CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Polynomial Functors in Agda: Theory and Practice

A formalization and collection of applications of categories of polynomial functors

Master's thesis in Computer science and engineering

Marcus Jörgensson

André Muricy Santos

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Polynomial Functors in Agda: Theory and Practice

A formalization and collection of applications of categories of polynomial functors

ANDRÉ MURICY SANTOS MARCUS JÖRGENSSON

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Polynomial Functors in Agda: Theory and Practice
A formalization and collection of applications of categories of polynomial functors
ANDRÉ MURICY SANTOS , MARCUS JÖRGENSSON

Supervisor: Felix Cherubini, Logic and types
Examiner: Andreas Abel, Logic and types

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Abstract

The category of polynomial functors - **Poly** - has been studied for over two decades and is well known for its relevance to computer science [1] and deep mathematical structure [2]. In recent years, new interpretations of **Poly** have been found in dynamical systems theory [3]. This thesis formalizes **Poly** in Cubical Agda and showcases its use in dynamical systems, contributing to the **Poly** code ecosystem.

The first part, theory, formalizes the category **Poly** itself and many of its categorical constructs, including the initial and terminal objects, product, coproduct, equalizer, composition and parallel product. Then **Chart** is formalized, together with its own categorical constructs. Finally, **Poly** and **Chart** morphisms are combined via commuting squares.

The second part, practice, builds upon the theory by using these categories to implement dynamical systems. It covers how morphisms in **Poly** represent dynamical systems, how polynomials act as interfaces, how systems are wired together, how behavior is installed into wiring diagrams, and how to run systems arriving at concrete programs. Many dynamical system examples are given, such as a Fibonacci sequence generator, the Lorenz system, the Hodkgin-Huxley model, and an Echo-State Network that learns the Lorenz dynamics. Commuting squares are used as a way of showing that one dynamical system can be made to simulate another.

Keywords: Category theory, Dependent types, Agda, Cubical, Polynomial functors, Dynamical systems, Complex systems

Acknowledgements

Firstly, we would like to acknowledge and thank Naïm Camille Favier for the help via the Agda Zulip in several proofs: solving a problem with characterizing lens equality, making the inclusion map in set coequalizers type-check, and for general guidance in Agda-related questions.

Secondly, we would like to acknowledge David Orion Girardo for his succinct and beautiful formulation of the composition operator we used, and for inspiration in Agda.

Thirdly, we thank David Spivak, co-author of the textbook we follow for this thesis and many of the papers we cite. We could not have done it without his and Nelson Niu's wonderful book. Or without his warm encouragement to pursue this project.

Fourth, André would like to thank Daniel Satanove, whose tutoring proved vital in understanding some of the more advanced categorical ideas.

Finally, we thank our supervisor Felix Cherubini, whose patience, guidance, and expertise were instrumental in bringing this thesis to life.

Contents

1

Introduction

Category theory is an area of mathematics concerned with abstraction and mathematical structure. The category of polynomial functors, named **Poly**, has been studied for over two decades. **Poly**[1] is known for its relevance to computer science, as well as for its deep mathematical structure. In recent years, researchers in the emerging field of applied category theory have found interpretations of **Poly** in several fields of science and engineering, including dynamical systems theory. The purpose of this thesis is to contribute to the code ecosystem of **Poly** and its use in dynamical systems.

1.1 Applied category theory and complex systems

In the study of complex systems, there is a concern of finding the right level of abstraction at which to seek commonalities between different systems. Sometimes, seemingly unrelated systems might reveal themselves to have some deep structural similarity, like critical transitions in ecosystems [4]. However, arriving at a fundamental mathematical theory of complex systems is profoundly difficult. The immense expressive power of category theory has been a staple of type theory and programming language theory for many decades, and this robust mathematical language is now finding its footing in expressing ideas from wider scientific fields as well, including fields under the umbrella of complex systems. Some of the applications include foundations of complex systems [5], game theory [6], probability [7], systems biology [8] and dynamical systems theory [9].

Polynomial functors and dynamical systems

In particular, **Poly** turns out to be useful in the modeling of so-called mode-dependent open dynamical systems. The "open" part of this class of systems refers to the fact that they not only have an internal dynamics and state space, but can interact with their environment by accepting external inputs and providing external outputs. The "mode-dependent" part refers to the fact that this paradigm of interaction *can change* depending on the internal state of the system, meaning they can, throughout time, accept inputs and provide outputs to different environments.

[1]**Poly** is also known as the category of containers.

This is a remarkably general form of system which **Poly** fully embeds in the categorical context, which means questions like "what does it mean to take the product of two systems?" often have a concrete and intuitive meaning for the dynamics of the resulting system.

1.2 The *poly-book*

Many aspects of the behavior and richness of **Poly** are collected in a newly written (still in draft at the time of writing) textbook, *Polynomial Functors: A Mathematical Theory of Interaction*, hereafter called the poly-book. The book is being written by David Spivak and Nelson Niu and consists of a deep dive into **Poly**, explaining its mathematical structure in detail as well as providing many practical consequences of this structure to modeling dynamical systems. The authors also gave a course based on the book, and the lectures are freely available on YouTube. In this lecture series, some categories closely related to **Poly** are explored, which hint at an even bigger potential for application.

1.3 Goal of this thesis

We believe that the surrounding ecosystem of **Poly** is lacking. It is difficult to find straightforward formalizations of the category, or programs/dynamical systems models that use its structure to their advantage. The goal of this thesis is to advance this ecosystem. We follow the poly-book and its associated lecture series as a blueprint, and rely on the explanations therein to contribute several categorical formalizations, as well as show how **Poly** can be used to model different examples of dynamical systems by providing implementations. Some of these examples are described informally in the poly-book, others we come up with. We do not aim to formalize all theorems or implement all the examples in the book, as this would be simply too much work for this thesis. Instead, we constrain ourselves to what we deem the most essential constructs and illustrative examples.

1.4 Outline

The outline of the thesis is as follows; first, some of the needed background is explained in chapter 2. Then, the theoretical part of the thesis is given, formalizing two categories of polynomial functors and their categorical constructs. The primary category, of lenses, is given in chapter 3 and charts in chapter 4. Chapter 5 consists of the practice part of the thesis, how **Poly** is used for dynamical systems, together with several examples. After, in chapter 6, a discussion of the formalization is given, as well as future work. Finally, in chapter 7, conclusions are drawn about how well the result of the thesis met its purpose.

All the code for this formalization is freely available on GitHub: https://github.com/amuricys/polynomi
functors/.

2

Background

This chapter gives a short introduction to the technology used, as well as the different areas of knowledge needed to understand this thesis. For the technology, **Agda** and **Cubical Agda** are explained, as they provide the framework for the implementation. For the theory, a brief explanation of **Category Theory** is given, as well as some references to previous work on the main category of this thesis, **Poly**. Introducing the concepts of **Poly** and **Chart** is done in their respective theory sections.

Some topics are not introduced in the background but at the point where they are used. There are also many citations and references to deeper explorations of most concepts mentioned.

2.1 Agda

Agda is a dependently typed programming language [10] created at Chalmers. It is a functional language with very similar syntax to Haskell, but with a more powerful type system that allows it to be used as a proof assistant. Dependent types mean that types can depend on values. For example, a function can have the type signature `natOrBool : (b : Bool) → if b then ℕ else Bool`, which means it returns a natural number if the value of the argument is true and a boolean if the value of the argument is false. In a non-dependent language, this would need to be expressed at the value level, so the function would have to return a value of type `Either ℕ Bool`.

2.1.1 Unicode characters

Agda, compared to most other languages, allows and makes heavy use of Unicode characters in symbol names. This can look daunting but provides for a concise syntax akin to mathematical notation.

2.1.2 Propositions as Types

The Curry-Howard isomorphism [11] is the idea that there is a one-to-one correspondence between types in programming languages, propositions in logic, and expressions in algebra. Under this correspondence (isomorphism), a value of a type can be seen as a proof of the proposition the type corresponds to [12]. For example, the following data type in Agda corresponds to the logical operator `or` (written $A \vee B$):

```
data Either (A B : Type) : Set where
    inj₁ : A → Either A B
    inj₂ : B → Either A B
```

Concretely, an element of type `Either A B` is either an element of `A`, contained in the first value constructor `inj`$_1$, or of the type `B`, contained in the second value constructor `inj`$_2$. This corresponds to the fact that that a proof of the logical proposition $A \vee B$ is either a proof of the proposition `A` or a proof of `B`. This, in turn, also corresponds to summing in algebra - $A + B$. A summary of the isomorphism is given below:

Algebra	Logic	Types
$A + B$	$A \vee B$	`Either A B`
$A \times B$	$A \wedge B$	`Pair A B`
B^A	$A \implies B$	`A -> B`
$A = B$	$A \iff B$	`Iso A B`
0	$false$	$\bot$
1	$true$	$\top$

The Curry-Howard isomorphism and Agda's powerful type system make Agda a powerful proof checker.

2.1.3 Agda examples

The `Either` example above shows how (polymorphic) data types are defined, similar to the GADTs [13] extension of Haskell.

It is also possible to create record types for data types with only one constructor.

```
record Action : Type where
  constructor mkAction
  field
    write : Alphabet
    move : Movement
```

The unit type and unit function are defined as:

```
data ⊤ : Type where
    tt : ⊤

unit : {A : Type} → A → ⊤
unit _ = tt
```

The bottom (or False) type and the absurd function from bottom:

```
data ⊥ : Type where

absurd : {A : Type} → ⊥ → A
absurd ()
```

The `natOrBool` function can be implemented as:

```
natOrBool : (b : Bool) → if b then ℕ else Bool
natOrBool true = 0
natOrBool false = true
```

More explanations and examples of Agda can be found in [10] and [12]. Otherwise, it should be possible to understand the more advanced concepts of Agda along the thesis.

2.1.4 Foreign function interface - Haskell

Agda has a fully-featured foreign function interface (FFI) with Haskell, providing access to the vast Haskell ecosystem. The FFI is important to the application section, to use well-established libraries for plotting graphs, writing versatile command-line interfaces, and fast matrix inversion. For a more detailed description of the supporting Haskell code and its use, see appendix A.

2.2 Cubical Agda

Cubical [14] is an extension of Agda, bringing ideas from Homotopy Type Theory (HoTT) [15]. The main contribution of Cubical Agda is how to deal with equality.

Normal Agda defines equality as a data type. However, this (propositional) equality has certain problems. For example, it is not possible to prove function extensionality, that two functions are equal if they are equal for all arguments. Functional extensionality, which is very useful, is directly provable in Cubical.

Cubical discards this data type and uses its own notion of equality, the path. An equality between elements $a : A$ and $b : A$ is a path from a to b. Cubical achieves this with a function $p : I \to A$ satisfying some conditions. Firstly, I is a special kind of variable representing an interval or a path ranging from 0 to 1. It can be thought of as a type representing an interval, although it isn't a type. Secondly, the beginning of the path must be a, that is $p(0) = a$. Finally, the end of the path must be b, $p(1) = b$. The important thing to note is that given `a b : A`, a type `a ≡ b` is a function `I → A` behind the scenes, satisfying the above conditions.

The equality `a ≡ b` is known as homogeneous equality because a and b have the same type. It is also possible to have an equality when a and b are of different types, named heterogeneous equality. Homogeneous equality is a special case of heterogeneous equality, where a and b have the same type. For this thesis, mostly equality between elements of a fixed type is used, so no further explanation of heterogeneous equality is needed.

It is also possible to combine Cubical Agda with propositional equality. Using the functions `pathToEq` and `eqToPath`, it is possible to go back and forth between the two different notions of equality, which is useful in some cases.

There exists a standard Cubical library [16] that has plenty of constructs and proofs from HoTT. The Cubical library is extensively used in this project.

A useful property of Cubical Agda is that the type of isomorphisms is the same as the type of equalities. It is possible to go back and forth using `isoToPath` and `pathToIso`, from the standard Cubical library. This means that one way to prove an equality is to prove an isomorphism.

A final property, which is used heftily in this thesis, is substitution. Substitution makes it possible to transfer a property over an equality. The declaration of substitution, somewhat simplified, is

```
subst : {A : Type} {x y : A} {B : A → Type} → (x ≡ y) → B x → B y
```

The next subsection gives some more examples of Cubical Agda. Further information about Cubical Agda and HoTT can be found in [17] and [15].

2.2.1 Cubical examples

The first example is to show that the absurd function, defined before, is unique. Given any other function from the empty type, it is the same function as absurd. Function extensionality is used to prove an equality between functions. Finally, the lemma is easily proved by pattern matching on the empty type.

```
absurdUnique : {A : Type} → (f : ⊥ → A) → f ≡ absurd
absurdUnique f = funExt lemma
    where
        lemma : (x : ⊥) → f x ≡ absurd x
        lemma ()
```

In propositional equality, `refl` is the constructor of the inductive data type representing equality, which provides a way to prove that any element a, is equal to itself. In Cubical, the function `refl` is used instead. This function satisfies the requirement of being a at both endpoints of the interval since it is constantly a.

```
refl : {A : Type} → {a : A} → a ≡ a
refl {a = a} i = a
```

2.3 Category Theory

Category theory (CT) is used heavily in this thesis. CT is the formal study of composition, building larger relationships from smaller ones. Its central objects of study are categories, where a category consists of some data and satisfies some laws. The most well-known category in computer science is the category of types and functions, but in this thesis, other categories are explored.

A *category* $\mathcal{C}$ consists of the following data:

1. A collection of objects $\mathrm{Ob}(\mathcal{C})$.

2. For every two objects A and B, a set of *arrows* (also called *morphisms* or *maps*) between them denoted by $\mathcal{C}$(A , B). Individual arrows, when they need to be named, are written like this: $f : A \to B$.
3. For every object A, an identity arrow $id : A \to A$.
4. A composition operator $\circ$, that gives rise to new arrows. For any three objects A, B and C and arrows $f : A \to B$ and $g : B \to C$, there exists an arrow $g \circ f : A \to C$.

A category must fulfill the identity and associativity laws:

1. $\forall f . f \circ id = id \circ f = f$ (identity)
2. $\forall f\ g\ h . f \circ (g \circ h) = (f \circ g) \circ h$ (associativity of composition)

2.3.1 Commuting diagrams

The usual way of representing statements in category theory is through *commuting diagrams*. Such as:

$$
\begin{tikzcd}
& D \arrow[dr, "k"] & \\
A \arrow[ur, "h"] \arrow[dr, "f"] & & C \\
& B \arrow[ur, "g"] &
\end{tikzcd}
$$

Here, the objects are A, B, C and D, and the arrows are $f : A \to B$, $g : B \to C$, $h : A \to D$ and $k : D \to C$. Composite arrows and identity arrows are usually omitted from the diagram but are always there implicitly. *Commuting* means that any path between two objects is the same; for this example $k \circ h = g \circ f$. In a sense, diagrams can be seen as category theory's analog of abstract algebraic equations. Like equations handle symbols that can stand for anything, diagrams are an abstract representation of some structure in a category, containing objects and morphisms labeled and subject to constraints like being equal or unique, and should be interpreted as a statement. It should be emphasized that in general settings of category theory, the meaning of the objects and arrows doesn't play any role - they are purposefully abstract. All that matters is the structure of the relationship between objects and arrows, which is why abstract commuting diagrams are useful. Concrete categories ought to have meaning to their arrows and what it means to compose them, like the category **Set**, where objects are sets and arrows are functions between them.

2.3.2 Applied category theory

Although the definition of a category may not seem like much, it reveals *the point* of category theory: it is a theory of *compositionality*. What it means to compose relationships between objects, what facts are preserved under different compositions, how composing can generalize: this and other deep structural questions are at the core of this theory.

In this spirit of investigating compositionality, *applied* category theory is done. The main introductory textbook on the subject, *Seven Sketches in Compositionality* [18], does a good job of justifying this approach. The book usually starts with motivating examples in the real world and what CT and its constructions and formalisms can bring to understanding them in better, or at least different ways. For this thesis, the order is reversed; the theory is covered first, but the goal of applying is always present.

There are many constructions and structures in CT which would not be feasible to explain in this section. Instead, they are introduced as they become relevant.

2.3.3 In code: Agda's `agda-categories` library

The poly-book demonstrates many category-theoretic constructs in the context of **Poly**, but this thesis is limited to the most essential ones. These tend to be quite ubiquitous, and as such, there are formulations of many of them in the Agda ecosystem: the three main ones are the 1lab [19], the Cubical library's own [20], and the `agda-categories` [21] library. For this thesis, the third one is picked for a few reasons: firstly, it is very well documented, with a sensible user interface and active community. Secondly, it has more constructs: for example, it is the only one of the three in which the exponential object, a key feature of **Poly**, is present. Thirdly, it is interoperable with Cubical, since the definition of a category requires a user to provide an equivalence relation to express equality between arrows, where cubical equality can be supplied.

2.4 Previous work on Polynomial Functors

The study of polynomial functors has an extensive literature in the pure category-theoretic setting, like the work of Joachim Kock [2] [22]. In contrast, the **poly-book**, the main book of this thesis, is focused on applied category theory, providing interpretations of abstract mathematical structures and mapping them to concrete concepts and applications. Some of the most mature interpretations are database theory, decision systems, and dynamical systems. One of the book's authors, David Spivak, has done work using **Poly** in the context of database theory [23]. The book also spends some time developing intuitions on the decision-making perspective, but not much work in this area could be found. Finally, there is the interpretation of **Poly** in dynamical systems, which the book provides many examples of.

3

Theory: Lenses

This chapter covers the first theoretical part of the work, consisting mainly of formal proofs of different categorical properties of the category **Poly**. All code is available open source at GitHub [24], which is recommended to follow while reading the thesis.

3.1 The category Poly

The category **Poly** is the category of polynomial endofunctors in **Set**, with morphisms as natural transformations. They are called *Polynomial* because these functors can be written down as high-school algebraic expressions, like:

$$p(y) = y^2 + 4y + 10$$

But instead of handling numbers, these polynomial expressions have *sets* plugged into the variable y. Sum, multiplication, exponentiation, and numeric literals, are all considered in the context of sets. A numeric literal such as 7 represents the set $\underline{7}$, a set with 7 elements. A $+$ symbol corresponds to *the categorical coproduct*, multiplication corresponds to *the categorical product* and exponentiation to the exponential object: in this context, it is the type of functions from the exponent to the base of the term. For example, $f : 7^2 \cong f : 2 \to 7$. This way of thinking algebraically is very convenient, because plugging in "numbers" into the polynomial expressions to get "numbers" back will turn out to be a very useful kind of operation.

3.1.1 Objects are polynomial functors

The objects of **Poly** are polynomial functors, which can be written as expressions like the one shown above. Since **Poly** handles sets, though, more unorthodox looking expressions can be created. For instance, the polynomial below is perfectly coherent:

$$p(y) = \mathbb{N}y^2 + 1 \tag{3.1}$$

and it can be thought of as taking a set as input, and sending it to the set of: either infinitely long tuples of functions from $\underline{2}$ to this set, or to the only element of the singleton set. Like the poly-book, each summand of a power of y in the polynomials are referred to as a *position*, and the exponent of that power as a *direction at that position* [1]. This suggests that a polynomial has a set of positions, and for each

[1]Note: The book acknowledges the unfortunate naming clash with the equivalent category of containers, where in that context, positions are called directions and directions are called shapes.

element in this set, a set of directions. Therefore, a natural way to represent these polynomials in Agda is as a dependent set:

```
record Polynomial : Set₁ where
    constructor mkpoly
    field
        position : Set
        direction : position → Set
```

The way to write the example $p(y) = \mathbb{N}y^2 + 1$ in Agda using this type is then

```
p = mkpoly (ℕ ⊎ ⊤)  λ {(inj₁ x) → Bool ; (inj₂ y) → ⊥}
```

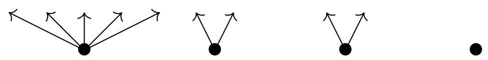
Sometimes, a polynomial is referred to as having "a single position" when it is a *monomial.* This means that, in the definition of such a polynomial, the set of directions does not depend on the set of positions. For instance, the monomial

$$p(y) = \mathbb{R}y^{\{a,b,c,d\}}$$

does not have any dependence between *the set of positions* (the real numbers) and *the sets of directions* (the set $\{a, b, c, d\}$). The sets of directions would normally be a family of sets, where each position could give a different one, but in this case, they're always the same set. This way of phrasing positions might be unsettling at first, but it makes more sense in everyday programming with polynomial functors, since what really matters when handling dependent sets is what distinction an indexing set performs. If two positions index the same set, they can be thought as the same thing.

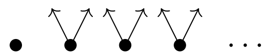
More perspectives on polynomials

Understanding polynomial functors is crucial, and there are many ways to think about them. The book showcases other standard ways to visualize polynomials, which are helpful to interpret the category in different applications and to build intuition about the purely categorical perspective. One of these is *corolla forests*:



(3.2)

The term *corolla* refers to a tree with a depth of one, and *forest* refers to the fact that there are many such small trees. The set of positions is the set of roots of the forest (in this case $\underline{4}$), and the set of directions at each position is the set of arrows at each root (in this case $\underline{5}$, $\underline{2}$, $\underline{2}$ and $\underline{0}$). This then corresponds to $p(y) = y^5 + y^2 + y^2 + 1 \cong y^5 + 2y^3 + 1$. This perspective lends intuition to the decision-making interpretation of polynomials. Very loosely speaking, the set of directions is "directions in which one can go" given that one is at that set's associated position. Since polynomials can handle infinite sets, even uncountably infinite ones, one can imagine such infinite corolla trees. For example, the polynomial 3.1 could be visualized as:



(3.3)

Another perspective on polynomials, emphasizing the container datatype view, is as Haskell algebraic data types. The polynomial 3.2, for instance, can be written in Haskell as:

```
data P y = Fst y y y y y | Snd y y | Trd y y | Frth
  deriving Functor
```

This type can trivially implement the `Functor` typeclass, so much so that it can be derived automatically as above (given that the language extension `DeriveFunctor` is enabled).

