The Recursion Schemes of Scientific Models
A Multi-Paradigm Study of the Logistic Map in Haskell

Baltasar Trancón y Widemann
Computer Science and Ecological Modelling
University of Bayreuth, Germany
Baltasar.Trancon@uni-bayreuth.de

The Squiggol approach to recursive algorithms separates computation and recursion scheme. We revisit the idea in a setting of simple but paradigmatic problems in scientific, particularly ecological, modelling. It is demonstrated that each stereotypic modelling task corresponds to a simple and well-understood recursion pattern, and vice versa. The logistic map is used as the running example of a scientific model. The Haskell implementation of the recursion framework and its applications is discussed. Its benefits from the perspective of application, compared with a previous set-theoretic presentation, are numerous: Static typing and higher-order functions promote abstraction and reuse, whereas immediately executable models make interactive approaches, adequate for modelling living systems, easy and convenient. From the perspective of functional programming, the general logic of scientific modelling is summarized in a concise and familiar language, blazing the trail for truly functional, real-world, non-number-crunching, scientific computing. The material presented here can be and has been used to instruct prospective natural scientists on mathematics they are usually not tought, but which are indespensable for understanding the logic of models, most notably recursion theory. It can also be used to demonstrate to functional programmers the deep and scarcely recognized empirical and epistemological implications of both dedicated simulation programs and very abstract formal algorithms.

1 Introduction

1.1 Motivation

Suppose you are a natural scientist and you are considering a scientific model, that is, a collection of mathematical concepts supposedly put together to correspond in a meaningful way to empirical data about the natural world. Which questions can and should you ask of your model in order to validate it and to learn from it? In [8], we have demonstrated how to organize a wide variety of modelling scenarios in a theoretical framework of (co)algebraic specification, expanding on an idea of the theoretical biologist Rosen [13] that pictures the model–data relationship as a homomorphism. In particular, different modelling scenarios are mapped to unique homomorphisms from initial algebras or to final coalgebras of a certain class of simple signature functors.

Now suppose that you are also a functional programmer. Can you leverage your exquisite skills to implement such (co)algebraically structured models in a disciplined way, thus allowing for their validation both statically by inspection and type checking, and dynamically by simulation, perhaps with greater clarity than a mere mathematically-minded scientist? Here, I shall argue that the answer is "yes, certainly!" To this end, I shall employ the Squiggol approach to recursive algorithms, in particular the concepts of *catamorphism* and *anamorphism* or, in the terminology of [12], *bananas* and *lenses*, respectively. In a fairly generic Haskell framework of (co)algebras and homomorphisms, implementations of the initial algebras and final coalgebras of the signature functors in question shall be given. Then

Submitted to:
TFPIE 2012

we shall use the logistic map as the running example of a scientific model, in this case from population biology, and implement each of the posed modelling problems in a single phrase of utterly simple, non-recursive Haskell code that specifies a cata- or anamorphism. The source code of this article is available[1] as executable, self-contained Literate Haskell code. It also comprises a few auxiliary modules and demonstrations, especially visualizations, which must be omitted here for the sake of brevity.

1.2 Context and Organization of This Article

Why would you resort to the theory of universal (co)algebra, deeply rooted in the obscure field of category theory and associated with suspicious concepts such as non-well-founded sets, when science has apparently thrived in blissful ignorance for centuries?

As a natural scientist, unless you happen to work in quantum physics and deal with Hopf algebras, you are not likely to have ever had any contact with either coalgebras or categories. However, high-level theoretical work in areas from physics [5] to biology [13] has long since discovered the homomorphic nature of scientific modelling. Writing down homomorphisms in an abstract notation that works by drawing lots of arrows is arguably a good idea. But there is more to it: We have argued in [8] that the fundamental metaphysical distinction and duality between structural and behavioural modelling approaches embodied in universal algebra and coalgebra, respectively, which has been worked out in great detail by Rutten [14], Adámek [1] and others, carries over nicely to empirical sciences. There it helps to recognize the duality of two modelling paradigms, termed *functional* and *interactive*, and to organize the notions of empirical reference, validation and logic pertaining to each.

As a functional programmer, you are far more likely to know category theory, and how universal (co)algebra relates to your discipline. Perhaps you are familiar with the introductions of [12] or [2] to a generic framework of data types and recursive algorithms, organized in terms of signature functors, initial algebras, final coalgebras and various homomorphisms, potentially enclosed in ornamental brackets. If you are, then Sects. 2 and 3 will just be a gentle reminder to you. Otherwise, nothing but sound working knowledge of Haskell is required to appreciate the individual definitions, although you might want to consult the cited literature for the bigger picture or for a gentler introduction than the one given below.

The running example for all applications shall be the *logistic map*, a family of polynomials $f_r(x) = rx(1-x)$ with a real-valued parameter $r > 0$. It is widely studied both as a simple model of biological populations [10], the discrete counterpart of the continuous logistic growth model of Verhulst, and as a dynamical system, prized for the richness of its bifurcating and chaotic behaviour [11]. Under both perspectives, not the function itself (a plain upside-down parabola with roots $x = 0$ and $x = 1$) is studied, but its *trajectories*; sequences arising from the recurrence relation $x_{n+1} = f_r(x_n)$. In settings of *symbolic dynamics*, the trajectories may even be mapped elementwise to a finite alphabet, making the resulting words and their statistics the objects of study. A variety of modelling scenarios, dealing with both structure and behaviour of this toy system, shall be studied within the (co)algebraic framework.

