harm here, poverty was the harmful effect there.

On the whole, there is an intention of making the context of vulnerability statements more precise than in the previous two sections. This is also apparent from the evolution of this definition, compare for example an earlier version (in the Second Assessment Report [38]): "vulnerability defines the extent to which climate change may damage or harm a system".

Along the same lines, the second sentence makes explicit some of the determinants of vulnerability. Besides a characterization of the climate change factors that vulnerability is considered relative to, one should also take into account the "sensitivity" and the "adaptive capacity", defined as follows.

> *sensitivity*: the degree to which a system is affected, either adversely or beneficially, by climate variability or change. The effect may be direct (e.g., a change in crop yield in response to a change in the mean, range or variability of temperature) or indirect (e.g., damages caused by an increase in the frequency of coastal flooding due to sea-level rise).

adaptive capacity: the ability of a system to adjust to climate change (including climate variability and extremes) to moderate potential damages, to take advantage of opportunities, or to cope with the consequences.

In the following, we present three representative assessments of vulnerability to climate change, all of which take the IPCC definitions as a starting point.

3.3.1 Vulnerability study of O'Brien et al. in India, 2004

In [31], O'Brien et al. describe their approach as "operationalizing the IPCC definition". The entities whose vulnerability to climate change was assessed were the agricultural units within the 466 (in 1996) districts of India: the same kind of calculation was done for each of them. Vulnerability was computed as the sum of two indices: an index of climate sensitivity under exposure, and another of adaptive capacity. Details about the data sources and the normalisation procedures used to compute each of the indices involved are given in [24].

The index of climate sensitivity under exposure results, in turn, from the average of two other indices: the monsoon dependency index and the dryness index. Monsoon dependency is considered proportional to the percentage of water received in the monsoon season (the four consecutive months with the largest amount of rainfall in the year). The dryness index is computed as a ratio of average potential evapotranspiration to precipitation. The data used to obtain these indices is produced by a downscaled general circulation model, HadRM2, and calibrated with measurements available for a period of thirty years (1961-1990).

The model HadRM2 is actually run twice: once, with control parameters set at the measured levels of 1990, another time with parameters consistent with a doubling of the CO2 levels in the atmosphere. The point-wise differences between the two runs are combined with the available measured data to obtain a final data set from which the two indices are calculated. This procedure is meant to give an estimate of the influence of climate change on the agricultural units in the region: the farmers in the regions with the highest monsoon dependency index are those most likely to develop farming practices which depend on the amount of rainfall received in the monsoon season, thus they are also the most likely to be affected by climate induced variations of rainfall. Variables describing climate change are estimated to lie within the range of values obtained when the model is driven by concentrations of CO2 levels in the atmosphere between current (when the study was undertaken) and double the current levels.

The adaptive capacity is calculated as the average of (normalized) values of several other indices, classified in biophysical (depth of soil cover, severity of soil degradation, amount of replenishable groundwater available annually), socioeconomic (adult literacy rates, degree of gender equity, percentage of workforce employed in agriculture, percentage of landless laborers in the agricultural workforce), and techological (net irrigated area as percentage of net sown area, the Infrastructure Development Index of the Center for Monitoring of Indian Economy).

It is argued that these indices provide an estimate of the entities' adaptive capacity. For example, when the adult literacy rates are higher, available information can be used to "moderate potential damages, take advantage of opportunities, or cope with the consequences" of climate change.

It is interesting to note that the indices which determine the adaptive capacity are measured at their present levels, whereas those that represent the sensitivity under exposure are calculated from estimated future values. Thus, in terms of the dictionary definition, the sensitivity under exposure would represent the potential aspect, the "possibility of harm", while the adaptive capacity would serve to characterize the "current position".

3.3.2 The ATEAM project

Within the ATEAM project (Advanced Terrestrial Ecosystem Analysis and Modeling) [29] the IPCC definition was taken as the starting point for an assessment of the vulnerability of various regions of Europe "to the loss of particular ecosystem services, associated with the combined effects of climate change, land use change, and atmospheric pollution". (Here we see "vulnerability to" used both with respect to the criteria of harm, "loss of ecosystem services", as above in the context of Development Studies, and with respect to the potential causes of this harm, "climate change, land use change, and atmospheric pollution", as common in the Climate Change community.)

In ATEAM, the second sentence of the IPCC definition is interpreted as asserting a functional dependence between vulnerability and exposure, sensitivity and adaptive capacity. In fact, the authors considered that exposure and sensitivity determine the potential impacts, and that they only enter the computation of vulnerability via these potential impacts. Vulnerability is then a function of the potential impacts and adaptive capacity only.

An impact was defined in terms of differences in ecosystem service levels in a region. The ecosystem services considered were agricultural (farmer livelihood, consumer food quality, support of rural communities, etc.), biodiversity related (number of plant species, bird species, etc.), energy or biomass related (carbon storage, energy supply), water related (drinking water, irrigation, hydro-power, etc.) and forestry (wood production and supply, carbon storage). The regions considered covered the entire territory of the European Union and not all ecosystem services were present in each region.

A level was computed for each ecosystem service, as the value of a relevant indicator, possibly with changed sign, so that the largest value represented the best ecosystem service level. For example, the "carbon storage" ecosystem service level was given by the net biome exchange, which was provided by the LPJ vegetation model.

The changes in the values of these levels were computed using various computer models for the evolution of the variables associated with the level indicators. For example, the evolution of the net biome exchange was computed using the LPJ model. The evolution of some indicators were computed using several models: evolution was considered non-deterministic. Several possibilities of evolutions were computed even for those indicators for which only one model was used, because each model was run several times with input data and parametrisations compatible to each of the four SRES storylines in [30].

For each resulting possible evolution, a potential impact for each ecosystem service level present in a region was computed. The potential impact was just the difference between (normalised) values of current and future ecosystem service levels. The normalisation was done in terms of the maximal level of the ecosystem service level in a cluster of regions which were *neighbour* of the region considered.

In order to compute the second argument of the vulnerability function, adaptive capacity, a distinction was made between "overarching management choices" and the "capacity of regions for macro-scale adaptation". The first one was represented by the use of the four SRES storylines, which had been used in the development of the possible evolutions of service indicators. The second one was computed for each region as an "adaptive capacity index" in a manner similar to that used by O'Brien et al. described in the previous section. The adaptive capacity index of a region was computed using a fuzzy aggregation of the values of several indices, including GDP per capita, female activity rate, income inequality, number of patents and age dependency ratio. Again, computational models were used to estimate the evolution of the adaptive capacity index in every region, according to each of the four SRES storylines.

After having computed the two arguments of the vulnerability function, the authors remark:

> The last step, the combination of the potential impact index (PI) and the adaptive capacity index (AC), is however the most dangerous step, especially when taking into account our limited understanding of adaptive capacity. We therefore decided to create a visual combination of PI and AC without quantifying their relationship.

We note that this can also be interpreted by choosing the function that gives the vulnerability as the identity function on $[-1, 1] \times [0, 1]$ (since the PI is a difference of normalised positive index and the adaptive capacity index is also normalised). For example, we may have only a partial order on the results of vulnerability assignements:

$$(\mathit{pi1}, \mathit{ac1}) < (\mathit{pi2}, \mathit{ac2}) = \mathit{pi1} < \mathit{pi2} \text{ `}\mathit{and}\text{` } \mathit{ac1} < \mathit{ac2}$$

3.3.3 The Luers et al. method for quantifying vulnerability

In [28], Luers et al. start from "a general definition of vulnerability as susceptibility to damage", which accords with the IPCC definition ("degree to which the system is susceptible [...] to adverse effects") and then "propose a new approach to quantifying vulnerability that integrates four essential concepts: the state of the system relative to a threshold of damage, sensitivity, exposure and adaptive capacity". The approach is then applied to assess the "vulnerability of wheat yields in the Yaqui Valley, Mexico, to climate variability and change, and market fluctuations".

Similar to the Development Studies approach, Luers et al. introduce a measure of well-being, and damage is considered to occur if the value of well-being falls below a given threshold. The well-being depends on the values of a "stressor": there is a parabolic function $W :: X \to \mathbb{R}+$ which, given the value of a stressor $x :: X$ returns the resulting value of the well-being, $w\ x \geqslant 0$. The set of stressor values is considered a subset of $\mathbb{R}$. The threshold well-being is denoted $\mathit{w\_0} :: \mathbb{R}$, and the stressor values are considered to be distributed according to a given probability distribution $p :: X \to [0, 1]$.

For a given stressor value, x, the sensitivity of the entity to that stressor is defined as the absolute value of the derivative of w computed at x:

$$\mathit{sensitivity}\ x = \mathit{abs}\ (\mathit{deriv}\ w\ x)$$

Since w is a parabolic function, it has only one optimum which we denote by $(\mathit{x\_opt}, \mathit{w\_opt})$. The derivative of w at this optimum is, of course, zero, and the sensitivity function thus defined induces the same preorder on X as the functions: $\lambda x \to \mathit{abs}\ (\mathit{x\_opt} - x)$ and $\lambda x \to \mathit{abs}\ (\mathit{w\_opt} - (w\ x))$ (we can drop the *abs* from the latter if the optimum point is a maximum). In other words, sensitivity can be seen as a kind of measure of the distance from optimality: the farther from optimality, the greater the sensitivity.

The vulnerability of an entity to a stressor distributed according to p is then computed as the expected value of the ratio between sensitivity and well-being relative to the threshold:

$$\mathit{vulnerability}\ p = \mathit{sum}\ [(\mathit{sensitivity}\ x\ /\ (w\ x\ /\ \mathit{w\_0})) * p\ x \mid x \leftarrow X]$$

In the deterministic case, in which the value x of the stressor is known, so that $p\ x = 1$ and $p\ x' = 0\ \mathit{for}\ x' \not\equiv x$, we have that the value of *vulnerability*, given by

$$sensitivity\ x\ /\ (w\ x\ /\ w\_0)$$

is again a measure of the distance from the optimal point. In the general case, vulnerability gives us a measure of the expected distance from the optimum. The induced preorder is independent of the threshold w_0.

The form of this vulnerability measure is very similar to the one given by Calvo and Dercon, discussed in 3.2. The function v used there depended on values of the outcome, not of a stressor, in other words, it was a function of $r = w\ x\ /\ w\_0$, and not of x. However, if we are just interested in the vulnerability "ranking", that can be just as well computed using the distance of $w\ x$ from w_opt instead of the sensitivity, that is, we could take v to be

$$v\ r = (w\_opt - r * w\_0)\ /\ r$$

(we assume that w_opt is a maximum). This would violate the assumption of *focus* that Calvo and Dercon put forward: $v\ r > 0$ even when $r > 1$, that is, when the outcome is above the threshold.

The problem with this analysis is that the threshold value plays different roles in the development measures described above and here. The measure of Luers et al. is, in fact, consistent with the Calvo and Dercon measure if one considers the threshold value to be w_opt, and not w_0. The latter plays here just the role of a scaling factor, unimportant if one is interested only (or mostly) in the induced preorders. The threshold value which satisfies the focus axiom and is consistent with the rest of the requirements of Calvo and Dercon is, in fact, w_opt.

The last "essential concept" in the framework proposed by Luers et. al, adaptive capacity, is defined as "the extent to which a system can modify its circumstances to move to a less vulnerable condition", which is consistent with the IPCC characterization as "the ability of a system [...] to moderate potential damages". The following measure of adaptive capacity is then proposed:

```
adaptive_capacity = v_e − v_m
  where
  v_e = ...   -- current vulnerability
  v_m = ...   -- potentially "better" vulnerability
```

that is, adaptive capacity is measured as the difference in the vulnerability under existing conditions and under the less vulnerable conditions to which the system could possibly shift. It is assumed that $v\_m < v\_e$. The authors argue that the IPCC characterization of vulnerability as "a function of [...] adaptive capacity" only applies to the potentially minimal vulnerability, that is, it considers only $v\_m = v\_e - adaptive\_capacity$, and not the existing vulnerability, v_e.

In the application of this approach to the assessment of the vulnerability to climate change of farmers in the Yaqui valley, the well-being was defined as the annual wheat yield obtained as a function of average temperature during the growth season. (A farmer in this study was identified with a farming unit, i.e. a 30 $m \times 30$ m parcel of agricultural land.) The function coefficients were obtained by regression on a data set consisting of yield and temperature values coming from satellite observations during four years. Statistical analysis of a time series of temperatures recorded in the Yaqui valley in a 30 years interval was used to determine the probability distribution of temperature values for the next year, and this was used in order to compute the vulnerability of each farmer. The difference between the vulnerability values thus computed and the best values of vulnerability actually observed in the past data was offered as a measure of adaptive capacity.

3.4 Conclusions

We have examined a number of definitions of vulnerability, focusing on the ones most important to applications in the field of Climate Change, but seeing also the natural language definition given by the Oxford Dictionary, and the technical one as it appears in the context of Development Studies. There are some common features, chief among which is that vulnerability is viewed as a measure of a potential harm, but also a number of differences:

1. differences in the way "vulnerability to" is interpreted in the various contexts: in the field of Climate Change and in the Oxford Dictionary, "vulnerability to" refers to the causes of the potential harm; within Development Studies, it refers to the harm or adverse effects themselves;

2. differences in the the entitities that vulnerability is predicated of: sometimes, vulnerability is said to be of a state, or of a position (as in the OED), other times of an individual or a region;

3. differences in which the various determinants of vulnerability, such as "adaptive capacity", are defined and measured, even in the context of the same field.

In the next chapter we shall develop a mathematical model to capture the common features of these definitions, help in understanding the differences, and serve as a starting point for building software for computational vulnerability assessment.

Chapter 4

A Mathematical Model of Vulnerability

In the previous chapter, we have reviewed a number of definitions of "vulnerability", as used in various contexts: in everyday language, in Development Studies and within the Climate Change community. In the following, we shall attempt to synthetize the common aspects of these definitions in a computational model. This will serve as a starting point for developing software for vulnerability assessment tasks, and help clarify the differences between the various usages of vulnerability and related terms.

4.1 The basic model

As we have seen, common to all the definitions and uses of vulnerability presented in the previous chapter is (at least) the idea of potential negative outcomes, formulated by the Oxford Dictionary as "exposed to the possibility of being harmed". A first difficulty in modeling comes from having to deal with the concept of "possibility". There are several alternatives: for example, one could try to formulate vulnerability within standard modal logic with possibility. We have chosen here to consider a temporal view of possibility: something is possible if it could happen in the future. This interpretation is consistent with the example of vulnerability assessments we have examined: computer programs, scenarios, or some form of statistical analysis are used to determine the possible future evolutions, or characteristics of these future evolutions.

We have also seen that these future evolutions are only considered for a limited time horizon (in Development Studies, this horizon is typically of one year, in Climate Change it is 100 years). Correspondingly, in the following we assume that we are interested in possible future evolutions over a given, fixed, time interval.

Consider a set *States* of "states of the world", or "states of affairs". Then we model possibility by a function which tells us, given a state, which are the possible future evolutions of the world:

$$possible :: States \rightarrow [Evolutions]$$

Thus, *possible* s is a set (represented by a list) of possible future evolutions, which all start in s. The states that we consider are assumed to be such that we can find out, examining a pair (s, s'), all that has happened in the evolution of the world between s and s'. In particular, if we consider continuous evolutions over a time t, then from s and s' we can obtain the entire trajectory as a function $trj :: [0, t] \rightarrow States$ such that $trj\ 0 = s$ and $trj\ t = s'$. Thus, *Evolutions* is just a type alias for $(States, States)$:

$$\textbf{type}\ Evolutions = (States, States)$$

With this interpretation of possibility, we can translate "exposed to the possibility of being attacked or harmed" as "among the *possible* future evolutions there is at least one in which harmful condition (or an attack) has actually occurred". The simplest way to model this is by a predicate on the set of evolutions which detects the occurrence of a harmful condition:

$$harm :: Evolutions \rightarrow Bool$$

The extension of the predicate harm, i.e. the set of those pairs (s, s') for which $harm\ s \equiv True$, represents the set of evolutions of the world in which a harm has befallen *the entity under consideration*, and not the pair or the state of the world itself. In other words, the definition of *harm* presupposes the selection from all the elements that compose a state of the world of those which describe the condition of an entity which might be exposed to harm.

Finally, we can use the Haskell prelude function *any*, which checks if there exists an element of a given list which fulfills a given predicate, in order to attempt a first definition of vulnerability:

$$\begin{aligned}&vulnerable :: States \rightarrow Bool\\&vulnerable = any\ harm \cdot possible\end{aligned}$$

We can also consider a more explicit version, in which *harm* and *possible* are given as parameters to the function:

$$\begin{aligned}vuln :: &(States \rightarrow [Evolutions] \rightarrow \quad \text{-- possible}\\&(Evolutions \rightarrow Bool) \quad \rightarrow \quad \text{-- harm}\\&States \rightarrow Bool \quad \text{-- resulting predicate}\\vuln\ p\ h &= any\ h \cdot p\end{aligned}$$

The extension of the predicate *vulnerable* is represented by the states in which the entity under consideration might come to harm (or be attacked) in a possible future evolution. As such, we have a mathematical analogue of the transferred epithet we found in the first example given by the OED: "we were in a vulnerable position", which seemed to assign the exposure to harm to the "position", instead of to "us".

The use of "vulnerable" in the game of bridge also accords with this interpretation. The entity exposed to the possibility of harm is the partnership, the harm being the loss incurred by the greater (double) penalties. The states of affairs in which such a possibility exists are those which occur, for example, after the partnership has won a game towards the rubber.

The boolean view of vulnerability has the advantage of simplicity, and of capturing an important structural aspect of all the vulnerability definitions that we have seen, but it cannot acount for the various uses of the concept which we have examined in the previous chapter. It is clear that we would like, not just in assessing vulnerability in technical situations, but also in natural language, to be able to compare the vulnerability of different entities, or of the same entity in different situations. We would like a model of vulnerability which goes beyond "all or nothing", or "true or false", allowing us to talk, as required for example by the IPCC definition, of *degrees* of vulnerability.

To guide us in going from a boolean view of vulnerability to a more nuanced one, we can try to find a mathematical analogue of the structural pattern expressed by $any\ harm \cdot possible$. For a given state s, $possible\ s$ is a set of possible outcomes, *harm* is a function assigning values to the elements of that set, and *any* "measures" the set of resulting values with a boolean value. The immediate analogue of aggregating values of possible outcomes is, in mathematics, the *expected value of a random variable.*

Assume that $possible\ s$ returns a simple probability on the set of evolutions, instead of just a set of possible evolutions. That is, we take $possible\ s :: SimpleProb\ Evolutions$. The predicate *harm* can then be seen as a random variable, say $harm :: Evolutions \to \mathbb{R}$. The simple probability distribution describing the values taken by *harm* along the possible evolutions is $fmap\ harm\ (possible\ s)$. Finally, instead of combining the values of a boolean predicate by taking, as it were, the maximum of these values using *any*, we instead compute the expected value of the resulting simple probability distribution:

```
$stoch\_vuln\ s = expected\ (fmap\ harm\ (possible\ s))$
**where**
  $expected\ (SP\ xs) = foldl\ f\ 0\ xs$
    **where**
    $f\ (x, p)\ avg = x * p + avg$
```

This is, in fact, the formula used by Calvo and Dercon, as described in 3.2 above. Indeed, the system they use describes discrete one-step stochastic evolutions, so that fixing the initial state we have that

```
type Evolutions = States
```

A possible evolution from the given fixed state is given by the immediate next state. The function *possible* is stochastic, that is

```
possible :: States → SimpleProb States
```

States are represented by natural numbers, that is

```
type States = ℕ
```

and the harm or damage measures the distance below the given threshold z:

```
harm i = v (x i)
```

This stochastic interpretation of vulnerability is a generalisation of the boolean one, at least if we interpret values of 0 as *False*, and any others as *True*. Indeed, since we assumed that we work with finite sets represented as lists, we can take

```
list2SP :: [a] → SimpleProb a
list2SP as = SP [(a, r) | a ← as]
  where r = 1 / length as
```

Then, we have

```
vuln p h s = expected (fmap harm′ (list2SP (possible s))) > 0
  where harm′ e = if harm e then 1.0 else 0.0
```

However, it is not the case that all vulnerability measurements are expressible in terms of expected values of damage or harm. For example, this was not the case in the studies carried out by ATEAM or O'Brien et al. described above. We can expect the set of possible evolutions to have a different structure from *SimpleProb Evolutions*. It might be *Id*, or [*a*], or *SimpleFuzzy* or any combination of these. A general representation which includes all these cases is *F Evolutions* where *F* is a type constructor for which an instance declaration **instance** *Functor F* **where**... has been defined.

The type of the function *possible* becomes then:

```
possible :: States → F Evolutions
```

Similarly, we do not want to assume that the function measuring the harm or damages along a possible evolution takes real values, but we do want them to be comparable:

```
harm :: Evolutions → V
```

where V is a partially ordered type, that is one for which an instance declaration **instance** *PartialOrd V* **where**... has been supplied.

Finally, we replace the functions *any* and *expected* by general "measures" of the sets of results given by applying *harm* to the possible evolutions:

$$measure :: F\ V \to W$$

where, again, W is an instance of *PartialOrd*.

The intended meaning of this measure imposes the following constraint: we expect that if the values in the structured set $xs :: F\ V$ increase, but the structure does not change, then the measure of the result should not decrease. It is here that the functoriality of F comes into its own: without the concept of a functor, it would be quite difficult to express precisely what it means for the values to increase, while the "structure does not change". As it is, we require:

The monotonicity condition for vulnerability measures:

For all increasing functions $inc :: V \to V$ and for all $xs :: F\ V$, we have:

$$measure\ xs \sqsubseteq measure\ (fmap\ inc\ xs)$$

Assembling all these elements, we obtain the basic model for vulnerability:

Definition 2 (Vulnerability model) *With the notation above, we define vulnerability to be the function given by*

$$vulnerability = measure \cdot fmap\ harm \cdot possible$$

This expresses in a very general fashion the idea, common to all definitions of "vulnerability" that we have seen, of a measure of possible harm.

Remarks

1. In the stochastic case we have $F = SimpleProb$, $V = \mathbb{R}$, $W = \mathbb{R}$, and $measure = expected$. In the boolean case, $F = [\,]$, $V = Bool$, $W = Bool$, and $measure = maximum$.

2. The monotonicity condition for vulnerability measures is satisfied by both *maximum* and *expected*. To see that it is not vacuous, consider the seemingly inocuous measure of "likeliest harm". Formally, $F = SimpleProb$ and

```
measure :: SimpleProb V → V
measure (SP xs) = snd (maximum (map swap xs))
  where
  swap (x, p) = (p, x)
```

   The *swap* is necessary because in Haskell the order on pairs is the lexicographical ordering. Take

$$
\begin{aligned}
&\mathit{xs} = \mathit{SP}\ [(10, 0.4), (0, 0.3), (1, 0.3)]\\
&\mathit{inc} :: \mathbb{N} \to \mathbb{N}\\
&\mathit{inc}\ 0 \qquad\quad = 1\\
&\mathit{inc}\ (n+1) = n + 1
\end{aligned}
$$

The function *inc* is obviously non-decreasing, but $\mathit{measure}\ \mathit{xs} = 10 > \mathit{measure}\ (\mathit{fmap}\ \mathit{inc}\ \mathit{xs}) = 1$. Thus, the measure "likeliest harm or impact" fails the monotonicity condition and cannot be used as a vulnerability measure. In 7 we show how to check the monotonicity condition using *QuickCheck*.

3. The two examples of types of *Evolutions* given above are themselves constructed functorially from *States*. In fact, we would probably reject a representation for *Evolutions* which is not so constructed. That means that in general *Evolutions* is going to take the form *G States* for some functor *G* and we'll have

   $$\mathit{possible} :: \mathit{States} \to F\ (G\ \mathit{States})$$

   That is, *possible* is a coalgebra of the functor $F \cdot G$ with carrier *States*. Coalgebras are extensively applied in the study of dynamical systems and modal logics, and indeed the intuition behind the *possible* function is that of a dynamical system (which describes possible future evolutions). We develop this point of view further in the next chapter.

4.2 "Vulnerability to" and sensitivity

The basic model of vulnerability we have developed in the previous section covers the common aspects we have seen when examining various definitions from Chapter 3. In particular, it can be specialized to obtain the vulnerability definition of Calvo and Dercon. This definition was quite similar to the one given by Luers et al., so it seems natural that the Luers formula, in turn, is obtained as a special case.

The transitions of the state of the world were described there as being induced by a stressor, element of a type X, so that every possible evolution was associated to one value $x :: X$. The harm registered along such an evolution was a function of just this x which was influencing the evolution. Since the states of the world play no role in this description, we can take $\mathit{States} = ()$ and $\mathit{Evolutions} = (X, ())$. There was a probability distribution $\mathit{xp} :: \mathit{SimpleProb}\ X$ given, and the magnitude of vulnerability was computed as the expected value of the random variable given by the function $f\ x = \mathit{sensivity}\ x\ /\ (w\ x\ /\ w\_0)$. Putting it all together we have

```
luers_vuln = measure · fmap harm · possible
  where
  possible :: () → SimpleProb (X, ())
  possible ()   = fmap (λ(x, p) → ((x, ()), p)) xp
  harm (x, ()) = sensitivity x / (w x / w_0)
  measure      = expected
```

We can see here that the computation of *sensitivity* affects the way that the damage or the harm is estimated. In the formulation of Calvo and Dercon, the impact or damage was given by a term similar to the one given here as 1 / (*w x* / *w_0*), an estimate of the distance below the threshold which was nondecreasing with respect to this distance. The Luers formula no longer displays this monotonicity: one could have an increase in the distance below the threshold, and yet a smaller estimate of harm, because of a corresponding decrease in sensitivity.

In other words, the estimate of the harm suffered by the entity considered is no longer just a function of the impacts, for example the distance below a threshold, but also of the sensitivity to the stressor associated with those impacts. The reason given for this by Luers et al. is that what is computed by this formula is "vulnerability to stressors such as climate change", and the *sensitivity* function measures the influence of the stressors on the evolution of the system. Thus, the sensitivity, like the estimation of the impacts, can be measured along a given evolution:

```
sensitivity :: Evolutions → V1
```

```
impacts :: Evolutions → V2
```

```
harm :: Evolutions → V
harm = combine · pair (impacts, sensitivity)
```

where *V1*, *V2*, *V* are ordered types.

The idea that sensitivity measures the influence of the factors of interest on the potential harm assessed by the vulnerability measure can help us translate the prey-predators example given in the Oxford Dictionary: "small fish are vulnerable to predators". We can assert this sentence in a context or state of the world s if among the possible evolutions the fish are, say, wounded by the predators. That is, we check each possible evolution for the case in which the fish have been wounded, but also for whether these wounds have been caused by the predators. Thus, we assume the existence of two predicates on the set of possible evolutions:

```
wounded :: Evolutions → Bool
predators :: Evolutions → Bool
```

The first of these represents the analogue of the *impacts* above, the second measures (in a boolean way) the contribution of the factors of interest to the impacts, and corresponds thus to the *sensitivity*. The intended combination is in this case logical conjunction:

$$\begin{aligned}
&\mathit{harm} = \mathit{combine} \cdot \mathit{pair}\ (\mathit{impacts}, \mathit{sensitivity})\\
&\quad \textbf{where}\\
&\quad \mathit{impacts} \ = \mathit{wounded}\\
&\quad \mathit{sensitivity} = \mathit{predators}\\
&\quad \mathit{combine} = \mathit{and}
\end{aligned}$$

In O'Brien et al., leaving aside for now the adaptive capacity index, we have that vulnerability is equated with the climate sensitivity index. The model HadRM2 which was used to compute the possible evolutions was driven by the given initial state and by a chosen concentration of CO2 levels in the atmosphere, i. e. it had the form:

$$\begin{aligned}
&\mathit{model} :: \mathit{States} \to \mathit{Concentration} \to \mathit{Evolutions}\\
&\mathit{model}\ s\ c :: \mathit{Evolutions}
\end{aligned}$$

Concentrations were expressed in terms of multiples of the standard concentration levels measured when the study was conducted, so that a concentration of c meant a level c times bigger than the standard one. The study assumed that c could take values in the range of $[1.0, 2.0]$, thus, if we discretize this interval with a step of size 0.1, we have that

$$\mathit{possible}\ s = [\mathit{model}\ s\ c \mid c \leftarrow [1.0, 1.1 \mathinner{..} 2.0]]$$

The sensitivity associated to a possible evolution was given by a normalization of the difference between the values taken by certain variables of interest, such as rainfall or evapotranspiration, along this evolution and those taken along the standard evolution given by $\mathit{std} = \mathit{model}\ s\ 1.0$. Thus, the computation had the form:

$$\mathit{sensitivity}\ \mathit{ev} = \mathit{normalize}\ (\mathit{rfall}\ \mathit{ev} - \mathit{rfall}\ \mathit{std})$$

Finally, as explained in the previous chapter, the harm sustained by the entities of interest is proportional to this sensitivity, for example, taking $k > 0$:

$$\mathit{harm}\ \mathit{ev} = k * \mathit{sensitivity}\ \mathit{ev}$$

The model was run only twice: once with the standard value $c = 1.0$ and once with the maximal value $c = 2.0$. This was because the vulnerability measure used was *maximum* and it was assumed that the sensitivity of evolutions driven by larger concentrations of CO2 levels will be larger, in other words, that the function *sensitivity* · *model s* is monotonous. Indeed, with this assumption we have:

$$
\begin{aligned}
& \mathit{vuln}\ s = \mathit{maximum}\ (\mathit{map}\ \mathit{harm}\ (\mathit{possible}\ s)) \\
\equiv\ & \{\ \text{definition of possible}\ \} \\
& \mathit{vuln}\ s = \mathit{maximum}\ (\mathit{map}\ \mathit{harm}\ [\mathit{model}\ s\ c \mid c \leftarrow [1.0, 1.1\,..\,2.0]]) \\
\equiv\ & \{\ \text{definition of map}\ \} \\
& \mathit{vuln}\ s = \mathit{maximum}\ [\mathit{harm}\ (\mathit{model}\ s\ c) \mid c \leftarrow [1.0, 1.1\,..\,2.0]] \\
\equiv\ & \{\ \text{definition of harm}\ \} \\
& \mathit{vuln}\ s = \mathit{maximum}\ [k * \mathit{sensitivity}\ (\mathit{model}\ s\ c) \mid \\
& \quad c \leftarrow [1.0, 1.1\,..\,2.0]] \\
\equiv\ & \{\ \text{sensitivity . model s monotonous}\ \} \\
& \mathit{vuln}\ s = k * \mathit{sensitivity}\ (\mathit{model}\ s\ 2.0) \\
\equiv\ & \{\ \text{definition sensitivity}\ \} \\
& \mathit{vuln}\ s = k * \mathit{normalize}\ (\mathit{rfall}\ (\mathit{model}\ s\ 2.0) - \mathit{rfall}\ (\mathit{model}\ s\ 1.0))
\end{aligned}
$$

In the study conducted by ATEAM, sensitivity was a measure of the variation of the levels of ecosystems services, having the form:

$$\mathit{Evolutions} = (\mathit{States}, \mathit{States})$$

$$\mathit{sensitivity}\ (s0, s1) = \mathit{ecoLevel}\ s1 - \mathit{ecoLevel}\ s0$$

Of course, as explained in the previous chapter, the actual operation used was more complicated than $(-)$, involving a local averaging of the variables involved, etc. and the result of this operation was a multidimensional index, not a real number. As in O'Brien's study, the variables of interest were computed running models with initial states and with various inputs considered to describe the exposure to climate change: thus, the variations of these variables were taken as a good measure of the influence of these inputs on the evolution of the entities considered.