As functors, polynomials act on both objects and functions, and this is defined as:

```
_⦇_⦈ : Polynomial → Set → Set
_⦇_⦈ (mkpoly position direction) Y = Σ position λ x → (direction x → Y)
infixl 30 _⦇_⦈

applyFn : {A B : Set} → (p : Polynomial) → (A → B) → p ⦇ A ⦈ → p ⦇ B ⦈
applyFn (mkpoly position direction) f (fst , snd) = fst , λ x → f (snd x)
```

Then, these definitions can be used to prove that **Poly**'s objects really are functors. A record in `agda-categories` is formulated corresponding to an endofunctor in the library's provided instance of **Set**:

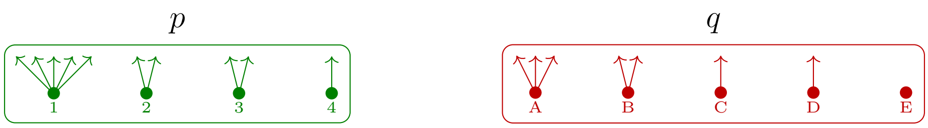
```
F-resp : {p : Polynomial} {A B : Set} {f g : A → B} {x : p ⦇ A ⦈ } →
    f ≡ g → applyFn p f x ≡ applyFn p g x
F-resp {x = posApp , dirApp} pr = λ i → posApp , (pr i) ∘  dirApp

conv : {A B : Set} {f g : A → B} → ({x : A} → f x Eq.≡ g x) → f ≡ g
conv p = funExt λ _ → eqToPath p

asEndo : (p : Polynomial) → Functor (Sets zero) (Sets zero)
asEndo p = record
    { F₀ = λ x → p ⦇ x ⦈
    ; F₁ = λ f → applyFn p f
    ; identity = Eq.refl
    ; homomorphism = Eq.refl
    ; F-resp-≈ = λ {_} {_} {f} {g} proof →
        pathToEq (F-resp {f = f} {g = g} (conv proof))
    }
```

3.1.2 Arrows are natural transformations - *dependent lenses*

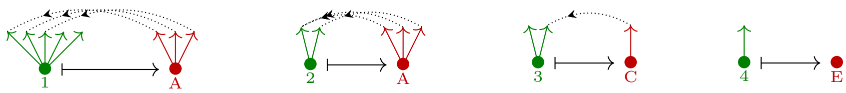
Arrows on **Poly** are so-called *dependent lenses*. The reason is that arrows between monomials are the usual definition of lenses in Haskell, as a pair of non-dependent "getter and setter" functions. For example, a lens between Sy^T and Ay^B consists of the functions `get : S → A` and `set : S → B → T`. A dependent lens has the additional property that "the types we are allowed to set depend on the value we get". This notion is explored further in the applications part of the thesis. Since non-dependent lenses (lenses between monomials) are a very special case of dependent lenses, the arrows in **Poly** are referred to as simply *lenses*.

Perspectives on lenses

Another use of the corolla forest perspective is that it makes visualizing maps between polynomials intuitive. Consider the polynomials $p(y) = y^5 + 2y^2 + y$ and $q(y) = y^3 + y^2 + 2y + 1$. First, their visualization as corollas is as:



A map between p and q can be visualized in the following way. Each position in p is sent to a position in q, and for each of these sendings, its directions are sent back to the directions at the original position:



If instead the map on the directions also goes forward, it would be another kind of arrow called charts. Charts are the arrows in the category **Chart**, which is the second category this thesis explores, in chapter 4.

In the Haskell view of polynomials, lenses can be given as the following type synonym (requires `RankNTypes` language extension):

```
type Lens p q = forall y. p y -> q y
```

An example of a lens between the two polynomials above would look like the following (requires `LambdaCase` language extension):

```
data P y = Fst y y y y y | Snd y y | Trd y y | Frth
  deriving Functor

data Q y = A y y y | B y y | C y | D y | E
  deriving Functor

exampleLens :: Lens P Q
exampleLens = \case
  Fst y1 y2 y3 y4 y5 -> _
  Snd y1 y2 -> _
  Trd y1 y2 -> _
  Frth -> _
```

The map on positions translates to a map on value constructors; for each of the summands of p, first a value constructor in q must be chosen. After choosing a constructor, its values must be filled in:

```
exampleLens :: Lens P Q
exampleLens = \case
  Fst y1 y2 y3 y4 y5 -> A y1 y2 y3
  Snd y1 y2 -> A y1 y1 y2
  Trd y1 y2 -> C y2
  Frth -> E
```

This recovers the idea of a function "back", that is a map on directions *at that position*: for each of the targeted value constructors in q, a lens must fill its slots with the source constructor's y values. For example, in this case, the three slots of the `A` constructor in the first branch must be filled by the five available choices in the source constructor `Fst`. In an abstract sense, we can see this as a function of type $\underline{3} \to \underline{5}$ - a choice of source out of 5 possible sources for each of the 3 slots.

Expressing more general kinds of lenses in Haskell using this scheme becomes hard quickly however. The dependency between the value of the map on positions and the type of the map of directions is hidden and baked into the syntax of the language. There are attempts to do this in Haskell [25] [26], but the Haskell view is given only for the sake of perspective. Polynomials and lenses are much more natural in Agda.

Lenses in Agda

Here is their definition:

```
record Lens (from to : Polynomial) : Set where
    constructor _⇆_
    open Polynomial
    field
        mapPosition : position from → position to
        mapDirection : (fromPos : position from) →
                       direction to (mapPosition fromPos) →
                       direction from fromPos
```

The constructor _⇆_ is an attempt at representing the two maps: on positions below and going forward (because they are the coefficients of the polynomials) and on directions on top going backwards (because they are exponents of y at each position).

As they are maps between functors, one can expect that lenses are natural transformations. Indeed, they are, and here is how to formulate them as such in the context of `agda-categories`:

```
asNatTransLens : {p q : Polynomial} →
            Lens p q →
            NaturalTransformation (asEndo p) (asEndo q)
asNatTransLens (f ⇆ f♯) = record {
    η = λ { X (posP , dirP) → f posP , dirP ∘ f♯ posP } ;
    commute = λ f₁ → Eq.refl ;
    sym-commute = λ f₁ → Eq.refl
    }
```

Two more pieces of data are needed to form a valid category: the identity arrow and composition.

3.1.2.1 The identity lens

The identity lens leaves the polynomial untouched in both the positions and the direction at each position, which is implemented by the identity function.

```
idLens : {A : Polynomial} → Lens A A
idLens = id ⇆ λ _ → id
```

3.1.2.2 Composing lenses

Composition of lenses is done in the natural way. The composed maps on positions are obtained with straightforward function composition. The same is true of the maps on directions, but with some extra work to deal with the fact that they are dependent functions.

```
_∘ₚ_ : {A B C : Polynomial} → Lens B C → Lens A B → Lens A C
_∘ₚ_ (f ⇆ f♯) (g ⇆ g♯) = (f ∘ g) ⇆ (λ i → g♯ i ∘ f♯ (g i))
```

3.1.3 Categorical axioms

That the associativity and identity laws hold for **Poly** follows directly from the definitions of these types, which means that they can be proved via Cubical Agda's `refl`. With all the pieces in place, the full formulation of **Poly** as a category in `agda-categories`' is provided:

```
-- Categorical/Poly/Instance.agda
∘-resp-≈ : {A B C : Polynomial}
          {f h : Lens B C}
          {g i : Lens A B} →
          f ≡ h → g ≡ i → (f ∘ₚ g) ≡ (h ∘ₚ i)
```

```
∘-resp-≈  p q ii = (p ii) ∘ₚ (q ii)

Poly : Category (lsuc lzero) lzero lzero
Poly = record
    { Obj = Polynomial
    ; _⇒_ = Lens
    ; _≈_ = _≡_
    ; id = idLens
    ; _∘_ = _∘ₚ_
    ; assoc = refl
    ; sym-assoc = refl
    ; identityˡ = refl
    ; identityʳ = refl
    ; identity² = refl
    ; equiv = record { refl = refl ; sym = sym ; trans = _∙_ }
    ; ∘-resp-≈ = ∘-resp-≈
    }
```

This record definition should not be surprising: **Poly**'s objects are `Polynomials`, its arrows are dependent `Lenses`, the notion of equality between arrows is defined to be cubical equality `_≡_`, the identity lens and composition are the previously defined values in Agda. The rest of the data provided to the record correspond to the categorical laws plus some convenience that `agda-categories` provides: for instance, the proof `sym-assoc` is not normally needed, but it makes it so that the opposite of opposite categories are equal "on the nose" to the original categories, and not just provably equal. A more detailed overview of the design decisions of `agda-categories` can be found in [21].

3.2 Polynomial equality

There are some occasions where polynomials need to be compared for equality. For example, to prove the right unit law for the parallel product (explained later) ⊗ with Y as unit: `rightUnit : (p q : Polynomial) → p ⊗ Y ≡ p`.

Since a polynomial is a record consisting of the `position : Set` and `direction : position → Set` a characterization of equality for this record is needed to make it easy to use and prove equalities between polynomials. An alternative definition of a polynomial as a Σ-type is used in aid to characterize the equality, to use the many properties and lemmas about Σ-type in the Cubical library. That Σ-type, along with proofs that the definitions are the same are:

```
PolyAsSigma : Set₁
PolyAsSigma = Σ[ position ∈ Set ] (position → Set)

polyToSigma : Polynomial → PolyAsSigma
polyToSigma (mkpoly position direction) = position , direction
```

```
polyFromSigma : PolyAsSigma → Polynomial
polyFromSigma (position , direction) = mkpoly position direction

poly≡polySigma : Polynomial ≡ PolyAsSigma
poly≡polySigma = isoToPath (iso polyToSigma
                                polyFromSigma
                                (λ _ → refl)
                                (λ _ → refl))
```

Equality for the Σ-type is then implemented by using `ΣPathTransport→PathΣ` from the Cubical library. It is important to note that an equality of a Σ-type is a Σ of equalities. This means that a proof that the map on positions is equal is needed, as well as a proof that the map on directions is equal. Substitution is needed for the proof on the directions to make the types match.

```
polySigmas≡ : (a b : PolyAsSigma)
    → (fstA≡fstB : fst a ≡ fst b)
    → (subst (λ x → x → Type) fstA≡fstB (snd a)) ≡ snd b
    → a ≡ b
polySigmas≡ a b fstA≡fstB sndA≡sndB
  = ΣPathTransport→PathΣ a b (fstA≡fstB , sndA≡sndB)
```

This definition of equality is then transferred to the record definition of Polynomials leading to the final characterization of polynomial equality:

```
poly≡ : {a b : Polynomial}
    → (fstA≡fstB : position a ≡ position b)
    → (subst (λ x → x → Type) fstA≡fstB (direction a)) ≡ direction b
    → a ≡ b
poly≡ {a} {b} fstA≡fstB sndA≡sndB i
  = polyFromSigma (polySigmas≡ (polyToSigma a)
                               (polyToSigma b)
                               fstA≡fstB
                               sndA≡sndB
                               i)
```

In addition, a forall-quantified version of equality is provided, which simplifies some proofs:

```
poly≡∀ : {a b : Polynomial}
    → (fstA≡fstB : position a ≡ position b)
    → ((posB : position b) →
       subst (λ x → x → Type) fstA≡fstB (direction a) posB ≡ direction b posB)
    → a ≡ b
poly≡∀ {a} {b} fstA≡fstB sndA≡sndB
  = poly≡ fstA≡fstB λ i x → sndA≡sndB x i
```

3.3 Lens equality

In category theory, it is frequently of interest to know when two morphisms are equal. Therefore, a characterization of equality for lenses is also needed, and it should be convenient enough to be usable in practical proofs.

A lens consists of two components, with the second component depending on the first. This suggests a Σ-type structure, which leads to a strategy similar to polynomial equality, but in lenses things are more complicated, since the underlying types are functions instead of simple sets. Still, representing lenses as Σ-types is straightforward:

```
LensAsSigma : Polynomial → Polynomial → Type
LensAsSigma (mkpoly posP dirP) (mkpoly posQ dirQ)
    = Σ[ mapPos ∈ (posP → posQ) ]
    ((fromPos : posP) → dirQ (mapPos fromPos) → dirP fromPos)

sigmaToLens : LensAsSigma p q → Lens p q
sigmaToLens (mapPos , mapDir) = mapPos ⇆ mapDir

lensToSigma : Lens p q → LensAsSigma p q
lensToSigma  (mapPos ⇆ mapDir) = mapPos , mapDir

lens≡lensSigma : (Lens p q) ≡ (LensAsSigma p q)
lens≡lensSigma = isoToPath (iso lensToSigma
                                sigmaToLens
                                (λ _ → refl)
                                (λ _ → refl))
```

Equality for this Σ-type is defined and then transferred to the lens type:

```
lensSigmas≡ : {p q : Polynomial} (f g : LensAsSigma p q)
    → (fstF≡fstG : fst f ≡ fst g)
    → subst (λ mapPos → (fromPos : position p) →
             direction q (mapPos fromPos) →
             direction p fromPos)
             fstF≡fstG
      (snd f) ≡ snd g
    → f ≡ g
lensSigmas≡ f g fstF≡fstG sndF≡sndG
  = ΣPathTransport→PathΣ f g (fstF≡fstG , sndF≡sndG)

lens≡ : {p q : Polynomial} {f g : Lens p q}
    → (mapPos≡ : mapPosition f ≡ mapPosition g)
    → subst (λ mapPos → (fromPos : position p) →
                        direction q (mapPos fromPos) →
                        direction p fromPos) mapPos≡
      (mapDirection f) ≡ mapDirection g
```

```
    → f ≡ g
lens≡ {p} {q} {f} {g} mapPos≡ mapDir≡ i
  = sigmaToLens ((lensSigmas≡ {q = q} (lensToSigma f)
                                      (lensToSigma g)
                                      mapPos≡
                                      mapDir≡
                                      i))
```

An all quantified version of lens equality is provided as well, named `lens≡∀∀`:

Both `lens≡` and `lens≡∀∀` provide a way to prove equalities between lenses. However, in some cases they are limited and almost impossible to use. The reason is that `subst` is not very deep in the term, and the `subst` is of a complicated dependent function type, where the dependent function returns a new function with a varying domain. Often, it is possible to use laws to push the `subst` operation further into the term, which makes lemmas easier to use. This idea is captured by the more powerful variant of lens equality below, `lens≡ₚ`. In it, `subst` is pushed further into the term making it more practical to use in proofs.

```
lens≡ₚ : {p q : Polynomial} {f g : Lens p q}
  → (mapPosEq : mapPosition f ≡ mapPosition g)
  → (
          (x : position p) →
          (y : direction q (mapPosition g x)) →
          mapDirection f x
          (subst (λ mapPos → direction q (mapPos x)) (sym mapPosEq) y)
          ≡
          mapDirection g x y
      )
  → f ≡ g
lens≡ₚ {f = f} {g = g} a b i
  = sigmaToLens (lensSigmas≡ₚ {f = f} {g = g} a b i)
```

To implement `lens≡ₚ`, or more specifically, `lensSigmas≡ₚ`, `ΣPathP` is used instead of `ΣPathTransport→PathΣ`.

```
lensSigmas≡ₚ : {p q : Polynomial} {f g : Lens p q}
  → (mapPosEq : Lens.mapPosition f ≡ Lens.mapPosition g)
  → ((x : position p) → (y : direction q (mapPosition g x)) → mapDirection f x
  → lensToSigma f ≡ lensToSigma g
lensSigmas≡ₚ {p = p} {q = q} {f = f} {g = g} mapPosEq mapDirEq
    = ΣPathP (mapPosEq ,
        funExt λ x →
        funExtDep λ {y1} {y2} p →
          cong (mapDirection f x) (fromPathP⁻ p) ∙ mapDirEq x y2)
```

If lenses and polynomials are both Σ-types in disguise, why not just use Σ-types?

The main reason is *usability*. This is a convenience-driven decision: we want our implementation of **Poly** to not only be amenable to formalization and having theorems proven about it, but also to be *nice to program in*. Working with Σ-types causes readability to suffer too much, because the names of the constructor and fields are lost. We then make the compromise of converting between the record Σ-type representations only in the equality modules, since the proofs that the representations are equal allows us to carry equality over between the types, a very useful feature of Cubical Agda.

3.4 Initial object

An object is called initial if there exists an unique arrow from it to every other object in the category. In **Poly**, the initial object is the polynomial $p(y) = 0$ with no positions (and therefore also no directions). It is the functor that sends any set to the empty set.

```
𝟘 : Polynomial
𝟘 = mkpoly ⊥ λ ()
```

The steps to prove that 𝟘 is the initial object is to firstly construct a lens to every other object, and to secondly show that this lens is unique.

Since 𝟘 has no positions (nor directions) both the map on positions and the map on directions to any other polynomial p is given by the function `λ ()` (often named absurd) from the empty type.

```
lensFromZero : {p : Polynomial} → Lens 𝟘 p
lensFromZero = (λ ()) ⇆ (λ ())
```

The final step is to show that this lensFromZero from 𝟘 to p is unique. This is done by assuming that there exists another lens f from 𝟘 to p, and then proving that f actually is the same lens as `lensFromZero`.

```
lensFromZeroUnique : {p : Polynomial} (f : Lens 𝟘 p) → lensFromZero ≡ f
lensFromZeroUnique _ = lens≡ (funExt λ ()) (funExt λ ())
```

This is implemented by using the lens equality function and the fact that the absurd function (from the empty type) is unique, utilizing function extensionality. We then plug these proofs into the `agda-categories` characterization of the initial object:

```
open import Categories.Object.Initial Poly

zeroIsInitial : IsInitial 𝟘
zeroIsInitial = record { ! = lensFromZero ; !-unique = lensFromZeroUnique }

initialZero : Initial
initialZero = record { ⊥ = 𝟘 ; ⊥-is-initial = zeroIsInitial }
```

3.5 Terminal object

Dually to the initial object, the terminal object is an object such that there is a unique arrow from every other object in the category to it. In **Poly**, the terminal object is defined as the polynomial with a single position, but no directions for that position. It is the functor that sends any set to the singleton set.

```
𝟙 : Polynomial
𝟙 = mkpoly ⊤ (λ _ → ⊥)
```

Similarly to the initial object, to show that 𝟙 is a terminal object amounts to first constructing a lens **from** every other object, and then showing that each of these lenses is unique.

The map on positions from any other polynomial to 𝟙 is the function always returning `tt`, the unique inhabitant of the singleton set ⊤. The function itself is also often called *unit*. For this position, the map on directions goes from ⊥ since 𝟙 has no directions, thus the map on directions is again the absurd function.

```
lensToOne : {p : Polynomial} → Lens p 𝟙
lensToOne = (λ _ → tt) ⇆ λ _ ()
```

Finally, for uniqueness, any other lens f from p to 𝟙 is the same as `lensToOne`. For map on positions equality, `refl` is used to show that any two functions (with same domain) to ⊤ is the same. For map on directions equality, uniqueness of the absurd function is used, as in the initial object. The more advanced version of lens equality is used to do the function extensionality in the background, allowing for more concise syntax.

```
lensToOneUnique : {p : Polynomial} (f : Lens p 𝟙) →  lensToOne ≡ f
lensToOneUnique _ = lens≡∀∀ refl (λ _ ())
```

And again we plug these proofs in to the `agda-categories` characterization of the initial object:

```
open import Categories.Object.Terminal Poly

oneIsTerminal : IsTerminal
oneIsTerminal = record { ! = lensToOne ; !-unique = lensToOneUnique }

terminalOne : Terminal
terminalOne = record { ⊤ = 𝟙 ; ⊤-is-terminal = oneIsTerminal }
```

3.6 Product

The product of two objects A and B is an object $A \times B$ together with arrows π_1 and π_2 such that, given any other object T and arrows from T to A and B, the diagram in Figure 3.1 commutes and the arrow ! exists and is unique. The arbitrary object T can be intuitively thought of as a "candidate" object to be the product. This is an

$$\begin{tikzcd}
 & T \arrow[ddl, "h"'] \arrow[d, dashed, "!"] \arrow[ddr, "k"] & \\
 & A \times B \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
A & & B
\end{tikzcd}$$

Figure 3.1: Product diagram of two objects A and B.

example of a *universal property* in category theory and the arrows h and k are said to be *factorized* through the compositions $\pi_1 \circ !$ and $\pi_2 \circ !$ respectively.

Poly has all products, which means that for any two polynomials p and q we can construct their product as the polynomial $p * q$.

Firstly, the product polynomial of two polynomials is defined. The position set is the product of p's and q's position sets, while the direction at a position is either a direction of p or a direction of q. This is another instance of the fact that these polynomials behave just like the ones in high school algebra: for example, given $p(y) = y^2$ and $q(y) = 5y^3 + 2y$, their product is given by $p * q(y) = 5y^5 + 2y^3$. In Agda, the product is given as:

```
_*_ : Polynomial → Polynomial → Polynomial
(mkpoly posP dirP) * (mkpoly posQ dirQ) =
    mkpoly (posP × posQ) (λ {(posP , posQ) → (dirP posP) ⊎ (dirQ posQ)})
```

A visualization of the product is provided in Figure 3.2.

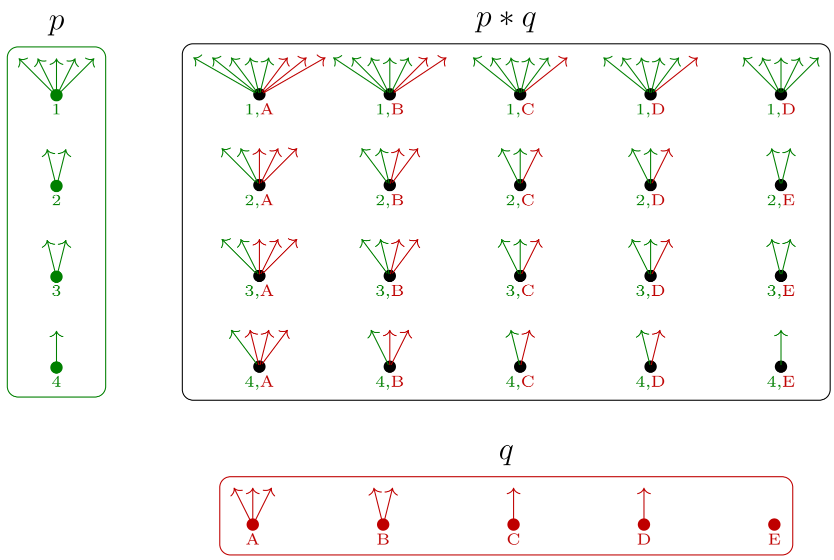


Figure 3.2: $p(y) = y^5 + 2y^2 + y$, $q(y) = y^3 + y^2 + 2y + 1$. Their product $p * q$ has both p and q's positions; each position of $p * q$ is a pair with one position of p and one of q. The direction at that position of $p * q$ is the disjoint union of the directions at the originals, with the arrow's color indicating where they came from.