To this end, Sect. 2 reviews the necessary basic notions of universal (co)algebra and their Haskell implementation. Sect. 3 introduces the class of functors under study and provides intuitive implementations of their initial algebras and final coalgebras, each associated with a (co)recursion scheme. Sect. 4 implements the logistic map and certain secondary computations, and uses these building blocks to implement each of the particular problems discussed in [8], and a few new ones, in terms of elementary Haskell expressions that refer to one of the previously elaborated schemes. Sect. 5 summarizes the available experience on classroom applications of the presented material and discusses future potential.

[1] http://www.bayceer.uni-bayreuth.de/mod/de/top/dl/94448/hasklog.zip

1.3 Words of Warning

As either a functional programmer or natural scientist, you are possibly in danger of falling into the trap of formalism—focussing on the equations in an article and ignoring the paragraphs in between. In that case, you are bound for disappointment here: The programming content is really elementary and hardly ground-breaking throughout. What matters is the discovery that a wide range of problems in a highly relevant application domain can be covered and organized by systematic exploration and clever interpretation of a very small set of elementary code fragments. Some particular concerns arising from formalistic disappointment are anticipated:[2]

1. *The (co)algebra framework is not necessary for the application.* It will be established as *useful* by discussing modelling scenarios that have an inherent catamorphic or anamorphic structure, respectively, and are hence more naturally related in the (co)algebra framework than in any other meta-theory. It will also be established as *sufficient* in the sense that the recursion scheme of all discussed scenarios is extracted completely, leaving the computational content in a simple and standardized form that can be easily understood and compared.
2. *The programs are unsurprising; I could have written them easily.* This should be regarded as evidence that skilled functional programmers could contribute greatly to the area of scientific computing, which is so far dominated by numerics and has much potential of development regarding logic and structure.
3. *Trajectories are merely lists; why not just use the Haskell list library?* It is true that the recursion schemes discussed below are already available as library functions. The focus here is on the observation that they are the *complete* set of solutions for a simple type-combinatorial puzzle, and on their duality relationships and uniform presentation.

Welcome to a special guided tour of the world of scientific models for functional programmers.

1.4 Educational Background

It is the proposition of this article to join functional programmers and scientific modelers in an interdisciplinary venture and discussion. Functional languages have so far rather little to contribute to the thinking and real-world programming of "hard" science, which is stubbornly loyal to the Fortran world. Functional presentations of algorithms are possibly recognized for their elegance and good parallelizability, but only in the role of operational tools. Only very recently, the approach of functional reactive programming [9] has produced concepts worth also for empirical scientists and philosophers of science thinking about, but the language mismatch between *program semantics* on the one hand and *model epistemology* on the other is tremendous and makes interdisciplinary communication and understanding extremely difficult.

The author of this article has had the privilege to work with a theoretician of scientific, in particular ecological, modeling for a couple of years.[3] In fact, he has been hired for an interdisciplinary project because of his partner's intuition that computer science might provide illuminating theory of scientific modeling of complex systems. The material presented here is therefore part of our effort to prove that it is so, in particular with respect to the fundamentals of recursion in functional programming. The results have so far been tried publicly on three years of students in ecology, and privately on a few functional

[2] In fact, they are quoted almost literally from an anonymous review of a previous version of this article.

[3] The first person plural shall refer to that team throughout this article.

programmers. We conclude that the approach is viable, and would like to invite and encourage others to base their own interdisciplinary efforts on our experience. To that end, we present in this article a self-contained, sound, fully executable, and hopefully convincing story of functional programming as a key to the logical and philosophical mysteries of scientific modeling. The bulk of this article is dedicated to storytelling; the reader interested in our classroom experience only may want to skip ahead to Sect. 5.

2 Universal Algebra and Coalgebra

This section reviews the basics of universal (co)algebraic recursive programming in Haskell. The reader familiar with the Squiggol approach will recognize the techniques underlying the infamous banana and lenses operators of [12].

```
module CoAlgebra where
```

2.1 Endofunctors

Consider a suitable category of Haskell types and pure, partial, potentially non-strict Haskell functions, for instance $\mathbf{Cppo}_\perp$ [7]. An endofunctor T of this category is a type constructor $t :: * \to *$ such that there is an **instance** *Functor t* that obeys the functor laws, namely a function $fmap :: (a \to b) \to t\ a \to t\ b$ with

$$fmap\ id \equiv id$$
$$fmap\ (f \circ g) \equiv fmap\ f \circ fmap\ g$$

2.2 Algebras

A T-algebra is a function $f :: t\ a \to a$, for some type a called its *carrier*.

```
type Algebra t a = t a → a
```

A T-algebra homomorphism between two T-algebras $f :: Algebra\ t\ a$ and $g :: Algebra\ t\ b$ is a function $h :: a \to b$ such that $h \circ f \equiv g \circ fmap\ h$. We allow for alternative representations of the type of T-algebras. The following type class abstracts from the actual representation type x.

```
class Algebraic t a x where algebra :: x → Algebra t a
instance Algebraic t a (Algebra t a) where algebra = id
```

A weakly initial T-algebra is an algebra $initial :: Algebra\ t\ a$ such that there is a homomorphism *cata* from it to every other T-algebra.

```
class Functor t ⇒ Initial t a | a → t where
  initial :: Algebra t a
  cata    :: Algebra t b → a → b
```