Potential impacts were then calculated as normalized sensitivities, in a manner consistent with viewing them as negative effects on the entity considered, so that

$$
\begin{aligned}
& \mathit{potential\_impacts}\ ev = \mathit{normalize}\ (\mathit{sensitivity}\ ev) \\
& \mathit{harm}\ ev = k * (\mathit{potential\_impacts}\ ev)
\end{aligned}
$$

for some $k > 0$.

We summarize the conclusions of this section in the following

Remarks.

1. In all the examples we have seen, sensitivity measures the influence of factors of interest (be they predators or climate change) along a

potential evolution, and the value of sensitivity enters the assessment of harm sustained along that evolution. The value of *harm* is either computed directly from the sensitivity, as in the studies of O'Brien and ATEAM, or combined with measurement of impacts, as in the study of Luers or in the Oxford Dictionary of English. In the latter case, the combination of the impacts and sensitivity is commutative: multiplication in Luers, logical conjunction in the ODE. The symmetry between impact measurements and sensitivity measurements is very similar to the symmetrical way in which the expression "vulnerability to" is used: Calvo and Dercon, who do not consider specific factors that influence the stochastic evolution of the system, say "vulnerability to poverty", where poverty can be regarded as the impacts or the harm suffered; O'Brien and ATEAM, who compute harm directly out of sensitivity, talk about "vulnerability to climate change", that is, vulnerability to those factors of interest whose contribution is measured by sensitivity. Luers et al., who have both impacts and sensitivity, use "vulnerability to stressors" whose effects are measured by sensitivity most of the time, but slip without comment into "vulnerability to poverty" or "vulnerability to food insecurity" when relating their work to existing literature. In a certain sense, the model of sensitivity outlined in this section justifies, or at least serves to explain, this usage.

2. When interested in assessing vulnerability *to* some factors of interest, then a measure of the contribution of these factors to potential harm may allow us, as in the O'Brien study, to reduce the number of potential evolutions to be considered. Often, it is difficult to measure the contribution of these factors on the resulting harm, but one can measure their influence on the evolution itself, and eliminate those evolutions on which this influence is negligible. This is consistent with the IPCC definition, where sensitivity is a measure of the "the degree to which a system is affected, *either adversely or beneficially*, by climate variability or change".

3. In some cases, sensitivity is used in an implicit way: a number of evolutions are modeled, and it is assumed that the resulting impacts are largely influenced by the factors of interest, e.g. climate change.

4.3 Adaptive Capacity

The major obstacle in carrying out a vulnerability assessment by applying the formula

$$\textit{vulnerability } s = \textit{measure } (\textit{fmap harm } (\textit{possible } s))$$

is, of course, that the set of possible evolutions is usually not computable. We have seen in the previous section that, when considering vulnerability *to* some factors of interest, we may be able to reduce, sometimes even drastically, the number of evolutions needed to compute the vulnerability. This was achieved by discarding the evolutions which were not at all, or insufficiently influenced by the factors of interest.

In this section, we describe a means of achieving a similar result, thinning the set of results produced by $possible\ s$, by considering the evolutions from the point of view of the entity under consideration.

The studies of vulnerability in the context of climate change which were examined in the previous chapter generally make use of the assumption that there are a number of "standard" scenarios which describe the possible evolutions of the climate. These scenarios are either summarized in the SRES storylines, as assumed in the study conducted by ATEAM, or derived from statistical data, as in the Luers et al. assessment, or both, as in the case of the study undertaken by O'Brien et al. These standard evolutions can be thought of as describing the possible evolutions at a macro scale, or with a lower degree of resolution.

Let us assume that the actual set of possible evolutions can be obtained from these standard evolutions by taking into account the actions of the entity considered. In particular, we want to consider the case in which each of the possible evolutions can be represented as a modification of a standard one depending on the course of action taken by the entity.

Formally, let *Actions* denote the type of *actions* or *courses of action* which the entity considered might take. In general, not all elements of *Actions* are available to the entity in a given context or state of the world s. For example, the set of potential investments is usually represented by $\mathbb{R}$, but the investments which can actually be carried out in a given context are limited by the available capital. We can represent the possibilities of actions of the entity in a given context by a function

$$doable :: States \rightarrow G\ Actions$$

where G is a functor. $doable\ s$ represents the structure of actions which may be undertaken in context s.

Let the standard evolutions be given by a function of the same kind as *possible*, but possibly returning a different structure of potential evolutions:

$$standard :: States \rightarrow H\ Evolutions$$

Each standard evolution may be modified by the actions of the entity. Let $\oplus :: H\ Evolutions \rightarrow G\ Actions \rightarrow F\ Evolutions$ be the operation which computes the resulting set of evolutions, that is

$$possible\ s = (standard\ s) \oplus (doable\ s)$$

For example, if $F = H = G = [\,]$ and every action $a :: \mathit{Actions}$ of the entity can modify an evolution according to a function $f :: \mathit{Evolutions} \rightarrow \mathit{Actions} \rightarrow \mathit{Evolutions}$, we have that

$$\begin{aligned}
&\mathit{possible}\ s = [f\ e\ a \mid e \leftarrow \mathit{standard}\ s, a \leftarrow \mathit{doable}\ s]\\
&\quad = (\mathit{standard}\ s) \oplus (\mathit{doable}\ s)\\
&\qquad \textbf{where}\\
&\qquad xs \oplus ys = [f\ x\ y \mid x \leftarrow xs, y \leftarrow ys]
\end{aligned}$$

Then:

$$\begin{aligned}
&\quad \mathit{vulnerability}\ s = \mathit{measure}\ (\mathit{fmap}\ \mathit{harm}\ (\mathit{possible}\ s))\\
\equiv &\quad \{\ \text{Express possible evolutions in terms of standard ones}\ \}\\
&\quad \mathit{vulnerability}\ s = \mathit{measure}\ (\mathit{fmap}\ \mathit{harm}\ ((\mathit{standard}\ s) \oplus\\
&\qquad (\mathit{doable}\ s)))\\
\equiv &\quad \{\ \text{introduce g, } \otimes \text{ explained below}\ \}\\
&\quad \mathit{vulnerability}\ s = (\mathit{std\_measure}\ (\mathit{fmap}\ \mathit{harm}\ (\mathit{standard}\ s)))\\
&\qquad \otimes (g\ (\mathit{doable}\ s))\\
\equiv &\quad \{\ \text{introducing stdVulnerability and adaptiveCapacity}\ \}\\
&\quad \mathit{vulnerability}\ s = (\mathit{stdVulnerability}\ s) \otimes (\mathit{adaptiveCapacity}\ s)
\end{aligned}$$

We have assumed that the functions *measure* and *harm* “distribute” over the computation of $\oplus$, so that the vulnerability can then be obtained as a measure of the potential harm registered along standard evolutions, combined with the measure of the local effects due to the actions of the entity. This term corresponds to what has been called “adaptive capacity” in the studies we have examined. Thus we have shown

Theorem 2 (Adaptive Capacity) *With the notation above, if for all* $he :: H\ \mathit{Evolutions}$ *and* $ga :: G\ \mathit{Actions}$

$$\begin{aligned}
&\mathit{measure}\ (\mathit{fmap}\ \mathit{harm}\ (he \oplus ga)) = (\mathit{std\_measure}\ (\mathit{fmap}\ \mathit{harm}\ he))\\
&\quad \otimes (g\ ga)
\end{aligned}$$

then

$$\mathit{vulnerability}\ s = (\mathit{stdVulnerability}\ s) \otimes (\mathit{adaptiveCapacity}\ s)$$

where

$$\mathit{stdVulnerability} = \mathit{std\_measure} \cdot \mathit{fmap}\ \mathit{harm} \cdot \mathit{standard}$$

and

$$\mathit{adaptiveCapacity} = g \cdot \mathit{doable}$$

1. In the study of Luers et al., "adaptive capacity" was obtained as the difference between two values of vulnerability: the "current" value of vulnerability and a past "optimal" one. This corresponds to the computation above, with $\otimes = +$ and the standard evolution taken to be the best evolution observed in the past.

2. In the study of O'Brien et al., "adaptive capacity" was a measure of the current state of the farming units under consideration from the point of view of their ability to adapt to future impacts, thus, a measure of the actions or courses of action available to them. The adaptive capacity index was subtracted from the macro-level climate sensitivity index, which, as we have seen in the previous section, corresponds to a vulnerability computation in which the values of harm are proportional to the sensitivity. Thus, in the formula above, the climate sensitivity index stands for *stdVulnerability* and $\otimes = -$.

3. In the study conducted by ATEAM, "adaptive capacity" was measured in a similar manner to the O'Brien study, therefore a measure of the "doable" actions. The last step, of combining the potential impacts, which can be understood as an indicator of standard vulnerability, with the adaptive capacity, was not undertaken: thus, $\otimes$ was left to the user of the assessment results.

Defining adaptive capacity as a measure of the influence of the actions or courses of action of the entity considered on vulnerability can also be seen as a mathematical formulation of the IPCC definition, especially if one consideres that the effect of these actions on vulnerability is expressed by "adjusting to climate change (including climate variability and extremes), moderating potential damages, taking advantage of opportunities, or coping with the consequences".

Within the climate change community, adaptive capacity is used to account for the lack of predictive models for the complex systems (especially the social systems) involved. However, as we have seen, the emphasis put on adaptive capacity in climate change vulnerability assessments can also be justified computationally: adaptive capacity is what allows us to actually compute the vulnerability by reducing the set of possible evolutions to manageable proportions.

4.4 Conclusions

In this chapter we have presented a mathematical model for vulnerability which captures the commonalities of the definitions and uses of vulnerability previously examined. The model expresses the idea that vulnerability is a measure of potential harm, generalizing the expected value models of Calvo and Dercon or Luers et al.

Additional concepts, such as sensitivity or adaptive capacity, refine our measurements of potential harm. Sensitivity, for example, is a way of taking into account the influence of factors of interest along a potential evolution, allowing us to model "vulnerability *to*" these factors (a very important aspect in climate change studies). Additionally, sensitivity may lead to a reduction of the computational effort, by eliminating from the structure of possible evolutions those which are not influenced (significantly) by the factors of interest.

A reduction of the computational effort can also be achieved by using a characterization of the initial state from the point of view of the actions available to the entity and their influence on the potential harm, and combining it with a vulnerability measured along a structure of standard evolutions. This usage of "adaptive capacity" is justified if the distributivity condition given above is fulfilled.

The computation of possible evolutions is complicated by the need to model the interactions between different types of systems: non-deterministic systems representing scenarios, deterministic models of physical processes or stochastic systems resulting from data analysis. The combination of such systems is an important topic of the next chapters.

Chapter 5

Dynamical Systems

In the previous chapter, we have modeled vulnerability as a measure of the potential future evolutions starting from a given state. In computational vulnerability, these future evolutions are described by using models of the various systems involved, which usually have heterogenous types. Indeed, it is a consensus in the Global Change and Sustainability Science communities that assessment of vulnerability in these fields can only be done by focusing on *social-ecological systems*, defined as "systems that include societal (human) and ecological (biophysical) subsystems in mutual interaction" ([13]). Such a complex system will replace the abstract *possible* function used in the previous chapter, providing the available information about the evolution of the system states.

The current emphasis in computational vulnerability assessment is on the usage of low and intermediate complexity models of the subsystems involved. That is, models which typically take minutes to run on modern PCs, in contrast, for example, with current state-of-the-art simulations of the climate system, which can take months to run on the fastest available parallel machines. The computational effort in vulnerability assessments arises mainly because of the interactions between the systems involved, and from the need to explore several scenarios of possible evolutions.

In order to assist the vulnerability assessment task, a software framework must allow the user to put together complex systems from simpler ones, to combine different types of systems and to take advantage of existing models. The design of such a framework is the subject of the next two chapters. We start by investigating the mathematical notion of "dynamical system". We then attempt to find the class of dynamical systems which represents the "minimal generality" necessary to represent the models used in practice, and which is closed with respect to the typical ways in which these models are combined.

The class of dynamical systems we settle on is that of "monadic systems". The next chapter then analyses various combinations of these systems, showing that they do indeed give rise to new monadic systems. We conclude with a discussion of the limitations of monadic systems.

5.1 Dynamical systems, classically

The usual way of defining a system in the fields of Engineering is in terms of the actions of a monoid on a set. A standard example is offered by Manfred Denker in [11] (page 4, my translation):

Definition 3 (Classical dynamical system) *Let T be a semigroup with unit element e and X a non-empty set. The tuple (X, T) is called a* dynamical system *if there exists an associative map*

$$\begin{array}{lcl} X \times T & \longrightarrow & X \\ (x,t) & \longrightarrow & tx \end{array}$$

for which the unit e acts as identity, that is, when the following two conditions hold:

$$(t_1 x, t_2) \longrightarrow t_2(t_1 x) = (t_2 t_1)x \text{ and } ex = x$$

By currying the map $X \times T \to X$, we obtain the somewhat simpler equivalent definition:

Definition 4 (Classical dynamical system, equivalent to 3) *Let $(T, +, 0)$ be a monoid and X an arbitrary set. A dynamical system is a monoid morphism from $(T, +, 0)$ to $(X \to X, \cdot, id)$, i.e. a function*

$$\phi :: T \to (X \to X)$$

such that

$$\begin{array}{ll} \phi\, 0 = & id \\ \phi\ (t\_1 + t\_2) = & \phi\ t\_2 \cdot \phi\ t\_1 \end{array}$$

The set X is called *the state space*, an element $x :: X$ is called a state of the system. The set T is usually intended to represent time, being usually $\mathbb{R}$ or $\mathbb{N}$, but it is important to understand that the elements of T are to be thought of as *time intervals* or *durations*, not as clock or calendar time. That is, $\phi\ t\ x$ represents the state of the system *after* t units of time since it was in state x.

If T is a monoid such as $\mathbb{R}$ or $\mathbb{N}$, then fixing $x_0 :: X$, we can compute the *trajectory* of the system 'starting' in x_0 as the graph of the function of time *flip* ϕ:

$$trj\ x_0 = \{(t, \phi\, x_0\ t) \mid t \in T\}$$

Of course, if, for example, $T = \mathbb{R}$, the subset of $\mathit{trj}\ x_0$ associated to negative elements of T will represent the past evolution of the system. For the case in which $T = \mathbb{N}$, we can use a list instead in order to define the trajectory:

$$[\phi\ x_0\ n \mid n \in \mathbb{N}]$$

The graph above can be obtained by pairing every element of the list with its index.

A *discrete dynamical system* is one for which $T = \mathbb{N}$ or $T = \mathbb{Z}$.

If $T = \mathbb{N}$, then denoting $f = \phi\ 1$, we have

$$\phi\ n = f^n$$

The function f is called the *transition function* of the system. When $T = \mathbb{N}$, the transition function is often identified with the system, since, by the above equation, any endo-function f uniquely determines a ϕ.

If $T = \mathbb{Z}$, the monoid morphism condition implies that the transition function is an isomorphism, with inverse $\phi\ (-1)$:

$$\phi\ 1 \cdot \phi\ (-1) = \phi\ (-1) \cdot \phi\ 1 = \phi\ 0 = \mathit{id}$$

In the following, we shall consider mainly the case $T = \mathbb{N}$ (and non-isomorphic transition functions).

In this classical setting, different types of system are distinguished by the existence of certain structures on X. Thus, in order to define linear systems, X must be a vector space, for stochastic systems we must have a probability space on X, for non-deterministic systems X must be a powerset, and so on. However, this structure on the state space is not sufficient to determine the type of the system. For example, for a liniar system, the transition function must be a liniar map, in other words an endo-map in $\mathit{Vect\_K}$, the category of vector spaces over a given field K.

Similarly, the powerset structure on X does not guarantee that the system is a non-deterministic one. Consider, for instance, the identity function on $P\ X$, the powerset of X. Intuitively, we would view this as the transition function of a deterministic system, whose states happen to be represented by subsets of X. The transition function of a non-deterministic system should be formulated in terms of a *relation*, that is, of a function which gives us for a given state a set of possible next states:

$$\mathit{nondet} :: X \to P\ X$$

The transition function would then be given by

$$\phi\ 1\ \mathit{xs} = \cup\{\mathit{nondet}\ x \mid x \in \mathit{xs}\}$$

Similarly, we would not view the identity function on $\mathit{SP}\ X$ as the transition function of a stochastic system, but as that of a deterministic system whose states happen to be represented by simple probability distributions. A stochastic system should have be formulated in terms of a function which tells us, given the current state, what the probability distribution of next states is, that is, should have the signature:

$$\mathit{stoch} :: X \to SP\ X$$

In particular, if X is finite, *stoch* could be represented as a stochastic matrix.

These and many other examples have led in Computer Science to the definition of dynamical system as *coalgebras*, that is, arrows of the form $X \to F\ X$ for some functor F. In the cases above, for $F = \mathit{Id}$ we obtain a deterministic system, for $F = P$ a non-deterministic system, and for $F = SP$ a stochastic one.

Remarks.

1. An arrow of the form $X \to F\ X$ defines, in the coalgebraic point of view, a *system*, not the transition function of a system, whereas in the examples of *nondet* and *stoch* above we were using the transition functions of discrete dynamical systems.

2. The classical definition contains, via the morphism of monoids, a notion of *dynamics*: $\phi\ t1\ x$ represents a state reached from x after $t1$ time, and $\phi\ t2\ (\phi\ t1\ x)$ represents a state reached from there after an additional $t2$ time, and this latter state coincides with the state reached after $t1 + t2$ from x. It is not immediately clear how such a notion of dynamics can be translated in the coalgebraic context.

3. Similarly unclear is the notion of a trajectory in the coalgebraic context: since the time is not explicit in this definition, we do not know what the source of the *trj* function should be. Moreover, the target of *trj* is also not a-priori determined: should it be X, or $F\ X$?

These questions will be addressed in the following section, in which we present the coalgebraic view of systems.

5.2 Coalgebras as general dynamical systems

The presentation we give here of the vast subject of coalgebras is of necessity brief, and guided by our main interest, which is the generic computation of trajectories of complex systems. Thus, important topics, such as the connection with modal logic, automata theory, program specification and verification, and so on, are touched upon only in passing or not at all. For a more complete introduction to this field we refer the reader to one of the many tutorial articles available (for example[21], [34], or [23]). A textbook presentation is being prepared by Bart Jacobs, the current version of which is available online ([20]).

5.2.1 Introduction

Central to the interpretation of coalgebras as dynamical systems is the idea that states are not always accessible to direct inspection. In the simplest case, the information that we can obtain about the states is summarized in a function $obs :: X \to O$, where O is a type of observable values, whereas the function gives the evolution of states has the form $f :: X \to F\ X$. The system proper is then the pair of these two functions:

$$\begin{aligned}&f' :: X \to (O, F\ X)\\&f' = pair\ (obs, id)\end{aligned}$$

Starting now with an element $x :: X$, we can apply the system to obtain a value $(o, xf) :: (O, F\ X)$ (the notation xf is meant to remind that the type of x is $F\ X$ and is similar to the convention of using xs for lists of "x"s). The value o is the only one which we can actually use: we have no direct access to xf. Applying $fmap\ f'$ to xf, we obtain a value of type $F\ (O, F\ X)$, from which we can extract, by $fmap\ fst$ and $fmap\ snd$ respectively, an **of** $:: F\ O$ and an $xff :: F\ (F\ X)$, the first being the only one which we can actually observe. Iterating, we obtain a sequence of observations $off :: F\ (F\ O)$, $offf :: F\ (F\ (F\ O)), \ldots.$

Let us consider as a simple example the following coalgebra of the list functor $[\,]$:

$$\begin{aligned}&f \quad :: \mathbb{N} \to [\mathbb{N}]\\&f\ n = [n-1, n+1]\end{aligned}$$

with the observation function

$$\begin{aligned}&obs :: \mathbb{N} \to Bool\\&obs = even\end{aligned}$$

so that

$$\begin{aligned}&f' :: \mathbb{N} \to (Bool, [\mathbb{N}])\\&f' = pair\ (obs, f)\end{aligned}$$

Starting from an arbitrary natural number, say 3, and following the process described above, we obtain a succession of values as follows:

$$\begin{aligned}&3\\&([2,4], False)\\&([[1,3],[3,5]], [True, True])\\&([[[0,2],[2,4]],[[2,4],[4,6]]], [[False, False],[False, False]])\end{aligned}$$

The observations are always, as it were, a step behind. We could say that we have a picture that emphasizes state. We obtain a much more accurate view of the coalgebraic point of view if we "blot out" the anyway inaccessible states:

$$\begin{array}{l} \circledast \\ (\circledast, \mathit{False}) \\ (\circledast, [\mathit{True}, \mathit{True}]) \\ (\circledast, [[\mathit{False}, \mathit{False}], [\mathit{False}, \mathit{False}]]) \end{array}$$

The first observable value, *False*, corresponds to the initial state, 3. After that, every state at a given level gives rise to a list of observable values at a lower level. Removing the $\circledast$s, we obtain an infinite tree-like data structure of more and more nested lists of boolean values. This infinite tree tells all that can be known about the observable evolution of the system from the initial state: let us call this tree *the behavior* of the system from that state.

The type of the behavior of the system does not depend on the the state space X. In our example, the same data structure of observations would arise if we were instead, considering the system obtained by pairing the function

$$\begin{array}{l} g :: \quad \mathbb{R} \to [\mathbb{R}] \\ g\ x = [x\ /\ 2, x * 2] \end{array}$$

with the observation function

$$\begin{array}{l} \mathit{obs}' :: \quad \mathbb{R} \to \mathit{Bool} \\ \mathit{obs}'\ x = x > 100 \end{array}$$

$$\begin{array}{l} g' :: \ \mathbb{R} \to (\mathit{Bool}, [\mathbb{R}]) \\ g' = \mathit{pair}\ (\mathit{obs}', g) \end{array}$$

We can now compare elements $x{::}\mathbb{R}$ with elements $n{::}\mathbb{N}$ according to whether the systems g' and f', exhibit the same behavior when started in x and n respectively. Thus, we can define an equivalence relation on $\mathbb{R} + \mathbb{N}$. In this case, it is obvious that no element of type $\mathbb{R}$ will be equivalent to an element of type $\mathbb{N}$. However, consider the next system:

$$\begin{array}{l} h :: \quad \mathit{Bool} \to [\mathit{Bool}] \\ h\ b = [\neg\ b, \neg\ b] \end{array}$$

$$\begin{array}{l} \mathit{obs}'' :: \ \mathit{Bool} \to \mathit{Bool} \\ \mathit{obs}'' = \mathit{id} \end{array}$$

$$\begin{array}{l} h' :: \ \mathit{Bool} \to (\mathit{Bool}, [\mathit{Bool}]) \\ h' = \mathit{pair}\ (\mathit{obs}'', h) \end{array}$$

Here, we have that the behavior of the initial system started from n coincides with that of h started from *even* n. Thus, every natural number n is equivalent to the boolean *even* n. Moreover, since the set of behaviors of both systems coincide, we can call the two systems equivalent.

In order to make these ideas more precise, let us take a closer look at the datatype of possible behaviors. This is, in fact, a familiar data structure: a rose tree (see, for example, section 6.4 in [1]). The data definition for this structure is:

$$\mathbf{data}\ \mathit{RoseT}\ o = \mathit{Node}\ (o, [\mathit{RoseT}\ o])$$

In our case, the type of the possible behaviors is *RoseT Bool*.

As explained in section 2.4, we have that *RoseT o* is (isomorphic to) the fixed point of the functor *FRoseT o* where

$$\begin{array}{l}\mathbf{newtype}\ \mathit{FRoseT}\ o\ x = R\ (o, [x])\\ \mathbf{instance}\ \mathit{Functor}\ (\mathit{FRoseT}\ o)\ \mathbf{where}\\ \quad \mathit{fmap}\ f\ (R\ (o, \mathit{xs})) = R\ (o, \mathit{map}\ f\ \mathit{xs})\end{array}$$

The initial algebra of *FRoseT o* is (isomorphic to) *Node*.

The behavior starting from a given initial state can be computed by

$$\begin{array}{l}\mathit{beh} :: (x \to (o, [x])) \to x \to \mathit{RoseT}\ o\\ \mathit{beh}\ f'\ x = \mathit{Node}\ (\mathit{obs}\ x, \mathit{map}\ (\mathit{beh}\ f')\ (f\ x))\\ \quad \mathbf{where}\\ \quad \mathit{obs} = \mathit{fst} \cdot f'\\ \quad f = \mathit{snd} \cdot f'\end{array}$$

The reason the definition of *beh* "works" is, as explained in 2.4, that *RoseT o* is the carrier of the final coalgebra of the functor *FRoseT o*, and *beh* is, in fact, *unfold* for this type:

$$\begin{array}{l}\mathit{unfoldR} :: (a \to (o, [a])) \to a \to \mathit{RoseT}\ o\\ \mathit{unfoldR} = \mathit{beh}\end{array}$$

The remarks about equivalent states made above can now be expressed formally:

For all $n :: \mathbb{N}$ and all $x :: \mathbb{R}$, we have that

$$\begin{array}{l}\mathit{beh}\ f'\ n \not\equiv \mathit{beh}\ g'\ x\\ (\mathit{beh}\ f'\ n = \mathit{beh}\ h'\ b) \equiv (\mathit{even}\ n = b)\\ \{\mathit{beh}\ f'\ n \mid n :: \mathbb{N}\} = \{\mathit{beh}\ h'\ b \mid b :: \mathit{Bool}\}\end{array}$$

In fact, there is one more system we have implicitely considered in this section, namely the one given by the inverse function of the isomorphism *Node*, that is, by the final coalgebra of the functor *FRoseT a*:

$$\begin{aligned}&\mathit{roseT} :: && \mathit{RoseT}\ o \rightarrow (o, [\mathit{RoseT}\ o])\\ &\mathit{roseT}\ (\mathit{Node}\ (o, \mathit{xts})) = (o, \mathit{xts})\end{aligned}$$

The set of states of this system is the set of all possible behaviors.

We can now compute the behavior of this system, by using the equivalent generic definitions:

$$\begin{aligned}& \mathit{beh}\ \mathit{roseT}\\ =\ & \{\text{ beh is unfold, roseT is out }\}\\ & \mathit{unfold}\ \mathit{out}\\ =\ & \{\text{ definition of unfold }\}\\ & \mathit{In} \cdot \mathit{fmap}\ (\mathit{unfold}\ \mathit{out}) \cdot \mathit{out}\end{aligned}$$

Therefore

$$\begin{aligned}& \mathit{unfold}\ \mathit{out} = \mathit{In} \cdot \mathit{fmap}\ (\mathit{unfold}\ \mathit{out}) \cdot \mathit{out}\\ \equiv\ & \{\text{ out is the inverse of In }\}\\ & \mathit{out} \cdot \mathit{unfold}\ \mathit{out} = \mathit{fmap}\ (\mathit{unfold}\ \mathit{out}) \cdot \mathit{out}\end{aligned}$$

But

$$\begin{aligned}& \mathit{out} \cdot \mathit{id} = \mathit{fmap}\ \mathit{id} \cdot \mathit{out}\\ \Leftarrow\ & \{\text{ fmap id = id }\}\\ & \mathit{out} = \mathit{out}\end{aligned}$$

which shows that, because of the unicity of *unfold out* we have

$$\mathit{unfold}\ \mathit{out} = \mathit{id}$$

In other words, the system started from a given behavior exhibts exactly this behavior.

5.2.2 Breadth-first traversal of observation trees

If we were to try printing the value of *beh* f' 3 using the automatic instance derivation of *Show*, we would obtain something like this:

```
Node (False,[Node (True,[Node (False,[Node (True,[Node ...
```

corresponding to a depth-first traversal of the tree. A better solution is to print the breadth-first levels of *RoseT o* values. This can be written as an unfold on lists, as described in [16].

$$\begin{aligned}&\mathit{levels} :: \mathit{RoseT}\ o \rightarrow [[o]]\\ &\mathit{levels} = \mathit{unfoldl}\ \mathit{null}\ (\mathit{map}\ \mathit{getObs})\ (\mathit{concat} \cdot \mathit{map}\ \mathit{getNext}) \cdot \mathit{wrap}\\ &\qquad \mathbf{where}\\ &\qquad \mathit{getObs}\ (\mathit{Node}\ (o, \mathit{rts})) \ = o\\ &\qquad \mathit{getNext}\ (\mathit{Node}\ (o, \mathit{rts})) = \mathit{rts}\\ &\qquad \mathit{wrap}\ \mathit{rt} \qquad\qquad\quad = [\mathit{rt}]\end{aligned}$$

The function *unfoldl* was defined in Section 2.4.

We can now print the observation values in a more informative manner:

```
levels (beh f' 3)
    ==> [[False],[True,True],[False,False,False,False],
         [True,True,True,True,True,True,True,True], ...
```

If we assume that we can observe the states directly, that is, we can use *id* as observation function, we can use *levels* to print the successive states that the system passes through:

```
levels (beh (pair (id, f)) 3)
    ==> [[3],[2,4],[1,3,3,5],[0,2,2,4,2,4,4,6], ...
```

Remember that at the end of Section 5.1 we have remarked that a discrete non-deterministic system should be expressible via a function of type $X \to P\ X$. We can consider the coalgebras of f as representations of such functions (modulo the presence of duplicates): what is the non-deterministic system they represent? The analogue of the $\cup$ operator on sets is *concat*, so we have that the transition function of such a system, call it ϕ, is given by

$$\phi\ 1\ xs = concat\ [f\ x \mid x \leftarrow xs]$$

or, equivalently

$$\phi\ 1 = concat \cdot map\ f$$

If we compute now the trajectory of this discrete system starting from the singleton set represented by [3], we obtain the infinite list

```
[phi n [3] | n <- [0 ..]]
    ==> [[3],[2,4],[1,3,3,5],[0,2,2,4,2,4,4,6], ...
```

This is the importance of this aside: the trajectory of the classical non-deterministic system coincides with the infinite list produced by the breadth-first traversal of the behavior tree. If we could generalize breadth-first traversal to the case of other types of coalgebras, not just those of [ ], we could perhaps obtain a "good" notion of trajectory for the case of coalgebraic systems.

5.2.3 Dynamical systems, behavioral equivalence, bismilarity

In view of our example, one might expect that the general definition of a system is given in terms of an explicit observation function paired with a coalgebra, that is, a system would be a function of type $X \to (O, F\ X)$. But, in fact, such a function is again a coalgebra: namely of the functor G where $G\ X = (O, F\ X)$. Thus, the following definition:

Definition 5 (Coalgebraic definition of dynamical systems) *A* dynamical system *is a coalgebra of a functor, that is an arrow of the form* $X \rightarrow F\ X$. *The set* X *is called the* state space *of the dynamical system. We call* F the functor *of the system.*

Coalgebras of a functor form a category, with arrows from a coalgebra $f :: X \rightarrow F\ X$ to $g :: Y \rightarrow F\ Y$ being given by arrows $arr :: X \rightarrow Y$ such that $g \cdot arr = \mathit{fmap}\ arr \cdot f$.

Definition 6 (Morphism of dynamical systems) *Let* f *and* g *be two dynamical systems with the same functor* F. *A* morphism *from* f *to* g *is an arrow from* f *to* g *in the category of coalgebras of* F.

In the following, equalities between types or type constructors are to be understood "up to isomorphism".

Example 1 *The functions* f', g' *and* h' *defined above are all coalgebras of the functor* $F\ X = (Bool, [X])$, *having as state spaces respectively* $\mathbb{N}$, $\mathbb{R}$ *and* $Bool$.