Continuing with Figure 3.1, π_1 and π_2 are defined to take the projections on the positions, and injections on the map on the directions back. In Figure 3.2, this can be imagined by selecting only the correspondingly colored part of a corolla in the product corolla forest.

```
π₁ : {p q : Polynomial} → Lens (p * q) p
π₁ = proj₁ ⇆ λ _ → inj₁

π₂ : {p q : Polynomial} → Lens (p * q) q
π₂ = proj₂ ⇆ λ _ → inj₂
```

Given an object c as the domain of two lenses f and g, $\langle f, g\rangle$ should be the factorization lens of f and g. This factorization must be unique, which is proved later as the last step. The factorization lens is defined by taking the pair of functions for map on positions. For the map on directions either f or g is used depending on the direction.

```
⟨_,_⟩ : {p q r : Polynomial} → Lens p q → Lens p r → Lens p (q * r)
⟨ f ⇆ f♯ , g ⇆ g♯ ⟩ = < f , g > ⇆ λ posP → [ f♯ posP , g♯ posP ]
```

Finally, we need to supply the proofs associated with the product. There are two proofs needed to show that the diagram 3.1 commutes, and one for showing the uniqueness of $\langle f, g\rangle$. The proofs for commuting is shown with `refl`, but the uniqueness is left out for space reasons - this will be done relatively frequently in the rest of the thesis, since many of these proofs get quite long. Interested readers can of course see the code for the formalization [24]. But the idea is that if we have another factorization lens h it must be the same as $\langle f, g\rangle$.

```
project₁ : {p q r : Polynomial} {f : Lens r p} {g : Lens r q} →
```

```
          π₁ ∘ₚ ⟨ f , g ⟩ ≡ f
project₁ = refl

project₂ : {p q r : Polynomial} {f : Lens r p} {g : Lens r q} →
          π₂ ∘ₚ ⟨ f , g ⟩ ≡ g
project₂ = refl

unique : {p q r : Polynomial} {h : Lens p (q * r)} {f : Lens p q}
    {g : Lens p r → (π₁ ∘ₚ h) ≡ f} →
    (π₂ ∘ₚ h) ≡ g →
    ⟨ f , g ⟩ ≡ h
unique = ...
    where
        lemma : ⟨ π₁ ∘ₚ h , π₂ ∘ₚ h ⟩ ≡ h
        lemma = ...
```

3.6.1 Productized lens

Two lenses between different pairs of polynomials can be represented as a single lens between two products, as follows.

```
⟨_×_⟩ : {a b c d : Polynomial} → (f : Lens a c) (g : Lens b d)
    → Lens (a * b) (c * d)
⟨ (f ⇆ f♯) × (g ⇆ g♯) ⟩
    = (λ {(a , b) → f a , g b}) ⇆
        λ {(a , b) (inj₁ dirC) → inj₁ (f♯ a dirC)
         ; (a , b) (inj₂ dirD) → inj₂ (g♯ b dirD)}
```

3.6.2 Generalized product

Similarly to how the Π type is a generalization of the product type in **Type**, we can create the product of all polynomials indexed by any type. Think of it as a "fold", in the sense that each element of a set produces a new polynomial that is productized with the "accumulator" of this fold (all the products taken so far). This construction will be particularly important for the exponential object.

```
ΠPoly : Σ[ indexType ∈ Set ] (indexType → Polynomial) → Polynomial
ΠPoly (indexType , generatePoly) = mkpoly pos dir
  where
    -- Embedding all polynomial positions into one position
    pos : Set
    pos = (index : indexType) → position (generatePoly index)

    -- Direction is exactly one of the polynomials' directions
    dir : pos → Set
    dir pos = Σ[ index ∈ indexType ] direction (generatePoly index) (pos index)
```

Using this generalized product type with `Bool` as the index type is equivalent to having a normal binary product - which makes sense, since it means taking a product of two polynomials. This is proved in `Various.agda`.

```
productIsΠPoly : {p q : Polynomial}
    → ΠPoly (Bool , tupleToFunFromBool (p , q)) ≡ (p * q)
productIsΠPoly = ...

tupleToFunFromBool : {ℓ : Level} {A : Set ℓ} → (A × A) → Bool → A
tupleToFunFromBool (a , b) true = a
tupleToFunFromBool (a , b) false = b
```

With the generalized product it is also useful to have the factorizer $\langle f, g\rangle$ generalized, now factorizing a generalized amount of lenses instead of just f and g.

```
universalPropertyProduct : {p : Polynomial} {Index : Type}
    {generate : Index → Polynomial}
    → Lens p (ΠPoly (Index , generate)) ≡ ((i : Index) → Lens p (generate i))
universalPropertyProduct = ...
```

3.6.3 Monoid

The product forms a monoid with $\mathbb{1}$, which we show by constructing a monoidal category (**Poly**, $\times$, $\mathbb{1}$). In fact, the product is a *symmetric* monoidal structure, since $A \times B$ is isomorphic to $B \times A$. This is one place where the existing structure of `agda-categories` is extremely useful, as we only need to call some of its helpers to prove this and rely on the fact that all cartesian categories form monoidal categories with respect to the product. The code for it is:

```
-- Categorical/Poly/Monoidal/Product.agda

open import Categories.Category.Monoidal
import Categories.Category.Cartesian as Cartesian

binaryProducts : Cartesian.BinaryProducts Poly
binaryProducts = record { product = prod }

cartesian : Cartesian.Cartesian Poly
cartesian = record { terminal = terminalOne ;
                     products = binaryProducts }

productMonoidal : Monoidal Poly
productMonoidal
    = Cartesian.CartesianMonoidal.monoidal Poly cartesian

open import Categories.Category.Monoidal.Symmetric productMonoidal
productSymmetricMonoidal : Symmetric
productSymmetricMonoidal
    = Cartesian.CartesianSymmetricMonoidal.symmetric Poly cartesian
```

3.7 Coproduct

The coproduct is the dual construction to the product, obtained from reversing all arrows, shown in figure 3.3.

$$\begin{tikzcd}
 & T & \\
 & A + B \arrow[u, dashed, "!"] & \\
A \arrow[uur, "h"] \arrow[ur, "\mathrm{i}_1"'] & & B \arrow[uul, "k"'] \arrow[ul, "\mathrm{i}_2"]
\end{tikzcd}$$

Figure 3.3: Coproduct diagram of A and B.

Poly has all coproducts, given any p and q, the coproduct $p + q$ is defined by taking the (Set) coproduct on the positions and the *projection* on the directions. Again, consider the analogy with high-school algebra. Given two polynomials $p(y) = 4y^2$ and $q(y) = y^2 + 3y$, their sum is given by $(p + q)(y) \cong 4y^2 + y^2 + 3y \cong 5y^2 + 3y$. In this example, four of the positions with 2 as a direction came from p and one of them came from q, hence in the Agda code for it below, we pattern match on the resulting polynomials's position. As a shorthand, we use Agda's `[_,_]` function.

```
_+_ : Polynomial → Polynomial → Polynomial
(mkpoly posA dirA) + (mkpoly posB dirB) = mkpoly (posA ⊎ posB) [ dirA , dirB ]
```

A visualization of an example coproduct is provided in figure 3.4.

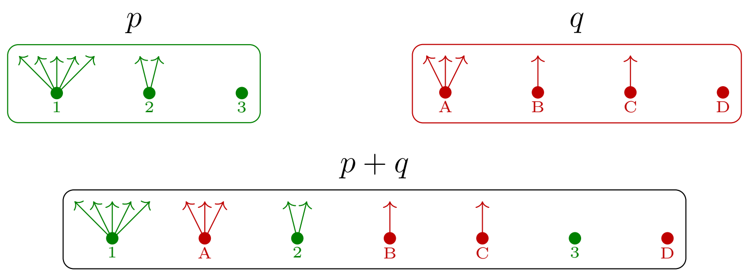


Figure 3.4: A sum of polynomials $p(y) = y^5 + y^2 + 1$ and $q(y) = y^3 + 2y + 1$. The resulting polynomial has the disjoint union of both as its positions, and each position has the directions at that position in the original polynomial it came from.

The injections i_1 and i_2 are defined by taking the injections on the positions and keeping the directions on that position using *id*. The example 3.3 shows how i_1 behave for one position.

```
i₁ : {p q : Polynomial} → Lens p (p + q)
i₁ = inj₁ ⇆ λ _ → id
```

```
i₂ : {p q : Polynomial} → Lens q (p + q)
i₂ = inj₂ ⇆ λ _ → id
```

Given an object c as the codomain of two functions f and g, $[f, g]$ should be the (unique) factorization lens of f and g. This function is defined by using f or g's behavior depending what the position corresponds to.

```
[_,_]ₚ : {p q r : Polynomial} → Lens p r → Lens q r → Lens (p + q) r
[ f ⇆ f♯ , g ⇆ g♯ ]ₚ = [ f , g ] ⇆ [ f♯ , g♯ ]
```

The proofs needed correspond to the ones from the product.

```
inject₁ : {f : Lens p a} {g : Lens q a} → [ f , g ]ₚ ∘ₚ i₁ ≡ f
inject₁ = refl

inject₂ : {f : Lens p a} {g : Lens q a} → [ f , g ]ₚ ∘ₚ i₂ ≡ g
inject₂ = refl

unique : {F : Polynomial} {h : Lens (A + B) F} {f₁ : Lens A F} {f₂ : Lens B F}
    → (h ∘ₚ i₁) ≡ f₁
    → (h ∘ₚ i₂) ≡ f₂
    → [ f₁ , f₂ ]ₚ ≡ h
unique p₁ p₂ = ...
    where
        lemma : [ h ∘ₚ i₁ , h ∘ₚ i₂ ]ₚ ≡ h
        lemma = ...
```

3.7.1 Generalized coproduct

The coproduct can also be generalized to work with any amount of polynomials indexed by an index-type. This corresponds to the sigma type, or exists operator.

```
ΣPoly : Σ[ indexType ∈ Set ] (indexType → Polynomial) → Polynomial
ΣPoly (indexType , generatePoly) = mkpoly pos dir
  where
    -- It is the positions of one of the polynomials
    pos : Set
    pos = Σ[ index ∈ indexType ] (position (generatePoly index))

    -- It is the direction of the polynomial for the position
    dir : pos → Set
    dir (index , positionAtIndex) = direction (generatePoly index)
                                              positionAtIndex
```

Using the generalized coproduct with bool as index type, we recover the normal binary coproduct. This is proved in `Various.agda`.

```
coproductIsΣPoly : {p q : Polynomial}
```

```
  → ΣPoly (Bool , tupleToFunFromBool (p , q)) ≡ p + q
coproductIsΣPoly = ...
```

The generalized factorizer for the generalized coproduct can now be defined.

```
-- A function (A+B)→C is the same as two functions A→C and B→C
universalPropertyCoproduct : {Index : Type} {generate : Index → Polynomial}
   → Lens (ΣPoly (Index , generate)) p ≡ ((i : Index) → Lens (generate i) p)
universalPropertyCoproduct = ...
```

3.7.2 Monoid

To show the coproduct is a monoid, we do almost the same as for the product. Again we rely on `agda-categories`' helpers to define the monoidal category (**Poly**, $+$, $\mathbb{0}$) - which is also *symmetric* monoidal. This time we rely on the fact that the category is *co*-cartesian, but the code is almost identical:

```
-- Categorical/Poly/Monoidal/Coproduct.agda

open import Categories.Category.Monoidal
import Categories.Category.Cocartesian as Cocartesian

binaryCoproducts : Cocartesian.BinaryCoproducts Poly
binaryCoproducts = record { coproduct = coprod }

coproductCocartesian  : Cocartesian.Cocartesian Poly
coproductCocartesian = record { initial = initialZero ;
                                coproducts = binaryCoproducts }

coproductMonoidal : Monoidal Poly
coproductMonoidal =
  Cocartesian.CocartesianMonoidal.+-monoidal Poly coproductCocartesian

open import Categories.Category.Monoidal.Symmetric coproductMonoidal
productSymmetricMonoidal : Symmetric
productSymmetricMonoidal =
  Cocartesian.CocartesianSymmetricMonoidal.+-symmetric Poly coproductCocartesian
```

3.7.3 Coproductized lens

It is also easy to define a parallel coproduct lens. Like for the product, two lenses can be made to represent a single lens between coproducts.

```
⟨_⊎_⟩ : {p q r w : Polynomial} → (f : Lens p r) (g : Lens q w)
         → Lens (p + q) (r + w)
⟨_⊎_⟩ {p} {q} {r} {w} (f ⇆ f♯) (g ⇆ g♯) = mp ⇆ md
    where mp : position (p + q) → position (r + w)
          mp = map f g
```

```
        md : (fromPos : position (p + q)) →
             direction (r + w) (mp fromPos) →
             direction (p + q) fromPos
        md (inj₁ x) d = f♯ x d
        md (inj₂ y) d = g♯ y d
infixl 30 ⟨_⊎_⟩
```

3.8 Parallel product

The parallel product (also called Dirichlet product) is another useful binary operator on polynomials. It is defined as the pair of positions and pair of directions. This is the first operator that does not have a direct high school algebra correspondence, as it multiplies on both coefficients and exponents: given $p(y) = 2y^4 + 3y^2$ and $q(y) = 2y^3$, $(p \otimes q)(y) = 4y^{12} + 6y^6$.

```
_⊗_ : Polynomial → Polynomial → Polynomial
(mkpoly posA dirA) ⊗ (mkpoly posB dirB)
    = mkpoly (posA × posB) (λ (posA , posB) → (dirA posA) × (dirB posB))
```

The parallel product will turn out to be extremely important in the context of dynamical systems. More on this in the applications chapter. A visualization of it is provided in Figure 3.5.

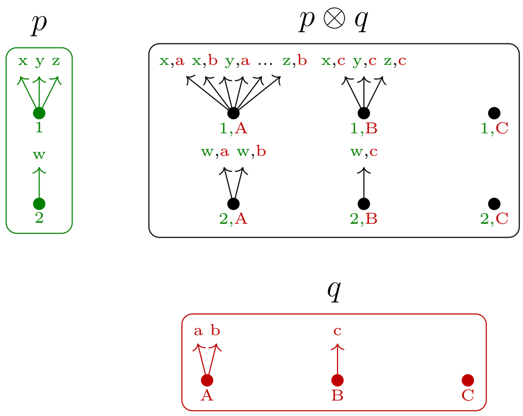


Figure 3.5: $p(y) = y^3 + y$, $q(y) = y^2 + y + 1$. Like the regular product, their parallel product $p \otimes q$ has both p and q's positions, with each position of $p \otimes q$ being a pair with one position of p and one of q. However, the directions at that position of $p \otimes q$ is the product of the directions at the originals, with each direction being a pair of the original directions.

3.8.1 Monoid

The parallel product forms a monoid with y. Since it is not the canonical product (or coproduct), as far as we know, we can't rely on the previous tricks to construct a monoidal category out of it. However, we can define the monoidal category construction manually, which is a lot of code in "volume", so we omit it from the text. The code can be found in `Categorical/Poly/Monoidal/ParallelProduct.agda`.

3.8.2 Parallel lens

The parallel product also enables turning to lenses, between individual polynomials, into one that goes between the parallel products. Here is the definition in Agda:

```
⟨_⊗_⟩ : {p q r w : Polynomial} → Lens p r → Lens q w →
      Lens (p ⊗ q) (r ⊗ w)
⟨_⊗_⟩ {p} {q} {r} {w} (f ⇆ f♯) (g ⇆ g♯) = mp ⇆ md
  where mp : position (p ⊗ q) → position (r ⊗ w)
        mp (posp , posq) = f posp , g posq
        md : (fromPos : position (p ⊗ q)) →
             direction (r ⊗ w) (mp fromPos) →
             direction (p ⊗ q) fromPos
        md (posp , posq) (dirr , dirw) = f♯ posp dirr , g♯ posq dirw
```

This particular action will have deep implications in the applications chapter.

3.9 Composition product

Since polynomials are functors, and in fact *endo*functors, they can be composed to give rise to new polynomial functors. The composition operator is given by the symbol $\triangleleft$. Here we can recover high-school algebra intuition. Composing functors corresponds to plugging in entire expressions, instead of numbers, into a polynomial. For instance, given $p(y) = y^2$ and $q(y) = y^2 + 3y$, the composition product is given as $(p \triangleleft q)(y) = (y^2 + 3y)^2$.

```
-- Proposition 5.2, page 158. Note: not same definition used.
-- We here treat positions  as inhabitants of the same set,
-- which makes a lot of proofs easier down the line.
_◃_ : Polynomial → Polynomial → Polynomial
p ◃ q = mkpoly pos dir
  where
    module p = Polynomial p
    module q = Polynomial q

    pos : Set
    pos = (Σ[ i ∈ p.position ] (p.direction i → q.position))

    dir : pos → Set
    dir (i , j) = Σ[ a ∈ p.direction i ] q.direction (j a)
infixl 27 _◃_
```

This definition is heavily inspired by David Orion Girardo's implementation here [27]. The key insight from his implementation is that the positions in the resulting polynomial are a pair, the first element of which is the same set as the postcomposed polynomial. This sticks closer to the idea that polynomials are in fact being composed. In terms of corolla forests, the composition product stacks trees on top of one another *in every possible way*, with the left polynomial in the bottom. The resulting polynomial's directions at any given position is "all directions that reach a height of two". A visualization is given in Figure 3.6.

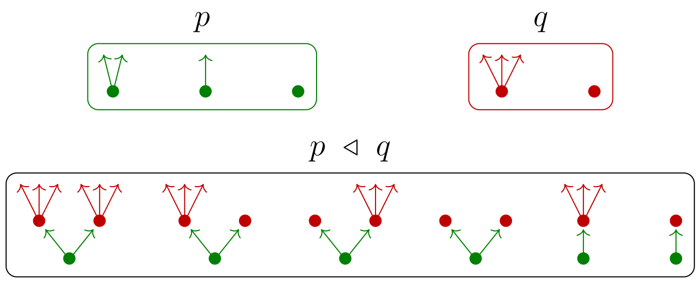


Figure 3.6: $p(y) = y^2 + y$ and $q(y) = y^3 + 1$. The composition product produces a polynomial that has as many positions as there are ways to stack q's trees on top of p's, and whose directions at each position are the ones that reach the top of this stack.

3.9.1 Monoid

The composition product is an important monoidal structure in **Poly**. It is not symmetric, like the previous three structures, but it also has an important interpretation in dynamics. It is also used in the definition of the exponential object. The proofs of left and right unit are complete, but the proof for association contains a hole, that is obviously fillable, but contains some constant transports that needs to be taken care of. The code exists in `Categorical/Poly/Monoidal/CompositionProduct`.

3.9.2 Composition lens

Again, it is possible to define a lens that goes to either side of the monoidal structure. This is the definition in Agda:

```
-- Apply lenses to both sides of the monoidal structure
⟨_◁_⟩ : {p q r w : Polynomial} → Lens p r → Lens q w →
      Lens (p ◁ q) (r ◁ w)
⟨_◁_⟩ {p} {q} {r} {w} (f ⇆ f♯) (g ⇆ g♯) = mapPos ⇆ mapDir
  where mapPos : position (p ◁ q) → position (r ◁ w)
        mapPos (posP , dirPToPosQ) = f posP , g ∘ dirPToPosQ ∘ f♯ posP
        mapDir : (fromPos : position (p ◁ q)) →
                 direction (r ◁ w) (mapPos fromPos) →
                 direction (p ◁ q) fromPos
        mapDir (posP , dirPToPosQ) (dirR , dirW) =
          (f♯ posP dirR) , g♯ (dirPToPosQ (f♯ posP dirR)) dirW
infixl 28 ⟨_◁_⟩
```

3.9.3 Composite power

Composing a polynomial with itself, `p ◁ p`, a repeated amount of times can be done with `compositePower`. Given the interpretation from figure 3.6 it builds a big tree of depth n with all combinations of p rooting from each node.

```
compositePower : Polynomial → ℕ → Polynomial
compositePower p zero = compositionUnit
compositePower p (suc n) = p ◁ (compositePower p n)
```

3.10 Cartesian closure

Poly is cartesian closed, which means it has exponential objects, that is, the family of lenses from any polynomial q to any polynomial r is found as an object r^q in **Poly**. This object is given by the following polynomial:

$$r^q = \prod_{j \in q(1)} r \triangleleft (y + q[j]) \tag{3.4}$$

where the notation $q[j]$ represents the set of directions of q at position j.

We define this polynomial as follows in Agda:

```
-- CategoryData/Exponential

-- Exponential object.
-- Theroem 4.27, page 130 in Poly book.
_^_ : (r : Polynomial) → (q : Polynomial) → Polynomial
r ^ (mkpoly posQ dirQ) = ΠPoly (posQ , λ j → r ◁ (Y + Constant (dirQ j)))
infixl 30 _^_
```

where `Constant` is a helper function that takes a set to the constant polynomial sending every set to that set.