The homomorphism law specializes to $cata\ g \circ initial \equiv g \circ fmap\ (cata\ g)$. [18] has shown, by invoking parametricity, that such a weakly initial algebra is initial proper, in the sense that *cata g* is unique. By Lambek's lemma, the operation *initial* is an isomorphism. Its inverse *laitini* can be implemented in terms of *initial* and *cata* [18], namely as $laitini \equiv cata\ (fmap\ initial)$. If we look at the situation the other way, and instead require an independent implementation as method of the type class *Initial*

```
laitini :: Coalgebra t a
```

that satisfies the laws $initial \circ laitini \equiv id$ and $laitini \circ initial \equiv id$, then we can give a generic implementation of *cata* by applying $(\circ laitini)$ to both sides of the initiality law:

$$cata\ g = h \textbf{ where } h = g \circ fmap\ h \circ laitini$$

2.3 Coalgebras

The definitions of T-coalgebras and final T-coalgebras are completely dual.

```
type Coalgebra t a = a → t a
class Coalgebraic t a x
  where coalgebra :: x → Coalgebra t a
instance Coalgebraic t a (Coalgebra t a)
  where coalgebra = id
```

```
class Functor t ⇒ Final t a | a → t where
  final  :: Coalgebra t a
  lanif  :: Algebra t a
  ana    :: Coalgebra t b → b → a
  ana g = h where h = lanif ∘ fmap h ∘ g
```

The functions *final* and *lanif* are assumed to be inverses of each other, that is $final \circ lanif \equiv id$ and $lanif \circ final \equiv id$. Thanks to laziness, the definition of *ana* can be productive even for infinite data structures if the recursive invocations of h are not forced by *fmap* or *lanif*, for instance by just passing them to a non-strict data constructor.

2.4 Generic Implementation

The Haskell datatype mechanism allows a free generic implementation that doubles as initial algebra and final coalgebra of certain well-behaved functors. The exact condition is not relevant here; we shall consider only functors that are well-behaved in the required sense in the next section.

```
newtype Fix t = In {out :: t (Fix t)}
```

```
instance Functor t ⇒ Initial t (Fix t) where
  initial = In
  laitini = out
```

```
instance Functor t ⇒ Final t (Fix t) where
  final  = out
  lanif  = In
```

Note that **newtype** is used instead of **data**, to the effect that both *In* and *out* are implemented as identity. Hence the inversion and productivity requirements are trivially satisfied. The desired effect of separating the recursion scheme and content of an algorithm is substantiated by the following observation:

> The definitions of *Fix*, *cata* and *ana* are the **only** and **adequate** instances of explicit recursion required to solve all the problems posed in this article!

3 Affine Functors

This section defines a particular, polymorphic class of endofunctors and gives more concrete implementations for the initial algebras and final coalgebras of special cases; cf. the categorical datatypes of [6]. By restricting the focus to instances where one of the class parameters is a trivial type, we obtain a small system of well-known elementary recursion schemes. It shall be demonstrated in the next section that they have a surprisingly wide range of applications in the domain of scientific modelling.

```
module Affine where
import CoAlgebra
import Data.List
```

This module uses a type *Natural* of unbounded natural numbers (the nonnegative subset of *Integer*).

```
import Natural
```

The class of affine functors in the category of sets is defined in [8] as $\mathscr{A}_B^A = (A \times -) + B$, for arbitrary sets A, B. We write *Go* and *Stop* for the injections, respectively.

```
data Aff a b x = Go !a x | Stop !b
```

The parameter positions a and b are made strict in the Haskell implementation, in order to make it behave more similar to the set-theoretic presentation. Note that, in contrast, the functor argument position x is explicitly non-strict to cater for a final coalgebra. The type constructor is extended to a functor in the obvious way.

```
instance Functor (Aff a b) where
  fmap f (Go a x) = Go a (f x)
  fmap f (Stop b) = Stop b
```

3.1 Generic Initial Algebra and Final Coalgebra

An invertible initial algebra can be given in terms of more familiar type constructors.[4]

```
instance Initial (Aff a b) ([a],b) where
  initial (Go a (as,b)) = (a:as,b)
  initial (Stop b)      = ([],b)
  laitini (a:as,b)      = Go a (as,b)
  laitini ([],b)        = Stop b
```

In the category of sets and total functions, the initial algebra is a proper substructure of the final coalgebra. In the category of Haskell types and functions, they are the same.

```
instance Final (Aff a b) ([a],b) where
  lanif = initial
  final = laitini
```

Note that the productivity requirement for *ana* is fulfilled by this definition.

3.2 Specific Cases

We obtain simpler cases of special interest by substituting the unit type () or an empty type (containing only $\perp$) for either a or b.

```
data Empty
```

[4] To be pedantic, a list type that is strict in its *head* argument would be required instead of the non-strict []. We silently assume that lists never contain $\perp$ elements.

3.2.1 The case $a = ()$

The case $a = ()$ has an initial algebra with the carrier $(Natural, b)$, by reading $[()]$ as a unary number system.

```
instance Initial (Aff () b) (Natural,b) where
  initial (Go () (n,b)) = (n+1,b)
  initial (Stop b)      = (0,b)
  laitini (0,b)         = Stop b
  laitini (n+1,b)       = Go () (n,b)
```

Algebras of this sort with carrier c can be represented as pairs of $f :: c \to c$ and $g :: b \to c$, playing the role of *Go* and *Stop*, respectively.

```
data Iter b c = Iter {igo :: c -> c, istop :: b -> c}
instance Algebraic (Aff () b) c (Iter b c) where
  algebra (Iter f g) (Go () x) = f x
  algebra (Iter f g) (Stop b)  = g b
```

The unique homomorphism from an initial algebra into algebras in *Iter* representation is an iteration operator.

```
iter :: (Initial (Aff () b) q) => Iter b c -> q -> c
iter = cata . algebra
```