Example 2 (Explicit observations) *A coalgebra of type* $f :: X \rightarrow (O, F\ X)$ *is a dynamical system with functor* $\mathit{FTree}\ F\ O$ *given by* $\mathit{FTree}\ F\ O\ X = (O, F\ X)$.

In Haskell, we have to provide an explicit instance declaration for $\mathit{FTree}\ F\ O$:

```
data FTree f o x = FT (o, f x)

instance Functor f => Functor (FTree f o) where
  fmap f (FT (o, xf)) = FT (o, fmap f xf)
```

Example 3 *An endo-function is a dynamical system with functor* Id.

Example 4 *The final coalgebra of a functor* F *is a dynamical system with the functor* F *and the state space* $\mathit{FixP}\ F$.

The next definition generalizes the notion of behavior, computed above by the function *beh*.

Definition 7 *Let* $f :: X \rightarrow F\ X$ *be a dynamical system. The datatype of* behaviors *of* f *is the fixed point of the functor of* f, $\mathit{FixP}\ F$. *If* x *is an element of type* X, *the* behavior *of* f *starting from* x *is*

$$\mathit{behavior}\ f\ x = \mathit{unfold}\ f\ x$$

Example 5 *The datatype of behaviors of the dynamical systems* f', g' *and* h' *defined above is* $\mathit{RoseT}\ Bool$.

Example 6 *The datatype of behaviors of a dynamical system with functor $FTree\ F\ O$ is $Tree\ O = FixP\ (FTree\ F\ O)$. Thus, in the previous example, we have $RoseT\ Bool = Tree\ Bool = FixP\ (FTree\ [\,]\ Bool)$. In Haskell, it is customary to write the datatype declaration of Tree explicitely, without using FixP:*

$$\textbf{data}\ Tree\ f\ o = T\ (o, f\ (Tree\ f\ o))$$

The unfold function is given by

$$unfoldT :: Functor\ f \Rightarrow (x \to (o, f\ x)) \to x \to Tree\ f\ o$$
$$unfoldT\ sys = T \cdot cross\ (id, fmap\ (unfoldT\ sys)) \cdot sys$$

Example 7 *The datatype of behaviors of a dynamical system with functor Id is isomorphic to a singleton set: $FixP\ Id = ()$. A dynamical system with functor $[\,]$ has the same datatype of behaviors: $FixP\ [\,] = ()$.*

Definition 8 (Final system) *A* final system *is a final coalgebra of its functor. Since all final coalgebras are isomorphic, we can talk about* the *final system. In particular, $out :: FixP\ F \to F\ (FixP\ F)$ is the final system of F.*

We have therefore that the datatype of behaviors of a system with functor F is the state space of the final system of F, and the function *behavior* is the unique morphism from the system to the final system of its functor.

Proposition 2 *The behavior of a final system is the identity.*

Proof.
We could just repeat the calculation done above in Section 5.2.1 for the behavior of the system $roseT$. Instead, we can just note that the behavior of a final system is a morphism from itself to itself, that the identity on its state space is also such an arrow, and that by definition there is only one such arrow.
□
We now generalize the equivalence relations discussed in our example.

Definition 9 (Behavioral equivalence of states) *Let $f :: X \to F\ X$ and $g :: Y \to F\ Y$ be two dynamical systems. The states $x :: X$ and $y :: Y$ will be called* behaviorally equivalent *if*

$$unfold\ f\ x = unfold\ g\ y$$

Remark. Obviously, the behavioral equivalence relation defined above is *not* an equivalence relation, since it is not an endo-relation. However, it can be extended to an equivalence relation $\sim$ on $X + Y$ in the natural way (remember that $X + Y$ is *Either* X Y in the Haskell notation):

$$\begin{aligned}
(\mathit{Left}\ x) \sim (\mathit{Left}\ x') &= \mathit{unfold}\ f\ x = \mathit{unfold}\ f\ x' \\
(\mathit{Right}\ y) \sim (\mathit{Right}\ y') &= \mathit{unfold}\ g\ y = \mathit{unfold}\ g\ y' \\
(\mathit{Left}\ x) \sim (\mathit{Right}\ y) &= \mathit{unfold}\ f\ x = \mathit{unfold}\ g\ y \\
(\mathit{Right}\ y) \sim (\mathit{Left}\ x) &= \mathit{unfold}\ f\ x = \mathit{unfold}\ g\ y
\end{aligned}$$

Definition 10 (Behavioral equivalence of systems) *Let $f :: X \to F\ X$ and $g :: Y \to F\ Y$ be two dynamical systems. They will be called* behaviorally equivalent *if*

$$\{\mathit{unfold}\ f\ x \mid x \in X\} = \{\mathit{unfold}\ g\ y \mid y \in Y\}$$

Example 8 *Every state of every endofunction exhibits the same behavior: endofunctions are dynamical systems with functor Id, therefore the final system is $() \to ()$ and the behavior of any system is const $()$, independent of the state from which it starts and from the system itself. The same holds for dynamical systems with functor $[\,]$ instead of Id. That is why it is extremely misleading to talk, when in a coalgebraic context, of endofunctions as "deterministic systems" or of functions of type $X \to [X]$ as "non-deterministic systems", etc.*

We note that although the fixed points of Id and $[\,]$ are isomorphic, we cannot compare systems with functor Id with systems with functor $[\,]$.

The study of such equivalences has been traditionally conducted using the concept of *bisimulation*. In order to define this, we first need the notion of a *relator*.

We remind that *Rel* denotes the category of relations, having as objects sets and as arrows $R : X \to Y$ subsets of the cartesian products $X \times Y$. The set of relations of type $X \to Y$ is a partially ordered set, with the order relation being given by set inclusion: $R \subseteq S$.

Definition 11 (Relator) *A* relator *is a monotonous endofunctor $F : \mathit{Rel} \to \mathit{Rel}$, that is, a functor such that:*

$$R \subseteq S \Rightarrow F\ R \subseteq F\ S$$

Since every function is also a relation, *Set* is a subcategory of *Rel*, and it is natural to consider extensions of endofunctors on *Set* to endofunctors on *Rel*.

Definition 12 (Extending a functor) *An endofunctor $F : \mathit{Set} \to \mathit{Set}$ is extended to an endofunctor $G : \mathit{Rel} \to \mathit{Rel}$ if $\mathit{Inc}\ (F\ f) = G\ (\mathit{Inc}\ f)$ for all functions f (where $\mathit{Inc} : \mathit{Set} \to \mathit{Rel}$ is the standard inclusion functor).*

An essential property of relators is summarized in the following

Theorem 3 *If two relators F and G agree on functions, then they are equal.*

For the proof, see for example [2].

The importance of this result is that it shows that an endofunctor on *Set* can be extended to *at most one* relator. There are functors that cannot be so extended, but a large class of functor can, including all polynomial functors.

With these preliminaries, we can define bismulation as follows:

Definition 13 (Bisimulation) *Let $f :: X \to F\ X$ and $g :: Y \to F\ Y$ be two dynamical systems and F' the extension to a relator of F. A* bisimulation *is a relation $R \subseteq X\ \times\ Y$ such that*

$$x\ R\ y \equiv (f\ x)\ (F'\ R)\ (g\ y)$$

Definition 14 (Bisimilarity of states) *Let $f :: X \to F\ X$ and $g :: Y \to F\ Y$ be two systems. The states $x :: X$ and $y :: Y$ will be called* bisimilar *if there exists a bisimulation $R \subseteq X\ \times\ Y$ such that $x\ R\ y$.*

If F can be extended to a relator *and* has a final coalgebra, then the bisimilarity coincides with behavioral equivalence (see [23]).

Theorem 4 *Let $f :: X \to F\ X$ and $g :: Y \to F\ Y$ be two coalgebras of a functor F which has a relator extension F' and a final coalgebra. Then, for any two states $x :: X$ and $y :: Y$ a bisimulation $R \subseteq X\ \times\ Y$ exists such that $x\ R\ y$ if and only if $unfold\ f\ x = unfold\ g\ y$.*

For a proof, see [34].

In particular, this is the case for all polynomial functors. Using bisimilarity over behavioral equivalence or the other way around is in such circumstances a matter of which leads to simpler proofs of equivalence. See, for example, [15].

Example 9 *We examine the notion of bisimulation for the case of the functor $FTree\ F\ O$. Assume that F can be extended to a relator F'. Then $FTree\ F\ O$ extends to the relator $Id\ \times\ F'$, that is, for any relation $R \subseteq X\ \times\ Y$ we have*

$$(ox, xf)\ ((Id\ \times\ F')\ R)\ (oy, yf) \equiv ox = oy \wedge xf\ (F'\ R)\ yf$$

Proof.

$\times$ is monotonous, therefore $Id\ \times\ F'$ is monotonous since both Id and F' are. Moreover, if f is a function we have

$(Id \times F')\, f$
$= \quad \{ \text{ Definition of } \times \}$
$(Id\, f) \times (F'\, f)$
$= \quad \{ \text{ Definition of } Id,\ F' \text{ extends } F \}$
$f \times (F\, f)$
$= \quad \{ \text{ definition of } FTree\ F\ O \}$
$(FTree\ F\ O)\, f$

and therefore $Id \times F'$ extends $FTree\ F\ O$ monotonously, and there can be only one such extension.

□

Therefore, given $f :: X \to FTree\ F\ O\ X$ and $g :: Y \to FTree\ F\ O\ Y$, R is a bisimulation over f and g iff

$x\ R\ y \equiv ox = oy \wedge xf\ (F'\ R)\ xy$
where
$(ox, xf) = f\ x$
$(oy, yf) = g\ y$

We see that bisimilarity of states translates to equality of observations over the behaviors started from those states.

5.3 Abstract datatypes

From the examples we have seen, it appears that the concepts of coalgebraic dynamical systems are clearest in the case in which the observation function is made explicit. The question is then whether there are indeed cases which would require more than functors of the form $FTree\ F\ O$. The answer is yes, and one example is the study of abstract datatypes. A detailed exposition of this subject can be found in [14]. Here we shall just summarize the main points of the approach.

An abstract datatype is described as a collection of operations acting on an internal state which is an element of some *inaccessible* type. The operations are usually of the form $op :: S \to I \to Either\ (O, S)\ E$ with the interpretation "the operation op acts on the hidden state of type S as a function of some input of type I and produces either an output of type O and a change in the state, or an error of type E. These operations all act on S, and so can be collected in one function with a sum-type value, having the general form $S \to F\ S$.

In Haskell, the "inaccessibility" of the state is ensured by using an existential type. Thus, the general definition of an abstract datatype is

data $Functor\ f \Rightarrow ADT\ f = \exists\ s \cdot D\ (s \to f\ s)\ s$

The Haskell keyword translated here by the symbol $\exists$ is, surprisingly, *forall*. The **data** declaration states that given an element $a :: ADT\ f$ there exists some type S such that a can be written uniquely as $D\ ops\ s$ where $ops :: S \to F\ S$ and $s :: S$. Since the type $ADT\ f$ is *not* parametrized on the type of the internal state s, there is no way that the programmer can operate on values of this type: the type checker would report an error.

Besides the coalgebra *ops*, the values of type $ADT\ f$ also encapsulate an internal state: this is the only possibility to ensure that *ops* is ever going to be applied. This also implies that the implementor of values of type $ADT\ f$ must supply at least one element of the type, which is going to act as a "factory" for other elements constructed on the basis of the operations provided (the internal state of this initial element is usually some form of the $\perp$ or the "zero" element of type S).

As an example, consider the datatype *Queue a* of queues with elements of some type a (see, for example, [1]). From the point of view of the user, the operations on *Queue a* are:

```
empty   :: Queue a                -- constructs an empty queue
put     :: Queue a → a → Queue a  -- adds an element to the rear
front   :: Queue a → a            -- removes an element from front
isEmpty :: Queue a → Bool         -- checks whether queue empty
```

These operations are related by a number of equations, for example

```
isEmpty (put x q) = False
isEmpty empty = True
```

and so on. The signatures of the operations, together with the equations these operations must fullfill, constitute the *signature* of the abstract datatype.

The implementor of *Queue a* will have to choose a representation of some internal state, say S. For him, *front* must act on this internal state and change it so that it corresponds to the initial queue with the front element removed, and so on. That is, the operations have the signatures:

```
put     :: s → a → s
front   :: s → (a, s)
isEmpty :: s → Bool
```

and *empty* is just going to correspond to the initial element the implementor provides, the empty queue.

Putting together the internal operations, we obtain the signature of the coalgebra characterizing the abstract datatype:

```
ops :: s → QueueF a s
```

where

$$\textbf{data}\ \mathit{QueueF}\ a\ x = Q\ (a \to x)\ (a, x)\ \mathit{Bool}$$

with accessor functions corresponding to

$$\begin{aligned}
&\mathit{putF}\ (Q\ p\ f\ b) &&= p\\
&\mathit{frontF}\ (Q\ p\ f\ b) &&= f\\
&\mathit{isEmptyF}\ (Q\ p\ f\ b) &&= b
\end{aligned}$$

$\mathit{QueueF}\ a$ is a functor:

```
instance Functor (QueueF a) where
  fmap f (Q p (a, x) b) = Q (f · p) (a, f x) b
```

and we can define $\mathit{Queue}\ a$ as

$$\textbf{type}\ \mathit{Queue}\ a = \mathit{ADT}\ (\mathit{QueueF}\ a)$$

The user-level functions are defined independently of a choice of representation for the internal state:

```
put              :: Queue a → a → Queue a
put (D ops s) a = D ops (putF (ops s) a)


front           :: Queue a → (a, Queue a)
front (D ops s) = (a, D ops s′)
                  where
                  (a, s′) = frontF (ops s)


isEmpty           :: Queue a → Bool
isEmpty (D ops s) = isEmptyF (ops s)
```

We can now implement a queue by choosing a particular representation for the internal state: the simplest is to choose a list.

```
empty :: Queue a
empty = D ops [ ]
        where
        ops s = Q put′ front′ isEmpty′
          where
          put′ x = x : s
          front′ = (last s, init s)
          isEmpty′ = null s
```

Alternatively, we can implement the internal state by a pair of lists, as in Section 8.6 of [1]:

```
empty1 :: Queue a
empty1 = D ops ([ ], [ ])
         where
         ops (xs, ys) = Q put' front' isEmpty'
           where
           put' x = if null xs then
                       (reverse (x : ys), [ ])
                       else (xs, x : ys)
           front'  = (head xs, (tail xs, ys))
           isEmpty' = null xs
```

The two implementations are intuitively equivalent, but how can this be stated formally and proved? Here the connection with dynamical systems comes in: the two implementations are equivalent if and only if the internal states [] and ([], []) are behaviorally equivalent. That is, if the observable behavior of the two datatypes are indistinguishable. The behaviors in this case are much more complicated than in the simple example of the non-deterministic system given above, due to the greater complexity of the operations, but they are still quite tame in comparison to the behaviors of the abstract datatypes which the average programmer is required to implement every day. However, the generic definition of equivalence:

```
equiv :: (Functor f) ADT f → ADT f → Bool
equiv (D ops1 s1) (D ops2 s2) = unfold ops 1 s1 ≡ unfold ops2 s2
```

is the appropriate conceptual tool for reasoning about implementation equivalence, and thus it would be inappropriate to restrict ourselves to functors of a simple type.

5.3.1 Abstract datatypes for dynamical systems

We have emphasized in the previous the notion of “internal” or “inaccessible” state, and we have seen that an (instance of an) abstract datatype can be represented as a colagebra packaged together with an internal state. It would therefore seem that coalgebraic dynamical systems with explicit observation functions, as introduced in the Example 2, are ideal candidates for being represented as abstract datatypes.

In the *Queue a* datatype, transitions such as *put* or *front* were modeled as functions which were changing the internal state, possibly using some input or providing some output. This change in state was deterministic, so for example, the function *put'* could have the signature $s \to a \to s$.

In the simple example we have used in Section 5.2.1, we had however a non-deterministic system, that is one in which the internal state is not changed to another state, but to a set or list of such states. This raises the question of how to model such a system, starting in an initial state, as an abstract datatype, because instances of this type will always have exactly one internal state.

The solution is to create as many instances of the type as there are new states after the transition. For a stochastic system, we would have to create a simple probability distribution over the possible next instances of the system, and so on. In general, if we have a dynamical system with functor $\mathit{FTree}\ F\ O$, we have to create an F-structure of possible next instances.

From the user point of view, the interface of the abstract datatype $\mathit{DynSys}\ F\ O$ of dynamical systems with functor F and observation type O is

$$\begin{aligned}&\mathit{observe} :: \mathit{DynSys}\ F\ O \to O\\&\mathit{transition} :: \mathit{DynSys}\ F\ O \to F\ (\mathit{DynSys}\ F\ O)\end{aligned}$$

At the implementational level, these functions are going to be translated in terms of the internal states X as

$$\begin{aligned}&\mathit{observe} :: X \to O\\&\mathit{transition} :: X \to F\ X\end{aligned}$$

The signature of the coalgebra that characterizes this abstract datatype is

$$\mathit{ops} :: X \to \mathit{DynSysF}\ F\ O\ X$$

where

$$\mathit{DynSysF}\ F\ O\ X = (O, F\ X) = \mathit{FTree}\ F\ O\ X$$

The functor that characterizes the signature of DynSys is $\mathit{FTree}\ F\ O$, hence the “ideal fit” between dynamical systems with explicit observations and abstract datatypes. In Haskell:

$$\textbf{type}\ \mathit{DynSysF} = \mathit{FTree}$$

and we introduct the accessor functions as usual:

$$\begin{aligned}&\mathit{observeF}\ (\mathit{FT}\ (o, \mathit{xf})) = o\\&\mathit{transitionF}\ (\mathit{FT}\ (o, \mathit{xf})) = \mathit{xf}\end{aligned}$$

Finally, we have

$$\textbf{type}\ \mathit{DynSys}\ f\ o = \mathit{ADT}\ (\mathit{FTree}\ f\ o)$$

The user level functions can then be defined as

$$\begin{aligned}&\mathit{observe}\ (D\ \mathit{ops}\ x) = \mathit{observeF}\ (\mathit{ops}\ x)\\&\mathit{transition}\ (D\ \mathit{ops}\ x) = \mathit{fmap}\ (\lambda x \to (D\ \mathit{ops}\ x))\ (\mathit{transitionF}\ (\mathit{ops}\ x))\end{aligned}$$

Let us now implement the system f' defined in Section 5.2.1. The functor of this system is $\mathit{FTree}\ [\,]\ \mathit{Bool}$: we can call f' a non-deterministic system (we are not exposed to the danger of calling a function of type $X \to [X]$ a non-deterministic system, which we have seen to be misleading in the coalgebraice context). Non-deterministic systems are special cases of *DynSys*:

$$\textbf{type}\ \mathit{NonDet}\ o = \mathit{DynSys}\ [\,]\ o$$

For every integer we can construct an instance of *NonDet Bool* which behaves like the example system f' defined above when started from that number:

$$\begin{array}{ll} \mathit{newnd} & :: \mathbb{N} \to \mathit{NonDet}\ \mathit{Bool} \\ \mathit{newnd}\ n & = D\ \mathit{ops}\ n \\ & \quad \textbf{where} \\ & \quad \mathit{ops} = \mathit{FT} \cdot f' \end{array}$$

We have

```
observe (newnd 3)
    ==> False

map observe (transition (newnd 3))
    ==> [True, True]

map (map observe) (map transition (transition (newnd 3)))
    ==> [[False, False], [False, False]]
```

An application of *transition* to an instance of *NonDet o* produces a list of new instances, each one with its own state, each one in turn being a valid starting point for a new transition, producing a list of states, and so on. Every instance of *NonDet o* results either from being constructed by the function *newnd*, or by repeated application of the *transition* function to an instance constructed by *newnd*. We can keep track of the states successively generated from the initial one to the state of a given instance, as in the following. We can access only the sequences of *observations*, of course, so let's assume that we have $o = \mathbb{N}$ and $obs = id$.

$$\begin{array}{ll} \mathit{ndHist} & :: \mathbb{N} \to \mathit{NonDet}\ [\mathbb{N}] \\ \mathit{ndHist}\ n & = D\ \mathit{ops}\ [n] \\ & \quad \textbf{where} \\ & \quad \mathit{ops}\ (x : xs) = \mathit{FT}\ (x : xs, \\ & \qquad [(x-1) : x : xs, (x+1) : x : xs]) \end{array}$$

We have

```
observe (ndHist 3)
    ==> [3]
```

```
map observe (transition (ndHist 3))
    ==> [[2,3],[4,3]]

map (map observe) (map transition (transition (ndHist 3)))
    ==> [[[1,2,3],[3,2,3]],[[3,4,3],[5,4,3]]]
```

These histories that instances of *NonDet* [$\mathbb{N}$] carry around and extend with each transition are potential candidates for a coalgebraic definition of trajectory. This definition, assuming we can actually give it, is emphatically *not* the one suggested, for the case of the same system f', in Section 5.2.2. There, we had a connection to the classical notion of trajectory of a non-deterministic dynamical system, via the accessibility relation. Here, the trajectories we obtain appear to be similar to those of a classical deterministic system: it's just that we obtain them embedded in a list, or a set. In the first case, we had one non-deterministic system, while here we treat the same situation as though we had a collection of deterministic systems.

We can look at the difference between these two notions also by analogy with a stochastic system, say a markov chain. The states of a markov chain are the elements over which probability distributions evolve as a result of the transitions. We can see this as a stochastic system, moving from probability distribution to probability distribution, or as a large collection of individual deterministic systems over whose trajectories we can make only stochastic evaluations.

Following a usage common in economics, we can call the states of the individual deterministic systems that make up the collection, *micro-states*, and the states of the system representing the collection, *macro-states*. Accordingly, the two types of trajectories suggested by the above considerations will be called *micro-trajectories* and *macro-trajectories*, respectively. Both notions have merit, and are encountered in practice. Ideally, we want to make both explicit and general enough to apply to any coalgebra. We shall attempt to do so in the next section.

5.4 Trajectories of coalgebraic dynamical systems

5.4.1 Micro-trajectories

We start with the micro-trajectories, because the programming "trick" we used to keep the history of a non-deterministic system is immediately generalizable to an arbitrary coalgebra:

$$\begin{array}{ll} \mathit{addHist} :: & \mathit{Functor}\ f \Rightarrow (x \to f\ x) \to [x] \to f\ [x] \\ \mathit{addHist}\ g\ (x : xs) = & \mathit{fmap}\ ({:}(x : xs))\ (g\ x) \end{array}$$

so that we have

$$ndHist'\ n = D\ (addHist\ (FT \cdot f'))\ [n]$$

addHist creates systems with the same functor as those given as input, but which have a different state space, reflecting the change that must be made in order to keep track of the history.

There are certain requirements for considering the lists built by successive calls to *addHist f* as micro-trajectories of f. We expect that the lists all have the same length: no trajectories are swept under the carpet, as it were. An application of *addHist f* leads to lists which are longer than the initial ones by exactly one element. The elements of micro-trajectories should all come from f: we don't want *addHist* to invent new elements.

We can express these requirements formally, and prove them:

Theorem 5 (*addHist*) *With the definitions above*

1. $fmap\ length\ (addHist\ f\ (x : xs)) =$
   $fmap\ (const\ (1 + length\ (x : xs)))\ (f\ x)$

2. $fmap\ head \cdot addHist\ f = f \cdot head$

Proof.

1. We have

$$\begin{aligned}
& fmap\ length\ (addHist\ f\ (x : xs)) \\
= & \quad \{ \text{ Definition of } addHist \ \} \\
& fmap\ length\ (fmap\ (:(x : xs))\ (f\ x)) \\
= & \quad \{ \ fmap \text{ functor } \} \\
& fmap\ (length \cdot (:(x : xs)))\ (f\ x) \\
= & \quad \{ \text{ Definition of } length \ \} \\
& fmap\ (\lambda z \to 1 + length\ (x : xs))\ (f\ x) \\
= & \quad \{ \text{ Definition of } const \ \} \\
& fmap\ (const\ (1 + length\ (x : xs)))\ (f\ x)
\end{aligned}$$

For the second property we need the following property of *head*:

Lemma:

$$head \cdot (:xs) = id$$

Proof.

$$
\begin{array}{ll}
 & (\mathit{head} \cdot (:\mathit{xs}))\ x \\
= & \{\ \text{Composition}\ \} \\
 & \mathit{head}\ (x : \mathit{xs}) \\
= & \{\ \text{Definition of } \mathit{head}\ \} \\
 & x \\
= & \{\ \text{Definition of } \mathit{id}\ \} \\
 & \mathit{id}\ x
\end{array}
$$

2. We have:

$$
\begin{array}{ll}
 & (\mathit{fmap}\ \mathit{head} \cdot \mathit{addHist}\ f)\ (x : \mathit{xs}) \\
= & \{\ \text{Composition, definition of } \mathit{addHist}\ \} \\
 & \mathit{fmap}\ \mathit{head}\ (\mathit{fmap}\ (:(x : \mathit{xs}))\ (f\ x)) \\
= & \{\ \mathit{fmap} \text{ functor}\ \} \\
 & \mathit{fmap}\ (\mathit{head} \cdot (:(x : \mathit{xs})))\ (f\ x) \\
= & \{\ \text{Lemma}\ \} \\
 & \mathit{fmap}\ \mathit{id}\ (f\ x) \\
= & \{\ \mathit{fmap} \text{ functor}\ \} \\
 & f\ x \\
= & \{\ \text{Definition of } \mathit{head}\text{, composition}\ \} \\
 & (f \cdot \mathit{head})\ (x : \mathit{xs})
\end{array}
$$

□

However, the situation is not entirely satisfactory. Trajectories of length n are embedded in a structure of type $F\ (F\ (...(F\ [X])))$ with $n - 1$ applications of the functor F. That is, if we want to obtain the trajectories of length n, we have to write a function *trj* such that

$$\mathit{trj}\ f\ n\ x :: F^n\ [x]$$

Such a function requires a dependently typed system, and is not implementable in Haskell.

In the next section, we attempt to generalize the breadth-first traversal of *Tree F O* values, in order to determine under which conditions we can compute such macro-trajectories. This investigation will lead us to restrictions on F which will allow us to solve the typing problem for *trj*.

5.4.2 Generalizing breadth-first traversal

The idea of macro-trajectories was suggested by the observation that

$$\mathit{levels}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f)))\ x = [\phi\ n\ [x] \mid n \leftarrow [0\,..]]$$

for the case of the specific f we had considered. Indeed, for an arbitrary argument to *beh*, this property does not hold:

```
levels ((beh (pair (id, \n -> if even n then [n-1, n+1] else []))) 3)
    ==> [3]

levels ((beh (pair (id, \n -> if even n then [n-1, n+1] else []))) 4)
    ==> [[4],[3,5]]
```

while in both cases the list on the right-hand side will be infinite. The reason this did not happen in the example we had is that the predicate given as first argument to the *unfoldl* function in *levels*, namely *null*, was always returning *False*. We can make a similar clame to the above if we replace *levels* by

$$\begin{array}{l} \mathit{levels}' = \mathit{unfoldl}\ (\mathit{const\ False})\ (\mathit{map\ getObs})\ (\mathit{concat} \cdot \mathit{map\ getNext}) \cdot \mathit{wrap} \\ \quad \textbf{where} \\ \quad \begin{array}{ll} \mathit{getObs}\ (\mathit{Node}\ (o, \mathit{rts})) & = o \\ \mathit{getNext}\ (\mathit{Node}\ (o, \mathit{rts})) & = \mathit{rts} \\ \mathit{wrap}\ \mathit{rt} & = [\mathit{rt}] \end{array} \end{array}$$

The infinite list on the right-hand side of the equation is computed by

$$\begin{array}{l} [\phi\ n\ [x] \mid n \leftarrow [0\,..]] \\ = \\ \mathit{unfoldl}\ (\mathit{const\ False})\ \mathit{id}\ (\mathit{concat} \cdot \mathit{map}\ f)\ (\mathit{wrap}\ x) \end{array}$$

(remember that $\phi\ 1 = \mathit{concat} \cdot \mathit{map}\ f$).

We want to generalize *levels′* and the construction of ϕ in order to define macro-trajectories for other functors, but there are obvious dependencies in our definitions on [ ]-specific functions: *map*, *concat* and *wrap*. It is natural to think of restricting the functors F for which we can define macro-trajectories to those that come equipped with similar functions, but what does “similar” mean? What properties of *map*, *concat* and *wrap* are required for the above equation to hold? In order to answer this question, we have only one solution: we must prove the above equation (which is not really obvious anyway), and check which properties of the three [ ] specific functions are used.

Proposition 3 *For any* $f :: X \to [X]$

$$\mathit{levels}' \cdot \mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f)) = [\phi\ n\ [x] \mid n \leftarrow [0\,..]]$$

Proof.

Expanding the definitions involved, we have to prove that

$$\begin{array}{l} \mathit{unfoldl}\ (\mathit{const\ False})\ (\mathit{map\ getObs})\ (\mathit{concat} \cdot \mathit{map}\ (\mathit{getNext})) \cdot \\ \mathit{wrap} \cdot \mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f)) \\ = \\ \mathit{unfoldl}\ (\mathit{const\ False})\ \mathit{id}\ (\mathit{concat} \cdot \mathit{map}\ f) \cdot \mathit{wrap} \end{array}$$

It is tempting to try to fuse the computations on each side of the equation. Unfortunately, the fusion theorem cannot be applied for any side. For example, on the right-hand side, the fusion theorem would require finding a function g' such that

$$\mathit{concat} \cdot \mathit{map}\ f \cdot \mathit{wrap} = \mathit{wrap} \cdot g'$$

The right-hand side always creates a singleton, but the one on the left-hand side in general does not, therefore there is no such g'.

We need to guess an intermediate step, one to which we might apply fusion. For example, let us try to find an equality which holds when we throw away the *wrap* on the right-hand side of the original equality. A moment's thought suggests the following

Lemma

$$\begin{array}{l} \mathit{unfoldl}\ (\mathit{const\ False})\ (\mathit{map\ getObs})\ (\mathit{concat} \cdot \mathit{map}\ (\mathit{getNext})) \cdot \\ \mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \\ = \\ \mathit{unfoldl}\ (\mathit{const\ False})\ \mathit{id}\ (\mathit{concat} \cdot \mathit{map}\ f) \end{array}$$

End of Lemma.

Assume we have shown the Lemma. Then, composing each side with *wrap*, we get on the right-hand side the desired result, and on the left we have

$$\begin{array}{ll} & \mathit{unfoldl}\ (\mathit{const\ False})\ (\mathit{map\ getObs})\ (\mathit{concat} \cdot \mathit{map}\ (\mathit{getNext})) \cdot \\ & \mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \cdot \mathit{wrap} \\ = & \{\ \mathit{wrap} \text{ natural transformation } \mathit{Id} \rightarrow [\,]\ \} \\ & \mathit{unfoldl}\ (\mathit{const\ False})\ (\mathit{map\ getObs})\ (\mathit{concat} \cdot \mathit{map}\ (\mathit{getNext})) \cdot \\ & \mathit{wrap} \cdot \mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f)) \end{array}$$

which is equal to the desired left-hand side. Therefore, we can establish the result if we can prove the Lemma. We note that we have used the naturality of *wrap*.

To prove the Lemma, we only need to check the three conditions for the application of fusion. We use the notation in Section 2.4, so that $p = \mathit{const\ False}$, $f = \mathit{map\ getObs}$, $g = \mathit{concat} \cdot \mathit{map}\ (\mathit{getNext})$, $h = \mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f)))$, $p' = \mathit{const\ False}$, $f' = \mathit{id}$, $g' = \mathit{concat} \cdot \mathit{map}\ f$. We have to check

1. $p' = p \cdot h$

   This expands to $\mathit{const\ False} = \mathit{const\ False} \cdot h$ which is trivially true.