We do not prove that this object is the exponential object via the universal property of exponential objects, which would employ `agda-categories`' `Exponential` construction. Instead, we prove an equivalent statement, related to currying: that the following natural isomorphism exists

$$\mathbf{Poly}(p, r^q) \cong \mathbf{Poly}(p \times q, r)$$

This isomorphism states that there is an equivalent number of ways to go from an arrow to an exponential object, as there are ways of going from an object and the exponential's argument to the exponential's output. This statement is the same as the poly-book proves. In fact, we follow the same chain of isomorphisms as the book to conclude this. In Agda, this looks like this:

```
-- Page 131
cartesianClosed : {p q r : Polynomial} → Lens p (r ^ q) ≡ Lens (p * q) r
cartesianClosed = zero · one · two · three · four · five
```

These steps' types are given next, but their proofs are omitted due to their lengths. They correspond exactly to the steps in the book.

```
-- By definition of exponential object. 4.27
zero : Lens p (r ^ q)
       ≡
       Lens p (ΠPoly (position q , λ j → r ◁ Y + Constant (direction q j)))
-- By universal property of products and coproducts.
one : ...
      ≡
      ((i : position p) →
       (j : position q) →
       Lens (purePower (direction p i)) (r ◁ Y + Constant (direction q j)))
two : ...
      ≡
      ((i : position p) →
       (j : position q) →
       (r ◁ Y + Constant (direction q j)) ⦅ direction p i ⦆)
three : ...
        ≡
        ((i : position p) →
         (j : position q) →
         Σ[ k ∈ position r ]
           ((direction r k) → (direction p i) ⊎ (direction q j)))
-- By 2.84
four : ...
       ≡
       ((i×j : position (p * q)) →
       Σ[ k ∈ position r ] (direction r k → direction (p * q) i×j))
-- By 2.54
five : ...
       ≡
       Lens (p * q) r
```

Under the hood, many different lemmas and helper function are needed for the proof. This includes the Yoneda lemma for **Poly**, generalized universal product (and coproduct) property, and a whole file (`Cubical/Utils/CoproductUtils`) with lemmas about the coproduct, which is needed due to how the exponential object is defined.

The exponential object implies the existence of the canonical evaluation lens - one that takes an exponential object paired with its argument to the output object. Its implementation is as follows:

```
eval : {p q : Polynomial} → Lens ((q ^ p) * p) q
eval {p} {q} = mapPos ⇆ mapDir
  where
      mapPos : position ((q ^ p) * p) → position q
```

```
    mapPos (posQ^P , posP) = fst (posQ^P posP)

    mapDir : (pos : position ((q ^ p) * p))
        → direction q (mapPos pos)
        → direction ((q ^ p) * p) pos
    mapDir (posQ^P , posP) dir
      = recoverLeft (snd (posQ^P posP) dir) λ x → posP , dir , x
```

There is another way to define the evaluation lens which is to transport the identity lens on q^p through the cartesianClosed equality. This is the definition used in the book.

```
eval' : {p q : Polynomial} → Lens ((q ^ p) * p) q
eval' {p} {q} = subst (λ x → x) path idLens'
    where
        path : Lens (q ^ p) (q ^ p) ≡ Lens (q ^ p * p) q
        path = (cartesianClosed {q ^ p} {p} {q})

        idLens' : Lens (q ^ p) (q ^ p)
        idLens' = idLens {q ^ p}
```

The cartesian closure of **Poly** has interesting implications, which we will consider when we talk about future work possibilities from this thesis.

3.11 Quadruple adjunction with Set

Poly has a deep relationship with **Set**. One place where this is reflected is with the connection between the following functors:

- Constant functor C : **Set** → **Poly**, which sends a set A to the constant polynomial $p(y) = A$; that is, the polynomial that sends all sets to A. This is a fully faithful functor, which means **Set** is a full subcategory of **Poly**.
- Linear functor L : **Set** → **Poly**, which sends a set A to the linear polynomial $p(y) = Ay$; that is, the polynomial that sends a set y to the set of A-tuples of y. This is also a fully faithful functor.
- Plug in $\underline{0}$ functor $p(0)$: **Poly** → **Set**, which sends a polynomial to a subset of its positions, namely the constant ones. For example, its action on $p(y) = 5y^{\mathbb{R}} + \mathbb{N}$ is $p(y) = 5 * 0^{\mathbb{R}} + \mathbb{N} = \mathbb{N}$. Again, $\underline{0}$ here stands for the set with 0 elements.
- Plug in $\underline{1}$ functor $p(1)$: **Poly** → **Set**, which sends a polynomial to the set of its positions. Its action on the example polynomial above is $p(y) = 5 * 1^{\mathbb{R}} + \mathbb{N} = 5 + \mathbb{N}$.

Note that the notation $p(1)$ here is being used to represent a functor from **Poly** to **Set**, but most of the time it is used to talk about the set of positions directly.

These functors turn out to form a chain of adjunctions: $L \dashv p(1) \dashv C \dashv p(0)$. Adjunctions are forms of weak equivalences between categories, given by pairs of functors: the left and right adjoints. The intuition is that the left adjoint "adds as much information" as possible to the objects and morphisms it acts on, whereas the right adjoint "removes as little information" as possible in order to make the objects and morphisms it acts on "fit" in its target category. In the case of the relationship between **Poly** and **Set**, there are two functors putting different copies of **Set** inside **Poly** (the fully faithful ones), and two functors that "forget" that they act on something *containing* sets.

For an illustration of what is meant by "forgetting", take the the first adjoint pair $L \dashv p(1)$ as an example. It acts the following way: $p(1)$ makes a polynomial functor "forget" it's a functor, while L "reminds" a set of "how to be" a functor.

The proof that this adjunction exists is relatively trivial for these functors, so we won't include it in the main text, but it is of course present in the accompanying formalization, under `Categorical/Adjunction`. It is done via the `agda-categories` scaffolding and via the unit-counit definition of adjunctions.

3.12 Equalizer

Equalizers are an important categorical structure, because when they exist in a category that also has products, it can be shown that the category is complete - in other words, it has all finite limits. A limit is an object, together with several morphisms from that object, in a category that "represent" or somehow summarize diagrams in that category.

An equalizer in general is an object E and a morphism eq, such that the following diagram commutes:

$$
\begin{tikzcd}
E \arrow[r, "eq"'] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \\
T \arrow[u, dashed, "!"] \arrow[ur, "h"] & &
\end{tikzcd}
$$

The objects A, B and T and the arrows f and g are arbitrary, and the morphism h is such that $f \circ h = g \circ h$. The equalizer object and its arrow are named this way for the intuitive reason that they select the part of A under which both arrows f and g agree, i.e. they "give the same output" in B. The notion of "giving output" is perhaps exploiting a **Set**-based intuition, but we believe **Poly** is close enough to **Set** to justify this. The object T relates to the universal property of the equalizer, in that it can be thought of as another "candidate" for the equalizer object along with its arrow h. What makes the actual equalizer E special is that, for any such candidate, there exists a unique arrow ! from that object to the equalizer, such that $eq \circ ! = h$.

The dual construction to the equalizer is the coequalizer, and as usual, is obtained by reversing all the arrows:

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "inc"'] \arrow[dr, "h"'] & C \arrow[d, dashed] \\
& & T
\end{tikzcd}$$

The intuition behind the coequalizer, in a **Set** context, is that it inspects the functions f and g, and any time these functions disagree on an input $a : A$, the two targets $b_1 : B$ and $b_2 : B$ get collapsed together by the arrow *inc*. This is done for all inputs. So, for example, if the functions disagree on all inputs, the set C becomes the singleton set, because *inc* will have to collapse every disagreeing output, eventually collecting all of them.

In **Poly**, the equalizer has two parts, as most constructions. For two given lenses $f : p \to q$ and $g : p \to q$, the equalizer is a polynomial that:

- Has the equalizer set of the map on positions of these lenses. This easily expressible with Σ-types:

  ```
  EqualizedPosition : Set
  EqualizedPosition = Σ (position p)
                        (λ z → mapPosition f z ≡ mapPosition g z)
  ```

- Has the *co*equalizer set of the map on directions of these lenses *at their equalized positions*. Normally, the partially applied functions `mapDirection f posp` and `mapDirection g posp` would have different types, since the second argument (a direction in q) depends on on the values of `mapPosition f posp` and `mapPosition g posp`. But in this case, we know these values to be the same, since the map on positions is the equalizer map, so the partially applied maps on directions have the same type, which means they can be coequalized. Cubical has a construction for this, `SetCoequalizer`.

Before we can introduce the equalizer polynomial and lens, we must address an issue: in the Cubical setting, a value of type `SetCoequalizer` can only be constructed out of functions that target types that are sets in the HoTT sense, so types of truncation level 0. In our case, this means that the directions of our polynomials must be sets, which is not a guarantee we have in the **Poly** category we have been working with; we only guarantee that these polynomials' targeted sets have *universe* level 0. To get around this, we work in a category that is identical to **Poly**, except the directions of the polynomials are guaranteed to be sets. This is the category **SetPoly**, where the objects are:

```
record SetPolynomial : Set₁ where
    constructor mksetpoly
    field
        poly : Polynomial
        isDirSet : ∀ {p : position poly} → isSet (direction poly p)
```

and the morphisms are lenses as usual, just wrapped in a new type constructor:

```
record SetLens (from to : SetPolynomial) : Set where
    constructor ⇆ˢ
    field
        lens : Lens (poly from) (poly to)
```

With that, we can introduce the equalizer polynomial, which is then given by:

```
eqPoly : Polynomial
eqPoly = mkpoly EqualizedPosition $
        λ ( posp , equal ) →
            SetCoequalizer (mdf posp)
                            (λ x → mdg posp
                                        (subst (direction q) equal x))
```

and the equalizer lens is:

```
mpe : position (eqObj) → position p
mpe = fst
mde : (fromPos : position (poly eqObj)) →
      direction p (mpe fromPos) →
      direction (poly eqObj) fromPos
mde _ dir = inc dir
eqLens : SetLens eqObj pˢ
eqLens = ⇆ˢ (mpe ⇆ mde)
```

Proving that this is in fact the equalizer of the **SetPoly** category is quite involved, so we will omit it from the text. It is of course included in the code. The idea behind the proof, however, is simple: since an equalizing lens of two lenses $f, g : p \to q$ must make sure following it with either f or g is equal, then the maps comprising f and g must be guaranteed to land in the same results after it. That is, the map on positions must be equalized on the inputs, and the map on directions must be equalized on the outputs. Equalizing two functions on outputs is, loosely speaking, what a coequalizer map does: it identifies potentially differing outputs of two functions.

3.13 Various properties of polynomials

We went over the intuition for polynomial functors as algebraic polynomials the beginning of this chapter, and many of these properties have been formalized. For example, here is a proof that the sum of two constant polynomials is still constant, exercise 4.1 in the poly-book:

```
isConstant : Polynomial → Type₁
isConstant (mkpoly pos dir) = (p : pos) → dir p ≡ ⊥

constantClosedUnderPlus : {p q : Polynomial} →
    isConstant p →
    isConstant q →
```

```
    isConstant (p + q)
constantClosedUnderPlus isConstantP isConstantQ (inj₁ x) = isConstantP x
constantClosedUnderPlus isConstantP isConstantQ (inj₂ y) = isConstantQ y
```

Other proofs of this kind are provided in the files `Cubical/Proofs.agda` and `Cubical/Various.agda`. Here is a list of them:

- The coproduct is the generalized coproduct indexed by $\underline{2}$.
- The product is the generalized product indexed by $\underline{2}$.
- A sum of two linear polynomials is still linear.
- A multiplication of two monomials is still a monomial.
- A lens from a polynomial p to y is the same thing as a function from positions to directions in p. The idea is that the map on positions has no choice (a single position), and the map on directions then picks out a direction in the source polynomial.
- Given $p(y) = y^2 + y$, $p(\underline{2}) = \underline{6}$.
- $p(1)$ is equal to the set of positions of p.
- Given two constant polynomials $p(y) = \underline{3}$ and $q(y) = \underline{2}$, $p^q(y) = \underline{9}$.
- Definition of cartesian and vertical lenses and some related properties.

3.14 Comonoids in Poly(y, $\triangleleft$) are small categories

A fact that the book spends a lot of time on is that comonoids in the monoidal category **Poly**($\triangleleft$, y) are isomorphic to small categories (categories where the collection of objects forms a set). This means that within **Poly**, at least the objects of the category **Cat** of small categories exist, and indeed there is an unusual category of categories called **Cat**$\sharp$ that can be modeled in **Poly**, where the morphisms are *co*functors, the duals to functors. This is a deep fact, and suggests that **Poly** is a setting in which a considerable amount of category theory itself can be studied.

A comonoid in this monoidal category, as the dual to the monoid structure, consists of the following data:

- A polynomial c, called the *carrier*.
- A morphism $\delta : c \to c \triangleleft c$ called the the *duplicator*.
- A morphism $\epsilon : c \to y$, called the *eraser*.

And the following laws:

- $\langle \mathrm{id}_c \triangleleft \delta \rangle \circ \delta = \langle \delta \triangleleft \mathrm{id}_c \rangle \circ \delta$, called the *coassociativity* law. It is the dual to a monoid's associativity law, and means that there is no difference between duplicating, then duplicating on the left and duplicating, then duplicating on the right.

- $\sim^L = \langle \epsilon \triangleleft \mathrm{id}_c \rangle \circ \delta$, the left counit law, saying that duplicating then erasing the left is the same as not doing anything.
- $\sim^R = \langle \mathrm{id}_c \triangleleft \epsilon \rangle \circ \delta$, the right counit law, saying that duplicating then erasing the right is the same as not doing anything.

The lenses $\sim^R : c \to c \triangleleft y$ and $\sim^L : c \to y \triangleleft c$ are just putting its source polynomial (in this case the carrier) in the right or left of the $\triangleleft$ product with a y, but because y is the monoidal unit for this product, this is roughly the same as not doing anything.

In Agda, the data is defined like the following type:

```
record Comonoid : Set₁ where
constructor Com
field
  c : Polynomial
  ε : Lens c Y
  δ : Lens c (c ◁ c)
  coassoc : ⟨ idLens {c} ◁ δ ⟩ ∘ₚ δ
            ≡
            transport (assoc⇆ {c}) (⟨ δ ◁ idLens {c} ⟩ ∘ₚ δ)
  leftCounit :  ~ᴸ ≡ ⟨ ε ◁ idLens {c} ⟩ ∘ₚ δ
  rightCounit : ~ᴿ ≡ ⟨ idLens {c} ◁ ε ⟩ ∘ₚ δ
```

The `transport` uses the proof of associativity of the $\triangleleft$ operation to send the right-hand side to the same type as the left side. This proof is complicated, and makes this record hard to work with. We were therefore not able to complete this proof, but our attempt is included in the code. It consists of constructing an `Iso` value, with the two legs `comonoidsAreCategories : Category zero zero zero → Comonoid` and `categoriesAreComonoids : Comonoid → Category zero zero zero` being mutual inverses. The attempt includes most of the second leg and parts of the first leg.

4

Theory: Charts

The previous chapter covered the category **Poly**, of polynomials and lenses. This chapter covers a closely related category, the category of polynomials and charts, **Chart** [1]. First, the category **Chart** is defined, then some universal constructions are proved, and finally the relation between **Poly** and **Chart** as commuting squares is shown.

4.1 The category Chart

The objects of **Chart** are exactly the same as **Poly**: polynomial functors. What differs are the arrows.

4.1.1 Chart

Arrows in **Chart** are called charts. The difference between charts and lenses is that while lenses has the map on directions going backwards, charts has the map on directions going forwards. This makes charts easier to reason about, since both the map on directions and the map on positions go forward, like figure 4.1 shows:

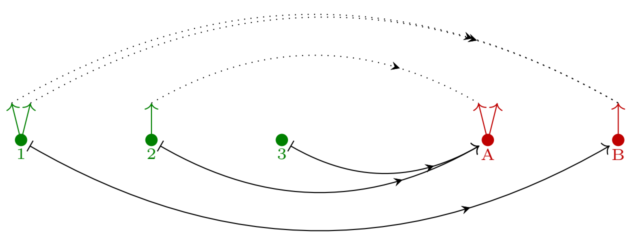


Figure 4.1: The polynomial $p = y^2+y+1$ on the left has 3 positions, and $q(y) = y^2+y$ on the right has 2 positions. This figure shows a chart from p to q. The map on position is a map between p's and q's positions, like in lenses. The map on direction is for each position of p, a map between that position's directions and the directions of the mapped position in q.

[1] The category **Chart** with charts as morphisms could also be called the "arrow-category" because it is the arrow-category of set.

The definition of `Chart` in Agda is as follows:

```
record Chart (p q : Polynomial) : Set where
    constructor _⇉_
    field
        mapPos : position p → position q
        mapDir : (i : position p) → direction p i → direction q (mapPos i)
```

The constructor ⇉ is used to indicate that both the maps go forward. Notably, despite being maps between functors, charts are *not* natural transformations. Trying to formulate it as an `agda-categories` natural transformation like below, with the `asEndo` function shown earlier, leads to an unfillable hole:

```
asNatTransChart : {p q : Polynomial} →
    Chart p q →
    NaturalTransformation (asEndo p) (asEndo q)
asNatTransChart (f ⇉ f♯) = record {
    η = λ { X (posP , dirP) → (f posP) , (λ x → dirP {!   !}) } ;
    ...
    }
```

In the hole, the desired type is `direction p posP`, but there is no such data to provide; `x` is of type `direction q (f posP)`.

4.1.2 Identity chart

The identity chart, in similarity to the identity lens, maps positions to itself and directions to itself. In code:

```
idChart : {p : Polynomial} → Chart p p
idChart = id ⇉ λ _ → id
```

4.1.3 Composing charts

Composition of charts is naturally defined as following all the arrows. For the map on position function composition is used. For the map on directions, function composition is used as well, but with some extra care in dealing with the dependency of the map on positions.

```
_∘c_ : {p q r : Polynomial} → Chart q r → Chart p q → Chart p r
(f ⇉ f♭) ∘c (g ⇉ g♭) = (f ∘ g) ⇉ (λ i → f♭ (g i) ∘ g♭ i)
```

4.1.4 Associativity and identity laws

The associativity, left identity, and right identity laws are proved directly by `refl`. This fulfills all the requirements of a category and thus **Chart** is fully defined.

4.2 Chart equality

In similarity to polynomials and lenses, charts also needs a characterization of equality. Thus the same procedure is used, to define charts as Σ-type, and show that the Σ-type is equal to the record definition.

```
ChartAsΣ : (p q : Polynomial) → Type
ChartAsΣ p q = Σ[ mapPos ∈ (position p → position q) ]
                   ((i : position p) → direction p i → direction q (mapPos i))

chartAsΣToChart : {p q : Polynomial} → ChartAsΣ p q → Chart p q
chartAsΣToChart (mapPos , mapDir) = mapPos ⇉ mapDir

chartToChartAsΣ : {p q : Polynomial} → Chart p q → ChartAsΣ p q
chartToChartAsΣ (mapPos ⇉ mapDir) = mapPos , mapDir

chartAsΣ≡Chart : {p q : Polynomial} → ChartAsΣ p q ≡ Chart p q
chartAsΣ≡Chart {p} {q} = isoToPath
    (iso chartAsΣToChart chartToChartAsΣ (λ b → refl) λ a → refl)
```

Equality of charts is now defined for the Σ-type and then transferred to the chart defined as a record. There are two variants of chart equality, the direct, and slightly weaker variant, `chart≡`. As well as the more powerful variant, `chart≡∀`.

The weak variant of chart equality is directly defined by using `ΣPathTransport→PathΣ`. This results in the following type.

```
chart≡ : {p q : Polynomial} {f g : Chart p q}
    → (mapPos≡ : mapPos f ≡ mapPos g)
    → subst (λ x → (i : position p) → direction p i → direction q (x i))
      mapPos≡ (mapDir f) ≡ mapDir g
    → f ≡ g
```

The `subst` for the map on direction equality proof can easily be simplified. This results in the more powerful variant of chart equality, corresponding to $\texttt{lens≡}_p$, which is defined by using `ΣPathP` and function extensionality twice. This result in the following type.

```
chart≡∀ : {p q : Polynomial} {f g : Chart p q}
    → (mapPos≡ : mapPos f ≡ mapPos g)
    → ((i : position p) → (x : direction p i)
        → subst (λ h → direction q (h i)) mapPos≡
          (mapDir f i x) ≡ mapDir g i x)
    → f ≡ g
```

With chart equality available, some universal constructions in **Chart** are proved.

4.3 Initial object

The initial object of **Chart** is the same object 𝟘 as for **Poly**. Of course, the chart from 𝟘 to any other polynomial is not the same as the lens from 𝟘, since the types are different. But the implementation is the same, since the position is ⊥, the absurd function is used.

```
chartFromZero : {p : Polynomial} → Chart 𝟘 p
chartFromZero = (λ ()) ⇉ (λ ())
```

The uniqueness proof is identical to the uniqueness proof for the initial object in **Poly**.

```
unique : {p : Polynomial} (f : Chart 𝟘 p) → chartFromZero ≡ f
unique _ = chart≡ (funExt λ ()) (funExt λ ())
```

4.4 Terminal object

The terminal object of **Chart** is not the same as for **Poly**! The terminal object of **Chart** is Y, having one position, and exactly one direction at that position.

```
Y : Polynomial
Y = mkpoly ⊤ (λ _ → ⊤)
```

The chart from any polynomial to Y is implemented as unit for both the map on positions and map on directions.

```
chartToY : {p : Polynomial} → Chart p Y
chartToY = (λ _ → tt) ⇉ (λ _ _ → tt)
```

Uniqueness of `chartToY` follows directly from the uniqueness of the unit function.

```
unique : {p : Polynomial} (f : Chart p Y) → chartToY ≡ f
unique _ = chart≡ refl refl
```

4.5 Product

The product of **Chart** is also not the same product as **Poly**; instead it is the parallel product ⊗ defined in 3.8. The projections from `p ⊗ q` is defined by projecting both the position and directions.

```
π₁ : {p q : Polynomial} → Chart (p ⊗ q) p
π₁ = fst ⇉ λ i → fst

π₂ : {p q : Polynomial} → Chart (p ⊗ q) q
π₂ = snd ⇉ λ i → snd
```

The factorization chart is given by running the two functions in parallel on both the map on position and map on directions.

```
⟨_,_⟩ : {p q r : Polynomial} → Chart p q → Chart p r → Chart p (q ⊗ r)
⟨ f ⇉ f♭ , g ⇉ g♭ ⟩ = (λ i → (f i , g i)) ⇉ (λ i d → f♭ i d , g♭ i d)
```

The proofs for commutation is given as `refl`, and uniqueness is given by slightly longer proof using a lemma. Full code can be found in `Categorical/Chart/Product.agda`.