Specialized to the initial algebra given above, it has the following property:

$$iter\ (Iter\ f\ g)\ (n,b) \equiv f^n (g\ b)$$

Iteration can be performed analogously in the Kleisli category of a monad.

```
data IterM m b c = IterM {igoM :: c -> m c, istopM :: b -> m c}
instance (Monad m) => Algebraic (Aff () b) (m c) (IterM m b c) where
  algebra (IterM f g) (Go () x) = f =<< x
  algebra (IterM f g) (Stop b)  = g b
iterM :: (Initial (Aff () b) q, Monad m) => IterM m b c -> q -> m c
iterM = cata . algebra
```

3.2.2 The case $b = ()$

The case $b = ()$ has an initial algebra with the carrier $[a]$, by dropping the second component, which is redundant because of strictness.

```
instance Initial (Aff a ()) [a] where
  initial (Go a as) = a : as
  initial (Stop ()) = []
  laitini []        = Stop ()
  laitini (a : as)  = Go a as
```

Algebras of this sort with carrier c can be represented by pairs of $f :: a \to c \to c$ and $e :: c$, alluding to the arguments of the operation *foldr* on (finite) lists.

```
data Fold a c = Fold {fgo :: a -> c -> c, fstop :: c}
```

```
instance Algebraic (Aff a ()) c (Fold a c) where
  algebra (Fold f e) (Go a x) = f a x
  algebra (Fold f e) (Stop ()) = e
```

The unique homomorphism from an initial algebra into algebras in *Fold* representation is a fold operator.

```
fold :: (Initial (Aff a ()) p) ⇒ (Fold a c) → p → c
fold = cata ∘ algebra
```

Specialized to the initial algebra given above, it is a well-known list function.

```
fold (Fold f e) ≡ foldr f e
```

3.2.3 The case $b = Empty$

The case $b = Empty$ has an empty initial algebra in the category of sets, but a final coalgebra of streams (infinite lists). In Haskell, the carrier of both is a type *Stream a*. Note that the *Stop* case of *Aff* is unreachable because of strictness.

```
data Stream a = (▷) !a (Stream a)
infixr 5 ▷
instance Final (Aff a Empty) (Stream a) where
  final (a▷as)   = Go a as
  lanif (Go a as) = a▷as
```

Coalgebras of this sort with carrier c can be represented by pairs of $h :: c \to a$ and $t :: c \to c$, alluding to the operations *head* and *tail* on (infinite) lists.

```
data Unfold a c = Unfold {ugo :: c → a, ustop :: c → c}
instance Coalgebraic (Aff a Empty) c (Unfold a c) where
  coalgebra (Unfold h t) x = Go (h x) (t x)
```

The unique homomorphism from a coalgebra in *Unfold* representation to a final coalgebra is an unfold operator.

```
unfold :: (Final (Aff a Empty) t) ⇒ Unfold a c → c → t
unfold = ana ∘ coalgebra
```

It can be specified using the dual of the list function *foldr*.

```
unfold (Unfold h t) ≡ list2stream ∘ unfoldr uncons
  where uncons x = Just (h x, t x)
```

A specification using more well-known functions is as follows.

```
unfold (Unfold h t) ≡ list2stream ∘ map h ∘ iterate t
```

The conversion *list2stream* is itself an unfolding.

```
list2stream :: [a] → Stream a
list2stream = unfold $ Unfold head tail
```

```
stream2list :: Stream a → [a]
stream2list = unfoldr f
  where f (h▷t) = Just (h,t)
```

3.2.4 The case $a = Empty$

The case $a = Empty$ is degenerate: Both *Go* and *Stop* are strict in their first argument, hence *Aff Empty b* is trivially isomorphic to b.

4 The Logistic Map

This section is structured along the lines of [8, Sect. 4]. Formal modelling scenarios are mapped onto the (co)algebraic framework and exemplified with the logistic map.

```
module Logistic where
import Control.Monad ((<=<))
import CoAlgebra
import Affine
import Natural
```

This module additionally uses a type *CDF* of continuous distribution functions.

```
import CDF           (CDF (..), eval)
import CDF.UInterval (subinterval, unit)
```

The logistic map is defined as expected, but restricted to the interval $[0,1]$ where its value is nonnegative. It is confined in this interval and hence total for $r \leqslant 4$, and surjective and hence (ambiguously) invertible for $r \geqslant 4$.

```
type Param = Double
type State = Double
type Step a = Param → a → a

logistic           :: Step State
logistic r x | r>0 = let y = r*x*(1-x)
                     in  if 0 ≤ y ∧ y ≤ 1 then y else error "out of range"
```

The definitions in this section use only a handful of stereotypic type signatures, *Step* being the first example. The others shall be introduced as we go, when their first instance arises.

As the primordial example of a map to a finite alphabet for the purpose of symbolic dynamics, consider the binary partition that halves the interval $[0,1]$. It is complementary to the logistic map in the sense that each preserves exactly the information the other destroys, and their pairing is injective and hence uniquely invertible.

```
color :: State → Bool
color x = x ≥ 0.5
```

The inverse of the logistic map has two symmetrical branches. We define an auxiliary function *loginv* that computes the left branch and another, *logbranch*, that selects a branch by colour.

```
loginv     :: Step State
loginv r y = 1/2 - sqrt (1/4 - y/r)

type Obs a = Param → State → a

logbranch       :: Obs (Bool → State)
logbranch r y a = if a then 1 - loginv r y else loginv r y
```

The function *logbranch* is the inverse of the pairing *logistic* and *color* in the following sense:

$$logbranch\ r\ (logistic\ r\ x)\ (color\ x) \equiv x$$

4.1 Direct Functional Modelling

Assuming that both the dynamics and the current state of the system under study is known, the direct approach to functional modelling addresses problems of *prediction*. Information about future, yet unobserved states is inferred from information about current or past, observed states. We have made the claim in [8] that the paradigm of direct functional modelling is the functor $Aff\ ()\ b$ for some type b that encodes predicates on the state space, together with *Iter* as an encoding of algebraic queries.