2. $f' = f \cdot h$

   Expanding $f \cdot h$ we have

$$\begin{aligned}
& (\mathit{map}\ \mathit{getObs}) \cdot \mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \\
= \quad & \{\ \mathit{map} \text{ functor }\} \\
& \mathit{map}\ (\mathit{getObs} \cdot \mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \\
= \quad & \{\ \text{Definitions of } \mathit{getObs} \text{ and } \mathit{beh}\ \} \\
& \mathit{map}\ \mathit{id} \\
= \quad & \{\ \mathit{map} \text{ functor }\} \\
& \mathit{id}
\end{aligned}$$

   which is f'. We have used the functoriality of *map*, which will translate to the functoriality of *fmap* for a general functor F.

3. $g \cdot h = h \cdot g'$

   Expanding the left-hand side, we have

$$\begin{aligned}
& (\mathit{concat} \cdot \mathit{map}\ \mathit{getNext}) \cdot \mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \\
= \quad & \{\ \mathit{map} \text{ functor }\} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{getNext} \cdot \mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \\
= \quad & \{\ \text{Definitions of } \mathit{getNext} \text{ and } \mathit{beh}\ \} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \cdot f) \\
= \quad & \{\ \mathit{map} \text{ functor }\} \\
& \mathit{concat} \cdot \mathit{map}\ (\mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f)))) \cdot \mathit{map}\ f \\
= \quad & \{\ \mathit{concat} \text{ natural transformation } [[\,]] \rightarrow [\,]\ \} \\
& \mathit{map}\ (\mathit{beh}\ (\mathit{pair}\ (\mathit{id}, f))) \cdot \mathit{concat} \cdot \mathit{map}\ f
\end{aligned}$$

   which is the right-hand side. Again we have used the functoriality of *map*, and, additionally, the naturality of *concat*.

All conditions check out: thus we have shown the Lemma, and consequently the Proposition.

□

We can now generalize this proposition to any functor F which is equipped with two natural transformations $\mathit{wrap} :: a \rightarrow F\ a$ and $\mathit{concat} :: F\ (Fa) \rightarrow F\ a$. The equivalent of Rose trees for such a functor is $\mathit{Tree}\ F\ X$, and the general breadth-first traversal is written as

```
bft = unfoldl (const False) (fmap getObs)
          (concat · fmap getNext) · wrap
  where
  getObs (T (o, xf)) = o
  getNext (T (o, xf)) = xf
```

We have

Theorem 6 (General breadth-first traversal) *For any colagebra $f :: X \rightarrow F\ X$, the general breadth-first traversal of behavior $(pair\ (id, f))$ starting from a state x is equal to the trajectory of the classical discrete dynamical system given by the transition function*

$$\phi\ 1 = concat \cdot fmap\ f$$

That is:

$$bft\ (unfoldT\ (pair\ (id, f))) = [\phi\ n\ (wrap\ x) \mid n \leftarrow [0\,..]]$$

By now, any Haskell programmer will have recognized that we propose to restrict our attention to the case in which F is a *monad*, replacing *wrap* by *return* and *concat* by *join*. This restriction is sufficient, although not necessary for the above result. For example, we could use, instead of the usual *wrap* for lists, the natural transformation $\lambda x \rightarrow [x, x]$, which no longer constructs a monad with the usual *concat* (it violates the law $concat \cdot wrap = id$), but with which we still have the above result.

However, the cases encountered in practice, such as [ ]. *SP* and so on, are all monads. Therefore, the over-specialization does not seem excessive.

As an example, let us consider a stochastic version of our simple non-deterministic system:

$$\begin{aligned} &st \quad :: Integer \rightarrow SimpleProb\ Integer \\ &st\ n = SP\ [(n-1, 0.4), (n+1, 0.6)] \end{aligned}$$

Using *bft*, we obtain:

```
bft (unfoldT (pair (id, st)) 3)
    ==> [SP [(3,1.0)],SP [(2,0.4),(4,0.6)],
         SP [(1,0.16),(3,0.48),(5,0.36)], ...
```

These are indeed the successive probability distributions over the states of the system *st*. The first one is the concentrated probability distribution corresponding to the given argument, and the rest result from the application of the conditional probability formula represented by *st* to the previous distribution. Such a computation is useful, for example, in many stochastic sequential decision problems where the control goal is to reach a certain macrostate after a number of states.

Perhaps surprisingly, since *addHist* does not change the functor of a coalgebraic system, we can use *bft* also to compute the micro-trajectories of the system:

```
bft (unfoldT (pair (id, addHist st)) [3])
    ==> [SP [([3],1.0)],SP [([2,3],0.4),([4,3],0.6)],
    SP [([1,2,3],0.16),([3,2,3],0.24),([3,4,3],0.24),([5,4,3],0.36),...
```

The macro-states of the *addHist st* coalgebra are simple probabilities over the micro-trajectories. We thus have a common way of dealing with both the micro-trajectories and the macro-trajectories of coalgebraic dynamic systems with monadic functors.

It is, however, still important to have a more general view of "system" which would allow us to understand how to deal with, for example, the case of continuous systems. Until now, we have been able to relate coalgebraic systems with discrete classical systems. In the following section, we take a closer look at this relation.

5.5 General dynamical systems

As explained in Section 2.2, in Haskell we use more commonly the "bind" operator, so that, in defining the transition function of the discrete system that corresponds to a given monadic coalgebra $f :: X \to M\ X$, we would write

$$\phi\ 1 = (f \lhd)$$

rather than $\phi\ 1 = \mathit{join} \cdot \mathit{fmap}\ f$.

Since we are always considering an *atomic* point of view, we can evaluate the macro-states resulting from successive applications of $\phi\ 1$ to some *return* x by applying the function

$$\mathit{iterate}\ (f \lhd) \cdot \mathit{return}$$

If we want to compute the nth macro-state, we can do that by

$$(\mathit{iterate}\ (f \lhd) \cdot \mathit{return}\ x)\ !!\ n$$

or, alternatively, by defining the function

$$\begin{aligned}&\mathit{app}\ f\ 0 = \mathit{return}\\ &\mathit{app}\ f\ (n+1) = (f \lhd) \cdot \mathit{app}\ f\ n\end{aligned}$$

which "applies" f n times. This way of computing the macro-states is perhaps the most familiar and the simplest, involving only repeated applications of $f \lhd$ without the construction of a *Tree M X* structure or of using an unfolding process. Moreover, the result of $\mathit{app}\ (\mathit{addHist}\ f)\ n\ [x]$ is the M-structure of micro-trajectories obtained in n steps starting from x, constructed without the need to refer to an intermediate *Tree M X* structure.

Using the Kleisli composition $\diamond$ and *unit* defined in Section 2.2, we have that

$$\mathit{app}\ f\ (n+1) = f \diamond (\mathit{app}\ f\ n)$$

The type of *app f* is

$$app\ f :: \mathbb{N} \rightarrow X \rightarrow M\ X$$

and it has the property:

$$app\ f\ (m + n) = app\ f\ n \diamond (app\ f\ m)$$

Therefore, *app f* is a monoid morphism from $(\mathbb{N}, +, 0)$ to $(X \rightarrow M\ X, \diamond, unit)$, or equivalently, to $Hom_{\mathbb{K}}\ (X, X), \cdot, id$ in the Kleisli category constructed with monad M. This is, in fact, very similar to the classical definition of a dynamical system: a morphism of monoids, whose target is a $Hom\ (x, x)$ set, a set of endo-arrows, except that the category in which these endo-arrows exist is not *Set* but the Kleisli category of m.

This justifies the following definition:

Definition 15 (Dynamical System) *Consider a monoid* $(T, +, 0)$*, a (locally small) category Cat and an object X of this category. A* dynamical system *is a* monoid morphism *from* $(T, +, 0)$ *to* $(Hom_{\mathbb{C}at}\ (X, X), \cdot, id$*, that is, a function*

$$sys :: T \rightarrow Hom_{\mathbb{C}at}\ (X, X)$$

such that

$$\begin{aligned} sys\ 0 &= id \\ sys\ (t1 + t2) &= sys\ t2 \cdot sys\ t1 \end{aligned}$$

This definition unifies the classical and the coalgebraic ones. Indeed, if $Cat = Set$ then we obtain the classical definition. For the coalgebraic point of view, let $F :: \mathbb{C} \rightarrow \mathbb{C}$ be an endofunctor and take *Cat* to be the category with the following elements:

1. Object: the same objects as $\mathbb{C}$
2. Arrows: there exist an arrow $(f, n) :: X \rightarrow Y$ in *Cat* for every arrow $f :: X \rightarrow F^n\ Y$ in $\mathbb{C}$, for any $n \in \mathbb{N}$. Here, F^n has the usual meaning of F composed with itself n times. We consider $F^0 = Id$, and therefore $\mathbb{C}$ can be seen as included in *Cat* (the inclusion functor sends every f to $(f, 0)$).
3. Composition: If $(f, m) :: X \rightarrow Y$ and $(g, n) :: Y \rightarrow Z$, then $(g, n) \cdot (f, m) = h\ (n + m)$ where $h = F^m\ g \cdot f$ in $\mathbb{C}$.

The identities on *Cat* are $(id, 0)$:

$$
\begin{aligned}
& (f, n) \cdot (id, 0) \\
= & \quad \{ \text{ Definition } \cdot \} \\
& (F^0\, f \cdot id, n + 0) \\
= & \quad \{ \text{ Definition } F^0 \} \\
& (Id\, f \cdot id, n) \\
= & \quad \{ \; Id \; \} \\
& (f, n)
\end{aligned}
$$

$$
\begin{aligned}
& (id, 0) \cdot (f, n) \\
= & \quad \{ \text{ Definition } \cdot \} \\
& (F^n\; id \cdot f, 0 + n) \\
= & \quad \{ \; F^n \text{ functor } \} \\
& (id \cdot f, n) \\
= & \quad \{ \; id \; \} \\
& (f, n)
\end{aligned}
$$

Composition is associative:

$$
\begin{aligned}
& (h, p) \cdot ((g, n) \cdot (f, m)) \\
= & \quad \{ \text{ Definition } \cdot \} \\
& (h, p) \cdot (F^m\; g \cdot f, n + m) \\
= & \quad \{ \text{ Definition } \cdot \} \\
& (F^(n + m)\; h \cdot (F^m\; g \cdot f), p + (n + m)) \\
= & \quad \{ \text{ Functors } \} \\
& (F^m\; (F^n\; h \cdot g) \cdot f, (p + n) + m) \\
= & \quad \{ \text{ Definition } \cdot \} \\
& (F^n\; h \cdot g, p + n) \cdot (f, m) \\
= & \quad \{ \text{ Definition of } \cdot \} \\
& ((h, p) \cdot (g, n)) \cdot (f, m)
\end{aligned}
$$

If $f :: X \to F\; X$, then the successive applications of f, $\mathit{fmap}\; f$, $\mathit{fmap}\; (\mathit{fmap}\; f)$, and so on, described above, are given by the arrows corresponding to $(f, 1)$, $(f, 1) \cdot (f, 1)$, $(f, 1) \cdot (f, 1) \cdot (f, 1)$ and so on, that is, by the iterations of $(f, 1)$, that is, by a general dynamical system $\mathit{sys} :: \mathbb{N} \to \mathit{Hom}_{\mathbb{C}at}\; (X, X)$ with $\mathit{sys}\; 1 = (f, 1)$.

5.6 Monadic systems and their trajectories

Finally, the systems obtained by pairing a function of type $X \to M\; X$, where M is a monad, with the identity, are obtained by taking Cat to be the Kleisli category of M:

Definition 16 (Monadic dynamical system) *A* monadic dynamical system *is a dynamical system in which the category is the Kleisli category of a monad M, that is, a monoid morphism sys from a monoid $(T, +, 0)$ to $(\mathit{Hom}_{\mathrm{K}}\; (X, X), \diamond, \mathit{unit})$. In Haskell, we have*

$$\begin{aligned}
&\mathit{sys} :: \mathit{Monoid}\ T \Rightarrow T \to (X \to M\ X)\\
&\mathit{sys}\ 0 = \mathit{return}\\
&\mathit{sys}\ (\mathit{t1} + \mathit{t2}) = (\mathit{sys}\ \mathit{t2} \lhd) \cdot \mathit{sys}\ \mathit{t1}
\end{aligned}$$

In the context of monadic dynamical systems, we are going to refer to T as the monoid *of the system, to X as* the state space *of the system, and to M as* the monad *of the system.*

We can now attempt to generalize the notion of trajectory, in order to take other cases into account. In the classical setting, trajectories were defined as graphs of functions $T \to X$ in the case in which T was an ordered monoid. This definition can be applied with no changes to the monadic case:

$$\mathit{trj}\ \mathit{sys} = \{(t, \mathit{sys}\ t\ x) \mid t \in T\}$$

which gives us, for every $x_0 :: x$ the graph function of a function $T \to M\ X$. We therefore have a general notion of trajectory for the macro-states.

We cannot hope for the same uniformity in computing the M-structure of possible micro-trajectories, because these are going to be in general, for uncountable T, in uncountable number. We can, however, define trajectories along a finite or countable list of elements of T. We remind that in the cases in which T stands for a "time"-set, the values $t :: T$ are interpreted as durations, and not as calendar time. Thus, $\mathit{sys}\ t\ x$ represents the macro-state resulting from x *after* t. Thus, for a list $[\mathit{t\_0}, \mathit{t\_1}, ..., \mathit{t\_n}]$, the trajectory along this list will represent "samples" after $\mathit{t\_0}$, $(\mathit{t\_0} + \mathit{t\_1})$, $\ldots$, $(\mathit{t\_0} + \mathit{t\_1} + ... + \mathit{t\_n})$.

$$\begin{aligned}
&\mathit{mtrj}\ \mathit{sys}\ x\ [\,] = \quad \mathit{return}\ [x]\\
&\mathit{mtrj}\ \mathit{sys}\ x\ (t : \mathit{ts}) = (\mathit{addHist}\ (\mathit{sys}\ t)) \lhd (\mathit{mtrj}\ \mathit{sys}\ x\ \mathit{ts})
\end{aligned}$$

With this definition, we can compute the micro-trajectories of the stochastic system *st* defined above, by first embeding the *st* transition function in a monadic system with the function *app*, and applying *mtrj* to lists of successive "durations" of 1:

```
mtrj (app st) 3 []
    ==> SP [([3],1.0)]

mtrj (app st) 3 [1]
    ==> SP [([2,3],0.4),([4,3],0.6)]

mtrj (app st) 3 [1, 1]
    ==> SP [([1,2,3],0.16),([3,2,3],0.24),([3,4,3],0.24),([5,4,3],0.36)]
```

In order to have a more uniform approach, we can also redefine *trj* to give us the trajectory of the macro-states along a list of elements in T:

$$\begin{aligned} &\mathit{trj}\ \mathit{sys}\ x\ [\,] &&= [\mathit{return}\ x] \\ &\mathit{trj}\ \mathit{sys}\ x\ (t : \mathit{ts}) &&= ((\mathit{sys}\ t) \lhd \mathit{mx}) : \mathit{mxs} \\ & &&\quad \textbf{where} \\ & &&\quad \mathit{mxs} = \mathit{trj}\ \mathit{sys}\ x\ \mathit{ts} \\ & &&\quad \mathit{mx} = \mathit{head}\ \mathit{mxs} \end{aligned}$$

For example:

```
trj (app st) 3 [1, 1]
   ==> [SP [(1,0.16),(3,0.48),(5,0.36)],
        SP [(2,0.4),(4,0.6)],SP [(3,1.0)]]
```

This approach works, of course, for any monadic system, independent of the properties of T, though the interpretation of "trajectory" is not always appropriate. In the next chapter, we shall use *mtrj* to compute micro-trajectories for the case where T is, in fact, not a "time"-set at all, when we study the representation of monadic systems with input.

There are a number of important relationships between *trj*, *sys* and *mtrj*. If *mplus* and e are the associative operation and the unit of the monoid T, let us denote by *sum* the function

$$\mathit{sum} :: [\,T\,] \to T < \mathit{sum} = \mathit{foldr}\ (\mathit{mplus})\ e$$

(This is a trivial generalization of the Haskell Prelude function *sum*, which works only for numeric types.)

By the definition of *foldr*, we have that

$$\mathit{sum}\ (t : \mathit{ts}) = t\ \mathit{mplus}\ (\mathit{sum}\ \mathit{ts})$$

Theorem 7 (*trj* and *sys*) *We have*

$$\mathit{head}\ (\mathit{trj}\ \mathit{sys}\ x\ \mathit{ts}) = \mathit{sys}\ (\mathit{sum}\ \mathit{ts})\ x$$

Proof. By induction over *ts*.

1. *Case* $[\,]$.

   For the left-hand side we have

   $$\begin{aligned} &\mathit{head}\ (\mathit{trj}\ \mathit{sys}\ x\ [\,]) \\ =\ &\quad \{\ \text{Definition of } \mathit{trj}\ \} \\ &\mathit{head}\ ([\mathit{return}\ x]) \\ =\ &\quad \{\ \text{Definition of } \mathit{head}\ \} \\ &\mathit{return}\ x \end{aligned}$$

   For the right-hand side we have

$$
\begin{aligned}
& \mathit{sys}\ (\mathit{sum}\ [\,])\ x \\
= \quad & \{\ \text{Definition of } \mathit{sum}\ \} \\
& \mathit{sys}\ e\ x \\
= \quad & \{\ \mathit{sys} \text{ monoid morphism}\ \} \\
& \mathit{return}\ x
\end{aligned}
$$

2. *Case* $(t : ts)$.

   For the left-hand side:

$$
\begin{aligned}
& \mathit{head}\ (\mathit{trj}\ \mathit{sys}\ x\ (t : ts)) \\
= \quad & \{\ \text{Definition of trj}\ \} \\
& \mathit{head}\ (((\mathit{sys}\ t) \lhd \mathit{mx}) : \mathit{mxs}) \\
& \textbf{where}\ \mathit{mxs} = \mathit{trj}\ \mathit{sys}\ x\ \mathit{ts} \\
& \quad \mathit{mx} = \mathit{head}\ \mathit{mxs} \\
= \quad & \{\ \text{Definition of } \mathit{head}\text{, induction hypothesis}\ \} \\
& (\mathit{sys}\ t) \lhd (\mathit{sys}\ (\mathit{sum}\ \mathit{ts})\ x) \\
= \quad & \{\ \text{Definition } \diamond\ \} \\
& (\mathit{sys}\ t \diamond \mathit{sys}\ (\mathit{sum}\ \mathit{ts}))\ x \\
= \quad & \{\ \mathit{sys} \text{ monoid morphism}\ \} \\
& \mathit{sys}\ (t\ \mathit{mplus}\ (\mathit{sum}\ \mathit{ts}))\ x \\
= \quad & \{\ \text{Property of } \mathit{sum}\text{, see above}\ \} \\
& \mathit{sys}\ (\mathit{sum}\ (t : ts))\ x
\end{aligned}
$$

□

Theorem 8 (*mtrj* and *sys*) *We have*

$$\mathit{fmap}\ \mathit{head}\ (\mathit{mtrj}\ \mathit{sys}\ x\ \mathit{ts}) = \mathit{sys}\ (\mathit{sum}\ \mathit{ts})\ x$$

Proof. By induction over *ts*.

1. *Case* $[\,]$.

$$
\begin{aligned}
& \mathit{fmap}\ \mathit{head}\ (\mathit{mtrj}\ \mathit{sys}\ x\ [\,]) \\
= \quad & \{\ \text{Definition of } \mathit{mtrj}\ \} \\
& \mathit{fmap}\ \mathit{head}\ (\mathit{return}\ [x]) \\
= \quad & \{\ \mathit{return} \text{ natural transformation}\ \} \\
& \mathit{return}\ (\mathit{head}\ [x]) \\
= \quad & \{\ \text{Definition of } \mathit{head}\ \} \\
& \mathit{return}\ x \\
= \quad & \{\ \text{Same as case } [\,] \text{ in previous proof}\ \} \\
& \mathit{sys}\ (\mathit{sum}\ [\,])\ x
\end{aligned}
$$

2. *Case* $(t : ts)$.

$$\begin{aligned}
& \mathit{fmap}\ \mathit{head}\ (\mathit{mtrj}\ \mathit{sys}\ x\ (t : ts)) \\
=\ & \quad \{ \text{ Definition of } \mathit{mtrj} \ \} \\
& \mathit{fmap}\ \mathit{head}\ (\mathit{addHist}\ (\mathit{sys}\ t) \lhd (\mathit{mtrj}\ \mathit{sys}\ x\ ts)) \\
=\ & \quad \{ \text{ Composition } \} \\
& (\mathit{fmap}\ \mathit{head} \cdot \mathit{addHist}\ (\mathit{sys}\ t)) \lhd (\mathit{mtrj}\ \mathit{sys}\ x\ ts) \\
=\ & \quad \{ \text{ Theorem 5 } \} \\
& (\mathit{sys}\ t \cdot \mathit{head}) \lhd (\mathit{mtrj}\ \mathit{sys}\ x\ ts) \\
=\ & \quad \{ \ \mathit{fmap} \text{ and } \lhd\text{, see Section 2.2 } \} \\
& (\mathit{sys}\ t) \lhd (\mathit{fmap}\ \mathit{head}\ (\mathit{mtrj}\ \mathit{sys}\ x\ ts)) \\
=\ & \quad \{ \text{ Induction hypothesis } \} \\
& (\mathit{sys}\ t) \lhd (\mathit{sys}\ (\mathit{sum}\ ts)\ x) \\
=\ & \quad \{ \ \mathit{sys} \text{ monoid morphism, property of } \mathit{sum} \ \} \\
& \mathit{sys}\ (\mathit{sum}\ (t : ts))\ x
\end{aligned}$$

□

Corollary 1 *For all $n \leqslant \mathit{length}\ ts$ we have*

1. $(\mathit{trj}\ \mathit{sys}\ x\ ts)\ !!\ n =$
$\quad \mathit{sys}\ (\mathit{sum}\ [ts\ !!\ i \mid i \leftarrow [n \mathinner{..} (\mathit{length}\ ts - 1)]])\ x$

2. $\mathit{fmap}\ (!!n)\ (\mathit{mtrj}\ \mathit{sys}\ x\ ts) =$
$\quad \mathit{sys}\ (\mathit{sum}\ [ts\ !!\ i \mid i \leftarrow [n \mathinner{..} (\mathit{length}\ ts - 1)]])\ x$

3. $\mathit{fmap}\ (!!n)\ (\mathit{mtrj}\ \mathit{sys}\ x\ ts) = (\mathit{trj}\ \mathit{sys}\ x\ ts)\ !!\ n$

Proof.

1. Induction on ts.

   (a) *Case* $[\,]$.
   We have that $n \leqslant \mathit{length}\ [\,] \equiv n = 0$.

$$\begin{aligned}
& (\mathit{trj}\ \mathit{sys}\ x\ [\,])\ !!\ 0 \\
=\ & \quad \{ \text{ Definition of } !! \ \} \\
& \mathit{head}\ (\mathit{trj}\ \mathit{sys}\ x\ [\,]) \\
=\ & \quad \{ \text{ Theorem 7 } \} \\
& \mathit{sys}\ (\mathit{sum}\ [\,])\ x \\
=\ & \quad \{ \text{ List comprehension } \} \\
& \mathit{sys}
\end{aligned}$$

(b) *Case* $(t : ts)$.
If $n > 0$ we have

$$
\begin{aligned}
& (\mathit{trj}\ \mathit{sys}\ x\ (t : \mathit{ts}))\ !!\ n \\
= \quad & \{\ \text{Definition of } !!\ \} \\
& (\mathit{tail}\ (\mathit{trj}\ \mathit{sys}\ x\ (t : \mathit{ts})))\ !!\ (n-1) \\
= \quad & \{\ \text{Definition of } \mathit{trj}\ \} \\
& (\mathit{tail}\ (((\mathit{sys}\ t) \lhd \mathit{mx}) : \mathit{mxs}))\ !!\ (n-1) \\
& \mathbf{where}\ \mathit{mxs} = \mathit{trj}\ \mathit{sys}\ x\ \mathit{ts} \\
& \quad\ \mathit{mx} = \mathit{head}\ \mathit{mxs} \\
= \quad & \{\ \text{Definition of } \mathit{tail}\ \} \\
& (\mathit{trj}\ \mathit{sys}\ x\ \mathit{ts})\ !!\ (n-1) \\
= \quad & \{\ \text{Induction hypothesis}\ \} \\
& \mathit{sys}\ (\mathit{sum}\ [\mathit{ts}\ !!\ i \mid i \leftarrow [n-1\,..\,(\mathit{length}\ \mathit{ts} - 1)]])\ x \\
= \quad & \{\ \text{List comprehension and } !!\ \} \\
& \mathit{sys}\ (\mathit{sum}\ [(t : \mathit{ts})\ !!\ i \mid i \leftarrow [n\,..\,(\mathit{length}\ (t : \mathit{ts}) - 1)]])\ x
\end{aligned}
$$

If $n = 0$ then

$$
\begin{aligned}
& (\mathit{trj}\ \mathit{sys}\ x\ (t : \mathit{ts}))\ !!\ 0 \\
= \quad & \{\ \text{Definition } !!\ \} \\
& \mathit{head}\ (\mathit{trj}\ \mathit{sys}\ x\ (t : \mathit{ts})) \\
= \quad & \{\ \text{Theorem 7}\ \} \\
& \mathit{sys}\ (\mathit{sum}\ (t : \mathit{ts}))\ x \\
= \quad & \{\ \text{List comprehension and } !!\ \} \\
& \mathit{sys}\ (\mathit{sum}\ [(t : \mathit{ts})\ !!\ i \mid i \leftarrow [0\,..\,(\mathit{length}\ (t : \mathit{ts}) - 1)]])\ x
\end{aligned}
$$

The proof of the second item is similar, and we omit it. The third item results immediately from transitivity of equality.

□

5.7 Conclusions

In order to make a vulnerability assessment, we have to consider possible evolutions of the current state of affairs. In computational vulnerability assessment, these possible evolutions are given by the trajectories of a representation of the *social-ecological system*, which representation is usually a combination of dynamical systems of heterogenous types: deterministic, non-deterministic, stochastic, or fuzzy.

In an attempt to interpret and compute in a uniform way these "possible evolutions", we have started by studying the notion of dynamical system.

Having reviewed the classical definition commonly used in Systems Theory, based on a morphism of monoids, we have examined the competing approach used mainly in Computing Science, which views dynamical systems as coalgebras. We have seen that these offer a satisfying view of dynamical systems with "hidden" states, and we have seen that the generality of the coalgebraic approach is made necessary by the study of more complicated systems, such as those of abstract datatypes.

The example of instances of abstract datatypes, objects with an internal state, has led us back to the study of trajectories. Writing an abstract datatype representation of discrete non-deterministic dynamical systems, we were led to a definition of "micro"-trajectories, computed by the standard programming trick of storing a history of the changes in the internal state of the datatype.

Finally, we have isolated a class of coalgebras, those given by monadic functors, for which we can compute trajectories in a particularly easy way. This led us to a point of view very similar to the classical one, which, in turn, suggested a general definition of dynamical systems which generalizes both the classical one and the coalgebraic one. Finally, we have returned to the monadic dynamical systems in this more general context, defined macro- and micro-trajectories for them in a general way, and given algorithms for computing both.

In the next chapter, we shall continue our examination of monadic systems. We shall examine monadic systems with input and ways in which to combine monadic systems of the same type, and of different types, to form new monadic systems.

Chapter 6

Working with Monadic Systems

In the previous chapter, we have identified a class of dynamical systems for which we could compute in a simple and uniform manner the micro- and macro-trajectories which represent the possible evolutions of an initial state. In this chapter, we continue the investigation of this class. We start by looking at systems "with input" in order to see whether they are instances of monadic systems. We then define and implement the elements of a simple algebra of monadic systems: ways of combining such systems in order to yield new ones, for example in series, in parallel and so on.

A word about the examples chosen to illustrate the concepts of this chapter. They are exceedingly simple, for the following reason. We will be dealing with quite abstract and general ways of computing trajectories of interacting systems. This is already hard enough: if we layer on top of this the additional complexity of the systems themselves, our chances of understanding the abstract operations and convincing ourselves of their correctness will be greatly diminished.

6.1 Monadic systems with input

In many contexts, the notion of a (discrete) system with input is associated to a function of the form $f :: (X, A) \to X$, the idea being of iterations which can be performed starting from a given initial state with an additional input being provided at each step of the iteration: from the initial x_0 and the additional initial input *a_0* we obtain x_1, this together with *a_1* give us x_2, and so on.

If we want to model such a process as a monadic system, we have to find a monoid T and a monad M such that the (micro- or macro-) trajectories of a system $T \to X \to M\ X$ correspond to the results described above.

Since the process of obtaining succesive states is a discrete one, it seems natural to choose $T = \mathbb{N}$. Then the system we seek is given by iterations of its transition function of type $X \to M\ X$, for a convenient M. Again, a natural idea is to curry f, thus obtaining $\mathit{curry}\ f :: X \to (A \to X)$. The functor $\mathit{Exp}\ A$, which associates to each type X the type of all functions $A \to X$ is, indeed, a monad:

```
newtype Exp a x = E (a → x)


unwrapE          :: Exp a x → a → x
unwrapE (E f) = f


instance Monad (Exp a) where
  return = E · const
  ef ▷ f = E g
           where
             g a = unwrapE (f (unwrapE ef a)) a
```

The verification of the monad laws is trivial. Haskell programmers are more familiar with *Exp* under the somewhat puzzling name *Reader*.

Now, if we have an $f :: (X, A) \to X$ we obtain a system $\mathbb{N} \to (X \to \mathit{Exp}\ A\ X)$ by using the function *app* defined in Chapter 5 with the uncurried version of f. For example, consider a simple adder:

```
adder          :: (ℝ, Char) → ℝ
adder (r, c) = r + (realToFrac (fromEnum c))
```

Here the type of the states is $\mathbb{R}$ and the type of the input is *Char* (the usage of *realToFrac* is mandated by the Haskell cast rules). The function *adder* increments the state represented by a real number r with the ASCII code of the input character. The transition function of the monadic system corresponding to *adder* is then

```
tadd    :: ℝ → Exp Char ℝ
tadd r = E (λc → adder (r, c))
```

and the system is given by *app tadd*. The trajectory of the macro-states along a list of type *Char* elements is a list of elements of type $\mathit{Exp}\ \mathit{Char}\ \mathbb{R}$. Since this is a type of functions, we cannot print the values of this type, but we can print the values of functions for some arguments, say for the first 5 characters in the alphabet:

```
toList          :: Exp Char a → [a]
toList (E f) = [f c | c ← ['a' .. 'e']]
```

We can now check the macro-trajectories along some simple lists:

```
map toList (trj (app tadd) 4.2 [])
    ==> [[4.2, 4.2, 4.2, 4.2, 4.2]]
```

No matter what the input, after *no* steps, the state remains the same. This is what we expected (by the definition of *return*).

```
map toList (trj (app tadd) 4.2 [1])
    ==> [[101.2,102.2,103.2,104.2,105.2],[4.2,4.2,4.2,4.2,4.2]]
```

After one step, the state changes to the initial state, plus the input. This agrees to our description of systems with input.

```
map toList (trj (app tadd) 4.2 [1, 1])
    ==> [[198.2,200.2,202.2,204.2,206.2],
         [101.2,102.2,103.2,104.2,105.2],
         [4.2,4.2,4.2,4.2,4.2]]
```

After two steps, the state changes to the initial state plus *twice* the input. Unfortunately, this no longer corresponds to the description above. It is a consequence of the definition of the monadic operators for *Exp a*: an input a is given once, and fixed for the rest of the trajectory. In other words, the process is described by: starting in x_0, an input a produces $x_1 = f\ (x_0, a)$, $x_2 = f\ (x_1, a)$, $\ldots x_n = f\ (x_{n-1}, a)$.