4.5.1 Monoid

Chart of course also forms a symmetric monoidal category with respect to the product and Y, and we prove this by again using the `agda-categories` helpers, similarly to **Poly**.

```
binaryProducts : Cartesian.BinaryProducts ChartCat
binaryProducts = record { product = prod }

cartesian : Cartesian.Cartesian ChartCat
cartesian = record { terminal = terminalY ;
                     products = binaryProducts }

productMonoidal : Monoidal ChartCat
productMonoidal
    = Cartesian.CartesianMonoidal.monoidal ChartCat cartesian

open import Categories.Category.Monoidal.Symmetric productMonoidal
productSymmetricMonoidal : Symmetric
productSymmetricMonoidal
    = Cartesian.CartesianSymmetricMonoidal.symmetric ChartCat cartesian
```

4.6 Coproduct

The coproduct of charts is the same as for lenses. The injections and factorization are defined the same as well.

```
i₁ : {p q : Polynomial} → Chart p (p + q)
i₁ = inj₁ ⇉ λ _ → id

i₂ : {p q : Polynomial} → Chart q (p + q)
i₂ = inj₂ ⇉ λ _ → id

[_,_]c : {p q r : Polynomial} → Chart p r → Chart q r → Chart (p + q) r
[ f ⇉ f♭ , g ⇉ g♭ ]c = [ f , g ] ⇉ [ f♭ , g♭ ]
```

The proofs are also similar, and can be found in `Categorical/Chart/Coproduct.agda`.

4.6.1 Monoid

Again, the coproduct with the initial object form a symmetric monoidal category, and we use the same tactic as before to define it. The code can be found in

`Categorical/Chart/Monoidal/Coproduct.agda`.

4.7 Commuting square between lenses and charts

One interesting thing about charts is how they behave together with lenses. This behavior can be condensed into a square of four polynomials, with lenses and charts going between them, such that the square commutes. Such a square is shown in figure 4.2.

$$
\begin{tikzcd}
p_1 \arrow[r, shift left, "f\flat"] \arrow[r, shift right, "f"'] \arrow[d, shift right, "w"'] & p_2 \arrow[d, shift right, "v"'] \\
p_3 \arrow[u, shift right, "w\sharp"'] \arrow[r, shift left, "g\flat"] \arrow[r, shift right, "g"'] & p_4 \arrow[u, shift right, "v\sharp"']
\end{tikzcd}
$$

Figure 4.2: A square consists of four polynomials. Vertically there is a lens from p_1 to p_3 and from p_2 to p_4. Horizontally there is a chart from p_1 to p_2 and from p_3 to p_4.

For a square to commute there are two requirements. The first requirement is that the map on positions commute. Starting at a position in p_1, following w and then g, ending up at a position in p_4, should be the same as following f and then v.

The second requirement is that the map on directions commute. That is, $w\sharp$ followed by $f\flat$ should be the same as $g\flat$ followed by $v\sharp$. Since these functions depend on the different map on positions there are some plumbing to make this type correct, which can be seen in the Agda definition down below. Furthermore, the type expressing the map on directions commuting depends on the map on positions commuting. This explains the use of a Σ-type as well as the use of `subst` to make the types match. Here's the code for this:

```
LensChartCommute : {p₁ p₂ p₃ p₄ : Polynomial} (w : Lens p₁ p₃) (v : Lens p₂ p₄)
    (f : Chart p₁ p₂) (g : Chart p₃ p₄) → Type
LensChartCommute {p₁} {p₂} {p₃} {p₄} (w ⇆ w♯) (v ⇆ v♯) (f ⇉ f♭) (g ⇉ g♭)
    = Σ mapPos≡ mapDir≡
    where
        mapPos≡ : Type
        mapPos≡ = (i : position p₁) → v (f i) ≡ g (w i)

        mapDir≡ : mapPos≡ → Type
        mapDir≡ p≡ = (i : position p₁) → (x : direction p₃ (w i))
            → f♭ i (w♯ i x) ≡
                v♯ (f i) (subst (direction p₄) (sym (p≡ i)) (g♭ (w i) x))
```

The use of commuting squares will be of importance in reducing dynamical systems to simpler ones, which is covered the applications chapter.

4.8 Composing squares between lenses and charts

Commuting squares are also composable in two different ways, horizontally and vertically, as shown in figures 4.3 and 4.4.

$$\begin{tikzcd}
p_1 \arrow[r, shift left, "f\flat"] \arrow[r, shift right, "f"'] \arrow[d, shift right, "w"'] & p_2 \arrow[r, shift left, "h\flat"] \arrow[r, shift right, "h"'] \arrow[d, shift right, "v"'] & p_5 \arrow[d, shift right, "m"'] \\
p_3 \arrow[r, shift left, "g\flat"] \arrow[r, shift right, "g"'] \arrow[u, shift right, "w\sharp"'] & p_4 \arrow[r, shift left, "r\flat"] \arrow[r, shift right, "r"'] \arrow[u, shift right, "v\sharp"'] & p_6 \arrow[u, shift right, "m\sharp"']
\end{tikzcd}$$

Figure 4.3: Horizontal composition, done through the h and r charts.

$$\begin{tikzcd}
p_1 \arrow[r, shift left, "f\flat"] \arrow[r, shift right, "f"'] \arrow[d, shift right, "w"'] & p_2 \arrow[d, shift right, "v"'] \\
p_3 \arrow[r, shift left, "g\flat"] \arrow[r, shift right, "g"'] \arrow[u, shift right, "w\sharp"'] \arrow[d, shift right, "h"'] & p_4 \arrow[u, shift right, "v\sharp"'] \arrow[d, shift right, "m"'] \\
p_5 \arrow[r, shift left, "r\flat"] \arrow[r, shift right, "r"'] \arrow[u, shift right, "h\sharp"'] & p_6 \arrow[u, shift right, "m\sharp"']
\end{tikzcd}$$

Figure 4.4: Vertical composition, done through the h and m lenses.

Translating these diagrams into Agda leads to the following types:

```
horizontalComposition : {p₁ p₂ p₃ p₄ p₅ p₆ : Polynomial}
    (f : Chart p₁ p₂) (g : Chart p₃ p₄) (h : Chart p₂ p₅) (r : Chart p₄ p₆)
    (w : Lens p₁ p₃) (v : Lens p₂ p₄) (m : Lens p₅ p₆)
    → LensChartCommute w v f g → LensChartCommute v m h r
    → LensChartCommute w m (h ∘c  f) (r ∘c g)


verticalComposition : {p₁ p₂ p₃ p₄ p₅ p₆ : Polynomial}
    (f : Chart p₁ p₂) (g : Chart p₃ p₄) (r : Chart p₅ p₆) (h : Lens p₃ p₅)
```

```
(w : Lens p₁ p₃) (v : Lens p₂ p₄) (m : Lens p₄ p₆)
→ LensChartCommute w v f g → LensChartCommute h m g r
→ LensChartCommute (h ∘ₚ w) (m ∘ₚ v) f r
```

Their definition is completed in the accompanying formalization for the map on positions part of the proofs. However, the parts dealing with map on directions is unfinished, due to the complexity of an explosion of `subst`'s, and left for future work, discussed further in 6.6.3.

5

Applications

As mentioned in the background section, polynomials have applications to many different areas of study, of which the main three highlighted in the poly-book are dynamical systems, decision systems and databases. Our implemented applications are all instances of the first: we pay no attention to the other two perspectives. Several of our examples are listed in the book and its associated YouTube lecture series, but a few are our own contribution.

It is worth mentioning that many of the ideas in this chapter come from the Categorical Systems Theory book [28], even though that book does not even mention polynomial functors. However, it explains how dynamical systems are modeled by lenses in extreme detail. It also pays a lot of attention to the distinction between different theories of dynamical systems: stochastic, discrete, continuous etc. This matters to us because several of our examples are continuous systems that are discretized, so strictly speaking we are working within a theory of discrete dynamical systems. David Jaz Myers, the author, is also responsible for the chart-related part of the applications, through his guest lectures on the YouTube series as well as his book.

We will start by talking about the class of systems that **Poly** mainly talks about - ones that can be expressed through lenses. We will then explain what our categorical constructs mean in the context of application to dynamical systems, and we will end by showcasing implemented examples of these dynamical systems, some of which clearly display the advantages of this categorical framework.

5.1 Mode-dependent open dynamical systems

First, let us define what a dynamical system is, starting with the simplest case: a closed dynamical system. It can be thought of as a machine with some functionality that "runs" through time. It has a notion of a state space S (in our case a set of values) and a function $f : S \to S$, called the *dynamics* of the system. The purpose of the dynamics is to reveal what the system's state is at each timestep. Given an initial state S_0, one can then run several steps of a system by simply composing f with itself that many times. This sort of system is called *closed* because it does not take any inputs apart from the initial state, and does not provide any outputs. The only thing one might be interested in is the state of the system. An example of a closed dynamical system is the logistic map, defined by the following equation:

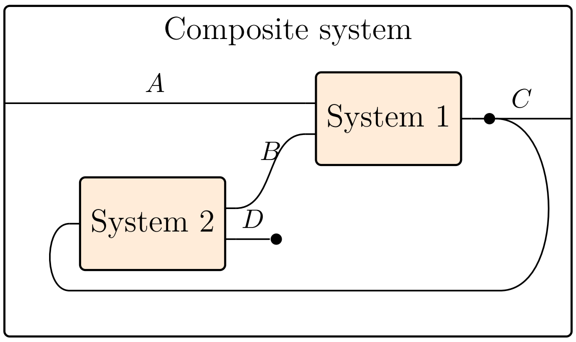
$$x_{t+1} = rx_t(1 - x_t)$$

The state space S here is $\mathbb{R}$, and the body of the function f is the expression on the right. The next state is defined in terms of the last state and a parameter r, so given an initial state x_0, one can generate an infinite sequence of states by running this function repeatedly. *Open* dynamical systems on the other hand are systems that, in addition to possessing a state space, also provide output to and receive input from their environment. "Environment" here is meant to be interpreted loosely: it is the collection of surrounding systems or user inputs, or some other source of data. Open dynamical systems are then composed of five pieces of data: an input set A, an output set B, a state space set S, and two functions: `update` $: S \times A \to S$ and `readout` $: S \to B$. This naturally suggests composition: one systems's output can be another system's input, if the types match.

Finally, *mode-dependent* open dynamical systems are ones in which the sets A and B can vary, and as such, which system is providing output to which other system's input can also change depending on the states of the system. This will become clearer in the next section, where we will provide visualizations of what is being described here.

5.1.1 Wiring diagrams

Wiring diagrams are a way of expressing morphisms of monoidal categories that is much more visually intuitive and applies to a wide array of contexts, including dynamical systems [29]. In our case, we will use them to express wiring patterns between open dynamical systems, and sometimes the systems themselves. A wiring pattern independent of the systems it is applied to might look the same as a concrete system in this scheme, but the distinction between them is important and will be explained. Here is an example of a wiring diagram, where each box represents an open system and the wires between them represents connections.



(5.1)

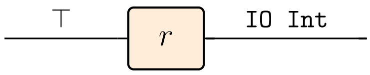
A, B, C and D here all abstractly represent sets/types. The wires do not have just any quantity flowing through them, but instead a specific kind of information. The outer system here arises out of a specific combination of the inner systems. An important fact about wiring patterns like the one above is that they give morphisms in **Poly**, while boxes give objects. This connection will be explored more deeply now.

5.2 Polynomial functors in dynamical systems

The three main objects of our study - polynomials, lenses and charts - have important roles in the study of dynamical systems, and the formalized categorical structures do as well, especially the parallel product $\otimes$.

5.2.1 Polynomials as interfaces

A polynomial can be seen as the interface or "API" of a dynamical system, by describing what its outputs and inputs are - the outputs are the set of positions, while the inputs correspond to directions at a particular position. The simplest way to visualize a polynomial as an interface is in the case of the monomial. Consider the monomial r, whose interface uses familiar types. $r(y) = \texttt{(IO Int)}y^{\top}$. As a box in the wiring diagram scheme, this corresponds to:

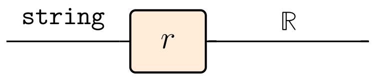


(5.2)

This is a straightforward view of an interface, because monomials cannot represent mode-dependence; that is, the types of outputs and inputs to a box like the one above cannot change, no matter what the state of a machine that implements this box is. If we consider a polynomial with different sets of summands however, the situation changes. Take the polynomial below for instance.

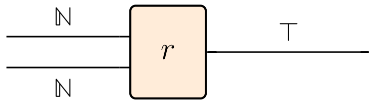
$$p(y) = \mathbb{R}y^{\texttt{string}} + \top y^{\mathbb{N}\times\mathbb{N}}$$

As an interface to an abstract system, it applies to any system that, when its state is such that it outputs real numbers, the possible inputs are strings. When its state is such that it is outputting $\top$ (the singleton set), then the possible inputs to give it are pairs of natural numbers. Notice that this says nothing about what the state space of a system should concretely be, only that its state can specify its interface. Mode-dependence makes it so that writing a polynomial as an abstract box in a wiring diagram is hard, because the input and output wires change. Consider the polynomial above. There are two possible boxes that it corresponds to, depending on its state. One of them looks like this:



(5.3)

And the second looks like this:

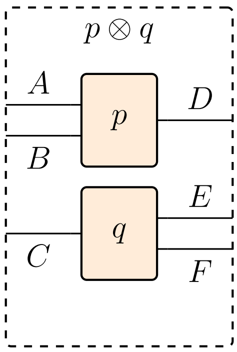


(5.4)

A good way of integrating this mode dependence inherent to polynomials into regular wiring diagrams is an open problem. Because of this, most wiring diagrams we will show are of non-mode dependent systems, and as such they represent only a small portion of the dynamical systems that **Poly** has the power to model.

5.2.2 The parallel product $\otimes$

We come to a great motivating factor for the naming of this structure: it corresponds to putting systems in parallel. Given two polynomials p and q with their own interfaces, taking their parallel product $p \otimes q$ corresponds to creating a polynomial that has *both interfaces simultaneously*. Intuitively, one can think that any system that has this interface can only run one timestep upon being given both inputs, and it will output both outputs. The structure, in some sense, does not "touch" the behavior of the systems at all. This is represented in the below picture, by considering $p(y) = Dy^{(A \times B)}$ and $q(y) = (E \times F)y^C$:



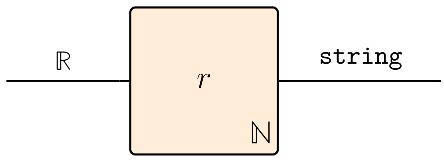
(5.5)

The resulting polynomial functor is the dashed line around the two inner boxes, and is given by $p \otimes q(y) = DEFy^{ABC}$. The parallel product also has an important implication for the behavior of systems, as explained next.

5.2.3 Lenses as behaviors

A particular kind of lens specifies behaviors/dynamics of systems, namely any lens of shape $f : Sy^S \to p$, where p is any polynomial and S is the state space set. Since lenses are maps on positions and on directions, in this case we get $f : S \to p(1)$; and $f\# : (\texttt{curState} : S) \to A(\texttt{f curState}) \to S$. This corresponds to our definition of open systems, with the added mode dependence: the map on positions is exactly the `readout` function, and the map on directions is the `update` function with the added constraint that not any A is valid - A is now a type family, so only the fibrations of A at the position (output) at the current state are valid.

A value of the type of lenses $f : Sy^S \to p$ can therefore be seen as a concrete dynamical system. At the end of the day, it is "only" a pair of functions: one to read out the exposed state of the system, and one to update the state of the system with some input. The way we will show behavior-specifying lenses in wiring diagrams is like the polynomials that represent their interfaces, but with the state space type denoted on the bottom like so:

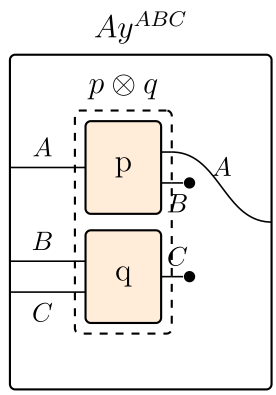


(5.6)

This can then be seen as a lens of type $S^S \to r$, where $S = \mathbb{N}$ and $r(y) = \texttt{string}y^{\mathbb{R}}$. Note that there are many values of this type, which is to say many ways to implement a dynamical system with this state space and this interface.

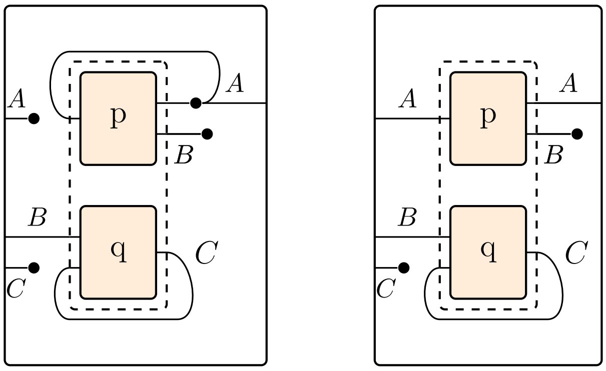
5.2.4 Lenses as wiring patterns

The other role that lenses play in systems of this kind is that they correspond to wiring patterns. For example, consider the wiring diagram below:



(5.7)

This corresponds to a lens from an abstract polynomial of the dashed box to the abstract polynomial of the outermost box. The inner dashed box is the polynomial obtained by taking the parallel product of p and q. We say "abstract" because we need to know nothing about the dynamics or concrete behavior of the systems here, all we need is the types of the interfaces to match, which means this particular lens is capable of wiring together *any* two systems that have the interfaces in question. There are of course other ways to wire these boxes together. Here are two examples:



And they all correspond to different lenses. The intuition for the lens structure is that the map on positions selects which of the outer outputs will be served by the inner outputs, and the map on directions will fill in the needed inputs to the inner boxes. Although sometimes lenses will feed an output of a system to itself, as above, a lens is still a morphism from some inner box (here the polynomial $p \otimes q(y) = ABCy^{ABC}$) to an outer one (here the polynomial Ay^{ABC}).

5.2.5 Commuting squares as system transformations

Commuting squares have a deep interpretation in this context: they correspond to how behaviour can transfer between different dynamical systems. Special kinds of commuting squares correspond to different kinds of behavior. For example, one behavior is for one system to simulate another system, this can be encoded by a simple kind of commuting square consisting of a function between the states of the two different systems. Two examples of such commuting diagrams are given in sections 5.3.1.2 and 5.3.1.3.

The fact that commuting squares compose also means that behaviour compose. For example, if system A simulates system B, and system B simulates system C, then system A simulates system C.

5.3 Implementing dynamical systems

Although dynamical systems are "just" lenses of a certain form, we choose a convenient representation for them so as to align with our thinking, analogously to how we choose to represent lenses, charts and polynomials as records and not Σ-types. Some of this code is directly translated from the Idris-based implementation in the poly-book's github repository. Our characterization is the following record:

```
record DynamicalSystem : Set₁ where
    constructor mkdyn
    field
```

```
    state : Set -- S
    interface : Polynomial -- p
    dynamics : Lens (selfMonomial state) interface -- Sy^S → p
```

When it comes to implementing a framework for coding up dynamical systems in this context, some patterns appear over and over. Some of these patterns are:

- Transforming a pure function $f : A \to B$ into a memoryless dynamical system, i.e. one that has A as an input, B as output, A as a state space and whose dynamics are given by running the function f on `readout` and just replace the state with the input in `update`.

  ```
  functionToDynamicalSystem : (A B : Set) → (A → B) → DynamicalSystem
  functionToDynamicalSystem A B f =
    mkdyn B (monomial B A) (id ⇆ (\_ → f))
  ```

- Create an `emitter` polynomial that represents an interface of no inputs (i.e. takes the singleton set) and always outputs the same type.

  ```
  emitter : Set → Polynomial
  emitter = linear
  ```

  It should not be surprising that this is just the linear polynomial; we give it a special name to bind it to the current interpretation, in the spirit of *concepts with an attitude* [30].

- Create a `haltingEmitter` polynomial that represents an interface of a system that must halt upon outputting its left side type.

  ```
  haltingEmitter : (A B : Set) → Polynomial
  haltingEmitter A B = mkpoly (A ⊎ B) halting
      where halting : {A B : Set} → (A ⊎ B) → Set
            halting (inj₁ _) = ⊥
            halting (inj₂ _) = ⊤
  ```

  A system inside this interface takes no inputs in a different, more strict sense: it is not that it does not receive any outside information, like the `emitter`, but rather that any use of it must account for the case when an input cannot be produced.

- Taking the parallel product of two systems. Recall that the parallel product is an operation on objects (polynomials) but it also it is also possible to construct a canonical lens from two lenses $f : A \to C$ and $g : B \to D$ of type $\langle f \otimes g \rangle : A \otimes B \to C \otimes D$. In the context of systems *being* morphisms, this corresponds to putting them "in parallel" and giving rise to a new system, with both state spaces, interfaces and behaviors simultaneously.

  ```
  _&&&_ : DynamicalSystem → DynamicalSystem → DynamicalSystem
  mkdyn stateA interfaceA dynamicsA &&& mkdyn stateB interfaceB dynamicsB
      = mkdyn (stateA × stateB)
              (interfaceA ⊗ interfaceB)
  ```

```
⟨ dynamicsA ⊗ dynamicsB ⟩
```

- Plugging dynamics and wiring patterns together. If lenses account both for dynamics *and* wiring patterns, then it seems natural that a dynamical system (really a lens) that just contains several "free-floating" boxes that are simply sitting in parallel could be composed with a lens that represents a wiring pattern to give rise to a system that is actually usable. Indeed, this is the case, and the function that does this is called `install` by both us and the poly example code:

```
install : (d : DynamicalSystem) →
          (p : Polynomial) →
          Lens (DynamicalSystem.interface d) p →
          DynamicalSystem
install d p l = mkdyn (DynamicalSystem.state d)
                      p
                      (l ∘ₚ (DynamicalSystem.dynamics d))
```

- Outer box lens representations. An outer box that represents an enclosure, as well as enclosures of a couple different possible inputs are given: we use `auto` for most systems, since we are interested in just running them as closed systems and seeing what happens, but sometimes we are interested in giving constant inputs that come from outside the environment, and for that we use `constI`:

```
encloseFunction : {t u : Set} → (t → u) → Lens (monomial t u) Y
encloseFunction f = (λ _ → tt) ⇆ (λ fromPos _ → f fromPos)

auto : {m : Set} → enclose (emitter m)
auto = encloseFunction λ _ → tt

constI : {m : Set} → (i : m) → enclose (selfMonomial m)
constI i = encloseFunction λ _ → i
```

  One can easily imagine extending `constI` to taking a function that operates on the number of timesteps so as to vary the input over time as well. This is something that is done in dynamical systems theory in a very informal capacity, under the intuition that the parameters of the system vary "slowly" with respect to the timescale of the state updates. With this framework, we gain the ability of making it precise for free. Of course, if we wanted to make the parameter variation respond to the system's dynamics, we would need to make it a system in itself; this is a design consideration that we hope future developers take into account. We take a mix of these two approaches for example 5.3.2.3.