4.1.1 Perfect Information

The basic prediction problem is solved by iteration of the step function of the system applied to an initial state.

$$\textbf{type}\ Predict\ a = Param \to Natural \to a \to a$$
$$predict\ \ :: Predict\ State$$
$$predict\ r = curry\ \$\ iter\ \$\ Iter\ (logistic\ r)\ id$$

Given a single initial state x, the expression *predict r n x* computes the n-th successor state, according to the logistic map with parameter r.

$$\textbf{type}\ Future\ a = Param \to Natural \to State \to a$$
$$predictTraj\ \ :: Future\ [State]$$
$$predictTraj\ r = curry\ \$\ iter\ \$\ Iter\ (push\ \$\ logistic\ r)\ return$$
$$\quad \textbf{where}\ push\ f\ s = f\ (head\ s) : s$$

Given a single initial state x, the expression $t = predictTraj\ r\ n\ x$ computes a reverse trajectory of x, according to the logistic map with parameter r. Namely, $reverse\ t\ !!\ k$ is the k-th successor state.

More complex prediction problems can be solved in the same way, by lifting the step function, here the logistic map, to a data type that describes coarser information about the initial state.

4.1.2 Imperfect Information: Nondeterminism

The obvious mathematical data structure to describe coarse state information is the powerset of the domain $[0,1]$. In Haskell we will have to make do with restricted, but computable classes of subsets. As an example, we calculate the lifting of the logistic map to closed intervals, represented by their end points:

$$\textbf{type}\ Interval = (State, State)$$
$$logimg :: Step\ Interval$$
$$logimg\ r\ (x1,x2) = (z1,z2)$$
$$\quad \textbf{where}\ [y1,y2] = map\ (logistic\ r)\ [x1,x2]$$
$$\qquad z1 \quad = min\ y1\ y2$$
$$\qquad z2 \quad = \textbf{if}\ color\ x1 \equiv color\ x2\ \textbf{then}\ max\ y1\ y2\ \textbf{else}\ r\ /\ 4$$

(Draw the graph of the logistic map to see how this definition works.) The lifted step function simply replaces the unlifted one in the specification of iteration.

```
predictIvl   :: Predict Interval
predictIvl r = curry $ iter $ Iter (logimg r) id
```

Given interval bounds (a,b), the expression *predictIvl r n* (a,b) computes the unique interval (c,d) such that $c \leqslant y \wedge y \leqslant d$ for the n-th successor state $y = predict\ r\ n\ x$, if and only if $a \leqslant x \wedge x \leqslant b$. It can demonstrate how, in the chaotic regime, minuscule intervals are stretched very quickly; for instance:

$$predictIvl\ 4\ 21\ (0.33, 0.330001) \equiv (0.0, 1.0)$$

4.1.3 Imperfect Information: Probabilism

As an example of a nontrivial description structure and lifting, consider the space of continuous probability distributions on $[0,1]$. The lifting of a distribution p, given by its support (a possibly unbounded interval) and cumulative distribution function $eval\ p :: Double \to Double$, can be calculated as follows; see [8] for proof.

```
logcdf :: Step CDF
logcdf r p | support p `subinterval` unit = CDF unit h
  where h y = eval p (1 / 2 - q y) + 1 - eval p (1 / 2 + q y)
        q y = sqrt (max (1 / 4 - y / r) 0)
predictCdf   :: Predict CDF
predictCdf r = curry $ iter $ Iter (logcdf r) id
```

Given an initial distribution p, the expression *predictCdf r n p* computes the state distribution after n steps. With a little auxiliary graphical code, the convergence towards stable distributions, for instance $\mathrm{Beta}(1/2, 1/2)$ in the case $r = 4$, can be visualized.

4.2 Inverse Functional Modelling

If either the exact dynamics or the current state of the system under study is not known, but there is data specifying the future states, an inverse modelling problem is posed. Information about model parameters or the current state are inferred from observations about future states. For empirical applications, the observations are typically imprecise and do not specify states uniquely. We have claimed in [8] that the paradigm of inverse functional modelling is the functor *Aff a* () for some observation type a, together with *Fold* as an encoding of algebraic queries.

As an example of data specifying future states, consider the reverse trace of colourings of a trajectory with a given length.

```
logobs             :: Future [Bool]
logobs r 0       x = []
logobs r (n + 1) x = map color $ predictTraj r n x
```

We consider only the case of inferring the current state. To this end, we define a partial variant of the step function that checks for consistency with some observation, in this case the binary colouring.

```
type Partial a b = Param → a → b → Maybe b
logifcol                    :: Partial Bool State
logifcol r a x | color x ≡ a = Just (logistic r x)
               | otherwise   = Nothing
```

The partial step function can be lifted, using the Kleisli composition of the *Maybe* monad, to an algebra that folds colouring data.

$$\begin{aligned}&\mathit{solve} \quad :: \mathit{Partial}\ [\mathit{Bool}]\ \mathit{State}\\&\mathit{solve}\ r = \mathit{fold}\ \$\ \mathit{Fold}\ ((<=<) \circ \mathit{logifcol}\ r)\ \mathit{Just}\end{aligned}$$