The micro-trajectories, stored in a structure of type *Exp* $\mathbb{N}$ $[\mathbb{R}]$, make this behavior even clearer:

```
toList (mtrj (app tadd) 4.2 [1, 1])
    ==> [[198.2,101.2,4.2],[200.2,102.2,4.2],[202.2,103.2,4.2],
         [204.2,104.2,4.2],[206.2,105.2,4.2]]
```

To each element of type $\mathbb{N}$ corresponds the trajectory of states obtained by giving that element as input, at every step.

Although this does not correspond to the idea of "system with input" which we wanted to capture, this behavior is quite important, since it describes *parametrized* systems. In programming, parameters are often organized in "environments": the behavior of the system depends on the environment, but the environment is chosen only once, in the beginning, and does not change in the course of a simulation.

Back to formulating a monadic representation of a system with input: choosing the *Exp A* monad has failed. This choice is, in fact, the standard one in the coalgebraic description of systems. The successive applications of the transition function lead to values of types *Exp A X*, *Exp A* (*Exp A X*), *Exp A* (*Exp A* (*Exp A X*)) and so on. In order to obtain a value representing a state after n steps, one has to provide n values of type A: one input per step. When working with monadic systems, however, the type of the value obtained after n steps is always the same, namely $\mathit{Exp}\ A\ X$. “Collapsing” the sequence of $\mathit{Exp}\ A\ (\mathit{Exp}\ A\ ...\ (\mathit{Exp}\ A\ X))$ down to $\mathit{Exp}\ A\ X$ is done by threading *one* input value through all the levels.

If the coalgebraic choice does not lead to satisfactory results in the context of monadic system, what about the classical approach? Here, one chooses a different *monoid* T to drive the dynamical system, namely $(T = [A], +\!\!+, [\,])$, that is, lists of elements of type A under concatenation, with the empty list as neutral element. The system corresponding to f is defined as

$$\begin{aligned} &\mathit{sys} && :: [A] \to X \to X \\ &\mathit{sys}\ [\,] && = \mathit{id} \\ &\mathit{sys}\ (a : \mathit{as})\ x && = f\ (\mathit{sys}\ \mathit{as}\ x, a) \end{aligned}$$

and it’s easy to see that this is indeed a morphism from $([A], +\!\!+, [\,])$ to $(X \to X, \cdot, \mathit{id})$.

In fact, the original definition of a dynamical system given in Section 5.1 was given in terms of a function $X \times T \to X$ with properties which made it equivalent to a morphism of monoids. Such a function is called a *monoid action*: the representation of systems with input offers an intuition for this terminology.

In general, we can obtain for any function $f :: (X, A) \to M\ X$ a system

$$\mathit{inpsys}\ f :: [A] \to X \to M\ X$$

that is, a monoid morphism from $([A], +\!\!+, [\,])$ to $(X \to M\ X, \diamond, \mathit{return}$, by defining

$$\begin{aligned} &\mathit{inpsys}\ f\ [\,] && = \mathit{return} \\ &\mathit{inpsys}\ f\ (a : \mathit{as}) && = f' \diamond \mathit{inpsys}\ f\ \mathit{as} \\ & && \quad \mathbf{where} \\ & && \quad f'\ x = f\ (x, a) \end{aligned}$$

The function *inpsys* is similar to the function *app* defined in Section 5.5. In particular, the function $f :: (X, A) \to M\ X$ can be considered the analog of the transition function.

To illustrate this definition, consider a stochastic version of the adder defined above:

$$\begin{aligned} &\mathit{sadder} :: && (\mathbb{R}, \mathit{Char}) \to \mathit{SimpleProb}\ \mathbb{R} \\ &\mathit{sadder}\ (x, c) = && \mathit{SP}\ [(x + n - 1, 0.4), (x + n + 1, 0.6)] \\ & && \mathbf{where}\ n = \mathit{realToFrac}\ (\mathit{fromEnum}\ c) \end{aligned}$$

We can use *inpsys sadder* directly in *trj* to obtain the macro-trajectories of the system:

```
trj (inpsys sadder) 4.2 []
```

```
    ==> [SP [(4.2,1.0)]]

trj (inpsys sadder) 4.2 [['a']]
    ==> [SP [(100.2,0.4),(102.2,0.6)],SP [(4.2,1.0)]]

trj (inpsys sadder) 4.2 [['a'], ['b']]
    ==> [SP [(197.2,0.16),(199.2,0.24),(199.2,0.24),(201.2,0.36)],
         SP [(101.2,0.4),(103.2,0.6)],
         SP [(4.2,1.0)]]
```

and in *mtrj* to obtain the micro-trajectories:

```
mtrj (inpsys sadder) 4.2 []
    ==> SP [([4.2],1.0)]

mtrj (inpsys sadder) 4.2 [['a']]
    ==> SP [([100.2,4.2],0.4),([102.2,4.2],0.6)]

mtrj (inpsys sadder) 4.2 [['a'], ['b']]
    ==> SP [([197.2,101.2,4.2],0.16),([199.2,101.2,4.2],0.24),
            ([199.2,103.2,4.2],0.24),([201.2,103.2,4.2],0.36)]
```

A similar approach allows us to treat continuous systems with input, that is, where the input is given as a function of one real variable taking values in a semi-open interval for some $t \in \mathbb{R}$, $t \geqslant 0$. Classically, such systems are presented as f::, with the interpretation that $f\ (x, inp)$ is the state after consuming input *inp* starting from state x.

We can define the input type in Haskell as

```
newtype $ContInp\ a = CI\ (\mathbb{R}, \mathbb{R} \to a)$

$apply$                  :: $ContInp\ a \to \mathbb{R} \to a$
$apply\ (CI\ (t, f))\ t' =$ if $t' \leqslant t \wedge 0 < t'$ then $f\ t'$
                          else $\perp$
```

ContInp a is then the relevant monoid for the definition of a dynamical system:

```
instance $Monoid\ (ContInp\ a)$ where
  $mempty = CI\ (0, \lambda t \to \perp)$
  $(CI\ (t1, f1))$ `mappend` $(CI\ (t2, f2)) = CI\ (t1 + t2, f)$
    where
    $f\ t =$ if $t \leqslant t1$ then $f1\ t$
      else if $t \leqslant t1 + t2$ then
        $f2\ (t1 + t2 - t)$
        else $\perp$
```

The names *mempty* and *mappend* are given to the neutral element and the operation of the monoid, respectively, in the Haskell library. The *ContInp a* monoid has as unit the empty function, and as operation the "pasting" together of the two arguments. Given an f as above, but with a general monadic output, $f :: (X, \mathit{ContInp}\ A) \to M\ X$, the system with input is constructed by the function *cinpsys* defined as

$$\mathit{cinpsys}\ f :: \mathit{ContInp}\ A \to X \to M\ X$$

$$\mathit{cinpsys}\ f\ \mathit{ci} = \lambda x \to f\ (x, \mathit{ci})$$

The trajectories of this system can then be computed using *trj* and *mtrj* without any changes.

6.2 Combining monadic dynamical systems

We have seen until now that *trj* and *mtrj* can be used to compute in a uniform way the macro- and micro-trajectories of many different types of systems: deterministic, non-deterministic, stochastic, fuzzy, each with or without input, discrete or continuous. In this section, we explore how monadic systems can be combined to yield more complex monadic systems.

It is, in fact, a common task to build a model of a complex system, such as the climate system, from building blocks: a model of the atmosphere, a model of the ocean, one for the ice-sheets at the poles, and so on. Traditionally, such components are represented in a similar way, for example, they are described as solutions to systems of partial differential equations, yielding deterministic systems. In such cases, the main problems are to ensure that the models are "compatible": that none of them violates assumptions that are vital to the functioning of the others, and that the meanings of various shared variables are not contradictory. Usually, there will be a need for pre- and post-processing the results of one model before they can be passed to the others: for example, data which is geographically situated might have to be interpolated to account for different resolutions, or it might have to be filtered or smoothened in order to fit with the numerical schemes used by the other models.

These are *not* the difficulties we address here. The problems we aim to solve are characteristic of the type of modeling needed in vulnerability studies, where the building blocks are represented in very different ways: physical systems might be described by deterministic or stochastic systems, while the social systems are usually described as non-deterministic or fuzzy. Even if all the systems operate on the same time scale, even if the geographical resolution is identical and all shared variables have uniform semantics, it is still not clear how to compute the trajectory of a combination of, say, fuzzy and stochastic systems.

A discussion of combining monadic systems can hardly be exhaustive. It is well known (see, for example, Section 10.4 in [1]) that there is no unique way of combining monads to yield monads. The same two monads may be combined differently, and which way is chosen is decided on the basis of pragmatic considerations. Accordingly, we shall present those combinations of systems which are most likely to be useful in the practice of computational vulnerability assessment.

6.2.1 Parallel combination

Conceptually, the simplest way of combining two systems is to put them in parallel. The trajectories of the two systems are computed independently, and are combined by tupling (product). The product of two monoids is a monoid:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Monoid}\ m, \mathit{Monoid}\ n) \Rightarrow \mathit{Monoid}\ (m, n)\ \textbf{where}\\ \quad \mathit{mempty} = (\mathit{mempty}, \mathit{mempty})\\ \quad \mathit{mappend}\ (a, b)\ (c, d) = (\mathit{mappend}\ a\ c, \mathit{mappend}\ b\ d)\end{array}$$

The product of two monads M and N always results in a monad $M \times N$: everything is done "component-wise":

$$\begin{array}{l}\mathit{return}\ (x, y) = (\mathit{return}\ x, \mathit{return}\ y)\\ (f, g) \lhd (mx, ny) = (f \lhd mx, g \lhd ny)\end{array}$$

Checking that the monad laws are satisfied is trivial.

Therefore, given two systems $\mathit{sys1} :: \mathit{T1} \to X \to M\ X$ and $\mathit{sys2} :: \mathit{T2} \to Y \to N\ Y$, putting them in parallel results in the system $\mathit{syspar}\ (\mathit{sys1}, \mathit{sys2}) :: (\mathit{T1}, \mathit{T2}) \to (X, Y) \to (M\ X, N\ Y)$. Again, it is trivial to check that $\mathit{syspar}\ (\mathit{sys1}, \mathit{sys2})$ is a monoid morphism.

Unfortunately, $\mathit{syspar}\ (\mathit{sys1}, \mathit{sys2})$ cannot be used with trj and mtrj, because, in fact, $M \times N$ cannot be declared as to be a monad in Haskell! The reason for that is that, in Haskell, we can only declare monads which are endo-functors on Set, while $M \times N$ is an endo-functor on the product category $\mathit{Set} \times \mathit{Set}$, so that the above instance declaration for $M \times N$ would be rejected. Another way of expressing this is that in Haskell monads are *unary* type constructors, while $M \times N$ is a binary type constructor.

In order to circumvent this problem, we first define the "pairing" of two monads:

$$\textbf{newtype}\ \mathit{PairM}\ m\ n\ a = \mathit{PM}\ (m\ a, n\ a)$$

$$\begin{array}{ll}\mathit{unwrapPM} :: & \mathit{PairM}\ m\ n\ a \to (m\ a, n\ a)\\ \mathit{unwrapPM}\ (\mathit{PM}\ x) = x & \end{array}$$

```
instance $(Monad\ m, Monad\ n) \Rightarrow Monad\ (PairM\ m\ n)$ where
  $return\ x = PM\ (return\ x, return\ x)$
  $(PM\ (mx, nx)) \rhd f = PM\ (mx', nx')$
    where
    $mx' = mx \rhd (fst \cdot unwrapPM \cdot f)$
    $nx' = nx \rhd (snd \cdot unwrapPM \cdot f)$
```

The next step is to "extend" the types of *sys1* and *sys2* to $(X, Y) \rightarrow M\ (X, Y)$ and $(X, Y) \rightarrow N\ (X, Y)$ respectively. This can be done by using the more general functions *pr* and *pl* (standing for "product right" and "product left", respectively).

```
$pr$          $:: Functor\ f \Rightarrow (f\ a, b) \rightarrow f\ (a, b)$
$pr\ (fa, b) = fmap\ (\lambda a \rightarrow (a, b))\ fa$
```

```
$pl$          $:: Functor\ f \Rightarrow (b, f\ a) \rightarrow f\ (b, a)$
$pl\ (b, fa) = fmap\ (\lambda a \rightarrow (b, a))\ fa$
```

Finally, we can define *syspar* as

```
$syspar :: (Functor\ m, Functor\ n) \Rightarrow$
           $(t1 \rightarrow x \rightarrow m\ x, t2 \rightarrow y \rightarrow n\ y) \rightarrow$
           $(t1, t2) \rightarrow (x, y) \rightarrow PairM\ m\ n\ (x, y)$
$syspar\ (sys1, sys2)\ (t1, t2)\ (x, y) = PM\ (pr\ (sys1\ t1\ x, y),$
  $pl\ (x, sys2\ t2\ y))$
```

The definition of *syspar* requires only the functoriality of m and n. If, however, m and n are monads and *sys1* and *sys2* are monoid morphisms, then $syspar\ (sys1, sys2)$ is also a monoid morphism, therefore a monadic system.

The elements constructed by *syspar* contain a lot of redundant information, which can be stripped away by the following function:

```
$strip :: (Functor\ m, Functor\ n) \Rightarrow PairM\ m\ n\ (x, y) \rightarrow (m\ x, n\ y)$
$strip\ (PM\ (mxy, nxy)) = (fmap\ fst\ mxy, fmap\ snd\ nxy)$
```

In fact, *strip* allows us to recover in a sense the intended combination of systems, namely $cross \cdot (sys1\ t1, sys2\ t2)$. We have

Proposition 4 (*strip*) *With the notations above, we have*

$$strip \cdot syspar\ (sys1, sys2)\ (t1, t2) = cross\ (sys1\ t1, sys2\ t2)$$

Proof.
We have

$$\begin{aligned}
& (\mathit{strip} \cdot \mathit{syspar}\ (\mathit{sys1}, \mathit{sys2})\ (\mathit{t1}, \mathit{t2}))\ (x, y) \\
= \quad & \{ \text{ Definition of } \mathit{syspar}\text{, composition } \} \\
& \mathit{strip}\ (\mathit{PM}\ (\mathit{pr}\ (\mathit{sys1}\ \mathit{t1}\ x, y), \mathit{pl}\ (x, \mathit{sys2}\ \mathit{t2}\ y))) \\
= \quad & \{ \text{ Definition of } \mathit{strip} \ \} \\
& (\mathit{fmap}\ \mathit{fst}\ (\mathit{pr}\ (\mathit{sys1}\ \mathit{t1}\ x, y)), \\
& \quad \mathit{fmap}\ \mathit{snd}\ (\mathit{pl}\ (x, \mathit{sys2}\ \mathit{t2}\ y))) \\
= \quad & \{ \text{ Definitions of } \mathit{pr}, \mathit{pl} \ \} \\
& (\mathit{fmap}\ \mathit{fst}\ (\mathit{fmap}\ (\lambda a \rightarrow (a, y))\ (\mathit{sys1}\ \mathit{t1}\ x)), \\
& \quad \mathit{fmap}\ \mathit{snd}\ (\mathit{fmap}\ (\lambda b \rightarrow (x, b))\ (\mathit{sys2}\ \mathit{t2}\ y))) \\
= \quad & \{ \text{ Functor } \} \\
& (\mathit{fmap}\ (\mathit{fst} \cdot (\lambda a \rightarrow (a, y)))\ (\mathit{sys1}\ \mathit{t1}\ x), \\
& \quad \mathit{fmap}\ (\mathit{snd} \cdot (\lambda b \rightarrow (x, b)))\ (\mathit{sys2}\ \mathit{t2}\ y)) \\
= \quad & \{ \text{ Definitions of } \mathit{fst}, \mathit{snd}, \mathit{id} \ \} \\
& (\mathit{fmap}\ \mathit{id}\ (\mathit{sys1}\ \mathit{t1}\ x), \mathit{fmap}\ \mathit{id}\ (\mathit{sys2}\ \mathit{t2}\ y)) \\
= \quad & \{ \text{ Functor } \} \\
& (\mathit{sys1}\ \mathit{t1}\ x, \mathit{sys2}\ \mathit{t2}\ y) \\
= \quad & \{ \text{ Definition of } \mathit{cross} \ \} \\
& \mathit{cross}\ (\mathit{sys1}\ \mathit{t1}, \mathit{sys2}\ \mathit{t2})\ (x, y)
\end{aligned}$$

□

To illustrate, consider the system *inpsys sadder* defined above with the discrete non-deterministic system given by the transition function

```
next2   :: Char → [Char]
next2 c = [toEnum ((n + 1 − a) `mod` m + a),
             toEnum (((n + 2 − a) `mod` m) + a)]
          where
          n = fromEnum c
          m = fromEnum 'z' − fromEnum 'a' + 1
          a = fromEnum 'a'
```

that is, *app next2*. We can combine this two systems in parallel with *syspar*, and we have, for example

```
syspar (inpsys sadder, app next2) ([' '], 1) (4.2, 'c')
    ==> (SP [((35.2,'c'),0.4),((37.2,'c'),0.6)],[(4.2,'d'),(4.2,'e')])

strip (syspar (inpsys sadder, app next2) ([' '], 1) (4.2, 'c'))
    ==> (SP [(35.2,0.4),(37.2,0.6)],"de")
```

We can compute macro- and micro-trajectories as usual with *trj* and *mtrj*:

```
map strip (trj (syspar (inpsys sadder, app next2)) (4.2, 'c') [])
    ==> [(SP [(4.2,1.0)],"c")]

map strip (trj (syspar (inpsys sadder, app next2))
                           (4.2, 'c') [(" ", 1)])
    ==> [(SP [(35.2,0.4),(37.2,0.6)],"de"),(SP [(4.2,1.0)],"c")]

map strip (trj (syspar (inpsys sadder, app next2))
                           (4.2, 'c') [(" ", 1), ("a", 1)])
    ==> [(SP [(131.2,0.16),(133.2,0.48),(135.2,0.36)],"effg"),
         (SP [(100.2,0.4),(102.2,0.6)],"de"),(SP [(4.2,1.0)],"c")]
```

The result of *mtrj* has the type *Pair SimpleProb* [] [(*Double*, *Char*)], so that the *strip* function defined above won't work: we have an additional level to strip into, as it were. We can generalize the function to act on structures of pairs:

$$\begin{aligned}&\mathit{genstrip} :: (\mathit{Functor}\ f, \mathit{Functor}\ m, \mathit{Functor}\ n) \Rightarrow\\ &\quad \mathit{PairM}\ m\ n\ (f\ (x, y)) \to (m\ (f\ x), n\ (f\ y))\\ &\mathit{genstrip}\ (\mathit{PM}\ (\mathit{mxys}, \mathit{nxys})) = (\mathit{fmap}\ (\mathit{fmap}\ \mathit{fst})\ \mathit{mxys}, \mathit{fmap}\ (\mathit{fmap}\ \mathit{snd})\ \mathit{nxys})\end{aligned}$$

strip is then a special case of *genstrip* for the case in which $f = \mathit{Id}$.

We have

$$\begin{aligned}&\mathit{genstrip}\ (\mathit{mtrj}\ (\mathit{syspar}\ (\mathit{inpsys}\ \mathit{sadder}, \mathit{app}\ \mathit{next2}))\\ &\quad (4.2, \texttt{'c'})\ [(\texttt{" "}, 1), (\texttt{"a"}, 1)])\\ &(\mathit{SP}\ [([131.2, 100.2, 4.2], 0.16), ([133.2, 100.2, 4.2], 0.24),\\ &\quad ([133.2, 102.2, 4.2], 0.24), ([135.2, 102.2, 4.2], 0.36)],\\ &\quad [\texttt{"edc"}, \texttt{"fdc"}, \texttt{"fec"}, \texttt{"gec"}])\end{aligned}$$

There is another common pattern of parallel combination of systems. This is the case when the systems all have the same monoid, state space and monad, they are started from the same initial state and are driven by the same list of elements. This is the case when these systems are supposed to represent alternative evolutions, such as different scenarios. Let us assume that the systems are indexed with elements from a set I:

$$\mathit{sysfam} :: I \to T \to X \to M\ X$$

Then, the parallel evolution of this family of systems is described by a system

$$\mathit{parfam}\ \mathit{sysfam} :: T \to X \to I \to M\ X$$

where $\mathit{parfam}\ \mathit{sysfam}\ t\ x\ i = \mathit{sysfam}\ i\ t\ x$ represents the result given by the ith system in the family when started from the common initial state x and driven by the common input t.

The combined system *parfam* has the same monoid and state space as the systems in the familiy, but a different monad, namely the one given by the functor $F\ X = I \to M\ X$. This is, in fact, the composition of the *Exp I* monad defined above with M, that is, $F = \mathit{Exp}\ I \cdot M$. In other words, a parametrized system can be seen as describing the parallel evolution of a familiy of systems indexed by the parameter.

In the Haskell terminology, an operation which constructs a monad such as $\mathit{Exp}\ I \cdot M$ from a monad M is called *monad transformer* (see, for example, section 10.4 of [1] or the article [26]).

In order to be able to use *parfam* with *trj* and *mtrj* we need to define $\mathit{Exp}\ I \cdot M$ explicitely as a monad.

```
newtype ExpT i m a = ExpT (i → m a)
```

```
unwrapExpT :: ExpT i m a → (i → m a)
unwrapExpT (ExpT f) = f
```

```
instance Functor m ⇒ Functor (ExpT i m) where
  fmap f (ExpT g) = ExpT (fmap f · g)
```

```
instance Monad m ⇒ Monad (ExpT i m) where
  return x = ExpT (const (return x))
  (ExpT f) ▷ g = ExpT h
    where
    h i = (f i) ▷ g′
          where
          g′ a = (unwrapExpT · g) a i
```

It is easy to see that the *return* and ▷ operators fulfill the necessary conditions that define a monad. For example, in order to prove the “associativity” of ▷, that is

```
((ExpT f) ▷ g) ▷ h = (ExpT f) ▷ g′
  where g′ a = ((unwrapExpT · g) a) ▷ h
```

we compute for the left-hand side:

$$
\begin{array}{ll}
 & ((\mathit{ExpT}\ f) \rhd g) \rhd h \\
= & \{\ \text{Definition of } \rhd \text{ for } \mathit{ExpT}\ \} \\
 & (\mathit{ExpT}\ (\lambda i \to ((f\ i) \rhd (\lambda a \to (\mathit{unwrapExpT} \cdot g)\ a\ i)))) \rhd h \\
= & \{\ \text{Definition of } \rhd \text{ for } \mathit{ExpT} \text{ and } \lambda \text{ application}\ \} \\
 & \mathit{ExpT}\ (\lambda i' \to (((f\ i') \gg= (\lambda a \to (\mathit{unwrapExpT} \cdot g)\ a\ i')) \\
 & \ \rhd (\lambda b \to (\mathit{unwrapExpT} \cdot h)\ b\ i'))) \\
= & \{\ m \text{ monad}, \rhd \text{ for } m \text{ is associative}\ \} \\
 & \mathit{ExpT}\ (\lambda i' \to ((f\ i') \rhd h')) \\
 & \mathbf{where} \\
 & h'\ a = ((\mathit{unwrapExpT} \cdot g)\ a\ i') \rhd (\lambda b \to (\mathit{unwrapExpT} \cdot h)\ b\ i') \\
= & \{\ \text{Definition of } \rhd\ \} \\
 & (\mathit{ExpT}\ f) \rhd h' \\
 & \mathbf{where} \\
 & h'\ a = \mathit{ExpT}\ (((\lambda i \to (\mathit{unwrapExpT} \cdot g)\ a\ i') \rhd \\
 & \quad (\lambda b \to (\mathit{unwrapExpT} \cdot h)\ b\ i')))
\end{array}
$$

and for the right-hand side:

$$
\begin{array}{ll}
 & (\mathit{ExpT}\ f) \rhd g' \\
 & \mathbf{where}\ g'\ a = ((\mathit{unwrapExpT} \cdot g)\ a) \rhd h \\
= & \{\ \text{Definition of } \rhd \text{ for } \mathit{ExpT}\ \} \\
 & (\mathit{ExpT}\ f) \rhd g' \\
 & \mathbf{where} \\
 & g'\ a = \mathit{ExpT}\ (\lambda i \to (((\mathit{unwrapExpT} \cdot g)\ a\ i) \\
 & \quad \rhd (\lambda b \to (\mathit{unwrapExpT} \cdot h)\ b\ i)))
\end{array}
$$

Since h' and g' denote the same function, both sides reduce to the same expression, which concludes the proof.

We can now define *parfam* as follows:

$$
\begin{array}{l}
\mathit{parfam} :: \mathit{Monad}\ m \Rightarrow (i \to t \to x \to m\ x) \to \\
\quad t \to x \to \mathit{ExpT}\ i\ m\ x \\
\mathit{parfam}\ \mathit{sysfam}\ t\ x = \mathit{ExpT}\ (\lambda i \to \mathit{sysfam}\ i\ t\ x)
\end{array}
$$

Just as the Haskell standard libraries use the name *Exp* instead of *Reader*, they call this transformer *ReaderT*. We will see an important usage of *ExpT* in the next section.

6.2.2 Serial combination

As one would expect, the situation is more complicated when the systems interact. In this section, we consider interaction over the state: the two systems are assumed to have the same state space X, although possibly different monoids and monads. We want to represent the following situation:

an initial state (element of X) is modified by the first system according to an input from its own monoid, the resulting macro-state is then modified by the second system according to an input from its monoid, the resulting macro-state is given again to the first system, and so on.

Let us consider first the case in which the two systems have the same monad, that is, they have the form $sys1 :: T1 \to X \to M\ X$ and $sys2 :: T2 \to X \to M\ X$. Then the combination we have described could be defined as:

$$comb\ (sys1, sys2)\ (t1, t2) = (sys2\ \ t2) \diamond (sys1\ \ t1)$$

but the result would not be, in general, a monadic system. Indeed, we have

$$\begin{aligned} & comb\ (sys1, sys2)\ (t1 + t1', t2 + t2') \\ = & \quad \{ \text{ definition of } comb \ \} \\ & (sys2\ \ (t2 + t2')) \diamond (sys1\ \ (t1 + t1')) \\ = & \quad \{ \ sys1 \text{ and } sys2 \text{ are systems } \} \\ & (sys2\ \ t2') \diamond (sys2\ \ t2) \diamond (sys1\ \ t1') \diamond (sys1\ \ t1) \end{aligned}$$

while

$$\begin{aligned} & (comb\ (sys1, sys2)\ (t1, t2)) \diamond (comb\ (sys1, sys2)\ (t1', t2')) \\ = & \quad \{ \text{ definition of } comb \ \} \\ & (sys2\ \ t2') \diamond (sys1\ \ t1') \diamond (sys2\ \ t2) \diamond (sys1\ \ t1) \end{aligned}$$

and, since $\diamond$ is not commutative, the two are in general different.

The difficulty comes from the fact that we do not want to consider the product monoid $(T1, T2)$. Rather, the pairs $(t1, t2)$ play the role of the *inputs* in Section 6.1, and the system resulting from the combination has as monoid $([(T1, T2)], +\!\!+, [\,])$:

$$\begin{aligned} & sysser\ (sys1, sys2) = inpsys\ f \\ & \quad \textbf{where} \\ & \quad f\ (x, (t1, t2)) = ((sys2\ \ t2) \diamond (sys1\ \ t1))\ x \end{aligned}$$

For example, consider combining with the system *app next2* defined in the previous section the system with input *inpsys choose* where

$$\begin{aligned} & choose :: && (Char, Char) \to [Char] \\ & choose\ (c1, c2) = && [c1, c2] \end{aligned}$$

(The idea is that a non-deterministic choice is made between continuing with the initial state or with the input.)

We have

```
sysser (app next2, inpsys choose) [(1, "f")] 'a'
    ==> "bfcf"
```

Starting in `'a'`, the list of reachable states after the application of *app next2* 1 is `"bc"`. Then *inpsys choose* `"f"` leads to a choice between any of `"bc"` and `'f'`. Since we use lists instead of sets, the `'f'` appears twice: we can obtain a leaner presentation by removing the duplicates by using the function *nub* provided by Haskell.

The trajectories of the combined system are computed as usual:

```
map nub (trj (sysser (app next2, inpsys choose)) 'a'
             [[(1, "g")], [(1, "f")]])
    ==> ["cgdhe","bfc","a"]

nub (mtrj (sysser (app next2, inpsys choose)) 'a'
           [[(1, "g")], [(1, "f")]])
    ==> ["cba","gba","dba","gfa","hfa","dca","gca","eca"]
```

We now consider the case in which the systems have different monads, that is, they have types $sys1 :: T1 \to X \to M\ X$ and $sys2 :: T2 \to X \to N\ X$. The idea is to find some "common" monad K to which both M and N can be "promoted", and use *sysser* with the transformed systems which are now $sys1' :: T1 \to X \to K\ X$ and $sys2' :: T2 \to X \to K\ X$. There is, as we have already said, no unique way of finding such a K. What we shall do in the rest of this section is to explore a number of typical patterns of monadic combinations, which often arise in practical problems.

Deterministic systems

As might be expected, the simplest case is that in which one of the two systems is deterministic, that is, when either M or N is the identity functor, Id. A deterministic system can be combined with any other system by simply turining it into a system of the appropriate type using *return*:

```
$detserl\ (detsys, sys) = sysser\ (detsys', sys)$
  **where**
  $detsys'\ t = return \cdot unwrapId \cdot (detsys\ t)$
$detserr\ (sys, detsys) = sysser\ (sys, detsys')$
  **where**
  $detsys'\ t = return \cdot unwrapId \cdot (detsys\ t)$
```

For example, consider the discrete dynamical system *inpsys lower* given by the transition function (with input):

```
$lower\qquad\quad :: (Char, Char) \to Id\ Char$
$lower\ (c1, c2) = Id\ (toLower\ c2)$
```

We can now combine *inpsys lower* with *app next2* by using either of the *detser* functions.

```
detserl (inpsys lower, app next2) [("C", 1)] 'a'
    ==> "de"

map nub (trj (detserl (inpsys lower, app next2)) 'a'
              [[("E", 1)], [("C", 1)]])
    ==> ["fg","de","a"]

mtrj (detserl (inpsys lower, app next2)) 'a'
              [[("E", 1)], [("C", 1)]]
    ==> ["fda","gda","fea","gea"]
```

But, just because one can always turn a deterministic system into a system with a different monad, does not mean that this is always the desired mode of combination. One could, instead, transform the other system in a deterministic one. However, there is in general no "canonical" way of doing that: each case must be treated on its own.

For our example we might wish to transform the *app next2* system in a deterministic one by taking the first element of the list of results.

$$detnext\ t\ c = Id\ (head\ (app\ next2\ t\ c))$$

We can combine the resulting system with *inpsys lower* using *sysser* directly:

```
sysser (inpsys lower, detnext) [("C", 1)] 'a'
    ==> 'd'

trj (sysser (inpsys lower, detnext)) 'a'
      [[("E", 1)], [("C", 1)]]
    ==> ['f','d','a']

mtrj (sysser (inpsys lower, detnext)) 'a'
       [[("E", 1)], [("C", 1)]]
    ==> "fda"
```

This kind of transformation is often used in combining stochastic systems with deterministic ones, usually using the "expected value" operator to go from *SimpleProb a* to *a*.