- Speedup of dynamics. The composition product can be thought of as a way to speedup the dynamics of a system: since it accounts for all possible inputs (recall the corolla forest visualization in 3.9), a step in a dynamical system that has as interface a composite of two polynomials can be thought of as a step in its root polynomial, *then* a step in the top polynomial. The code for it is this:

```
speedup◃ : {p : Polynomial} {S : Set} →
           Lens (selfMonomial S) p →
           Lens (selfMonomial S) (p ◃ p)
speedup◃ {p} {S} l = dup ∘ₚ fixState
  where dup : Lens ((selfMonomial S) ◃ (selfMonomial S))
                   (p ◃ p)
        dup = ⟨ l ◃ l ⟩
        fixState : Lens (selfMonomial S)
                        (selfMonomial S ◃ selfMonomial S)
        fixState = (λ s₀ → s₀ , λ s₁ → s₁)
                   ⇆
                   λ { s₀ (s₁ , s₂) → s₂ }
```

An example usage of this speedup functionality is provided in the Fibonacci example 5.3.1.1.

- Finally, there is a function that does what one accustomed to dynamical systems simulations would expect: given an abstract representation of a dynamical system (a value of type `DynamicalSystem` in our case), and an initial condition, it produces an infinite stream of outputs.

```
{-# TERMINATING #-}
run : (d : DynamicalSystem) →
      enclose (DynamicalSystem.interface d) →
      DynamicalSystem.state d →
      Stream (Polynomial.position (DynamicalSystem.interface d)) _
run d e initialState =  [ output ] ++ (run d e next)
    where
      output : Polynomial.position (DynamicalSystem.interface d)
      output = Lens.mapPosition (DynamicalSystem.dynamics d)
                                initialState
      next : DynamicalSystem.state d
      next = Lens.mapDirection (DynamicalSystem.dynamics d)
                               initialState
                               (Lens.mapDirection e output tt)
```

We need the Agda pragma `TERMINATING` here because the termination checker cannot be sure that this code terminates, since `run` calls itself on input that is not guaranteed to be smaller. We can then `take` on the resulting stream to produce regular Agda lists or vectors, and use them to render plots or print to the screen, and as long as we always run this in the context of a finite `take` call, the program should always terminate.

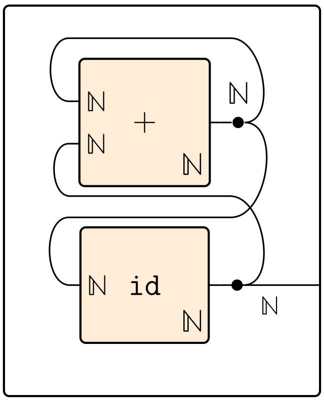
There are more helpers and utilities peppered throughout the code, but this is the basic glue needed to work with **Poly** in the context of dynamical systems, as the examples will demonstrate.

5.3.1 Examples: Discrete systems

We split the example implementations in the class of discrete- and continuous-time. As mentioned previously, all examples are in the strict sense discrete; the continuous ones are simply discretized with some dt. However, the split is still valid if one considers the spirit of the implementations.

5.3.1.1 Fibonacci

We start with the most basic example of all: a Fibonacci sequence generator implemented as a dynamical system in **Poly**. The wiring diagram of this system is as follows:



(5.8)

To break it down: the $+$ function takes two natural numbers and produces one output, and the *id* function takes a natural number and outputs a natural number. In terms of dynamical systems, however, we have to think about what the `readout` and `update` functions are doing for each system: `readout` can only read what the current state is, and `update` can only update the state given an input. At every timestep, all systems produce their outputs based on the current state, then all the inputs are read, and the states of all systems are updated. Therefore, the effect of the `id` function is only felt one timestep later. Hence it plays the subtle role of storing the Fibonacci value computed in the previous timestep, which is why its output is fed as an input to the $+$ function. The code for this system is this:

```
delay : (A : Set) → DynamicalSystem
delay A = functionToDynamicalSystem A A id

plus : DynamicalSystem
plus = functionToDynamicalSystem (Nat × Nat) Nat (uncurry _+ℕ_)

prefib : DynamicalSystem
prefib = plus &&& delay Nat

fibWiringDiagram : Lens (interface prefib) (emitter Nat)
fibWiringDiagram = (λ {(sumOutput , idOutput) → idOutput})
                    ⇆
```

```
                    (λ {(sumOutput , idOutput) _ →
                        (idOutput , sumOutput) , sumOutput })

fibonacci : DynamicalSystem
fibonacci = install prefib (emitter Nat) fibWiringDiagram
```

This example provides a good opportunity to understand dynamical systems in the context of polynomials, so let us take the chance to dig in further. In keeping with the previously mentioned intuition, the `fibWiringDiagram` lens is just one of many possible wiring patterns for an emitter-enclosed system that contains two "floating" dynamical systems that have the interfaces of `plus` and `delay Nat`. The outer outputs are the target of the lens, so the polynomial `emitter Nat`, which is a polynomial that takes the singleton set as input and outputs natural numbers. The inner inputs are given by the wiring pattern in the lens, specifically in the second function (the map on directions). This part of the code:

```
(λ {(sumOutput , idOutput) _ →
    (idOutput , sumOutput) , sumOutput })
```

is, on the right hand side, saying that the two inputs to `plus` are given by the output of `delay Nat` and the its own output, named `idOutput` and `sumOutput` respectively in the lambda. The input to `delay Nat`, on the other hand, is given by the output of the sum: in the next time step, this value will appear in the variable `idOutput`.

Two final remarks about the Fibonacci sequence generator: the first is that it is actually an efficient way to express Fibonacci dynamics, since there is no recursive calls involved - and the second is that we went into as much detail as possible explaining it so that we can rely on some of the intuition built here to explain the more complicated systems that follow.

5.3.1.2 Flip Flop

Flip Flop is a simple example with two dynamical systems showing how one system can simulate the behavior of another system. This simulation is exactly a special kind of commuting square between charts and lenses.

Both systems needs to have the same interface, which in this case is a linear polynomial `{on, off}`y, that can output either on or off, and always takes unit as input. However, the state and the behavior of the systems can differ. One of the systems is a flip flop system, with states on or off, with behaviour that outputs this state, and flips the state in the update. The other system has natural numbers as states, outputs the two different states depending on the natural number modulus 2, and updates the state by always taking the successor of the state. Running these systems has the exact same output. This can be expressed concisely by a chart transforming the state space of the natural numbers to either on or off. Then by putting this chart in a commuting square such as shown in diagram 5.1.

```
data Switch : Set where
    on : Switch
    off : Switch
```

```
toggle : Switch → Switch
toggle on = off
toggle off = on

-- | Commonly used where input to enclosed dynamical system where
--   updateState only depends on current state.
ignoreUnitInput : {A B : Set} → (A → B) → A → ⊤ → B
ignoreUnitInput f a tt = f a

-- | Note: linear interface is used to accept only 1 possible input.
--   Readout defined as id to expose state.
flipFlop : DynamicalSystem
flipFlop = mkdyn Switch (linear Switch) (id ⇆ ignoreUnitInput toggle)

-- | Result is: on, off, on, off...
flipFlopRan : Vec Switch 10
flipFlopRan = take 10 $ run flipFlop auto on

modNat : Nat → Switch
modNat n = if n % 2 == 0 then on else off

-- | To compare flipFlop and counter they need to have the same interface.
counter : DynamicalSystem
counter = mkdyn Nat (linear Switch) (modNat ⇆ ignoreUnitInput suc)

-- | Result is: on, off, on, off...
counterRan : Vec Switch 10
counterRan = take 10 $ run counter auto 0

-- | Morphism between p dynamicalSystems with states Nat and Switch.
morphSystem : Nat → Switch
morphSystem = modNat

-- | The square expressing the simulation.
square : LensChartCommute (dynamics counter)
                          (dynamics flipFlop)
                          (morphSystem ⇉ λ _ → morphSystem)
                          idChart
square = law₁ , law₂
```

$$\begin{tikzcd}
Ny^1 \arrow[r, shift left, "\%2"] \arrow[r, shift right, "\%2"'] \arrow[d, shift right, "\%2"'] & 2y^1 \arrow[d, shift right, "id"'] \\
2y^1 \arrow[u, shift right, "suc"'] \arrow[r, "id_{chart}"] & 2y^1 \arrow[u, shift right, "toggle"']
\end{tikzcd}$$

Figure 5.1: A special kind of commuting square. The identity chart turns this into a commuting triangle. The chart at the top is defined as the function reducing the states between the systems. The laws are fulfilled if this diagram commuting encodes what it means for a system to simulate another.

This showcased how a simple chart in a commuting square can compare behaviour of two different dynamical systems. By generalizing, and using more advanced charts and commuting squares, more advanced comparisons can be made.

5.3.1.3 Simulating state machines

A more advanced example, in comparison to flip flop, is how one state machine can simulate another. This is often expressed as that one state machine is more minimal than another, that is, it has less states but achieves the same behaviour. This is exactly what the special commuting square can express. Because the same idea is followed as for the flip flop example, this example will not be discussed further, but can be found in `Dynamical/Chart/SimulateStateMachine.agda`.

5.3.1.4 Moore & Mealy machines

In automata theory, a Moore machine is defined to be an abstract machine comprised of the following:

- A state-space set S
- An input set A
- An output set B
- A readout function `readout` : $S \to B$
- An update function `update` : $S \to A \to S$

and a Mealy machine is a similar machine with the difference that reading the state and updating it happen in a single function `updateRead` : $S \to A \to S \times B$.

These machines correspond directly to lenses in **Poly**: Moore machines are straightforward dynamical systems, i.e. lenses of form $Sy^S \to By^A$. Mealy machines are lenses to the exponential object between linear polynomials, so lenses of form $Sy^S \to (Ay \to By)$. Because of the isomorphism shown in 3.10, this can be reduced to $Sy^S * Ay \to By$, which is easier to work with and think about.

Both machines can be defined as their own types as follows:

```
record MooreMachine {State Input Output : Set} : Set where
    constructor mkmoore
    field
        readout : State → Output
        update : State → Input → State

record MealyMachine {State Input Output : Set} : Set where
    constructor mkmealy
    field
        updateRead : State → Input → (State × Output)
```

And both can be shown to be equal to the aforementioned lenses. The proof for the Moore machine is trivial, while the Mealy one is more complex - there is an attempt at proving this isomorphism that has a hole, again due to difficulty in handling pattern matching on sum types during normalization. This code can be found in `LensIsMooreMachine.agda` and `LensIsMealyMachine.agda`.

5.3.1.5 Deterministic finite automaton

A deterministic finite state automaton (DFA) is also a special kind of lens, the lens to the monomial $2y^{alphabet}$. A DFA can be defined as:

```
record DFS {State Alphabet : Set} : Set where
    constructor mkDFS
    field
        -- Transition function
        update : State → Alphabet → State
        -- Partitions all states into recognized and unrecognized states.
        recognized : State → Bool
```

The equality can be defined as a simple isomorphism.

The same trick as for the Moore machine can be used to provide an initial state to the DFA.

5.3.1.6 Turing machines

One of the examples given by the authors is of a Turing machine. A Turing machine consists of two components:

- An infinitely long tape, indexed by integers, which:
    - Has a notion of what its current index is.
    - At each index, contains symbols of an alphabet set $A := \{\mathbb{0}, \mathbb{1}, \mathbb{b}\}$
    - Can receive commands in an alphabet set $C := \{\leftarrow, \rightarrow\} + A$ to change its current index by $+1$ or -1 and or to change which symbol is at the current index.

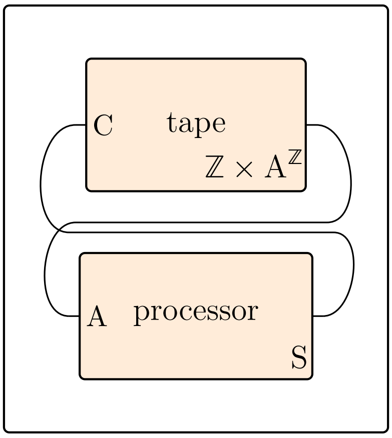


Figure 5.2: Turing machine as a wiring diagram. The processor issues commands and reads the tape's current index.

- A processor, which can:
  - Output the aforementioned commands to the tape according to some internal logic that operates on a state S.
  - Read the symbol at the tape's current index.

These two components can be represented as dynamical systems, and then wired together to give a Turing machine, like in the wiring diagram in Figure 5.2.

A real Turing machine must be able to output something to the outer box, which happens once it is halted, while Figure 5.2 only shows them while running/communicating. This is an example of mode-dependence being hard to express.

Implementation outline

We can implement the Turing machine setup above by first defining the sets used:

```
data Alphabet : Set where
    𝟘 𝟙 𝕓 : Alphabet

data Movement : Set where
    𝕝 𝕣 : Movement
```

And then defining each part of the system. Since Turing machines typically have a halting state, we represent this for the processor:

```
ProcessorState : Set
ProcessorState = ℕ

data ProcessorOutput : Set where
    move : Movement → ProcessorOutput
    write : Alphabet → ProcessorOutput
    halt : ProcessorOutput
```

```
data ProcessorInput : Set where
    instruction : Alphabet → ProcessorInput

procInputFromOutput : ProcessorOutput → Set
procInputFromOutput (move x) = ProcessorInput
procInputFromOutput (write x) = ProcessorInput
procInputFromOutput halt = ⊤

processorInterface : Polynomial
processorInterface = (mkpoly ProcessorOutput procInputFromOutput)
```

The `ProcessorState` type is a dummy type, since there will be no Turing program written, so it is fine to represent the state as a simple natural number. The `procInputFromOutput` function is capturing the idea that once the processor outputs `halt`, it accepts only a mindless input with no meaning. This could also be represented as the empty type, as in NO input, but it becomes somewhat harder to synchronize with the tape's own possible inputs, as will be demonstrated next.

Now the analogous types are provided for the tape:

```
data TapeInput : Set where
    write : Alphabet → TapeInput
    move : Movement → TapeInput
    halt : TapeInput

TapeAt : Set
    TapeAt = ℤ → Alphabet

data TapeOutput : Set where
    goOut : Alphabet → TapeOutput
    haltOut : TapeAt → TapeOutput

tapeInputFromOutput : TapeOutput → Set
tapeInputFromOutput (goOut x) = TapeInput
tapeInputFromOutput (haltOut x) = ⊥

tapeInterface : Polynomial
tapeInterface = mkpoly TapeOutput tapeInputFromOutput

data TapeState : Set where
  go : TapeAt → TapeState
  halt : TapeAt → TapeState
```

The tape's associated types and polynomial are more complex. There are a few things to notice.

Firstly, the tape does in fact not accept any more inputs when it is outputting a halting state. The reason it can get away with this, but not the processor, is that the processor *decides* when to halt the program, and so its output of `halt` needs to reach the tape - for that, at least one more input is needed.

Second, the tape state contains *only* the $A^{\mathbb{Z}}$ part of the diagram, and not the notion of which index it is at. The trick is to always consider the index to be zero, and treat the move directions as post composing the tape with an increment or decrement function. This idea comes from the book's associated YouTube lectures. Since the tape behavior is always the same for any Turing machine, we provide it:

```
tapeBehavior : Lens (selfMonomial TapeState) tapeInterface
tapeBehavior =
    readout ⇆ update
    where readout : TapeState → TapeOutput
          readout (go tapeAt) = goOut (tapeAt 0ℤ)
          readout (halt tapeAt) = haltOut tapeAt
          update : (x : TapeState) →
                   (tapeInputFromOutput (readout x)) →
                   TapeState
          update (go tapeAt) (write x) = go $ λ tapeIndex →
            if ⌊ tapeIndex ≟ 0ℤ ⌋
                then x
                else tapeAt tapeIndex
          update (go tapeAt) (move x) = go (moveTape x tapeAt)
              where moveTape : Movement → TapeAt → TapeAt
                    moveTape 𝕝 f = f ∘ (1ℤ +ℤ_ )
                    moveTape 𝕣 f = f ∘ (1ℤ -ℤ_ )
          update (go tapeAt) halt = halt tapeAt
```

Finally, there is the need to parallel-productize the two systems together and then wire them, finding a suitable run-time representation for the tape state. We chose a vector that represents the indices 0-255 of it.

```
tape : DynamicalSystem
tape = mkdyn TapeState tapeInterface tapeBehavior

preTuring : DynamicalSystem
preTuring = tape &&& processor

open DynamicalSystem
Word : Set
Word = Vec ℤ 256
toInt : Alphabet → ℤ
toInt 𝟘 = 0ℤ
toInt 𝟙 = 1ℤ
toInt 𝕓 = 1ℤ +ℤ 1ℤ
turingWiringDiagram : Lens (interface preTuring) (haltingEmitter Word ⊤)
turingWiringDiagram = outerOutput ⇆ fillInputs
  where outerOutput : TapeOutput × ProcessorOutput → (Word ⊎ ⊤)
```

```
outerOutput (goOut x , procOut) = inj₂ tt
outerOutput (haltOut x , _) = inj₁ $
    Data.Vec.map (toInt ∘ x) (tabulate (+_ ∘ toℕ))
fillInputs : (fromPos : position (interface preTuring)) →
             direction (haltingEmitter Word ⊤)
                       (outerOutput fromPos) →
             direction (interface preTuring)
                       fromPos
fillInputs (goOut tapeInstruction , move procInstruction) tt =
    move procInstruction , instruction tapeInstruction
fillInputs (goOut tapeInstruction , write procInstruction) tt =
    (write procInstruction) , instruction tapeInstruction
fillInputs (goOut _ , halt) tt = halt , tt
```

A remarkable thing to notice about this system wiring is that `fillInputs` does not even allow us to pattern match on an invalid combination of inputs to the system, like `haltOut _ , tt`, since it is a dependent function on the result of `outerOutput`, which guarantees a `haltOut` always maps to `inj`$_1$ and this corresponds to no input. Dependent types allows us to make illegal states truly unrepresentable.

5.3.2 Examples: Continuous systems

Now we move on to discretized continuous-time systems. They may be treated in much more rigorous mathematical detail than we do here, by representing the state space as topological spaces or manifolds [28], but as we have mentioned, to showcase their implementation in **Poly** we discretize them simply according to a differential time dt supplied by the user.

Supporting Haskell code

Unfortunately, not all the code in this thesis could be kept in Agda, since it would require too much reimplementation of basic user functionality, or would not be efficient. Therefore, there are the following accompanying tiny Haskell libraries:

- A command-line interface library for running the continuous-time systems, since each of them requires different parameters, and using the `optparse-applicative` [31] library.
- A plotting library, which renders each system as a collection of values of the system at a sequence of timsteps, written using the `Chart` [32] library (not to be confused, of course, with the Chart arrow between polynomials).
- A matrix-inversion library with a single function which takes a list-of-lists representation of a matrix and returns its pseudoinverse. This is needed for the last example. Taking the pseudoinverse of a matrix is a costly operation, and there was an attempt to do it in Agda, but the associated proofs were not trivial and the performance was prohibitively slow. Therefore, we outsource matrix inversion to the `hmatrix` [33] Haskell library, which relies on underlying

BLAS [34] algorithms for speed, while passing `trustMe` proofs to guarantee matrix sizes at the Agda level.

For more information on this supporting code, see the Appendix A. With this context, we move on the the systems in question.

5.3.2.1 Lotka-Volterra

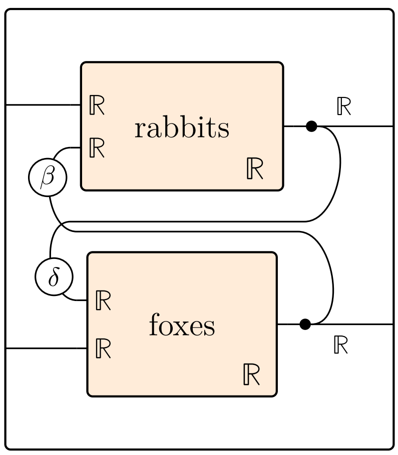
The Lotka-Volterra predator-prey model is an essential system [35] that portrays how the population of predators interacts with the population of prey in a given environment. The two main quantities in the model are two real numbers, abstractly representing the amount of the two types of animals - these are its states, in our context. The model is given by the two equations below, where r and f represent our populations of rabbits and foxes - our chosen species of prey and predator:

$$\dot{r} = \alpha r - \beta f r \tag{5.9}$$

$$\dot{f} = \delta f r - \gamma f \tag{5.10}$$

The logic of it is that, for rabbits, they will reproduce naturally at a rate α, while being hunted at a rate βf. In other words, their encounters with and subsequent predation by foxes is modeled by multiplying the number of foxes by a "fox appetite" parameter β. For foxes, their population can be thought of as flourishing or growing at the rate that they can encounter and eat rabbits. This aspect is captured by the term $\delta f r$. They naturally die at a rate of γ. Note that δ does not have to match β - the rate at which foxes eat rabbits is not necessarily the same as the rate at which fox populations grow as a result of being well-fed, though the intuitive relationship between these parameters could be captured by a more sophisticated model.

This system can be represented in terms of our open dynamical systems by considering the rabbit and fox populations as each inhabited by an individual system. They output their population (so `readout` is the identity function) and that gets fed as an input to the other system, along with any other parameters - in this case α, β, δ and γ. Here is the wiring diagram for it:



(5.11)

The circles with the constants signify multiplication of the values by the constants β and δ. This is something we express at the wiring pattern lens' level in this system.