Given a finite list of colourings w of length n, the expression $\mathit{solve}\ r\ w\ x$ yields $\mathit{Just}\ (\mathit{predict}\ r\ n\ x)$ if and only if $\mathit{logobs}\ r\ n\ x \equiv w$, and *Nothing* otherwise. Again, the class of subsets of $[0,1]$, in this case the computable subsets, is too large to be directly useful. Apart from heuristic probing, we can just state the "if" part of the above statement in a form amenable to testing, for instance with QuickCheck [3]:

$$\begin{aligned}&\mathit{solvePositive} :: \mathit{Future}\ \mathit{Bool}\\&\mathit{solvePositive}\ r\ n\ x = \mathit{solve}\ r\ w\ x \equiv \mathit{Just}\ (\mathit{predict}\ r\ n\ x)\\&\quad \textbf{where}\ w = \mathit{logobs}\ r\ n\ x\end{aligned}$$

A more effective class of subsets is again the closed intervals. We can give an inverse of *logimg* that calculates either half of the preimage of an interval, selected by colouring.

$$\begin{aligned}&\mathit{logpre} :: \mathit{Partial}\ \mathit{Bool}\ \mathit{Interval}\\&\mathit{logpre}\ r\ a\ (y1,y2) \mid y1 > r\,/\,4\ = \mathit{Nothing}\\&\qquad\qquad\qquad\quad \mid \mathit{otherwise} =\\&\quad \textbf{let}\ x1 = \mathit{loginv}\ r\ y1\\&\qquad x2 = \mathit{loginv}\ r\ (\mathit{min}\ y2\ (r\,/\,4))\\&\quad \textbf{in}\ \ \textbf{if}\ a\ \textbf{then}\ \mathit{Just}\ (1-x2, 1-x1)\ \textbf{else}\ \mathit{Just}\ (x1,x2)\end{aligned}$$

This function allows us to lift the inverse problem to intervals.

$$\begin{aligned}&\mathit{solveIvl} :: \mathit{Partial}\ [\mathit{Bool}]\ \mathit{Interval}\\&\mathit{solveIvl}\ r = \mathit{flip}\ \$\ \mathit{fold} \circ \mathit{Fold}\ ((=\!\ll) \circ \mathit{logpre}\ r) \circ \mathit{Just}\end{aligned}$$

Given a finite list of colourings w of length n and interval (a,b), the expression $\mathit{solveIvl}\ r\ w\ (a,b)$ yields the unique interval (c,d) such that $c \leqslant y \wedge y \leqslant d$, if and only if $y \equiv \mathit{predict}\ r\ n\ x$ for some x with $a \leqslant x \wedge x \leqslant b$ and $\mathit{logobs}\ r\ n\ x \equiv w$.

4.3 Interactive Modelling

So far we have only considered functional approaches to modelling where state is studied as the cause of behaviour. Now the perspective is turned around, thus enabling behavioural and interactive approaches. Not surprisingly, this move coincides with the move from algebra (*cata*) to coalgebra (*ana*). We have claimed in [8] that the paradigm of behavioural modelling is the functor *Aff a Empty* for some observation type a, together with *Unfold* as an encoding of coalgebraic evaluation.

$$\begin{aligned}&\mathit{classify} \quad :: \mathit{Obs}\ (\mathit{Stream}\ \mathit{Bool})\\&\mathit{classify}\ r = \mathit{unfold}\ \$\ \mathit{Unfold}\ \mathit{color}\ (\mathit{logistic}\ r)\end{aligned}$$

For $r \leqslant 4$, the expression $\mathit{classify}\ r\ x$ unfolds to an infinite binary stream (for $r > 4$, it eventually aborts with an out-of-range error almost surely). The resulting objects can then be studied for similarity according to finite prefixes of those streams, corresponding to exponentially shrinking neighbourhoods for some suitable metric. This modelling approach, true to the paradigm of symbolic dynamics, can be found in [14], presented as a coalgebraic specification in the category of complete metric spaces. It is by generalization of that idea that our work in [8] has been conceived.

By simply running the dynamics backwards, the stream of colourings is turned from an output of an *autonomous* system to the input of a *reactive* system, more adequate for living systems that are mostly governed by the (re)actions of their inhabitants.

$$\begin{aligned}&\mathit{offline} \quad :: \mathit{Obs}\ (\mathit{Stream\ Bool} \to \mathit{Stream\ State})\\&\mathit{offline}\ r = \mathit{curry}\ \$\ \mathit{unfold}\ \$\ \mathit{Unfold\ fst\ back}\\&\quad \textbf{where}\ \mathit{back}\ (y, a \triangleright as) = (\mathit{logbranch}\ r\ y\ a, as)\end{aligned}$$

The above model introduces an external decision-making entity, but is not yet fully interactive, because there is no provision for feedback of the outcome of past decisions into the stream of future decisions; hence the name *offline*. For a fully interactive recursion scheme, a slightly more complicated functor than those considered before is needed. The reader may recognize it as the signature of Moore automata with input type u and output type i. We shall use it to model a two-player game from the perspective of the first player.

$$\begin{aligned}&\textbf{data}\ \mathit{Game\ u\ i\ x} = \mathit{Game}\ \{\mathit{imove} :: i, \mathit{iwait} :: u \to x\}\\&\textbf{instance}\ \mathit{Functor}\ (\mathit{Game\ u\ i})\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Game\ m\ w}) = \mathit{Game\ m}\ (f \circ w)\end{aligned}$$

Truly interactive modelling scenarios, as required for *adaptive* living systems, can be expressed as *hylomorphisms* (*ana* followed by *cata*, or *envelopes* in the nomenclature of [12]) of this functor. Note that, unlike the previous subsections, this technique requires the coincidence of initial algebra and final coalgebra; it is sound in non-strict Haskell but not in plain set theory.

$$\begin{aligned}&\mathit{online} \quad :: \mathit{Obs}\ (\mathit{Fix}\ (\mathit{Game\ Bool\ State}))\\&\mathit{online}\ r = \mathit{ana}\ \mathit{iplay}\\&\quad \textbf{where}\ \mathit{iplay}\ y = \mathit{Game}\ y\ (\mathit{logbranch}\ r\ y)\end{aligned}$$

The expression *online r y* denotes a game where the first player announces the post-state y and moves to one of the two pre-states according to *logbranch r y a*, depending on the input of a colour bit a from the second player. The parameter r remains hidden.