Non-deterministic systems

When combining non-deterministic systems with other types of systems one often takes advantage of simple ways of transforming a set into an instance of a more complex structure. For example, a finite set can be turned into a simple probability distribution by taking the uniform distribution over the set, and into a simple fuzzy set by assigning to all members of the set the membership degree 1.0. Conversely, a simple probability or a simple fuzzy set can be turned into a set by using *the forgetful functor* which discards the probability or membership information and returns the support of the simple probability distribution or of the fuzzy set. In all these cases, the monad K is, as in the case of combinations with a deterministic system, one of the initial monads of the systems involved.

An alternative, familiar to functional programmers, is to combine the monads M and $[\,]$ to yield a monad $M\ [\,]$, by using the *list transformer*, an element of the Haskell monad transformers refered to above in Section 6.2.1.

The list transformer is defined in the Haskell standard libraries as follows (slightly rewritten for readability). This definition is based on the proposal by Jones and Duponcheel in [22].

```
newtype ListT m a = ListT (m [a])

unwrapListT                :: ListT m a → m [a]
unwrapListT (ListT mxs) = mxs

instance (Monad m) ⇒ Monad (ListT m) where
  return           = ListT · return · return
  (ListT mas) ▷ f = ListT mbs
                    where
                    mlmbs = fmap (map (unwrapListT · f)) mas   -- :: m [m [b]]
                    mmbss = fmap sequence mlmbs   -- :: m (m [[b]])
                    mmbs = fmap (fmap concat) mmbss   -- :: m (m [b])
                    mbs   = join mmbs   -- :: m [b]
```

The function *sequence* is the decisive component in this definition: everything else could be generalized to any other monads. This function is part of the standard Haskell prelude, and is defined as:

```
sequence             :: Monad m ⇒ [m a] → m [a]
sequence []          = return []
sequence (m : ms) = m ▷ (λa →
                        (sequence ms ▷ (λas →
                        return (a : as))))
```

sequence takes a list of m-structures and forms an m-structure of lists. In terms of trajectories, it constructs all possible micro-trajectories from a given macro-trajectory.

As an example, let us combine the stochastic system *inpsys sadder* defined above in Section 6.1 with the following non-deterministic system:

```
simpleND :: ℕ → ℝ → [ℝ]
simpleND = app (λx → [−5.0, 5.0])
```

We now have a system *inpsys sadder* of type $[\mathit{Char}] \to \mathbb{R} \to \mathit{SimpleProb}\ \mathbb{R}$ and another one, *simpleND*, of type $\mathbb{N} \to \mathbb{R} \to [\mathbb{R}]$. We want to unify their types, and have two systems with the *ListT SimpleProb* monad:

$$
\begin{aligned}
&\mathit{sys1} \quad :: [\mathit{Char}] \to \mathbb{R} \to \mathit{ListT}\ \mathit{SimpleProb}\ \mathbb{R} \\
&\mathit{sys1}\ \mathit{cs} = \mathit{ListT} \cdot \mathit{fmap}\ \mathit{return} \cdot \mathit{inpsys}\ \mathit{sadder}\ \mathit{cs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sys2} \quad :: \mathbb{N} \to \mathbb{R} \to \mathit{ListT}\ \mathit{SimpleProb}\ \mathbb{R} \\
&\mathit{sys2}\ n = \mathit{ListT} \cdot \mathit{return} \cdot \mathit{simpleND}\ n
\end{aligned}
$$

```
sysser (sys2, sys1) [(1, "C")] 0.0
    ==> SP [([61.0,71.0],0.16),([61.0,73.0],0.24),
            ([63.0,71.0],0.24),([63.0,73.0],0.36)]
```

As can be seen, the combined system operates in the following way: the initial state is transformed by the non-deterministic system in a list of possible next states. This list is then turned into a concentrated probability distribution. The stochastic system is then applied to every element in this list: the result is a concentrated probability distribution over lists of simple probability distributions. In order to turn this into a simple probability distribution over lists, the function *sequence* is then applied. The list within the concentrated probability distribution is turned into a simple probability distribution over lists as follows: a list is obtained by picking an element from the support of each simple probability distribution in the list: the list then is assigned the probability resulting from the multiplication of the probabilities of its elements (as given by their original probability distributions).

```
trj (sysser (sys2, sys1)) 0.0 [[(1, "E")], [(1, "C")]]
    ==> [SP [([63.0,73.0],0.0256),([63.0,73.0,75.0],0.07680001),
         ([63.0,65.0,73.0],0.07680001),([63.0,65.0,73.0,75.0],0.2304),
         ([63.0,75.0],0.05760001),([63.0,65.0,75.0],0.1728),
         ([65.0,73.0],0.05760001),([65.0,73.0,75.0],0.1728),
         ([65.0,75.0],0.1296)],
        SP [([61.0,71.0],0.16),([61.0,73.0],0.24),
         ([63.0,71.0],0.24),([63.0,73.0],0.36)],
        SP [([0.0],1.0)]]

mtrj (sysser (sys2, sys1)) 0.0 [[(1, "C")]]
    ==> SP [([[61.0,0.0],[71.0,0.0]],0.16),([[61.0,0.0],[73.0,0.0]],0.24),
            ([[63.0,0.0],[71.0,0.0]],0.24),([[63.0,0.0],[73.0,0.0]],0.36)]
```

The computation of micro-trajectories results in a simple probability distribution over lists of trajectories. The output above shows that, for example, starting from 0.0 we can have the non-deterministic evolution given by the trajectories $[[63.0, 0.0], [73.0, 0.0]]$ with probability 0.36. Because we assign probabilities to non-deterministic evolutions, instead of to individual trajectories, there is a certain amount of duplication involved: for example, the trajectory $[73.0, 0.0]$ appears twice: in the second possible non-deterministic evolution, and in the last.

If what is of interest is just the trajectories and the probabilities with which they appear, and not the context of non-deterministic evolution in which they appear, then we can obtain this by the much simpler transformation of the non-deterministic into a stochastic one, which we described in the beginning of the section:

$$
\begin{aligned}
&\mathit{sys2'} :: \mathbb{N} \rightarrow \mathbb{R} \rightarrow \mathit{SimpleProb}\ \mathbb{R}\\
&\mathit{sys2'}\ n\ r = \mathit{SP}\ [(x, p) \mid x \leftarrow \mathit{xs}]\\
&\qquad\qquad \mathbf{where}\\
&\qquad\qquad \mathit{xs} = \mathit{simpleND}\ n\ r\\
&\qquad\qquad p = 1.0\ /\ (\mathit{realToFrac}\ (\mathit{length}\ \mathit{xs}))
\end{aligned}
$$

```
mtrj (sysser (sys2', inpsys sadder)) 0.0 [[(1, "C")]]
    ==> SP [([61.0,0.0],0.2),([63.0,0.0],0.3),
           ([71.0,0.0],0.2),([73.0,0.0],0.3)]
```

The problem is that in practice we often find the need for a different kind of combination: we would like to obtain not $M\ [A]$, for example probabilities over possible states, but rather $[M\ A]$, possible probabilities over states. This is the case whenever we combine a system with some M monad with a non-deterministic system representing possible *scenarios*: we want to obtain M-type evolutions for each possible scenario. As an example, the IPCC explicitely states that the SRES scenarios used in evaluating possible future evolutions of the global socio-ecological system are *not* stochastic: they are independent possibilities which should be explored independently, and taking, say, the expected value of a variable along evolutions induced by the scenarios is not a meaningful operation.

Some scenarios, such as the SRES ones, are actually a collection of deterministic systems which are not meant to be "mixed". One should not evolve a system with a scenario, and then switch to another scenario after that. In this sense, these scenarios represent a family of deterministic systems, rather than a non-deterministic system.

Let $\mathit{sysfam} :: I \rightarrow \mathit{T1} \rightarrow X \rightarrow \mathit{Id}\ X$ be a family of scenarios indexed by I. Then, combining $\mathit{sysfam}\ i$ with a monadic system $\mathit{sys} :: \mathit{T2} \rightarrow X \rightarrow M\ X$ is done using *detserl* or *detserr* defined in the section on deterministic systems. We obtain a family of combined systems:

$$\mathit{combfaml}\ (\mathit{sysfam}, \mathit{sys})\ i = \mathit{detserl}\ ((\mathit{sysfam}\ i), \mathit{sys})$$

or

$$\mathit{combfamr}\ (\mathit{sys}, \mathit{sysfam})\ i = \mathit{detserr}\ (\mathit{sys}, (\mathit{sysfam}\ i))$$

depending on which is used: *detserl* or *detserr*. Finally, we construct the parallel combination of the systems of the family using *parfam*:

$$\mathit{sysserfaml}\ (\mathit{sysfam}, \mathit{sys}) = \mathit{parfam}\ (\mathit{combfaml}\ (\mathit{sysfam}, \mathit{sys}))$$
$$\mathit{sysserfamr}\ (\mathit{sys}, \mathit{sysfam}) = \mathit{parfam}\ (\mathit{combfamr}\ (\mathit{sys}, \mathit{sysfam}))$$

A non-deterministic process is like a collection of scenarios which may be "mixed". Every entire relation $r :: A \rightarrow [B]$ (that is, $r\ a \not\equiv [\,]$ for all a), can be represented as a set of functions

$$\mathit{funs}\ r = \{f \mid f :: A \rightarrow B, f\ a \in r\ a\}$$

Uncurrying a non-deterministic system $\mathit{nondet} :: T \rightarrow X \rightarrow [X]$ we have a relation of type $(T, X) \rightarrow [X]$. The set of functions corresponding to this relation is then

$$\mathit{funs}\ (\mathit{uncurry}\ \mathit{nondet}) = \{f \mid f :: (T, X) \rightarrow X,$$
$$f\ (t, x) \in \mathit{nondet}\ t\ x\}$$

Finally, uncurrying the functions in this set, we have

$$\mathit{funs}'\ \mathit{nondet} = \mathit{fmap}\ (\mathit{uncurry})\ (\mathit{funs}\ (\mathit{uncurry}\ \mathit{nondet}))$$
$$= \{f \mid f :: T \rightarrow X \rightarrow X,$$
$$f\ t\ x \in \mathit{nondet}\ t\ x\}$$

funs′ decomposes a non-deterministic system into a the set of all deterministic systems compatible with it. This statement can be formalized as follows:

Proposition 5 *Let $f :: T \rightarrow X \rightarrow X$. Then*

1. $f \in \mathit{funs}'\ \mathit{nondet} \equiv \forall\ ts :: [T], x :: X : \mathit{mtrj}\ f\ x\ ts \in \mathit{mtrj}\ \mathit{nondet}\ x\ ts$

2. $f \in \mathit{funs}'\ \mathit{nondet} \equiv \forall\ ts{::}[T], x{::}X{:}\mathit{head}\ (\mathit{trj}\ f\ x\ ts) \in \mathit{head}\ (\mathit{trj}\ \mathit{nondet}\ x\ ts)$

Proof.

We prove the first equivalence. For the implication $\Leftarrow$ consider an arbitrary $t :: T$ and $x :: X$. We have that

$$
\begin{aligned}
& \mathit{mtrj}\ f\ x\ [t] \in \mathit{mtrj}\ \mathit{nondet}\ x\ [t] \\
\equiv\ & \{ \text{ definition of } \mathit{mtrj} \ \} \\
& (\mathit{addHist}\ (f\ t)) \lhd (\mathit{mtrj}\ f\ x\ [\,]) \in (\mathit{addHist}\ (\mathit{nondet}\ t)) \lhd \\
& (\mathit{mtrj}\ \mathit{nondet}\ x\ [\,]) \\
\equiv\ & \{ \text{ definition of } \mathit{mtrj}\text{, monad laws } \} \\
& \mathit{addHist}\ (f\ t)\ [x] \in \mathit{addHist}\ (\mathit{nondet}\ t)\ [[x]] \\
\equiv\ & \{ \text{ definition of } \mathit{addHist} \ \} \\
& \mathit{fmap}\ (:[x])\ (f\ t\ x) \in \mathit{fmap}\ (:[[x]])\ (\mathit{nondet}\ t\ x) \\
\equiv\ & \{ \ \mathit{fmap} \text{ for } \mathit{Id} \text{ and } [\,] \ \} \\
& [f\ t\ x, x] \in [[y, x] \mid y \leftarrow \mathit{nondet}\ t\ x] \\
\equiv\ & \{ \text{ `elem`, lists } \} \\
& \exists\ y \in \mathit{nondet}\ t\ x : f\ t\ x = y \\
\equiv\ & \{ \text{ simplify } \} \\
& f\ t\ x \in \mathit{nondet}\ t\ x
\end{aligned}
$$

Since t and x were arbitrary, we have that $f \in \mathit{funs}'\ \mathit{nondet}$.

In the other direction, $\Rightarrow$, consider an arbitrary $x :: X$. We are going to use induction on ts:

Case $ts = [\,]$:

$$
\begin{aligned}
& \mathit{mtrj}\ f\ x\ [\,] \\
=\ & \{ \text{ definition of } \mathit{mtrj}, \} \\
& [x]
\end{aligned}
$$

and

$$
\begin{aligned}
& \mathit{mtrj}\ \mathit{nondet}\ x\ [\,] \\
=\ & \{ \text{ definition of } \mathit{mtrj} \ \} \\
& [[x]]
\end{aligned}
$$

Therefore, $\mathit{mtrj}\ f\ x\ [\,] \in \mathit{mtrj}\ \mathit{nondet}\ x\ [\,]$.

Case $(t : ts)$

We have

$$
\begin{aligned}
& \mathit{mtrj}\ f\ x\ (t : ts) \\
=\ & \{ \text{ definition of } \mathit{mtrj} \ \} \\
& (\mathit{addHist}\ (f\ t)) \lhd (\mathit{mtrj}\ f\ x\ ts) \\
=\ & \{ \text{ definition of } \lhd \text{ for } \mathit{Id} \ \} \\
& \mathit{addHist}\ (f\ t)\ (\mathit{mtrj}\ f\ x\ ts) \\
=\ & \{ \text{ definition of } \mathit{addHist} \ \} \\
& \mathit{fmap}\ (:(\mathit{mtrj}\ f\ x\ ts))\ (f\ t\ (\mathit{head}\ (\mathit{mtrj}\ f\ x\ ts))) \\
=\ & \{ \ \mathit{fmap} \text{ for } \mathit{Id} \ \} \\
& (f\ t\ (\mathit{head}\ (\mathit{mtrj}\ f\ x\ ts))) : (\mathit{mtrj}\ f\ x\ ts)
\end{aligned}
$$

and

$$\begin{array}{ll}
 & \mathit{mtrj}\ \mathit{nondet}\ x\ (t : \mathit{ts}) \\
= & \{ \text{ definition of } \mathit{mtrj} \ \} \\
 & (\mathit{addHist}\ (\mathit{nondet}\ t)) \lhd (\mathit{mtrj}\ \mathit{nondet}\ x\ \mathit{ts}) \\
= & \{ \text{ definition of } \lhd \text{ for } [\,] \ \} \\
 & \mathit{concat}\ [\mathit{addHist}\ (\mathit{nondet}\ t)\ \mathit{xs} \mid \mathit{xs} \leftarrow \mathit{mtrj}\ \mathit{nondet}\ x\ \mathit{ts}] \\
= & \{ \text{ definition of } \mathit{addHist} \ \} \\
 & \mathit{concat}\ [\mathit{fmap}\ (:\mathit{xs})\ (\mathit{nondet}\ t\ (\mathit{head}\ \mathit{xs})) \mid \\
 & \quad \mathit{xs} \leftarrow \mathit{mtrj}\ \mathit{nondet}\ x\ \mathit{ts}] \\
= & \{ \text{ definition of } \mathit{fmap} \text{ for } [\,] \ \} \\
 & \mathit{concat}\ [[y : \mathit{xs} \mid y \leftarrow \mathit{nondet}\ t\ (\mathit{head}\ \mathit{xs})] \mid \\
 & \quad \mathit{xs} \leftarrow \mathit{mtrj}\ \mathit{nondet}\ x\ \mathit{ts}]
\end{array}$$

But, by the induction hypothesis, we have that $\mathit{mtrj}\ f\ x\ \mathit{ts} \in \mathit{mtrj}\ \mathit{nondet}\ x\ \mathit{ts}$ and, since $f \in \mathit{funs}'\ \mathit{nondet}$ we have that $f\ t\ (\mathit{head}\ (\mathit{mtrj}\ f\ x\ \mathit{ts})) \in \mathit{nondet}\ t\ (\mathit{head}\ (\mathit{mtrj}\ f\ x\ \mathit{ts}))$. Therefore, we find $\mathit{mtrj}\ f\ x\ \mathit{ts}$ among the xs and $f\ t\ (\mathit{head}\ (\mathit{mtrj}\ f\ x\ \mathit{ts}))$ among the y values that make up $\mathit{mtrj}\ \mathit{nondet}\ x\ (t{:}\mathit{ts})$, therefore we have that $\mathit{mtrj}\ f\ x\ (t{:}\mathit{ts}) \in \mathit{mtrj}\ \mathit{nondet}\ x\ (t{:}\mathit{ts})$.

The proof of the second equivalence is similar: $\Rightarrow$ by induction, $\Leftarrow$ by specializing the right-hand side with arbitrary x and $[t]$.

□

The significance of this proposition is that we can study the combination of a monadic system with a non-deterministic system by replacing the latter with the family of deterministic systems given by funs' (we can simply take $\mathit{funs}'\ \mathit{nondet}$ to be the indexing set, and $\mathit{sysfam}\ i\ t\ x = i\ t\ x$), using one of the functions *sysserfam* defined above.

Of course, this is not a useful *computational* characterization: in general, the number of elements of $\mathit{funs}'\ \mathit{nondet}$ is going to be uncountable. Consider, for example, the case of $\mathit{nondet} :: \mathbb{N} \to \mathbb{N} \to \mathit{Bool}, \mathit{nondet} = \mathit{app}\ (\lambda n \to [\mathit{True}, \mathit{False}])$. The transfer function of *nondet* is the total relation $\mathbb{N} \times \mathit{Bool}$: any function $f :: \mathbb{N} \to \mathbb{N} \to \mathit{Bool}$ is a member of $\mathit{funs}'\ \mathit{nondet}$. Thus, $\mathit{funs}'\ \mathit{nondet}$ has the same cardinality as $\mathbb{R}$.

Nevertheless, this characterization can be a useful conceptual tool. We can use it to give precise specifications of problems, and as starting point for the derivation of solutions.

For example, consider the following simple control problem. We start with a discrete "runaway" stochastic process:

```
run (r, c) = SP [(−2 ∗ x − 0.5, 0.5), (2 ∗ x + 0.5, 0.5)]
  where
  x = r + c
```

The process, started in the initial state 0.0, tends to "run away" from this initial state, faster and faster, if the control variable c is not suitably chosen.

In turn, the control variable can be chosen to be either 0.0, or 3.0 or −3.0. The choice of 0.0 is always available, of the other two, only the one with the opposite sign to the state of the process may be chosen:

$$\mathit{choice}\ r = \mathbf{if}\ r > 0\ \mathbf{then}\ [-3.0, 0.0]\ \mathbf{else}\ [0.0, 3.0]$$

The control problem is, starting with 0.0, to choose controls in such a way that the state of the system is maintained, for a given number of steps, say 10, within a given interval, for example $[-5.0, 5.0]$ with at least 0.5 probability, while minimizing the number of control choices different from 0.0

We can formalize the problem by considering a system that gives us all possible trajectories of the stochastic process depending on the control choices made.

First, let us examine more closely the two processes involved. The first can be written as a stochastic system

$$\begin{aligned}
&\mathit{runsys} :: \mathbb{N} \to (\mathbb{R}, \mathbb{R}) \to \mathit{SimpleProb}\ (\mathbb{R}, \mathbb{R})\\
&\mathit{runsys} = \mathit{app}\ \mathit{run}'\\
&\quad \mathbf{where}\\
&\quad \mathit{run}'\ (r, c) = \mathit{fmap}\ (\lambda x \to (x, c))\ (\mathit{run}\ (r, c))
\end{aligned}$$

The choice of the control can be expressed as a non-deterministic system:

$$\begin{aligned}
&\mathit{ctrlsys} :: \mathbb{N} \to (\mathbb{R}, \mathbb{R}) \to [(\mathbb{R}, \mathbb{R})]\\
&\mathit{ctrlsys} = \mathit{app}\ \mathit{ctrl}'\\
&\quad \mathbf{where}\\
&\quad \mathit{ctrl}'\ (r, c) = \mathit{map}\ (\lambda c' \to (r, c'))\ (\mathit{choice}\ r)
\end{aligned}$$

A "control choice", a potential solution to the problem, can be seen as a deterministic system:

$$\mathit{ctrl} :: \mathbb{N} \to (\mathbb{R}, \mathbb{R}) \to \mathit{Id}\ (\mathbb{R}, \mathbb{R})$$

such that

$$\mathit{ctrl}\ t\ (r, c) \in \mathit{choice}\ t\ (r, c)$$

that is, one that always chooses an allowed element.

The set of all potential solutions is then *funs′ ctrlsys*. One such solution can be combined with *runsys* in order to give a probability distribution over possible trajectories:

$$\mathit{onemtrj} = \mathit{mtrj}\ (\mathit{detserl}\ (\mathit{onesol}, \mathit{runsys}))\ 0.0\ (\mathit{replicate}\ 10\ [(1, 1)])$$

The function *replicate* creates a list of as many copies of the second argument as specified by the first argument. Here, we want to make 10 steps starting form 0.0. We advance both systems equally at each step, and the trajectories include the states reached at each step.

In order to evaluate such a probability distribution over trajectories, we need to determine the probability that the trajectory stays within the given bounds, and to count the number of control choices different from 0.0. It is not clear from the problem information whether only the choices along successful trajectories count, or all should be added up: we choose the second alternative.

$$\begin{aligned}
&\mathit{within\_bounds} :: [(\mathbb{R},\mathbb{R})] \to \mathit{Bool}\\
&\mathit{within\_bounds} = \mathit{null} \cdot \mathit{filter}\ \mathit{bad}\\
&\qquad\qquad\qquad\quad \textbf{where}\\
&\qquad\qquad\qquad\quad \mathit{bad}\ (r, c) = \mathit{abs}\ r > 5.0
\end{aligned}$$

$$\begin{aligned}
&\mathit{cost} :: [(\mathbb{R},\mathbb{R})] \to \mathbb{N}\\
&\mathit{cost} = \mathit{length} \cdot \mathit{filter}\ \mathit{expensive}\\
&\qquad \textbf{where}\\
&\qquad \mathit{expensive}\ (r, c) = c \not\equiv 0.0
\end{aligned}$$

$$\begin{aligned}
&\mathit{prob\_qual} \qquad\quad :: \mathit{SimpleProb}\ (\mathit{Bool}, \mathbb{N}) \to (\mathbb{R}, \mathbb{N})\\
&\mathit{prob\_qual}\ (\mathit{SP}\ \mathit{bns}) = (\mathit{sum}\ \mathit{qs}, \mathit{sum}\ \mathit{cs})\\
&\qquad\qquad\qquad\qquad \textbf{where}\\
&\qquad\qquad\qquad\qquad \mathit{qs} = [\,q \mid ((b, n), q) \leftarrow \mathit{bns}, b\,]\\
&\qquad\qquad\qquad\qquad \mathit{cs} = [\,n \mid ((b, n), q) \leftarrow \mathit{bns}\,]
\end{aligned}$$

$$\mathit{eval} = \mathit{prob\_qual} \cdot \mathit{fmap}\ (\mathit{pair}\ (\mathit{within\_bounds}, \mathit{cost}))$$

The computation of all trajectories, for all potential solutions, is defined by combining *runsys* with the family of allowable systems *sysfam* by using *sysserfaml*. We do not want to add up the probability of identical trajectories coming from different control choices, which is why a combination based on turning *ctrlsys* into a stochastic system is inappropriate.

$$\begin{aligned}
&\mathit{sysfam} :: \mathit{funs}'\ \mathit{ctrlsys} \to \mathbb{N} \to (\mathbb{R},\mathbb{R}) \to \mathit{Id}\ (\mathbb{R},\mathbb{R})\\
&\mathit{sysfam}\ f\ n\ (r, c) = f\ n\ (r, c)
\end{aligned}$$

At this point we are no longer in the realm of legal Haskell. The indexing set *funs′ ctrlsys* is not a legal type for a function declaration, and moreover it is uncountable.

$$\begin{aligned}
&\mathit{alltrj} = \mathit{mtrj}\ (\mathit{sysserfaml}\ (\mathit{sysfam}, \mathit{runsys}))\ 0.0\\
&\quad (\mathit{replicate}\ 10\ [(1, 1)])
\end{aligned}$$

The type of *alltrj* is $ExpT\ (funs'\ ctrlsys)\ SimpleProb\ [(\mathbb{R},\mathbb{R})]$, isomorphic to $funs'\ ctrlsys \to SimpleProb\ [(\mathbb{R},\mathbb{R})]$

The quality of a particular solution $f \in funs'\ ctrlsys$ is given by

$$qual\ f = eval\ (unwrapExpT\ alltrj\ f)$$

A solution to the control problem is a deterministic system $sol \in funs'\ ctrlsys$ such that:

$$\begin{array}{l} fst\ (qual\ sol) > 0.5 \\ \forall\ f \in (funs'\ ctrlsys) : fst\ (qual\ f) > 0.5 \Rightarrow snd\ f > snd\ sol \end{array}$$

We can move towards a solution of this problem by implementing a subset of *sysfam*. For example, we can discretize the domain of *choice* and keep just an interval from it. Since any state outside $[-5.0, 5.0]$ is already "lost", we can just concentrate on a subinterval of $[-5.0, 5.0]$. For example:

$$dom = [-4.0\,..\,4.0]$$

On this small discrete domain, the relation *choice* can be fully decomposed in functions. The graphs of these functions can be represented as lists:

```
$funlists = foldr\ f\ [[\,]]\ dom$
  where
  $f\ r\ rcss = [(r, c) : rcs \mid rcs \leftarrow rcss,$
                         $c \leftarrow choice\ r]$
```

There are 512 functions in total. In order to combine these functions with *runsys*, we have to turn each of them into a deterministic dynamical system defined on $(\mathbb{R},\mathbb{R})$, which is a problem of interpolation. For example:

```
$mkfun\ xs\ (r, c)$
  $\mid r < -4.0$   $= Id\ (r, 3.0)$
  $\mid r > 4.0$    $= Id\ (r, -3.0)$
  $\mid r \equiv -4.0$ $= Id\ (xs\ !!\ 0)$
  $\mid otherwise$ $= Id\ (r, c)$
              where
              $Just\ i = findIndex\ (\lambda(x, y) \to r \leqslant x)\ xs$
              $(rl, cl) = xs\ !!\ (i - 1)$
              $(rr, cr) = xs\ !!\ i$
              $c = $ if $r - rl < rr - r$ then $cl$ else $cr$
```

Then, the functions in the family are defined by

$$\begin{array}{l} funs :: [\mathbb{N} \to (\mathbb{R},\mathbb{R}) \to Id\ (\mathbb{R},\mathbb{R})] \\ funs = map\ (app \cdot mkfun)\ funlists \end{array}$$

The family of systems is given by:

```
$sysfam ::$               $\mathbb{N} \rightarrow \mathbb{N} \rightarrow (\mathbb{R}, \mathbb{R}) \rightarrow Id\ (\mathbb{R}, \mathbb{R})$
$sysfam\ n\ t\ (r, c) = (funs\ !!\ n)\ t\ (r, c)$
```

We can now compute all the trajectories by using *sysserfaml*:

```
$alltrj = mtrj\ (sysserfaml\ (sysfam, runsys))\ (0.0, 0.0)$
  $(replicate\ 10\ [(1, 1)])$
```

Evaluating all trajectories, we obtain

```
$quals = [\,eval\ (unwrapExpT\ alltrj\ n) \mid n \leftarrow [0\,..\,(length\ funlists - 1)]\,]$
```

Finally, the graph of the best function is given by

```
$best = funlists\ !!\ (snd\ (head\ (sortBy\ f\ (zip\ (filter\ g\ quals)\ [0, 1\,..]))))$
  **where**
  $g\ (p, n) = p \equiv 1.0$
  $f\ ((p1, n1), i1)\ ((p2, n2), i2)$
    $\mid min\ p1\ p2 \geqslant 0.5 = compare\ n2\ n1$
    $\mid max\ p1\ p2 < 0.5 = compare\ p1\ p2$
    $\mid p1 < 0.5 \wedge p2 \geqslant 0.5 = LT$
    $\mid p1 \geqslant 0.5 \wedge p2 < 0.5 = GT$
```

In our case, we have

```
best = [(-4.0, 3.0), (-3.0, 3.0), (-2.0, 0.0), (-1.0, 0.0), (0.0, 0.0),
        (1.0, 0.0),  (2.0, 0.0), (3.0, -3.0), (4.0, -3.0)]
```

This keeps *runsys* always within the given bounds, and the total number of controls different from 0.0 is 2418. In fact, this is also the solution with the smallest number of controls different from 0.0, so that minimizing this "cost" automatically brings the system within the desired range. This is a feature of the way we interpolated the graphs of the functions in *funlists*, selecting non-zero controls for the case in which the state of the system is out of bounds.

Of course, computing *alltrj* is a very inefficient way of going about finding a solution to the control problem. Rather, one would try to find conditions under which the number of potential solutions can be decreased at every step of computing potential trajectories, leading to thinning algorithms, such as those presented, with illustrative examples, in [2], [9], [10].

6.2.3 Input-output connections

In the example of the previous section we started with a stochastic process given by the function $run :: (\mathbb{R}, \mathbb{R}) \rightarrow SimpleProb\ \mathbb{R}$ and the relation $choice :: (\mathbb{R}, \mathbb{R}) \rightarrow [\mathbb{R}]$. When viewing these as dynamical systems, we considered them as discrete systems $runsys :: \mathbb{N} \rightarrow (\mathbb{R}, \mathbb{R}) \rightarrow SimpleProb\ (\mathbb{R}, \mathbb{R})$ and

$ctrlsys :: \mathbb{N} \to (\mathbb{R}, \mathbb{R}) \to [(\mathbb{R}, \mathbb{R})]$ respectively. However, both *run* and *choice* have the appropriate form to be used with *inpsys*, the function defined above in Section 6.1. Using *inpsys*, we obtain

$$inpsys\ run :: [\mathbb{R}] \to \mathbb{R} \to SimpleProb\ \mathbb{R}$$
$$inpsys\ choice :: [\mathbb{R}] \to \mathbb{R} \to [\mathbb{R}]$$

Let us use R for the real numbers representing the states of *run* and C for those representing the states of *choice*:

$$inpsys\ run :: [C] \to R \to SimpleProb\ R$$
$$inpsys\ choice :: [R] \to C \to [C]$$

We could now pose the problem of combining such systems by noticing that the monoid of each system is related to the states of the other system. In fact, by rewriting *run* and *choice* to act on lists rather than elements by wrapping the states and the results, the state of each system would be *equal* to the monoid of the other system.

The general form of this combination is then: given

$$sys1 :: T1 \to T2 \to M\ T2$$
$$sys2 :: T2 \to T1 \to N\ T1$$

to combine them by using the output states of one system as elements of the monoid of the other system.