Thinking of the systems individually allows us a "separation of concerns" in specifying these systems' behavior that is not there in environments like Matlab or Python. The code for the rabbit system is:

```
rabbits : DynamicalSystem
rabbits = mkdyn ℝ (mkpoly ℝ λ _ → ℝ × ℝ) (readout ⇆ update)
  where readout : ℝ → ℝ
        readout state = state
        update : ℝ → ℝ × ℝ → ℝ
        update state (birthRabbits , deathRabbits) =
            state + dt * (state  * (birthRabbits - deathRabbits))
```

The differential equation is directly discretized, so the differential amount of time dt goes to the right hand side of 5.9 and multiplies the entire thing; this is then added to the current state to produce a new state.

Similarly, the fox system is given as:

```
foxes : DynamicalSystem
foxes = mkdyn ℝ (mkpoly ℝ λ _ → ℝ × ℝ) (readout ⇆ update)
  where readout : ℝ → ℝ
        readout state = state
        update : ℝ → ℝ × ℝ → ℝ
        update state (birthFoxes , deathFoxes) =
            state + dt * (state * (birthFoxes - deathFoxes))
```

Note that each of these systems have only two parameters, whereas by the equations 5.9 and 5.10, they ought to have three: 5.9 should get α and β from "outside" the system, whereas 5.10 should get δ and γ. It is a somewhat arbitrary choice that we make to make the β and γ parameters part of the larger system, that is the one given rise to by the composition of these systems parallel-productized with the wiring diagram, and not constants within the system themselves, for instance.

The final system is then a composition with a certain wiring diagram lens:

```
preLV : DynamicalSystem
preLV = rabbits &&& foxes

-- Wiring diagram is an lens between monomials
lotkaVolterraWiringDiagram : ℝ → ℝ →
    Lens (DynamicalSystem.interface preLV) (selfMonomial (ℝ × ℝ))
lotkaVolterraWiringDiagram foxPerCapDeath foxHunger =
  outerOutput ⇆ innerInput
    where outerOutput : ℝ × ℝ → ℝ × ℝ
          outerOutput (rabbitOutput , foxOutput) =
            rabbitOutput , foxOutput
          innerInput : (outputs : ℝ × ℝ) →
```

```
                    direction (selfMonomial (ℝ × ℝ))
                              (outerOutput outputs) →
                    direction (interface preLV) outputs
          innerInput (r , f) (rabMaxPerCapGrowth , howNutritiousRabbitsAre) =
            (rabMaxPerCapGrowth , foxHunger * f) ,
            (foxPerCapDeath * r , howNutritiousRabbitsAre)
-- Final system is composition of wiring diagram and dynamics
lotkaVolterra : ℝ → ℝ → DynamicalSystem
lotkaVolterra β γ = install preLV
                         (selfMonomial (ℝ × ℝ))
                         (lotkaVolterraWiringDiagram β γ)
```

Running this system through the command line interface produces the expected oscillating dynamics of the Lotka-Volterra model, albeit with some numerical instability causing the oscillations to grow slightly. Figure 5.3 shows this dynamics, obtained by running the system through the command line interface with the following parameters.

```
./Plot LotkaVolterra --alpha 0.2 --beta 0.2 --delta 0.5 \
                     --gamma 0.4 --r0 1.0 --f0 0.9 --dt 0.1
```

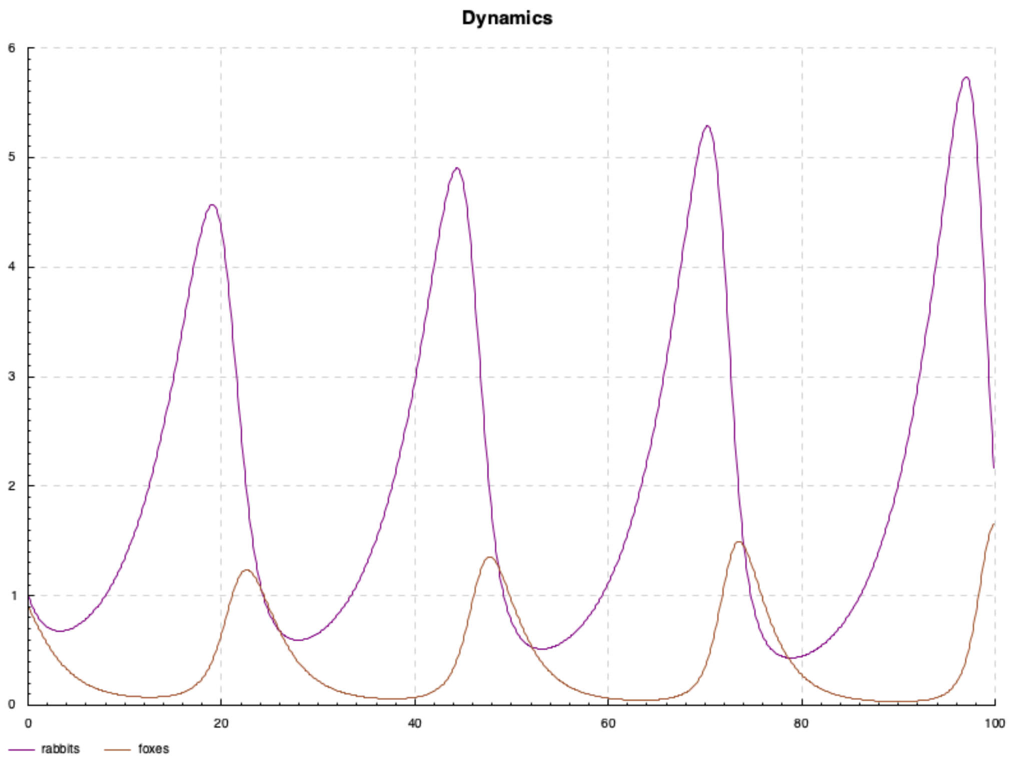


Figure 5.3: Lotka-Volterra oscillatory dynamics. As the population of rabbits grows, the population of foxes catches up, causing it to go down.

5.3.2.2 Lorenz system

The Lorenz system [36] is what is known as a chaotic system, in an elementary form. It was introduced by Edward Lorenz as a simplified model of atmospheric convection. The three dimensionless (as in, pure proportions with no physical units) variables x, y and z correspond to rate of convection, temperature difference between ascending and descending air currents, and how temperature changes with height. It is given by the following differential equations:

$$\dot{x} = \sigma(y - x) \tag{5.12}$$

$$\dot{y} = x(\rho - z) - y \tag{5.13}$$

$$\dot{z} = xy - \beta z \tag{5.14}$$

The system has three numeric parameters, σ, ρ and β and for specific values of these parameters ($\sigma = 10.0$, $\rho = 28.0$, $\beta = 8/3$), it displays chaotic dynamics - dynamics that have the property of becoming hard to predict as time goes on. What this means is that for two arbitrarily close initial conditions, the distance between the trajectories followed by running the system with these initial conditions grows exponentially, until the trajectories are maximally distant. Figure 5.4 shows an example trajectory of the system with the parameters mentioned above.

This is known as a "strange attractor": the system's trajectories eternally oscillate between two attracting states, but they never converge to a single orbit, jumping back and forth. The system is a greatly simplified model of the atmosphere, but provides a good "essence" of chaotic dynamics that are hard to predict, even in just three variables.

Implementing the Lorenz system in terms of **Poly** follows the same pattern as the Lotka-Volterra model, by capturing each equation in its own box with its own internal state, with the difference that the external parameters are now fixed, so the inputs are now only the external variables upon which each box should depend. In Agda, the systems are given as:

```
-- First order differential equations
x : ℝ → DynamicalSystem
x dt = mkdyn X (mkpoly X λ _ → Y) (readout ⇆ update)
  where readout : X → X
        readout state = state
        update : X → Y → X
        update (xnt state) (ynt y) =
            xnt (state + dt * (σ * (y - state)))

y : ℝ → DynamicalSystem
y dt = mkdyn Y (mkpoly Y λ _ → X × Z) (readout ⇆ update)
  where readout : Y → Y
        readout state = state
        update : Y → X × Z → Y
```

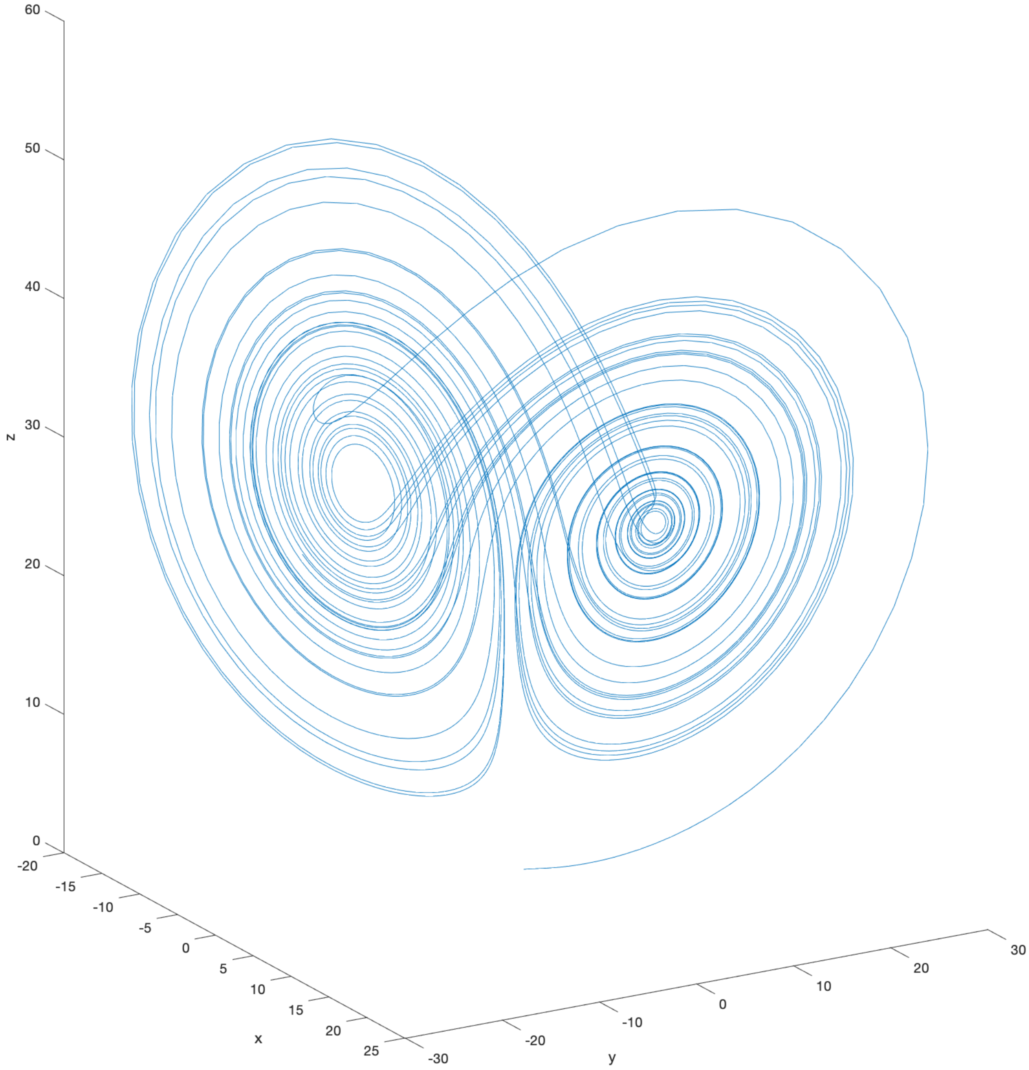


Figure 5.4: "Lorenz butterfly", a visualization of a trajectory of the Lorenz system. dt is set to 0.01.

```
        update (ynt state) ( xnt x , znt z ) =
            ynt (state + dt * (x * (ρ - z) - state))

z : ℝ → DynamicalSystem
z dt = mkdyn Z (mkpoly Z λ _ → X × Y) (readout ⇆ update)
  where readout : Z → Z
        readout state = state
        update : Z → X × Y → Z
        update (znt state) (xnt x , ynt y) =
            znt (state + dt * (x * y - β * state))
```

And they are again parallel-productized, producing a system with 3 inputs and 3 outputs, then wired in the right way (x is given as an input to y and z, y to x and z, and z to y, as per the system equations)

```
preLorenz : ℝ → DynamicalSystem
preLorenz dt = x dt &&& y dt &&& z dt

-- Wiring diagram is an lens between monomials
lorenzWiringDiagram : Lens (interface (preLorenz _))
                           (emitter (X × Y × Z))
lorenzWiringDiagram = mp ⇆ md
  where mp : X × Y × Z → X × Y × Z
        mp (x , y , z) = x , y , z
        md : X × Y × Z → ⊤ → Y × (X × Z) × (X × Y)
        md (x , y , z) _ = y , (x , z) , (x , y)

-- Final system is composition of wiring diagram and dynamics
lorenz : ℝ → DynamicalSystem
lorenz dt = install (preLorenz dt)
                    (emitter (X × Y × Z))
                    lorenzWiringDiagram
```

An example run of this system, obtained by `take` of 1000 timesteps at $dt = 0.01$, is given in Figure 5.5.

5.3.2.3 Hodgkin-Huxley model

The two examples given in this section are closed systems, meaning they take no input - what this really means is that they take the singleton set as input, so each timestep the element `tt` is given as a value and the system produces the next output. Now we come to a slightly bigger and more realistic system with 4 equations: the Hodgkin-Huxley [37] model. It is a system that models action potentials in the giant squid neuron, by thinking of the neuron membranes as circuits, so that the relevant quantities are all electrical (charge, current, capacitance etc). This model captures the flow of sodium and potassium ions through ion channels in the membrane, like in Figure 5.6, obtained from [38].

The interesting thing about the Hodgkin-Huxley model is that it actually lends

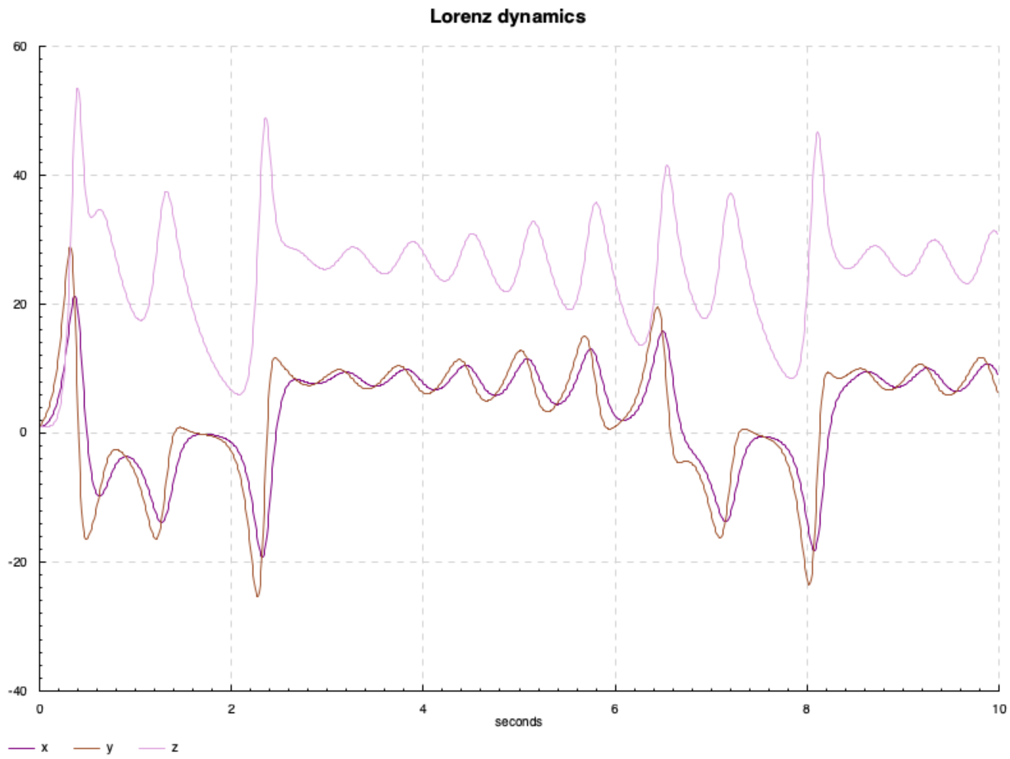


Figure 5.5: Lorenz system chaotic dynamics. The three variables keep oscillating in ever-changing patterns, never settling to a stable cycle.

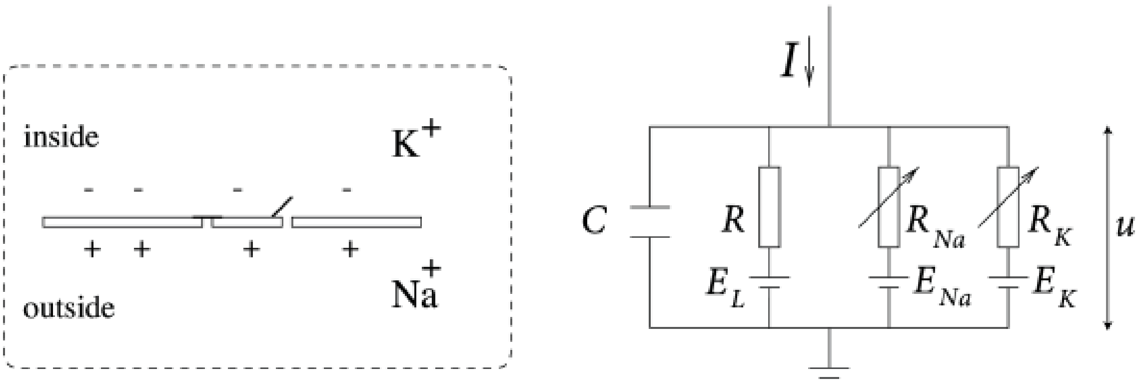


Figure 5.6: Analogy schematic of a neuronal membrane modeled as a circuit, from the book Neuronal Dynamics.

itself to being implemented as an open system. There are several equivalent ways of formulating HH as a 4-dimensional dynamical system, but the equations we use are according to this online tutorial by Mark Kramer and given by:

$$\dot{V} = G_{na}m^3h(E_{na} - V_s) + G_kn^4(E_k - V_s) + G_L * (E_L - V_s) + I_e \tag{5.15}$$

$$\dot{m} = \alpha_m(V)(1.0 - m) - \beta_m(V)m \tag{5.16}$$

$$\dot{h} = \alpha_h(V)(1.0 - h) - \beta_h(V)h \tag{5.17}$$

$$\dot{n} = \alpha_n(V)(1.0 - n) - \beta_n(V)n \tag{5.18}$$

The variables here mean the following: V is the resulting voltage between the intra and extracellular mediums, m encodes potassium channel activation probability, h encodes sodium channel activation probability and n encodes sodium channel *inactivation* probability, and I_e is the driven input voltage. All other symbols are constants and helper functions that have their own deep theoretical justification which we will not get into, we will just explain the basic logic of the system. The idea is that potassium and sodium channels are positively charged, so their flow across their membrane corresponds to current. However, their flow is not only affected by voltage, but also by concentration: in a chemical setting, substances can saturate a solution regardless of their charge. The interacting probabilities of channels activating (opening, thus allowing particles to flow from more to less concentrated environments) or inactivating give rise to interesting voltage dynamics - the voltage is in fact the output we are interested in for this particular application. Given a strong enough input current, the system can be made to display oscillating behavior, which corresponds to the neuron firing.

The input current can be a constant value, in which case it might as well be treated as a parameter, or an input to an open dynamical system as we have been describing in the framework we work in. This allows us to model "changing a parameter" of a dynamical system, but in a way that is totally internal to our modeling language and does not rely on ad-hoc formulations or programming [1].

The code for the equations themselves, represented as systems, follows the same pattern as the two previous examples. It also involves many fixed constants, discovered experimentally, and so it is quite big, so for the sake of clarity we omit most of it for this system. Instead we focus on the wiring pattern, as well use of `constI` as a constant input to the system to showcase running a system:

```
preHH : ℝ → DynamicalSystem
preHH dt = voltage dt &&&
           potassiumActivation dt &&&
           sodiumActivation dt &&&
           sodiumInactivation dt
```

[1]For a description of the informal process of analyzing changing a parameter as "turning a knob" at a timescale that is much bigger than the system's dynamics, see this lecture by Shane Ross on the book Nonlinear Dynamics and Chaos by Strogatz: https://www.youtube.com/watch?v=BBd68_q3Dgg.

```
hodgkinHuxleyWiringDiagram : Lens (interface (preHH _)) (selfMonomial ℝ)
hodgkinHuxleyWiringDiagram =
    (λ {(v , m , h , n) → v }) ⇆
    (λ {((v , m , h , n)) Ie → (Ie , m , h , n) , v , v , v })

hodgkinHuxley : ℝ → DynamicalSystem
hodgkinHuxley dt = install (preHH dt)
                           (selfMonomial ℝ)
                           hodgkinHuxleyWiringDiagram

hhSeq : ℝ → Stream ℝ _
hhSeq dt = run (hodgkinHuxley dt) (constI Ie) (V₀ , m∞ , n∞ , h∞)
  where V₀ : ℝ
        V₀ = -70.0
        m∞ : ℝ
        m∞ = 0.05
        n∞ : ℝ
        n∞ = 0.54
        h∞ : ℝ
        h∞ = 0.34
        Ie : ℝ
        Ie = 10.0
```

Notice the constant driven input current of 10nA. An example run of the system under this setup, with $dt = 0.02$ and run for 5000 steps, is seen in Figure 5.7.

Since this is a system with input, however, we can exploit this in order to do parameter variation: if we want to see how the system's dynamics vary with a change in parameters that is comparatively slow, we can create a new system whose responsibility is to output the changing current.