To actually play the game, the action of the second player is wrapped in a monadic computation. The alternating execution of half-turns can be defined as the catamorphism of a single full-turn.

$$\begin{aligned}&\mathit{runGame} \quad :: (\mathit{Monad}\ m, \mathit{Initial}\ (\mathit{Game\ u\ i})\ g) \Rightarrow (i \to m\ u) \to g \to m\ ()\\&\mathit{runGame}\ f = \mathit{cata}\ \mathit{uplay}\\&\quad \textbf{where}\ \mathit{uplay}\ g = f\ (\mathit{imove}\ g) \gg\!= \mathit{iwait}\ g\end{aligned}$$

A simple monadic interface for the second player is console I/O.

$$\begin{aligned}&\mathit{shell} :: (\mathit{Read\ u}, \mathit{Show\ i}) \Rightarrow i \to \mathit{IO\ u}\\&\mathit{shell} = (\gg \mathit{readLn}) \circ \mathit{print}\end{aligned}$$

The expression *runGame shell* \$ *online r y* runs the game. Here is a suggestion how to play it: choose $r = 4$ for maximum freedom of possibilities. Start with some state, say $y = 0.3$, and find a sequence of binary choices that takes you as close to another state, say $x = 0.4$, as possible. (Hint: You might cleverly use some of the previous code to cheat.) For instance, the following sequence of moves will take you within 0.001:

$$[\mathit{False}, \mathit{False}, \mathit{False}, \mathit{True}, \mathit{True}, \mathit{False}, \mathit{True}, \mathit{False}]$$

5 Classroom Experience

The theoretical concepts underlying the material presented here have been tested and proven useful over several years in courses in ecology and environmental informatics at the University of Bayreuth. The ingenious insight of the biologist and philosopher Rosen [13] that scientific modeling is about the *recursive* structure of empirical data, hampered by that author's own limited grasp of theoretical computer science, can be explicated elegantly and effectively by the theory and practice of functional programming.

The duality of algebra and coalgebra offers the additional possibility of addressing natural scientists at two levels of abstraction and intellectual difficulty. At the lower level, the usual reasoning from structure, cause and effect can be mapped to universal algebra and *cata*-style recursion, as a gentle introduction to the world of discrete mathematics that is typically not covered in the scientific curriculum. At the higher level, universal coalgebra and *ana*-style recursion can be contrasted with the above as the discrete mathematics of behavior, serving the twin purposes to replace the prevalent informal and fuzzy notions of behavior by a precise and useful one, and uncovering the intricate dualities of structural and behavioral theories, models, arguments, problems and computations.

The basic concepts, especially of direct and inverse functional modelling, are well-suited even for courses at the bachelor level, such as our own *Introduction to Ecological Modeling*. The *cata*-style recursive organization of data, such as a trajectory predicted from an assumed dynamical law and initial state, or an initial state or parameter extracted from redundancy in observed time series, is grasped easily by students with no formal training the underlying mathematics. It allows them to appreciate complex behavioral features exhibited even by toy models, such as *stability*, *sensitivity* and *equifinality* by means of hands-on exercise calculations.

More advanced concepts, such as the scientific interpretation of *ana*-style recursion and the fundamentals of category theory required to state the duality of the algebraic and coalgebraic approach, can be used with careful guidance in master-level courses, such as our own *Computer-Based Modelling of Ecosystems*. Even if students become neither proficient category theorists nor programmers of recursive algorithms after a single such course, they can be trained to appreciate the algebra-versus-coalgebra distinction in hybrid models such as the seminal Lindenmayer model of vegetative growth, alias *L-Systems*, in elementary terms of philosophy of science such as *causal interpretation of internal structure* versus *unfolding of interface behavior*. Note that, while many well-known epistemological consequences can be deduced from the algebra–coalgebra distinction, it is a rather novel perspective on the theory of scientific modeling, as our own more recent publications [16, 15] show. It provides a systematic framework for modeling methodology that is invaluable for teaching the whole story: even the existence of equally formal and sound alternatives to the functional modeling paradigm, taken for granted in computer science and embodied in established theories such as Bialgebraic Structural Operational Semantics [17], is very hard to communicate in lower-level terms, and hence largely unknown even to fully trained scientific modelers.

The Haskell implementation of the concepts discussed above, even while not given directly to students, has already proven valuable in two ways. Firstly, as a quick and accurate consistency check of both publications and lecture notes, by the simple means of type checking and executable test cases; neither kind of written material would have attained the present level of correctness and maturity without continual hands-on evaluation. Secondly, as a generator of visual demonstrations to highlight and complement the theoretical discussion of selected properties of toy models.

The obvious third step, namely to use the Haskell code directly as lecture material has not yet been tried on students, simply because we dot not have suitable preliminary functional programming, let alone Haskell, courses at all. But individual experience with fellow senior scientist from fields as far as

econometrics has shown that the code itself can be a powerful tool for communicating very abstract ideas to those with merely basic technical knowledge of Haskell, in a more concrete and applicable way than a mathematical treatise such as [8]. We would welcome any opportunity to give a full-fledged course of scientific modeling for functional programmers ourselves, or others to take up the challenge to design one.