But, in fact, there is no need to introduce this special form of combination, as the above example shows. We can transform both systems in discrete systems which have the same state space, by:

$$sys1' :: \mathbb{N} \to (T1, T2) \to M\ (T1, T2)$$
$$sys1' = app\ f\ \textbf{where}\ f\ (t1, t2) = fmap\ (\lambda t \to (t, t2))\ (sys1\ t1\ t2)$$
$$sys2' :: \mathbb{N} \to (T1, T2) \to N\ (T1, T2)$$
$$sys2' = app\ g\ \textbf{where}\ g\ (t1, t2) = fmap\ (\lambda t \to (t1, t))\ (sys2\ t2\ t1)$$

Thus, the problem of combining *sys1* and *sys2* is reduced to finding a serial combination of *sys1′* and *sys2′*.

6.3 Conclusions

The main ideas of this chapter are expressed in the various constructors and combinators of monadic dynamical systems. The function *inpsys*, defined in the first section and used throughout, constructs a monadic dynamical system from a system "with input", classically presented as a function $f :: (X, A) \to X$. The combinators *parsys* and *parfam* construct monadic systems by putting in parallel two different systems or a family of identically typed systems, respectively. The combinator *sysser* allowed us to combine two monadic dynamical systems which interact via a common state, in the case in which they have the same monad. We have then discussed several solutions for the case in which the two systems do *not* have the same monad. Special attention was paid to the combination, frequently encountered in practice, of a stochastic system with a non-deterministic one, where a first failed attempt at representing systems with input came surprisingly handy. Finally, we have seen that the case of systems interacting via their monoids can be reduced to that of systems interacting via their common state.

All the systems resulting from these constructions are monadic systems, and their trajectories, both macro and micro, can be computed with *trj* and *mtrj*. In the following chapter, when we return to the context of vulnerability, we will interpret *mtrj* as the implementation of the function *possible* which was returning the structures of possible future evolutions. If *mtrj* were restricted, say, to computing the trajectories of stochastic systems, its use as *possible* would be questionable. Due to the flexibility of monadic systems, however, we can hope to use *mtrj* in all situations that arise in practice.

In the course of this chapter we have also seen a number of limitations or awkward effects of using monadic systems. In the case of systems with input, the first attempt of obtaining a monadic system failed, resulting instead in a representation of parameterized systems. Moreover, there is in general no guidance about how to combine systems with different monads. Sometimes, as in the case of combinations involving deterministic systems, the two monads are "promoted" to their composition, but this is by no means a universal solution. In the important case of combining non-deterministic systems serially with other systems, the common monad is a version of the parameterized systems monad.

Despite these flaws, we were always able to formulate a satisfactory, even if not always natural, monadic version of the systems involved. More experience with these combinators and others which might arise in practice will hopefully lead to a smoother presentation of these versions, or will reveal a simpler, better structure than that of the monad.

Chapter 7

Monadic Systems and Vulnerability

In this chapter, we return to the model of vulnerability developed in Chapter 4. We use monadic dynamical systems to specialize some of its components, in particular the function *possible* which was assumed to generate the future evolutions of the world. We discuss how to define measures of harm on a structure of possible evolutions which are compatible with other measures of harm defined on a different structure. We implement a small example and show how a number of conditions can be computationally tested using QuickCheck.

7.1 Micro-trajectories as possible evolutions

The model of vulnerability developed in Chapter 4, expressed as

$$\begin{aligned}
&\mathit{possible} :: \mathit{States} \to F\ \mathit{Evolutions}\\
&\mathit{harm} \quad :: \mathit{Evolutions} \to V\\
&\mathit{measure} :: F\ V \to W\\
&\mathit{vulnerability} :: \mathit{States} \to W\\
&\mathit{vulnerability} = \mathit{measure} \cdot \mathit{fmap}\ \mathit{harm} \cdot \mathit{possible}
\end{aligned}$$

where F is a functor, V and W are ordered sets, and *measure* satisfies the monotonicity condition, left open the question of how its components might be implemented. In this chapter, we propose the following answer: the function *possible* is computed as the structure of micro-trajectories of a monadic dynamical system.

This answer implies that the type of future evolutions is the same as that of micro-trajectories, that is, $[\mathit{States}]$. The states in question have been "narrowed down", from general states of the world, to states of the dynamical system involved in the computation. Additionally, we have the restriction that F is a monad.

Formally, we have:

```
newtype VulnerabilityContext m v w x t
            = VC (t → x → m x, [x] → v, m v → w)
system (VC (sys, h, m)) = sys
harm   (VC (sys, h, m)) = h
measure (VC (sys, h, m)) = m


vulnerability vc x ts = (measure vc) (fmap (harm vc)
                                           (mtrj (system vc) x ts))
```

We compute vulnerability in the context of a given monadic dynamical system, with given functions *harm* and *measure*. In such a context, vulnerability is computed in an initial state, along a given trajectory of inputs (which could be, for example, intervals of time).

This represents an implementation of the basic model of vulnerability. In the Sections 4.2 and 4.3, we have presented extensions involving the usage of sensitivity and adaptive capacity. We can implement these extensions by extending the context of vulnerability with the missing elements.

```
newtype SensitivityContext x v1 v2 v
   = SC ([x] → v1, [x] → v2, (v1, v2) → v)
impacts  (SC (i, s, c)) = i
sensitivity (SC (i, s, c)) = s
combine (SC (i, s, c)) = c


vulnerability_sensitivity sys meas sc
   = vulnerability (VC (sys, harm, meas))
     where harm = (combine sc) · pair (impacts sc, sensitivity sc)


newtype AdaptiveCapacityContext x w w′ w″ = AC (x → w′, (w, w′) → w″)
adaptive_capacity (AC (ac, adj)) = ac
adjust (AC (ac, adj)) = adj
vulnerability_adcap vc acc x ts
   = (adjust acc) ((vulnerability vc x ts), adaptive_capacity acc x)
```

7.2 Measures of harm on monadic structures

Computational vulnerability assessment is generally considered an exploratory activity: various ways of computing the possible evolutions of the systems involved are tried out, the models representing the systems are often changed, and different ways of assessing harm, impacts or sensitivity are used. One of the most important problems in all these changes is that of the type of the *measure* function, which is estimating a structure of possible harms, or impacts, along a trajectory of states. When the system giving us this trajectory changes, for example, when a deterministic system is replaced by a stochastic one, the *harm* function generally does not need to change: its type remains $[X] \to V$, but the *measure* function has to change: the new type is given by the monad of the new system. In our example, initially we had $\mathit{measure} :: \mathit{Id}\ V \to W$, and after the change $\mathit{measure} :: \mathit{SimpleProb}\ V \to W$.

It is interesting, especially when comparing the results of these explorations, to know whether the measures used are in some sense *compatible*. Further, it would be desirable if one could take advantage of an existing *measure* function in order to devise others which are compatible with it, but work on different structures.

The feasability of this depends on the precise sense of "compatible". Since we are assuming that W is a partially ordered set, we have that any vulnerability measure $m :: M\ V \to W$ induces a preorder on $M\ V$: $mv_1 \sqsubseteq_{M\ V} mv_2 \equiv m\ (mv_1) \sqsubseteq_W m\ (mv_2)$. In order to decide whether two measures with the same target defined on different structures of the same underlying set, $m_1 :: \mathit{M1}\ V \to W$ and $m_2 :: \mathit{M2}\ V \to W$ are compatible or not, we can examine the preorders they induce on their sources.

Let us give some examples.

1. If X is a set and M is a monad, a preorder $\sqsubseteq_X$ on X can be considered compatible with a preorder $\sqsubseteq_{M\ X}$ on $M\ X$ if

   $$\forall\, x_1, x_2 :: X : x_1 \ \sqsubseteq_X \ x_2 \equiv (\mathit{return}\ x_1) \ \sqsubseteq_{M\ X} \ (\mathit{return}\ x_2)$$

2. The preorder $\sqsubseteq_1$ on $[\mathbb{R}]$ given by

   $$\mathit{xs1} \sqsubseteq_1 \mathit{xs2} \equiv (\mathit{average}\ \mathit{xs1}) \leqslant (\mathit{average}\ \mathit{xs2})$$

   seems to be more compatible with the preorder $\sqsubseteq_2$ on $\mathit{SimpleProb}\ \mathbb{R}$ given by

   $$sp_1 \sqsubseteq_2 sp_2 \equiv (\mathit{expected}\ sp_1) \leqslant (\mathit{expected}\ sp_2)$$

   than with the preorder $\sqsubseteq_3$ on $\mathit{SimpleProb}\ \mathbb{R}$ given by

   $$sp_1 \sqsubseteq_3 sp_2 \equiv (\mathit{likeliest}\ sp_1) \leqslant (\mathit{likeliest}\ sp_2)$$

   where *likeliest sp* chooses the greatest real number assigned the greatest probability (ties between equally likely elements are broken by using *max*)

3. It seems reasonable to consider that the preorder $\sqsubseteq_3$ above is compatible with the preorder $\sqsubseteq_4$ on $\mathit{SimpleFuzz}\ \mathbb{R}$ given by

$$sf_1 \sqsubseteq_4 sf_2 \equiv (\mathit{highestDeg}\ sf_1) \leqslant (\mathit{highestDeg}\ sf_2)$$

where *highestDeg sf* returns the real number with the greatest degree of membership to *sf* and ties are broken, as with $\sqsubseteq_4$, by using *max*.

On the other hand, $\sqsubseteq_3$ does not seem intuitively to be compatible with $\sqsubseteq_5$ where

$$sf_1 \sqsubseteq_5 sf_2 \equiv (\mathit{cog}\ sf_1) \leqslant (\mathit{cog}\ sf_2)$$

where *cog sf* computes the "center of gravity" of the fuzzy set *sf* (similar to taking the expected value in *SimpleProb* $\mathbb{R}$).

The reason the choices made in these examples between compatibility and non-compatibility seem so obvious is probably that in each case, although we have not made it explicit, we have a transformation between two different types of structure. The transformation is natural: in programming terms, the functions it defines are polymorphic, and injective: all functions in the transformation have left inverses. In each case, the preorders are judged compatible when *both* the transformation between the structures and its left inverses, restricted to the image of the transformation, are monotonous, and otherwise they are classified as incompatible. In other words, we can take the following as definitions of "compatibility".

Definition 17 *Compatible preorders.*
Let M and N be functors and $\tau :: M\ a \rightarrow N\ a$ be an injective natural transformation. Two preorders $\sqsubseteq_M$ on $M\ X$ and $\sqsubseteq_N$ on $N\ X$ are compatible with respect to τ *if $M\ X$ and the image of $M\ X$ under τ are order isomorphic, that is*

$$mx_1 \sqsubseteq_M mx_2 \equiv \tau\ (mx_1) \sqsubseteq_N \tau\ (mx_2)$$

Definition 18 *Compatible vulnerability measures.*
Two vulnerability measures $m_1 :: M1\ V \rightarrow W$ and $m_2 :: M2\ V \rightarrow W$ are compatible with respect to an injective natural transformation $\tau :: M1\ a \rightarrow M2\ a$ if the preorders induced by them are compatible with respect to τ.

We can now state the main result about "translating" vulnerability measures from one type of structure to another.

Theorem 9 *Translating vulnerability measures.*
Let $\tau :: M\ a \rightarrow N\ a$ be an injective natural transformation, and let $\tau^{-1} :: N\ a \rightarrow M\ a$ be a left inverse of τ which is also a natural transformation. Then:

1. *If $m_M :: M\ V \rightarrow W$ is a vulnerability measure, then $m_M \cdot \tau^{-1} :: N\ V \rightarrow W$ is a vulnerability measure compatible with it with respect to τ.*

2. *If $m_N :: N\ V \to W$ is a vulnerability measure, then $m_N \cdot \tau :: M\ V \to W$ is a vulnerability measure compatible with it with respect to τ.*

Proof.

1. Let $inc :: V \to V$ be an increasing function, and $nv :: N\ V$. We have

$$\begin{array}{ll} & m_M\ (\tau^{-1}\ (N\ inc\ nv)) \\ = & \{\ \text{naturallity of } \tau^{-1}\ \} \\ & m_M\ (M\ inc\ (\tau^{-1}\ nv)) \\ \sqsubseteq & \{\ m_M \text{ vulnerability measure }\} \\ & m_M\ (\tau^{-1}\ nv) \end{array}$$

and therefore, $m_M \cdot \tau^{-1}$ is a vulnerability measure.

Remark. Since only the naturality of τ^{-1} has been used, we have the more general result that the composition of a vulnerability measure with a natural transformation is a vulnerability measure.

Let $mv_1, mv_2 :: M\ V$. We have

$$\begin{array}{ll} & m_M\ mv_1 \sqsubseteq m_M\ mv_2 \\ \equiv & \{\ \tau^{-1} \cdot \tau = id\ \} \\ & m_M\ (\tau^{-1}\ (\tau\ mv_1)) \sqsubseteq m_M\ (\tau^{-1}\ (\tau\ mv_2)) \\ \equiv & \{\ \text{function composition }\} \\ & (m_M \cdot \tau^{-1})\ (\tau\ mv_1) \sqsubseteq (m_M \cdot \tau^{-1})\ (\tau\ mv_2) \end{array}$$

therefore, m_M and $m_M \cdot \tau^{-1}$ are compatible with respect to τ.

2. $m_N \cdot \tau$ is a vulnerability measure, as shown above.

Let $mv_1, mv_2 :: M\ V$. We have

$$\begin{array}{ll} & (m_N \cdot \tau)\ mv_1 \sqsubseteq (m_N \cdot \tau)\ mv_2 \\ \equiv & \{\ \text{composition }\} \\ & m_N\ (\tau\ mv_1) \sqsubseteq m_N\ (\tau\ mv_2) \end{array}$$

which shows that $m_N \cdot \tau$ and m_N are compatible with respect to τ.

□

The "compatibility" between the examples with which we started and the formal definitions above is summarized in the following proposition.

Proposition 6 *Examples of compatibility.*

With the notations in the examples above, we have:

1. *$\sqsubseteq_X$ and $\sqsubseteq_{M\ X}$ are compatible with respect to return if return is injective. They are compatible with respect to an injective inverse of return, if one exists.*

2. *average and expected are vulnerability measures, compatible with the following natural transformation:*

   $$l1ToSP :: List1\ x \rightarrow SimpleProb\ x$$

   $$l1ToSP\ xs = SP\ [(x,p) \mid x \leftarrow xs]\ \textbf{where}\ p = 1.0\ /\ realToFrac\ (length\ xs)$$

   The preorder $\sqsubseteq_3$, induced by likeliest which is not *a vulnerability measure, is not compatible with the preorder induced by average, $\sqsubseteq_1$.*

3. *$\sqsubseteq_3$ and $\sqsubseteq_4$ are compatible with respect to the natural transformation*

   $$spToSF :: SimpleProb\ x \rightarrow SimpleFuzzy\ x$$
   $$spToSF\ (SP\ xs) = SF\ xs$$

   The preorder $\sqsubseteq_5$ induced by cog, which is *a vulnerability measure, is not compatible with $\sqsubseteq_3$ with respect to spToSF, but is compatible with $\sqsubseteq_2$.*

The proofs are just routine verification of the definitions involved.

7.3 Computational testing using QuickCheck

7.3.1 Introducing QuickCheck

QuickCheck is a Haskell module with functionality which allows the user to express, in a language somewhat resembling first-order logic, properties of Haskell functions, to generate data required to test these properties, and finally to test these properties using the generated data.

QuickCheck is presented in the online manual [18] and in a number of publications ([5], [6], [7] and others). Here, we briefly present the components we will use in the sequel.

The most important of these is the function *quickCheck* which verifies testable properties. The simplest testable properties are boolean valued functions, for example:

$$prop\_addition\_associative\ x\ y\ z = (x + y) + z = x + (y + z)$$

The names of such functions start, by convention, with *prop_*. Arguments of *prop_* functions are universally quantified, so *prop_Something* $x\ y\ z$ = *some_property* is read: *for all* x, y and z, the property *some_property* holds.

The function *quickCheck* uses data generators in order to test the property for random values of the arguments. One cannot generate in Haskell arbitrary values for polymorphic properties such as *prop_addition_associative*, therefore a specific type signature has to be provided:

```
quickCheck (prop_addition_associative::Nat -> Nat -> Nat -> Bool)
    ==> OK, passed 100 tests.

quickCheck (prop_addition_associative::Double -> Double
                                        -> Double -> Bool)
    ==> Falsifiable, after 1 tests:
        -2.0
        2.25
        1.66666666666667
```

Addition on integers is associative, on floating point numbers it is not. The number of tests *quickCheck* executes in order to verify a property is 100 by default, and can be changed by using a different configuration.

The arguments of *prop_* functions need to be instances of the type class *Arbitrary*, providing the data generator *arbitrary*. The QuickCheck module provides many instances of this type class, for the primitive types, for data structures such as tuples, lists and even for functions. Among the datatypes for which QuickCheck does *not* provide such instances are *Id a*, *SimpleProb a* and *ExpT a m b*, which we need in order to test properties of monadic dynamical systems. Therefore, we define them here.

```
instance Arbitrary a ⇒ Arbitrary (Id a) where
  arbitrary = (return · Id) ⊲ arbitrary
```

```
instance (Arbitrary a, Arbitrary (m b)) ⇒
  Arbitrary (ExpT a m b) where
  arbitrary = (return · ExpT) ⊲ arbitrary
```

These instance declarations take advantage of the predefined ones, and construct arbitrary elements of the required type from their components.

In defining the data generator for *SimpleProb a* we must also check that the lists of elements generated are not empty (that is, that we do not generate *SP* [ ]) and that the elements have non-zero probabilities (otherwise the scaling of probabilities might fail with a divide-by-zero error).

```
**instance** $Arbitrary\ a \Rightarrow Arbitrary\ (SimpleProb\ a)$ **where**
  $arbitrary = arbitrary \rhd f$
    **where**
    $f\ (x : xs) = return\ (SP\ xs')$
      **where**
      $xs'' = map\ (cross\ (id, abs'))\ (x : xs)$
           **where** $abs'\ x =$ **if** $x = 0$
                              **then** $0.00001$
                              **else** $abs\ x$
      $tot\ = sum\ (map\ snd\ xs'')$
      $xs'\ = map\ (cross\ (id, (/tot)))\ xs''$
    $f\ [\,] = (return \cdot return) \lhd arbitrary$
```

Boolean-valued properties are not the only testable properties. The combinator $\Rightarrow$ introduces *conditional properites*, whose values are not *Bool*, but rather the QuickCheck type *Property*. The combinator allows to formulate a precondition on the values of the arguments of the *prop_* function before testing the property. For example

$$prop\_has\_inverse\ x = (1\ /\ x) * x = 1$$

fails when checked for *Rational*

```
quickCheck (prop_has_inverse::Rational -> Bool)
    ==> 2
        Program error: Ratio.%: zero denominator
```

because 0 has no inverse. What we want is rather

$$prop\_has\_inverse\ x = x \neq 0 \Rightarrow (1\ /\ x) * x = 1$$

which has the type

$$prop\_has\_inverse :: Rational \rightarrow Property$$

```
quickCheck(prop_has_inverse::Rational -> Property)
    ==> OK, passed 100 tests.
```

Some preconditions are fulfilled only for a small subset of the datatype's space, and generating arguments until such conditions are met can be very costly. In order to avoid is, one can use the *forAll* combinator, which allows one to provide a custom generator. For example, instead of

$$prop\_transitive\ x\ y\ z = x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$$

which gives

```
quickCheck (prop_transitive::(Real, Real) -> (Real, Real) ->
                             (Real, Real) -> Property)
    ==> Arguments exhausted after 77 tests.
```

it is better to use

$$\begin{array}{l} \mathit{prop\_transitive}\ x = \mathit{forAll}\ (\mathit{gen\_bigger}\ x)\ (\lambda y \rightarrow \\ \qquad\qquad (\mathit{forAll}\ (\mathit{gen\_bigger}\ y)\ (\lambda z \rightarrow \\ \qquad\qquad\quad x \leqslant z))) \\ \quad \mathbf{where} \\ \quad \mathit{gen\_bigger}\ x = (\mathit{return} \cdot (+x)) \lhd \mathit{choose}\ (0.0, 1.0) \end{array}$$

This time, *quickCheck* has no problems:

```
quickCheck (prop_transitive::Real -> Property)
    ==> OK, passed 100 tests.
```

7.3.2 Testable conditions

About the simplest condition we can test when given a *VulnerabilityContext* is that the monoid of the system giving us the possible evolutions is indeed a monoid. In Haskell, the typeclass *Monoid* ensures that the datatype of the monoid of the system is equipped with the function *mappend* and with the unit *mempty*, but checking associativity and unit properties cannot, in general, be done by the type system. However, both properties are easy to define in QuickCheck:

$$\begin{array}{l} \mathit{prop\_mappend\_associative}\ x\ y\ z = x\ `\mathit{mappend}`\ (y\ `\mathit{mappend}`\ z) = \\ \quad (x\ `\mathit{mappend}`\ y)\ `\mathit{mappend}`\ z \end{array}$$

$$\begin{array}{l} \mathit{prop\_mempty\_unit}\ x = (x\ `\mathit{mappend}`\ \mathit{mempty} = x) \wedge \\ \quad (\mathit{mempty}\ `\mathit{mappend}`\ x = x) \end{array}$$

Similarly, if a type constructor such as *SimpleProb* has been declared to be of the type *Monad*, then instances of the types constructed by it are guaranteed to be equipped with *return* and $\rhd$ operations, but the properties of these operations cannot be checked by the type system. When writing tests for these operations, we have two additional difficulties as compared to the case above. First, we have to test equality between values which represent functions, second, we have to generate functions in order to be able to test these properties. For example:

$$\mathit{prop\_return\_left\_unit}\ f\ x = (\mathit{return}\ x) \rhd f = f\ x$$

Checking that *return* is a left unit of $\rhd$ requires testing the equality of the functions $(f\lhd) \cdot \mathit{return}$ and f. We can, in general, only test equality of functions *extensionally*, therefore we need to provide the argument of the function explicitely (remember that arguments of *prop_* functions are universally quantified).

Functions can be generated by QuickCheck if elements in the source and target can be generated. In general, however, it is better to use a custom generator. In our case, we could use the generator that the given *VulnerabilityContext* provides for us. The given system, say $sys :: T \to X \to M\ X$ gives us, for every t, a function $X \to M\ X$, namely $sys\ t$. We can then reformulate the property using *sys* as a custom generator:

$$sysfuns\ sys = (return \cdot sys) \lhd arbitrary$$

$$\begin{aligned} &test\_return\_left\_unit\ sys\ x = forAll\ (sysfuns\ sys)\\ &\quad (\lambda f \to ((return\ x) \rhd f) = (f\ x)) \end{aligned}$$

The property to be tested is *test_return_left_unit sys*, not *test_return_left_unit*. If the latter were the case, *sys* would be universally quantified, and QuickCheck would attempt to generate random instances of the type of *sys*. In order to respect the convention that arguments of *prop_* functions are universally quantified, we prefix properties that depend on a paramenter with *test_*, rather than *prop_*.

We can use custom generators to state the associativity of $\rhd$ as well:

$$\begin{aligned} &test\_bind\_assoc\ sys\ mx = forAll\ (sysfuns\ sys)\\ &\quad (\lambda f \to (forAll\ (sysfuns\ sys)\\ &\quad (\lambda g \to (((mx \rhd f) \rhd g) =\\ &\qquad (mx \rhd (\lambda x \to (f\ x) \rhd g)))))) \end{aligned}$$

Finally, the right unit property of *return* is simpler to formulate:

$$prop\_return\_right\_unit\ mx = (mx \rhd return) = mx$$

Once we have tested the monoid and monad properties, we can test that the given system represents a monoid morphism.

$$\begin{aligned} &test\_monoid\_morphism\ sys\ t1\ t2\ x = (sys\ (t1\ `mappend`\ t2)\ x) =\\ &\quad (sys\ t1 \lhd sys\ t2\ x) \end{aligned}$$

The most vulnerability-specific property we can check when given a *VulnerabilityContext* is the monotonicity condition for the vulnerability measure. Again, this property requires generating functions, but in this case the functions must be non-decreasing. We could implement a test for non-decreasing functions and specify the monotonicity condition as

$$\begin{aligned} &test\_monotonicity\ measure\ inc\ mv = nondecreasing\ inc \Rightarrow\\ &\quad mv \sqsubseteq measure\ (fmap\ inc\ mv) \end{aligned}$$

This specification is in most cases inadequate because arbitrarily generated functions are unlikely to be non-decreasing, and QuickCheck will stop with an inconclusive result once it reaches the maximum number of attempts for which it is configured. Thus, it is better to use a custom generator which guarantees that the functions it generates are non-decreasing.

$$\begin{aligned}&\mathit{test\_monotonicity}\ \mathit{measure}\ \mathit{geninc}\ \mathit{mv} = \mathit{forAll}\ \mathit{geninc}\\ &\quad (\lambda \mathit{inc} \to ((\mathit{measure}\ \mathit{mv}) \sqsubseteq (\mathit{measure}\ (\mathit{fmap}\ \mathit{inc}\ \mathit{mv}))))\end{aligned}$$

The last condition we deal with here is the compatibility of two vulnerability measures, as defined above. We need for this an injective natural transformation. The injectivity condition is checked by

$$\mathit{test\_injective}\ f\ x_1\ x_2 = x_1 \neq x_2 \Rightarrow (f\ x_1) \neq (f\ x_2)$$

We can then translate the definition of compatibility of vulnerability measures in QuickCheck as follows:

$$\begin{aligned}&\mathit{test\_compatibility}\ \tau\ m_M\ m_N\ mx_1\ mx_2 =\\ &\quad ((m_M\ mx_1) \sqsubseteq (m_M\ mx_2)) \equiv\\ &\quad ((m_N\ (\tau\ mx_1)) \sqsubseteq (m_N\ (\tau\ mx_2)))\end{aligned}$$

This test fails if τ is not monotonous with respect to the preorders induced by the vulnerability measures (the implication $\Rightarrow$ does not hold), or if there exist mx_1 and mx_2 for which $\tau\ mx_1$ and $\tau\ mx_2$ are comparable, but mx_1 and mx_2 are not.

7.3.3 Example

In many ways, the example of the interacting stochastic and non-deterministic systems given in Section 6.2.2 is representative for the problems which appear in vulnerability studies. The systems involved have different types, the evolutions are not deterministic, and the objective is to keep the controlled system in a certain range (in the Climate Change community, this is often called the *coping range*, see, for example, Chapter 2 in [19]). Only one feature is missing from it in order for it to be a good "toy model" for vulnerability assessments: we have seen that the goals of keeping the system within the desired range and minimizing the costs were very much compatible: minimizing the costs led to the desired behavior of the system. In vulnerability studies, the situation is mostly the opposite: the goal of emission reduction (keeping the climate system within certain bounds), for instance, is often seen as conflicting with that of economic growth. The latter could be represented in our example if we choose to *maximize*, rather than minimize, the number of controls different from 0.0.

Disclaimer: this example is not meant to be in any way a realistic representation of the interaction between the social and the ecological systems, beyond the features explicitely stated in the above paragraph. And much of the research in climate impact studies aims to show that, in fact, the goals of environmental-friendly developement and economic growth are quite compatible with each other.

In our example, the ecological system is represented by *runsys* and the social system is represented by a family of systems included in *sysfam*.

The first thing to check is that the monoid, monad and monoid morphism conditions are in fact fullfilled (of course, we *know* that in this case they are, but in general the models are presented as executable code, and the conditions should be tested).

The monoids involved are $\mathbb{N}$ for *runsys* and the systems in *sysfam*, $[(\mathit{Int}, \mathit{Int})]$ for the combination $\mathit{sysserfaml}\ (\mathit{sysfam}, \mathit{runsys})$. We have

```
quickCheck (prop_mappend_associative::Int ->
               Int -> Int -> Bool)
    ==> OK, passed 100 tests.

quickCheck (prop_mempty_unit::Int -> Bool)
    ==> OK, passed 100 tests.

quickCheck (prop_mappend_associative::[(Int, Int)] ->
               [(Int, Int)] -> [(Int, Int)] -> Bool)
    ==> OK, passed 100 tests.

quickCheck (prop_mempty_unit::[(Int, Int)] -> Bool)
    ==> OK, passed 100 tests.
```

No surprises there. As an aside, we note that a frequent case in which a function *sys* fails the monoid morphism test is when $\mathit{sys}\ t :: X \to M\ X$ depends on the argument t. Consider, for example,

$$\mathit{detsys}' :: \mathbb{N} \to \mathbb{N} \to \mathit{Id}\ \mathbb{N}$$
$$\mathit{detsys}'\ t\ n = \mathit{Id}\ (t * n)$$

We do have that $\mathit{detsys}'\ 0 = \mathit{return}$, but the monoid operation is no longer preserved. QuickCheck gives us

```
quickCheck (test_monoid_morphism detsys')
    ==> Falsifiable, after 0 tests:
        2
        0
        1
```

Indeed, $\mathit{detsys}'\ (2 + 0)\ 1 = 2$, while $\mathit{detsys}'\ 2\ (\mathit{detsys}'\ 0\ 1) = 0$.

The monads involved are *SimpleProb* for *runsys*, *Id* for the systems in *sysfam* and *ExpT Int SimpleProb* for the combination $\mathit{sysserfaml}\ (\mathit{sysfam}, \mathit{runsys})$.

```
quickCheck (prop_return_right_unit::SimpleProb Real -> Bool)
    ==> OK, passed 100 tests.

quickCheck (prop_return_right_unit::Id Real)
    ==> OK, passed 100 tests.

quickCheck (prop_return_right_unit::ExpT Int SimpleProb Real)
    ==> OK, passed 100 tests.

quickCheck (test_return_left_unit runsys)
    ==> OK, passed 100 tests.

quickCheck (test_return_left_unit (sysfam 0))
    ==> OK, passed 100 tests.

quickCheck (test_return_left_unit (sysserfaml (sysfam, runsys)))
    ==> OK, passed 100 tests.

quickCheck (test_bind_assoc runsys)
    ==> OK, passed 100 tests.

quickCheck (test_bind_assoc (sysfam 0))
    ==> OK, passed 100 tests.

quickCheck (test_bind_assoc (sysserfaml (sysfam, runsys)))
    ==> OK, passed 100 tests.
```

The monoid morphism property for the systems involved is checked by

```
quickCheck (test_monoid_morphism runsys)
    ==> OK, passed 100 tests.

quickCheck (test_monoid_morphism (sysfam 0))
    ==> OK, passed 100 tests.

quickCheck (test_monoid_morphism (sysserfaml (sysfam, runsys)))
    ==> OK, passed 100 tests.
```

Next, we have to define the function *harm* to be used in estimating the vulnerability. Harm is measured along two dimensions in our example: *runsys* going out of bounds, or having a small number of controls different from 0.0. Along the first dimension, we can borrow the idea of Luers et al. described in Section 3.3.3 and compute a function of the distance from the desired range:

```
$out\_of\_bounds :: \mathbb{R} \to \mathbb{R}$
$out\_of\_bounds\ r$
    $\mid r \geqslant -5.0 \wedge r \leqslant 5.0 = 0.0$
    $\mid r < -5.0 = abs\ (-5.0 - r)$
    $\mid r > 5.0 \quad = abs\ (r - 5.0)$
```

```
$total\_oob :: [(\mathbb{R}, \mathbb{R})] \to \mathbb{R}$
$total\_oob = sum \cdot map\ (out\_of\_bounds \cdot fst)$
```

Along the second dimension, we can choose a definition of "small": say, under half of the controls are different from 0.0. We can just count the "missing" elements:

```
$misses \quad :: [(\mathbb{R}, \mathbb{R})] \to \mathbb{N}$
$misses\ xs = \mathbf{if}\ k > (n\ `div`\ 2)\ \mathbf{then}\ k\ \mathbf{else}\ 0$
              $\mathbf{where}$
              $n = length\ xs - 1$
              $k = length\ (filter\ (\lambda(r, c) \to c = 0.0)\ xs) - 1$
```

```
$h1 :: [(\mathbb{R}, \mathbb{R})] \to (\mathbb{R}, \mathbb{N})$
$h1 = pair\ (total\_oob, misses)$
```

We have to define a partial order on $(\mathbb{R}, \mathbb{N})$. The simplest one is given by the dominance relation:

```
$\mathbf{instance}\ (Ord\ a, Ord\ b) \Rightarrow PartialOrd\ (a, b)\ \mathbf{where}$
  $(a1, b1) \sqsubseteq (a2, b2) = (a1 \leqslant a2) \wedge (b1 \leqslant b2)$
  $(a1, b1) \sqsupseteq (a2, b2) = (a1 \geqslant a2) \wedge (b1 \geqslant b2)$
```

We now have to choose a measure for vulnerability, which aggregates the measures along all trajectories.