6 Conclusion

A wide variety of simple universal modelling problems have been reduced to a seemingly trivial pattern: An invocation of an administrative recursion scheme (such as $curry \circ iter$) applied to a terse specification of algorithmic content (such as $Iter\ (logistic\ r)\ id$). This does not imply that the content is trivial, but rather that the separation of concerns has been accomplished effectively. The resulting style is an elegant, precise and fully executable alternative to the set-theoretic effects commonly achieved with sequences, ellipses and index magic.

It is probably too immodest to announce an era of non-numeric scientific computation, but certainly justified to propose that structured functional programming is a treasure trove of tools even for disciplines with a traditional predilection for number crunching.

The profound change in formal scientific thinking brought about by the study of living and interacting systems has been noticed by many researchers. [4] hits the nail on the head:

> *Mathematics is biology's next microscope, only better;*
> *biology is mathematics' next physics, only better.*

I am confident that an important part of these mathematics is of the conceptual kind that can be expressed in categories and (co)algebras, and implemented in functional programming. But the move from mere possibility to real scientific business will require people trained in both ways of thinking. This article is intended as a motivation and aid to bridge the gap from either side.

Acknowledgments

Thanks to Michael Hauhs for fruitful interdisciplinary collaboration; to Jan Rutten for inspiring writings and encouraging discussions; to Markus Lepper for valuable suggestions; to several anonymous referees for their comments on previous incarnations of this manuscript for various occasions.

References

[1] J. Adámek (2005): *Introduction to coalgebra*. *Theory and Applications of Categories* 14, pp. 157–199.

[2] Richard Bird & Oege de Moor (1997): *Algebra of Programming*. *International Series in Computing Science* 100, Prentice Hall.

[3] K. Claessen & J. Hughes (2000): *Quickcheck: a lightweight tool for random testing of Haskell programs*. In: *Proceedings 5th International Conference on Functional Programming (ICFP 2000)*, ACM, pp. 268–279, doi:10.1145/351240.351266.

[4] J. E. Cohen (2004): *Mathematics is biology's next microscope, only better; biology is mathematics' next physics, only better*. *PLoS Biol* 2(12):e439.

[5] Robert Geroch (1985): *Mathematical Physics*. Chicago University Press.

[6] Tatsuya Hagino (1987): *A Typed Lambda Calculus with Categorical Type Constructors*. In David H. Pitt, Axel Poigné & David E. Rydeheard, editors: *Category Theory and Computer Science*, *Lecture Notes in Computer Science* 283, Springer, pp. 140–157, doi:10.1007/3-540-18508-9_24.

[7] Makoto Hamana (2007): *What is the category for Haskell?* Available at http://www.cs.gunma-u.ac.jp/~hamana/Papers/cpo.pdf. Talk slides.

[8] Michael Hauhs & Baltasar Trancón y Widemann (2010): *Applications of Algebra and Coalgebra in Scientific Modelling, Illustrated with the Logistic Map*. *Electronic Notes in Theoretical Computer Science* 264(2), pp. 105–123, doi:10.1016/j.entcs.2010.07.016.

[9] Paul Hudak (1999): *Functional Reactive Programming*. In S. Swierstra, editor: *Programming Languages and Systems*, *Lecture Notes in Computer Science* 1576, Springer, p. 67, doi:10.1007/3-540-49099-X_1.

[10] Robert M. May (1974): *Biological populations with nonoverlapping generations: stable points, stable cycles, and chaos*. *Science* 186, pp. 645–647.

[11] Robert M. May (1976): *Simple mathematical models with very complicated dynamics*. *Nature* 261(5560), pp. 459–467.

[12] Erik Meijer, Maarten M. Fokkinga & Ross Paterson (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. In John Hughes, editor: *Proceedings 5th International Conference on Functional Programming Languages and Computer Architecture (FPCA 1991)*, *Lecture Notes in Computer Science* 523, Springer, pp. 124–144, doi:10.1007/3540543961_7.

[13] Robert Rosen (1991): *Life Itself: A Comprehensive Inquiry into the Nature, Origin, and Fabrication of Life*. Columbia University Press, New York.

[14] Jan Rutten (2000): *Universal coalgebra: a theory of systems*. *Theoretical Computer Science* 249(1), pp. 3–80, doi:10.1016/S0304-3975(00)00056-6.

[15] Baltasar Trancón y Widemann (2012): *Lindenmayer Systems, Coalgebraically*. In: *Proceedings 11th International Workshop on Coalgebraic Methods in Computer Science (CMCS 2012)*. Short contribution.

[16] Baltasar Trancón y Widemann & Michael Hauhs (2011): *Distributive-Law Semantics for Cellular Automata and Agent-Based Models*. In Andrea Corradini, Bartek Klin & Corina Cîrstea, editors: *Proceedings 4th International Conference on Algebra and Coalgebra (CALCO 2011)*, *Lecture Notes in Computer Science* 6859, Springer, pp. 344–358, doi:10.1007/978-3-642-22944-2_24.

[17] Daniele Turi & Gordon D. Plotkin (1997): *Towards a Mathematical Operational Semantics*. In: *Proceedings 12th International Conference on Logic in Computer Science (LICS 1997)*, IEEE, pp. 280–291, doi:10.1109/LICS.1997.614955.

[18] Philip Wadler (1990): *Recursive types for free!* Available at http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt. Online manuscript.