Let us first consider the case in which the family of systems representing the social system is reduced to just one, for example, the optimal choice of the previous chapter

```
$prevbest = app\ (mkfun\ ([(-4.0, 3.0), (-3.0, 3.0), (-2.0, 0.0),$
                          $(-1.0, 0.0), (0.0, 0.0), (1.0, 0.0),$
                          $(2.0, 0.0), (3.0, -3.0), (4.0, -3.0)]))$
```

The combined system is

```
$combsys1 = detserl\ (prevbest, runsys)$
```

The structure of all trajectories is a *SimpleProb* $[(\mathbb{R}, \mathbb{R})]$, and the vulnerability measure thus has to act on *SimpleProb* $(\mathbb{R}, \mathbb{N})$. Again, following Luers et al., we can take the expected value along the two dimensions:

$$\begin{aligned}&m_1 :: \mathit{SimpleProb}\ (\mathbb{R}, \mathbb{N}) \rightarrow (\mathbb{R}, \mathbb{R})\\ &m_1 = \mathit{pair}\ (\mathit{expected} \cdot \mathit{fmap}\ \mathit{fst},\\ &\qquad\qquad \mathit{expected} \cdot \mathit{fmap}\ (\mathit{realToFrac} \cdot \mathit{snd}))\end{aligned}$$

In order to check that the monotonicity condition is fulfilled, we need to write a generator for increasing functions on $(\mathbb{R}, \mathbb{N})$. For example, we can take

$$\begin{aligned}&\mathit{geninc} :: \mathit{Gen}\ ((\mathbb{R}, \mathbb{N}) \rightarrow (\mathbb{R}, \mathbb{N}))\\ &\mathit{geninc} = \mathbf{do}\\ &\qquad \mathit{dx} \leftarrow \mathit{choose}\ (0, 10)\\ &\qquad \mathit{dn} \leftarrow \mathit{choose}\ (0, 10)\\ &\qquad \mathit{return}\ (\lambda(x, n) \rightarrow (x + \mathit{dx}, n + \mathit{dn}))\end{aligned}$$

The monotonicity condition is then checked by

```
quickCheck (test_monotonicity m1 geninc)
    ==> OK, passed 100 tests.
```

The vulnerability of *sys1* in the state $(0.0, 0.0)$ within 10 steps is

```
vulnerability (VC (combsys1, h1, m1)) (0.0, 0.0)
                        (replicate 10 [(1, 1)])
    ==> (0.0,7.638671875)
```

Let us now consider the case in which the family of systems is composed of several members. For example:

$$\begin{aligned}\mathit{lists} = [&[(-4.0, 3.0), (-3.0, 3.0), (-2.0, 3.0), (-1.0, 3.0),\\ &(0.0, 3.0), (1.0, -3.0), (2.0, -3.0), (3.0, -3.0),\\ &(4.0, -3.0)], \quad \text{-- a greedy strategy}\\ &[(-4.0, 3.0), (-3.0, 3.0), (-2.0, 0.0), (-1.0, 0.0),\\ &(0.0, 3.0), (1.0, 0.0), (2.0, 0.0), (3.0, -3.0),\\ &(4.0, -3.0)], \quad \text{-- a version of prevbest}\\ &[(-4.0, 3.0), (-3.0, 3.0), (-2.0, 3.0), (-1.0, 3.0),\\ &(0.0, 3.0), (1.0, 0.0), (2.0, 0.0), (3.0, -3.0),\\ &(4.0, -3.0)], \quad \text{-- greedy prevbest}\\ &[(-4.0, 3.0), (-3.0, 3.0), (-2.0, 0.0), (-1.0, 0.0),\\ &(0.0, 3.0), (1.0, -3.0), (2.0, -3.0), (3.0, -3.0),\\ &(4.0, -3.0)]] \quad \text{-- greedy prevbest}\end{aligned}$$

The system family is constructed as in the previous chapter:

$$\begin{aligned}&\mathit{sysfam2}\ n\ t\ (r, c) = (\mathit{funs}'\ !!\ n)\ t\ (r, c)\\ &\qquad\qquad \mathbf{where}\\ &\qquad\qquad \mathit{funs}' = \mathit{map}\ (\mathit{app} \cdot \mathit{mkfun})\ \mathit{lists}\end{aligned}$$

The combined system is

$$combsys2 = sysserfaml\ (sysfam2, runsys)$$

Now the structure of the trajectories is $ExpT\ \mathbb{N}\ SimpleProb\ [(\mathbb{R}, \mathbb{R})]$. Correspondingly, the vulnerability measure will act on $ExpT\ \mathbb{N}\ SimpleProb\ (\mathbb{R}, \mathbb{N})$. An intuitively reasonable way of aggregating this structure is to take the average across *all* trajectories:

$$\begin{aligned}
&m_2 :: \mathbb{N} \rightarrow ExpT\ \mathbb{N}\ SimpleProb\ (\mathbb{R}, \mathbb{N}) \rightarrow (\mathbb{R}, \mathbb{R})\\
&m_2\ numsys\ (ExpT\ f) = (avg1, avg2)\\
&\qquad\qquad\qquad\qquad\quad \mathbf{where}\\
&\qquad\qquad\qquad\qquad\quad xs = map\ m_1\ [f\ n \mid n \leftarrow [0\,..\,numsys - 1]]\\
&\qquad\qquad\qquad\qquad\quad avg1 = sum\ (map\ fst\ xs)\ /\ l\\
&\qquad\qquad\qquad\qquad\quad avg2 = sum\ (map\ snd\ xs)\ /\ l\\
&\qquad\qquad\qquad\qquad\quad l = realToFrac\ (length\ xs)
\end{aligned}$$

The measure is $m_2\ numsys$, not m_2. We have to pass explicitely the number of systems in the family, in this case 4. We can check that this is indeed a vulnerability measure:

```
quickCheck (test_monotonicity (m2 4) geninc)
    ==> OK, passed 100 tests.
```

Is this compatible with the expected value measure? In order to answer this, we have to find a natural transformation $SimpleProb \rightarrow ExpT\ \mathbb{N}\ SimpleProb$ with respect to which we can test monotonicity. Fortunately, this is not difficult:

$$\begin{aligned}
&\tau\quad :: SimpleProb\ a \rightarrow ExpT\ \mathbb{N}\ SimpleProb\ a\\
&\tau\ sp = ExpT\ (const\ sp)
\end{aligned}$$

For every type A, the function $\tau :: SimpleProb\ A \rightarrow ExpT\ \mathbb{N}\ SimpleProb\ A$ is injective:

$$\begin{aligned}
&\quad \tau\ sp_1 = \tau\ sp_2\\
&\equiv \quad \{\ \text{definition } \tau\ \}\\
&\quad ExpT\ (const\ sp_1) = ExpT\ (const\ sp_2)\\
&\equiv \quad \{\ \text{constructors are injective}\ \}\\
&\quad const\ sp_1 = const\ sp_2\\
&\equiv \quad \{\ const\ \}\\
&\quad sp_1 = sp_2
\end{aligned}$$

Of course, checking with QuickCheck confirms this:

```
quickCheck (test_injective
    (tau::SimpleProb Real -> ExpT Nat SimpleProb Real))
     ==> OK, passed 100 tests.
```

The compatibility condition can then be checked:

```
quickCheck (test_compatibility tau m1 (m2 4))
     ==> OK, passed 100 tests.
```

The vulnerability of the combined system in state $(0.0, 0.0)$ within 10 steps is

```
vulnerability (VC (combsys2, h1, m2 numsys)) (0.0, 0.0)
                (replicate 10 [(1, 1)])
          where numsys = length lists
     ==> (521.5,0.0)
```

Let us represent the social system by the collection of all systems in *sysfam*, and find out the optimal policies with respect to our two goals. Combining each system in *sysfam* with *runsys*, we obtain

$$\begin{aligned}&\mathit{allsyss} = [\,\mathit{detserl}\ (\mathit{sysfam}\ n, \mathit{runsys}) \mid \\ &\qquad\qquad n \leftarrow [0\,..\,(\mathit{length}\ \mathit{funlists} - 1)]\,]\end{aligned}$$

We can compute the vulnerability of each of these systems by

$$\begin{aligned}&\mathit{allvulns} = [\,\mathit{vulnerability}\ (\mathit{VC}\ (\mathit{sys}, \mathit{h1}, m_1))\ (0.0, 0.0) \\ &\qquad\qquad (\mathit{replicate}\ 10\ [(1,1)]) \mid \mathit{sys} \leftarrow \mathit{allsyss}\,]\end{aligned}$$

If we give priority to keeping *runsys* within the prescribed bounds, we are interested in the systems for which the vulnerability along the first component is 0.0 and the second component is as small as possible:

$$\begin{aligned}&\mathit{bestBounds} = \mathit{takeWhile}\ (p\ (\mathit{head}\ \mathit{xyzs}))\ \mathit{xyzs} \\ &\qquad\qquad \mathbf{where} \\ &\qquad\qquad p\ ((a, b), c)\ ((x, y), z) = y \leqslant b \\ &\qquad\qquad \mathit{xyzs} = \mathit{sort}\ (\mathit{filter}\ (\lambda((a, b), i) \rightarrow a = 0.0) \\ &\qquad\qquad\qquad\qquad (\mathit{zip}\ \mathit{allvulns}\ [0\,..]))\end{aligned}$$

```
bestBounds
     ==> [((0.0,0.474609375),14),((0.0,0.474609375),15)]
```

The skeleton of the 15th member of the system is

$$\begin{aligned}&[(-4.0, 3.0), (-3.0, 3.0), (-2.0, 3.0), (-1.0, 3.0), \\ &\quad (0.0, 0.0), (1.0, -3.0), (2.0, -3.0), (3.0, -3.0), (4.0, -3.0)]\end{aligned}$$

On the other hand, if we give priority to minimizing the number of controls of 0.0, we have

```
bestCtrls = takeWhile (p (head xyzs)) xyzs
        where
        p ((a, b), c) ((x, y), z) = x ⩽ a
        xyzs = sort (filter (λ((a, b), i) → b = 0.0)
                    (zip allvulns [0 ..]))
```

All the elements in *bestCtrls* have a vulnerability of $(512.5, 0.0)$. Their indices in the family of systems are given by

$$[16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,\\ 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,\\ 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,\\ 112, 113, 114, ...508, 509, 510, 511]$$

(in total 256).

As we see, the two goals are not compatible: there is no policy which keeps the system within bounds and which always fulfills the "economic" objective. One could try, for example, to increase the bounds within which *runsys* is to be kept: this is similar to the discussions in climate reasearch about the climate change range with which society can cope. Alternatively, one could try to bring about a "technological improvement", which would represent in this case a control range of more than just the values -3.0 and 3.0. Indeed, for example the policy described by the "skeleton"

$$[(-4.0, 3.0), (-3.0, 3.0), (-2.0, 2.0), (-1.0, 2.0),\\ (0.0, 2.0), (1.0, -2.0), (2.0, -2.0), (3.0, -3.0), (4.0, -3.0)]$$

leads to a combined system whose vulnerability is

$$(0.0, 0.0)$$

Bringing about a situation as close as possible to this one is the goal of climate impact research.

7.4 Conclusions

In this final chapter, we have gathered together the main threads running through this work. Using the monadic dynamical systems developed in the second part, we have been able to give a surprisingly simple implementation of the model of vulnerability developed in the first part. The main theoretical result of this chapter, the compatibility condition on vulnerability measures, allows us to check to what extent various computational vulnerability assessments may be compared, and to extend vulnerability measures defined on one type of structure to other types by means of composing them with natural transformations.

The various conditions that can be posed on the components of the implemented models of vulnerability can be expressed in a form suitable for automatic testing. We have presented the main components of such a testing tool, the popular Haskell utility QuickCheck, and we have shown how to implement the conditions in terms of QuickCheck combinators.

Finally, we have shown how the model can be used in an idealized example of a vulnerability assessment-like problem. This "toy example" exhibits many of the features of full-blown vulnerability assessments: possible evolutions given by combined systems of different types, conflicting definitions of "harm", partial ordering of vulnerability measurements. We have used the implemented tests to verify the various components of the model and to ensure that the vulnerability measures introduced were compatible. The analysis of the compromises that need to be made between the conflicting goals, and the possible changes to the problem that could lead to better results, have many parallels in the "real-world" discussions on the impacts of climate change.

Chapter 8

Conclusions and Perspectives

8.1 A look back

We started our journey by examining a number of definitions of vulnerability, both in the ordinary language context, as well as in fields of Development Studies and Climate Change. We have attempted to leverage the work of synthesis done by Calvo and Dercon in [4] and by the IPCC in [19]. The common theme of all these definitions is that vulnerability represents a measure of the harm that might occur in future evolutions. There were, as we have seen, also a number of important differences. The Development Studies community, for example, favors expressions such as "vulnerable to poverty", that is, vulnerability is relative to the harm suffered, while the Climate Change community prefers to refer to "vulnerability to climate change", emphasizing the causes of the potential harm. The model we have developed has helped us understand this (and other) differences: in development studies, the measurement of harm along a possible evolution focuses on poverty, neglecting other negative effects, but considering all causes of this poverty, while climate change studies typically try to assess many different kinds of impacts, but only taking into account those caused (mainly) by climate change. The symmetry involved in these exclusion/inclusion decisions is exhibited in our model by the similar role played by the impacts (negative effects) and the sensitivity to certain factors (the causes in whose effects we are interested).

In order to unify the various definitions of vulnerability, we have taken a very general view of what a "possible future evolution" is. We did not want to be limited to "a set of evolutions", because, for example, we wanted to also represent "probable future evolutions" in the same framework as a probability distribution over evolutions. At this point, we made use of elementary concepts of Category Theory, representing "possible evolutions" as functorially constructed structures of evolutions. Computing the impacts along these evolutions is done by "mapping" the function that evaluates the impacts on the structure of evolutions, while vulnerability represents a measurement on the entire structure of impact values.

Similarly, we were able to formulate precisely the natural condition of consistency between the impact evaluations and the vulnerability measure: ceteris paribus, if all impacts in the structure increase, then the vulnerability measure should not decrease. It is hard to imagine how one might capture this requirement in a precise fashion without the notion of functor.

Once we had developed the model, we returned to the original definitions, especially to those which had been expressed in mathematical terms, in order to see whether we could now interpret all of them as special cases of the more general, unifying version. We also formalized the derived concepts of sensitivity and adaptive capacity, and gave conditions for their usage.

Once this was done, we had a certain confidence that the model was adequate to account for the concept of vulnerability and its usage. However, what we had was only a formalization (or specification): to develop software, we needed to refine this specification.

The major problem in operationalizing our specification of vulnerability was how to compute the structure of future evolutions. We decided to view the possible evolutions as given by a dynamical system, in conformity to actual usage, where the systems considered are interacting models of the physical, social, or economical world. In attempting to compute the trajectories of systems of different types (deterministic, stochastic, non-deterministic, and combinations thereof), we examined the mathematical literature. The classical work on systems theory, as exemplified by the textbook of Denker ([11]) provided the means of computing trajectories of systems viewed as monoid actions, but did not account for the structure of these trajectories in a satisfactory fashion. The modern work presented, for example, in [34], which views systems as coalgebras, preserved the structure, but made computing the trajectories difficult.

The solution was to settle on a particular class of coalgebras, namely the coalgebras of monadic functors. Monads come equipped with a kind of natural "iteration" which is given by the Kleisli composition. Thus, monadic coalgebras were able to represent the structural properties of the systems we considered, and to allow the generic computation of trajectories.

In monadic systems, in fact, we encountered two kinds of trajectories: those that are the perfect analogues of the traditional trajectory of a dynamical system, and which we called macro-trajectories, and those that more resembled the results of iterating coalgebraic systems, which we called micro-trajectories. Both kinds of trajectories are interesting in applications, and both can be computed generically.

At this point, there could arise the question: do we have three kinds of systems, or only one? Is there no way of unifying the various definitions so as to be able to translate various results from one kind of system to the other? In fact, it turned out that classical systems, coalgebras and monadic coalgebras were all instances of the same defintion: a system is a morphism of monoids whose target is a set of endo-arrows.

In this way we could also treat the continuous case and the case of systems with input, by taking over the definitions from the classical systems (endo-arrows in *Set*) and translating them into endo-arrows of a Kleisli category.

We could now compute trajectories of monadic systems in a generic fashion, but, in computer parlance, we were missing the *constructors* of monadic systems. We saw that the "elementary" components of the complex systems typically considered in vulnerability assessments, the deterministic, stochastic, fuzzy, non-deterministic systems were all monadic, but what about their combinations? It is well known that there is no uniform way of combining monads, and that the same two monads can be combined to yield a monad in different ways. For better or worse, the same flexibility is available to monadic systems as well. We developed generic functions for the most frequent combinations, and we showed that different combinations of the same systems are possible and useful.

Armed now with a collection of operations on monadic systems, enabling us to build up complex systems out of simpler ones and to compute their trajectories, we returned to the design and implementations of components for vulnerability assessment. At this point, we could fully flesh out our model: the various components received computational expression, and we could implement and test a number of examples.

Since vulnerability assessments are often exploratory, it is common to want to replace a system of one type, say deterministic, by another which represents the same "real-world component" but has a different type, say stochastic. Since the types of the components of a complex system influence the type of the complex system itself and ultimately the structure of its trajectories, and since vulnerability measures are defined on these structures, it follows that changes in the types of components attract changes in the vulnerability measure. On the other hand, since the complex system is still meant as a representation of the same "reality", we can ask what is the relation between the two vulnerability measures? When are they compatible? What can be reused in the passage from one structure to another? We were able to give a general answer to these questions, again using elementary categorial machinery, in this case, natural transformations.

Speaking of reuse, when putting together complex systems there is always the possibility of error: are all the components compatible? do all components fulfill their requirements? and so on, are typical questions that arise in software engineering. An advantage of a precise mathematical specification of the compatibility conditions and component requirements is that it can the be translated into tests which can be automatically performed using a tool such as QuickCheck. We showed how this is done, using simple examples for all tests of interest.

As a final proof of concept, we conducted a “toy vulnerability assessment”, where we encountered all the problems of realistic cases in laboratory conditions: combined systems of different types, conflicting definitions of “harm”, partial ordering of vulnerability measurements, etc., thus enabling us to conclude, with a certain degree of confidence, that the model proposed and the attendant software infrastructure are adequate for the purposes of computational vulnerability assessments in the context of climate change.

8.2 A look ahead

The research reported here was partly financed by the European Commission within the ADAM project (Adaptation and Mitigation Strategies Supporting European Climate Policy, http://www.adamproject.eu), where it is used to guide the meta-analysis task (a comparative study of approximately two hundred case studies of climate change impacts, vulnerability and adaptation) and the design of a digital atlas of vulnerability. Our model of vulnerability was also used to formalize the notion of “risk” as it appears in the Natural Hazards community, and to clarify the similarities and differences between the usage of “vulnerability” in this field and Climate Change.

In the longer term, the model of vulnerability is expected to play a bigger part in the analysis of vulnerability studies, along the lines of Chapter 4. This will involve taking a closer look at the models used in the various case studies, in order to arrive at the computational expressions of the structures of possible evolutions that were used for the assessment. Many studies do not end up with a vulnerability measurement, but present instead the possible evolutions in a synoptic form, as was done in the ATEAM project discussed in Subsection 3.3.2 where the introduction of a vulnerability measure was considered as “the most dangerous step”. We aim to use the results developed in this thesis in order to make the introduction of vulnerability measures simpler and eliminate at least some of the pitfalls associated to it.

This brings us to the next application of our thesis: the developement of software components for vulnerability assessment. Taking the work presented here as a starting point, Daniel Lincke from the Potsdam Institute for Climate Impact Research is currently developing a library of generic concepts (in the C++ sense) together with implementations of important instances of these concepts, in order to put the foundations of a computer assisted vulnerability assessment tool. The current status of his work is reported on in [27].

Also working at the Potsdam Institute for Climate Impact Research, Sarah Wolf is investigating additional examples of uncertainty representation in dynamical systems. She is currently focusing on finitely additive probability theory, a generalization of the classical probability theory (which requires countable additivity), and which promises to also encompass fuzziness, possibility functions, and default reasoning, and thus serve as a unifying tool for uncertainty management (see [8]). The question that arises is: do finitely additive probability distributions form a monad? Or is there a more general structure that should be used instead?

The functions *trj* and *mtrj* can be defined as folds on lists. Indeed, we have

```
trj sys x = foldr f [return x]
            where
            f t (mx : mxs) = ((sys t) ◁ mx) : mxs

mtrj sys x = foldr f (return [x])
             where
             f t = (addHist (sys t)) ◁
```

One can therefore pose the question of writing other versions of these functions, for other structures of type *Fix f* than [ ]. Remember that the list of values along which trajectories are computed is not always a list of time intervals, but can also be a list of commands or inputs. If we want to compute trajectories along different lists of commands, we can do that list by list, but we risk being very inefficient if the lists have, say, initial segments in common. In this case, we could for example arrange the commands in a tree structure, so that they branch out where they differ, but the initial identical parts are computed only once.

An even more important source of inefficiency is the combinatorial explosion which results from combining non-deterministic systems with other types of systems. In the examples we have seen in Chapters 6 and 7, the set of all trajectories that needed to be computed grows exponentially with the number of steps taken. For optimization problems, building this set of trajectories may be avoided if we can eliminate non-optimal trajectories while they are being constructed. For the case in which the system to be combined with the non-deterministic one is deterministic, there exist well known algorithms which allow us to do just this: see, for example, the chapters on thinning and dynamic programming in [2]. The same algorithms can be applied also to the case in which the system to be combined with the non-deterministic one is stochastic, but the resulting gain is smaller: there is already a growth in the support of the probability distributions over the possible trajectories. What is necessary here is to generalize thinning to the general case of monadic systems, from the current case where the monad is [ ]. Another necessary step is to investigate what type of thinning may be achieved when the vulnerability measure is not of the "worst impact" type, but has a more integrative structure.

Finally, thinning algorithms achieve a similar effect to using sensitivity and adaptive capacity contexts in a vulnerability assessment: they eliminate possible evolutions from those that need to be considered. Investigating the relationship between these three is an interesting topic for future research.

The conditions for this elimination are described in the chapters on thinning and dynamic programming in [2].

Bibliography

[1] R. Bird. *Introduction to Functional Programming using Haskell.* International Series in Computer Science. Prentice Hall, second edition edition, 1998.

[2] R. S. Bird and O. de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.

[3] R. S. Bird and P. Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[4] C. Calvo and S. Dercon. Measuring individual vulnerability. Economics Series Working Papers 229, University of Oxford, Department of Economics, 2005. Available at http://ideas.repec.org/p/oxf/wpaper/229.html.

[5] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP).* ACM SIGPLAN, 2000.

[6] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Proc. of Haskell Workshop.* ACM SIGPLAN, 2002.

[7] K. Claessen and J. Hughes. Specification based testing with QuickCheck. In *The Fun of Programming*, Cornerstones of Computing, pages 17–40. Palgrave, 2003.

[8] G. Coletti and R. Scozzafava. *Probabilistic Logic in a Coherent Setting.* Springer, 2002.

[9] O. de Moor. A generic program for sequential decision processes. In *PLILPS '95: Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs*, pages 1–23, 1995.

[10] O. de Moor. Dynamic programming as a software component. In N. Mastorakis, editor, *Proceedings of the 3rd WSEAS International Conference on Circuits, Systems, Communications and Computers.* WSES Press, 1999.

[11] M. Denker. *Einfuhrung in die Analysis dynamischer Systeme*. Springer, 2005.

[12] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

[13] G. Gallopin. Linkages between vulnerability, resilience, and adaptive capacity. *Global Environmental Change*, 16(3):293–303, 2001.

[14] J. Gibbons. Unfolding abstract datatypes. In *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 110–133. Springer, 2008.

[15] J. Gibbons and G. Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, 2005.

[16] J. Gibbons and G. Jones. The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34(1), pages 273–279. ACM Press, 1998.

[17] R. Goldblatt. *Topoi, The Categorial Analysis of Logic*. Dover Publications, Inc., 2006.

[18] J. Hughes. QuickCheck: An automatic testing tool for haskell. User manual avaiable online at http://www.cs.chalmers.se/ rjmh/QuickCheck/manual.html.

[19] IPCC. Impacts, adaptation, and vulnerability: Contribution of working group II to the fourth assessment report of the intergovernmental panel on climate change. In M.L. Parry, Canziani O.F., J.P. Palutikof, P.J. van der Linden, and C.E. Hanson, editors, *Climate Change 2007*, page 976 pp., Cambridge, UK, 2007. Cambridge University Press.

[20] B. Jacobs. Introduction to Coalgebra. Towards Mathematics of State and Observations. In preparation. Current version available at http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf.

[21] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[22] Mark P. Jones and Luck Duponcheel. Composing monads. Technical report, 1993.

[23] A. Kurz. Coalgebras and modal logic. In *Proceedings of Advances in Modal Logic '98*, pages 222–230. Uppsala, 1998.

[24] R. Leichenko, K. O'Brien, G. Aandahl, H. Tompkins, and A. Javed. Mapping vulnerability to multiple stressors: A technical memorandum. Technical report, CICERO, Oslo, Norway, 2004.

[25] D. Lemmen and F. Warren, editors. *Climate Change Impacts and Adaptation: A Canadian Perspective.* Natural Resources Canada, 2004.

[26] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco,California,January 22–25,1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[27] D. Lincke, C. Ionescu, and N. Botta. A generic library for earth system modelling based on monadic systems. Technical report, Potsdam Institute for Climate Impact Research, Potsdam, Germany, 2008.

[28] A.L. Luers, D.B. Lobell, L.S. Sklar, C.L. Addams, and P.A. Matson. Method for quantifying vulnerability, applied to the agricultural system of the Yaqui Valley, Mexico. *Global Environmental Change*, 13(4):255–267, 2003.

[29] M.J. Metzger and D. Schröter. Towards a spatially explicit and quantitative vulnerability assessment of environmental change in europe. *Regional Environmental Change*, 6(4):201–216, 2006.

[30] N. Nakicenovic and R. Swart. IPCC special report on emission scenarios (SRES). Technical report, Intergovernmental Panel on Climate Change, 2000.

[31] K. O'Brien, R. Leichenko, U. Kelkar, H. Venema, G. Aandahl, H. Tompkins, A. Javed, S. Bhadwal, S. Barg, L. Nygaard, and J. West. Mapping vulnerability to multiple stressors: Climate change and globalization in india. *Global Environmental Change*, 14(4):303–313, 2004.

[32] M. Parry, O. Canziani, J Palutikof, P. van der Linden, and C. Hanson, editors. *Climate Change 2007: Impacts, Adaptation and Vulnerability. Contritbution of Working Group II to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change.* Cambridge University Press, 2007.

[33] S. Peyton Jones and J.-M. Eber. How to write a financial contract. In *The Fun of Programming*, Cornerstones of Computing, pages 105–129. Palgrave, 2003.

[34] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[35] C. Soanes and A. Stevenson, editors. *Oxford Dictionary of English.* Oxford University Press, second edition (revised) edition, 2005.

[36] K. Thywissen. Components of risk, a comparative glossary. *SOURCE - Studies Of the University: Research, Counsel, Education*, 2, 2006.

[37] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[38] R.T. Watson, M.C. Zinyowera, and R.H. Moss, editors. *Climate Change 1995: Impacts, Adaptations and Mitigation of Climate Change: Scientific-Technical Analyses Contribution of Working Group II to the Second Assessment of the Intergovernmental Panel on Climate Change.* Cambridge University Press, 1995.

[39] The World Bank. *World Development Report 2000/2001. Attacking Poverty.* Oxford University Press, 2001.

Zusammenfassung

In der vorliegenden Arbeit wurde ein mathematisches Modell von Vulnerabilität und verwandten Begriffen (Sensitivität und Anpassungsfähigkeit) eingeführt. Die genannten Begriffe sind zentrale Bestandteile der Fachgebiete "Globaler Wandel" und "Klimafolgenforschung" und werden hier entsprechend verwendet.

Es wurde gezeigt, dass verschiedene repräsentative Definitionen Spezialfälle dieses allgemeinen Modells sind, dies zeigt, dass es die Möglichkeit einer mathematischen Metaanalyse von Vulnerabilitätsassessments gibt.

Der Bedarf für ein allgemeines Model begründet sich unter anderem aus der Tatsache, dass es viele verschiedene Definitionen für Vulnerabilität in der Literatur gibt. Diese Vulnerabilitätsdefinitionen sind im Allgemeinen nur auf einen bestimmten Typ eines dynamischen Systems zugeschnitten: deterministisch, nichtdeterministisch (szenariengesteuert), stochastisch, fuzzy, usw. Um die verschiedenen Definitionen zu vereinheitlichen wurde die Klasse der monadischen dynamischen Systeme identifiziert. Diese Klasse beinhaltet alle vorher erwähnten Systeme, sowohl in kontinuierlicher, als auch in diskreter Zeit, mit und ohne Input. Das heißt, eine Vulnerabilitätsdefinition, welche sich auf allgemeine monadische Systeme bezieht, kann man auf alle Systeme eines üblichen Typs anwenden. Spezifische Definitionen erhält man durch die Auswahl einer Monade, eines Zustandtyps, einer Schadensfunktion und einer Funktion für die Vulnerabilitätsabschätzung. Es wurden Bedingungen für die Konsistenz von Vulnerabilitätsabschätzungen und für die Kompatibilität der Vulnerabilitätsabschätzungen, die auf unterschiedlichen Typen von Systemen definiert wurden, formuliert.

Operationen wurden auf monadischen dynamischen Systemen definiert. Die wichtigsten Operationen sind allgemeine Funktionen für die Berechnung von Trajektorien solcher Systeme und Operationen, die aus zwei oder mehreren monadischen Systemen ein neues monadisches System erzeugen.

Das Vulnerabilitätsmodell und die Kombinatoren monadischer Systeme wurden in der funktionalen Programmiersprache Haskell implementiert. Die Verwendung der Kombinatoren wurde exemplarisch in einem vereinfachten Modell dargestellt, welches die praxisbezogenen Probleme von Vulnerabilitätsassessments widerspiegelt. Es wurde gezeigt wie man die vorher erwähnten Konsistenz- und Kompatibilitätsbedingungen automatisch testen kann.

Erklärung

Hiermit versichere ich, dass ich die Dissertation selbstständig verfasst habe und dass ich keine anderen als die in der Arbeit genannten Hilfsmitteln und Quellen benutzt habe.

Berlin, den 15.11.2008 Cezar Ionescu