Adding this is simple: we take our finished Hodgkin-Huxley system and parallel-product it with a new system whose only responsibility is to grow its own output at a rate slower than HH's. This "current grower" system is so simple we inline it, using the helper `functionToDynamicalSystem`. It could have been implemented in other ways - and in fact, this could be its own investigation, like how changing parameters quickly/slowly/exponentially or whatever, could land systems in different dynamics landscapes. We choose a simple ticking, linear increase at each timestep here.

```
preHHWithInput : ℝ → DynamicalSystem
preHHWithInput dt = hodgkinHuxley dt &&&
                    functionToDynamicalSystem ℝ ℝ
                                              λ x → x + (dt * 0.02)
```

We then wire it with HH, taking care to provide the current increaser this time using the `auto` helper to always give `tt` as input to the resulting system; we no longer need to provide it an external input of any kind, since this is now built into the system. We also take care with the wiring pattern to give our "current grower"

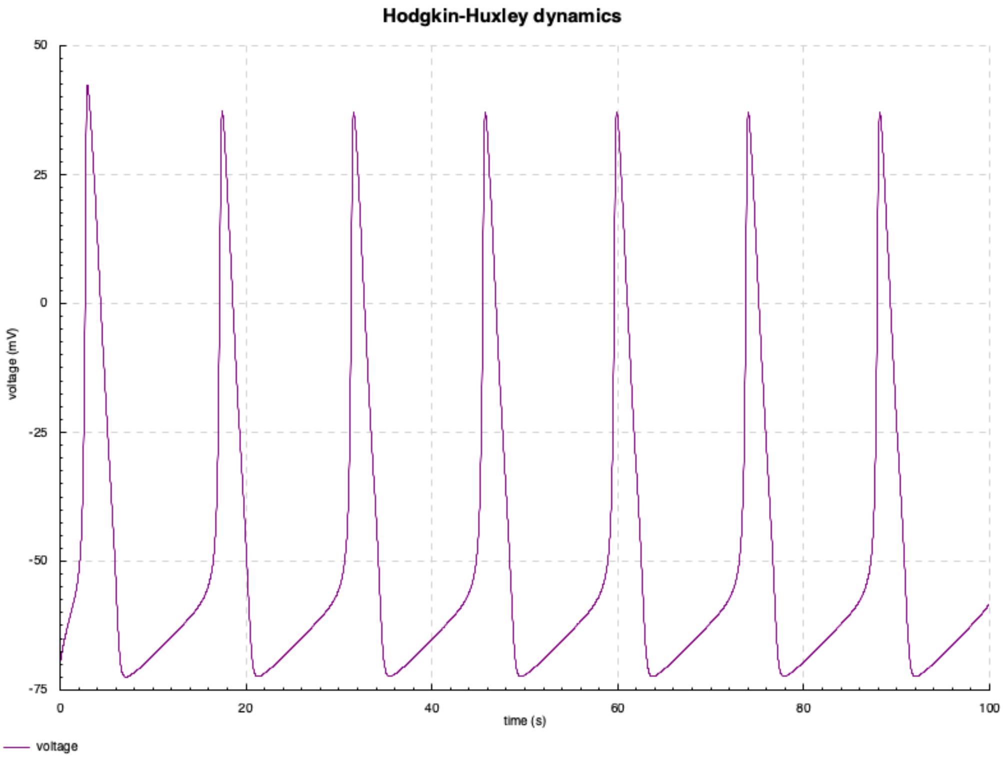


Figure 5.7: Hodgkin-Huxley model dynamics with input current $I_e = 10$nA and $dt = 0.02$. The simulated neuron fires repeatedly.

system's output to both itself and to HH.

```
hhWithInputWiring : Lens (interface (preHHWithInput 0.0)) (emitter ℝ)
hhWithInputWiring =
    (λ { (hhOut , _) → hhOut}) ⇆
     λ { (hhOut , fnOut) _ → fnOut , fnOut }

hhWithInput : ℝ → DynamicalSystem
hhWithInput dt = install (preHHWithInput dt) (emitter ℝ) hhWithInputWiring

hhSeqWithInput : ℝ → Stream ℝ _
hhSeqWithInput dt = run (hhWithInput dt) auto ((V₀ , m∞ , n∞ , h∞) , -5.0)
  where V₀ : ℝ
        V₀ = -70.0
        m∞ : ℝ
        m∞ = 0.05
        n∞ : ℝ
        n∞ = 0.54
        h∞ : ℝ
        h∞ = 0.34
```

The resulting dynamics is what one would expect from Hodgkin-Huxley at the input current levels explored by the current grower, and can be seen in Figure 5.8.

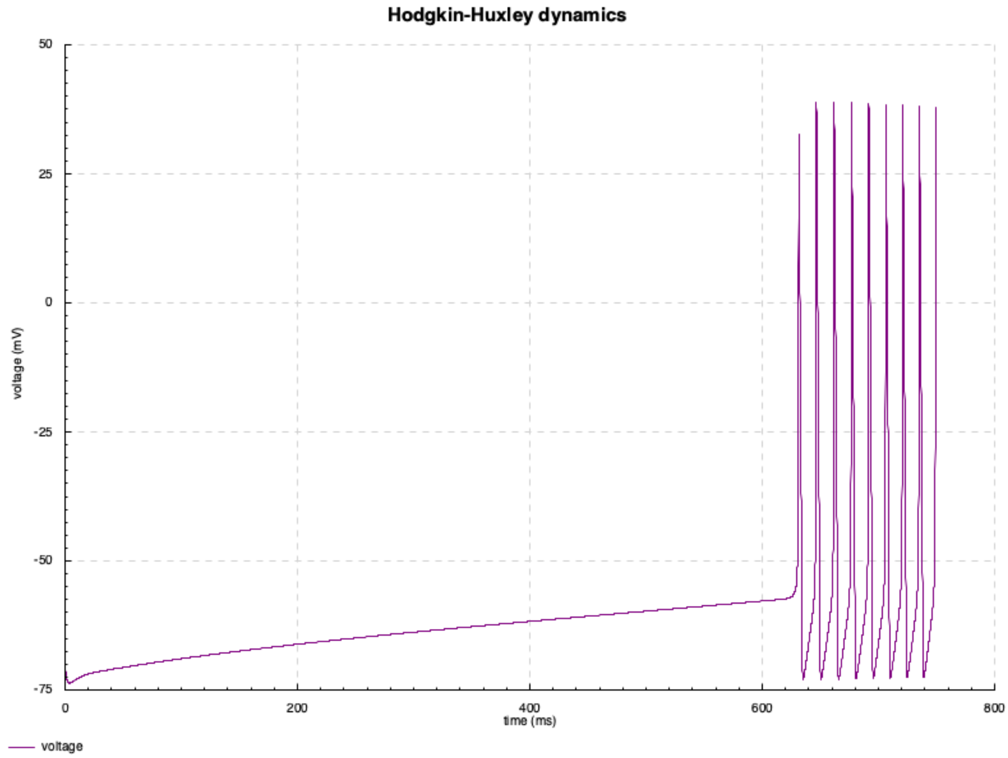


Figure 5.8: Hodgkin-Huxley model dynamics with $dt = 0.05$ and input current growing linearly from -5nA at a rate of $0.02dt$ per timestep. The simulated neuron starts by not firing, with the resting state slowly growing as the input current grows, then fires repeatedly.

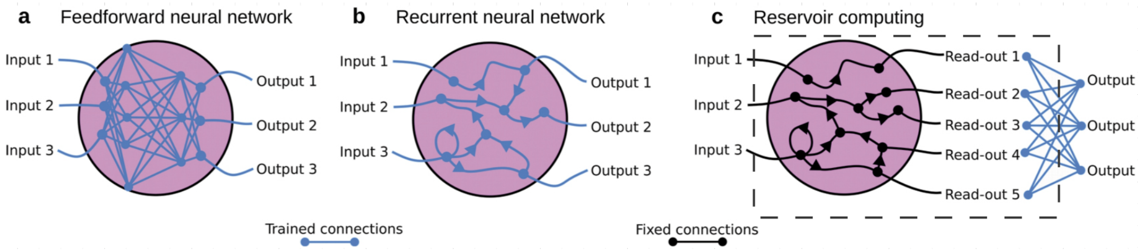


Figure 5.9: Reservoir setup when compared to neural networks. The dashed box represents an input/output boundary around the reservoir. Figure with modification from Cucchi et al. (2022) [42]. Licensed under CC BY.

5.3.2.4 Reservoir computer

Reservoir computing (RC) refers to a broad range of machine learning techniques [39] in the general paradigm of recurrent neural networks. Generally speaking, they involve exploiting the intrisic computations that happen in random, fixed recurrent neural networks (RNNs) to learn how to reproduce complex dynamics i.e. nonlinear timeseries, and have found many applications [40][41]. What makes the RC approach attractive is that, since the source of the dynamics is fixed, training is simplified when compared to regular neural networks, which require backpropagation: RCs can be trained by simple linear regression on the output layer. Another advantage is that again, since the dynamics are fixed, they do not need to be implemented in software, and can instead be implemented in physical systems [42]. Figure 5.9 contains a schematic showcasing the RC approach compared to neural networks.

The most representative type of reservoir computer is the *Echo State Network* (ESN) [43]. In this setup, the reservoir is composed of a large collection of stateful nodes. All nodes are connected to all other nodes, and these connections are weighted. Nodes also carry their own state. The reservoir has an input layer, through which data points of the timeseries to be learned is fed, and an output layer, through which the reservoir's output dynamics are exposed.

To actually use a reservoir to predict a timeseries, the process is organized in three stages, as follows:

- **Training / data collection**: In this stage, the reservoir accumulates training data from an instance of the nonlinear timeseries it should learn to predict. Both the states and inputs to the system are stored, and this stage ends with the output layer being trained with some form of linear regression.
- **Warm-up / touching**: The reservoir states are reset, and another instance of the nonlinear timeseries (the test data) is provided as input to the reservoir for a limited number of timesteps. This is done so that the system's states have time to reflect an input series and not its own initial values.
- **Prediction / going**: The output of the reservoir is now looped back on itself and given as input. Now it is running essentially as a closed dynamical system, and its outputs should be considered its predictions of how it "thinks" the test timeseries will continue throughout time.

The *touch and go* terminology is to invoke an intuition of gradually letting go of some system and letting it run on its own after guiding it for a while. Concretely, we use the following update rule for the reservoir states:

$$res(t+1) = tanh(W_{res} * res(t) + W_{in} * input) \tag{5.19}$$

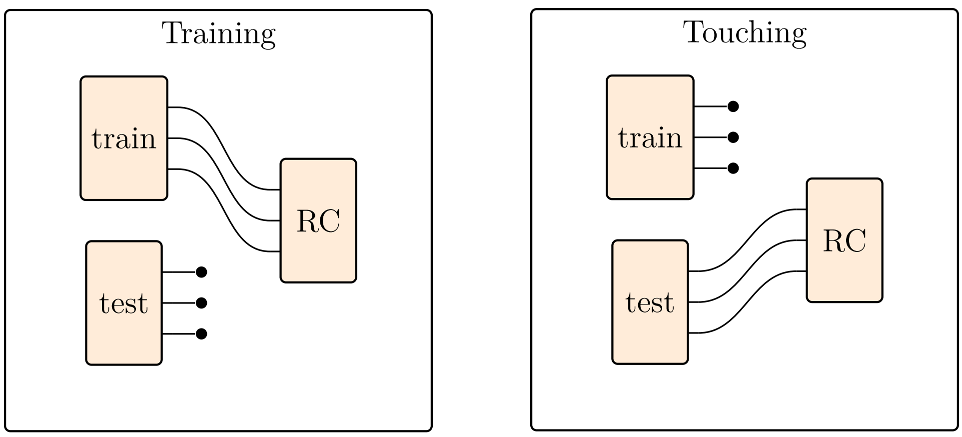
where *res* is a vector containing the reservoir states, and the following training method:

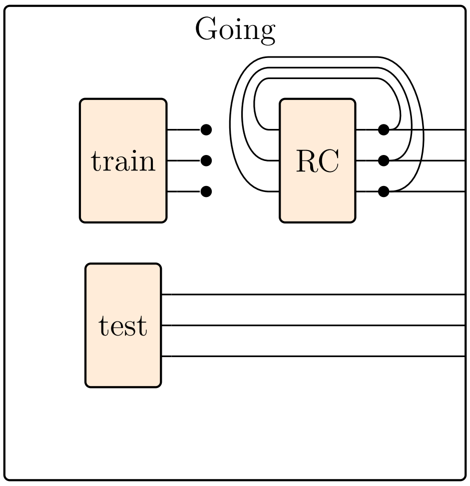
$$W_{out} = (H_{states}^T * H_{states} + rI)^{-1} * (H_{states}^T * H_{inputs}) \tag{5.20}$$

To explain the variables in order: H_{states} is a matrix representation of the history of states of the reservoir, accumulated in stage 1, and given by 5.19. r is the ridge parameter, which multiplies the identity matrix I. Ridge regression is helpful to avoid overfitting. H_{inputs} is a history of system inputs.

Reservoir computer in terms of Poly

We will show an implementation of a reservoir computer that learns the dynamics of the Lorenz system, whose implementation we have already shown. The dashed box in Figure 5.9 is reminiscent of a polynomial as an interface, and the system taking in inputs from different sources depending on its outputs suggests a mode-dependent dynamical system. Mode-dependence on the type level is not needed, since the systems in question always receive inputs and provide outputs of the same types, but the wiring pattern changes through time. Our situation is with the three wiring patterns below, each corresponding to one of the stages mentioned:





The boxes `train` and `test` represent instances of the previously implemented Lorenz system, starting at different initial conditions. The outputs we will care to plot are those at *Going* stage, when the reservoir is outputting its attempt at predicting the input sequence from the point it was left off from the touching stage. Not all of the Agda code is included, since it is quite big and again, readers can consult the implementation in `Dynamical/Reservoir/ModeDependent.agda`. But some key areas to highlight are the following definitions.

Firstly, the state of the reservoir system:

```
data ReservoirState (numNodes : ℕ) (systemDim : ℕ) : Set where
  Coll : (nodeStates : Vec ℝ numNodes)
         (counter : ℕ)
         (statesHistory : Vec (Vec ℝ numNodes) counter)
         (systemHistory : Vec (Vec ℝ systemDim) counter) →
         ReservoirState numNodes systemDim
  Touch : (nodeStates : Vec ℝ numNodes)
          (counter : ℕ)
          (outputWeights : OutputWeights numNodes systemDim)
          →
          ReservoirState numNodes systemDim
  Go : (nodeStates : Vec ℝ numNodes)
       (outputWeights : OutputWeights numNodes systemDim)
       →
       ReservoirState numNodes systemDim
```

The reservoir ought to start in the *collection* state, where it will accumulate some amount of states AND system history, as well as keep a current set of states for the nodes. It then goes into the *touching* state, where it has a separate counter keeping track of how many state updates to perform before starting prediction. Although the touching state does not use the trained output weights obtained from the collection state, it needs to carry it around in order to pass it to the *going* state, where the reservoir is now only concerned with its dynamics.

Second, the output of the reservoir:

```
data ReservoirOutput (systemDim : ℕ) : Set where
    stillColl : ReservoirOutput systemDim
    stillTouch : ReservoirOutput systemDim
    predicting : Vec ℝ systemDim → ReservoirOutput systemDim
```

A regular sum type, since we do not care about what the output of the system is until it is actually ready to predict.

Thirdly, the parallelizing of the systems in question. The interesting thing about this is the `reservoir` function, which takes some parameters to construct a `DynamicalSystem`. Here it is made clear that the input and reservoir weights are fixed:

```
preLorRes : (numNodes trainingSteps touchSteps : ℕ) →
            (dt : ℝ) →
            InputWeights numNodes 3 →
            ReservoirWeights numNodes →
            DynamicalSystem
preLorRes numNodes trainingSteps touchSteps dt inputWeights reservoirWeights =
  -- Training system
  lorenz dt &&&
  -- Test system
  lorenz dt &&&
  -- Reservoir of dynamics + readout layer
  reservoir numNodes 3 trainingSteps touchSteps inputWeights reservoirWeights
```

The types `InputWeights` and `ReservoirWeights` are just synonyms for matrices. Finally, the wiring pattern lens, which captures the changing inputs depending on system outputs:

```
lrWiringDiagram : (numNodes trainingSteps touchSteps : ℕ) →
                  (dt : ℝ) →
                  (iw : InputWeights numNodes 3) →
                  (rw : ReservoirWeights numNodes) →
                  Lens (interface (preLorRes numNodes
                                             trainingSteps
                                             touchSteps
                                             dt
                                             iw
                                             rw))
                       (emitter (ReservoirOutput 3 × (X × Y × Z)))\
lrWiringDiagram numNodes trainingSteps touchSteps dt iw rw =
    outerOutputsFrom ⇆ innerInputsFrom
    where outerOutputsFrom : (X × Y × Z) ×
                             (X × Y × Z) ×
                             ReservoirOutput 3 →
                             ReservoirOutput 3 × (X × Y × Z)
          outerOutputsFrom (_ , test , ro) = ro , test
          -- Provide inputs from different sources
          -- depending on reservoir's output
```

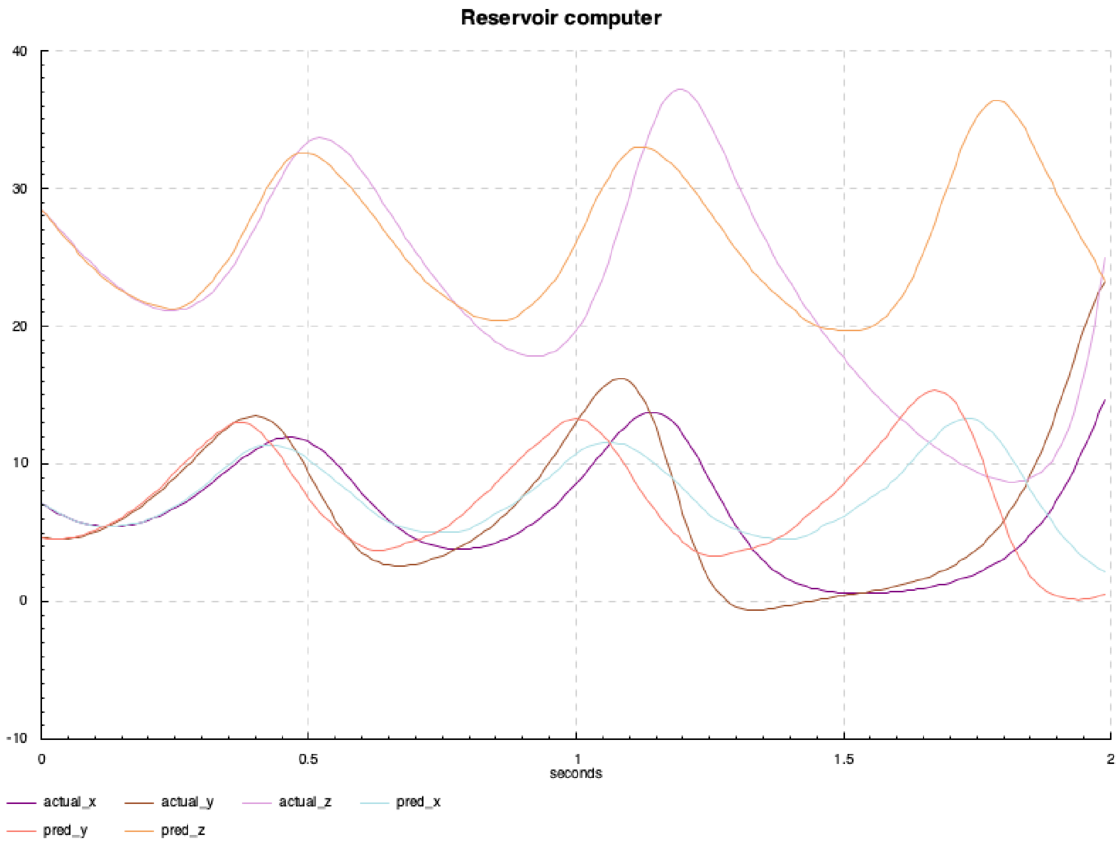


Figure 5.10: Reservoir dynamics: the reservoir's output along with the test sequence it is trying to predict along all dimensions.

```
innerInputsFrom : (X × Y × Z) ×
                  (X × Y × Z) ×
                  ReservoirOutput 3 →
                  ⊤ →
                  (⊤ × ⊤ × Vec ℝ 3)
innerInputsFrom (lorOut , lorTestOut , stillColl) tt =
  tt , tt , Lorenz.outToVec lorOut
innerInputsFrom (lorOut , lorTestOut , stillTouch) tt =
  tt , tt , Lorenz.outToVec lorTestOut
innerInputsFrom (lorOut , lorTestOut , predicting x) tt =
  tt , tt , x
```

The `outerOutputsFrom` function just always provides the reservoir output along with the test sequence, so as to facilitate exposing the data to a list we can plot. The `innerInputsFrom` function, which takes care of filling inputs, is key to representing the three stages: the input to the reservoir is *its own output* when it has started predicting.

For deep mathematical reasons, reservoirs (or any other model of chaotic systems) cannot perfectly reproduce the learned time series. For an in-depth explanation of this, see [44]. However, they can track the system's dynamics quite well for a while before diverging. Figure 5.10 shows the result of our run over 200 predictions, with 8000 training steps and 600 reservoir nodes - the system, although small, has clearly learned some of the intrinsic properties of the input sequence.

6

Discussion

This chapter covers a discussion on certain topics about the formalization of **Poly**. The characterization of equality between lenses is discussed, how useful Cubical Agda was in the formalization, as well as the ease of use of **Poly** in implementing dynamical systems. Further, the consequence of mixing Cubical Agda with `agda-categories` is discussed, and the notion of wild polynomials. Finally, some areas of future work are given.

6.1 Equality of lenses

The fact that lenses (as well as polynomials and charts) are defined as (dependent) sigma types makes equality of lenses (as well as polynomials and charts) a major pain point. The first component of the sigma type is directly comparable, but for the second component, `subst` is needed to make the types comparable. This problem shows up every time an equality between lenses needs to be proved. If the proof of the first component being equal is `refl`, the `subst` is easily removed by using `substRefl`. But, anytime the proof of the first component is non-trivial, the complexity of the equality proof quickly increases. Therefore, the more powerful variant of lens equality was developed, `lens`$\equiv_p$, where the `subst` is pushed inside the term as much as possible. This variant of lens equality has shown to be very useful and is used in most places in the code. Hopefully, the efforts to characterize lens equality are useful to any future formalizations of **Poly**.

Although the same solution is used for charts, there is a possible alternative approach. A chart can be represented as a single function, satisfying a predicate instead of being defined as two different maps in a sigma type. This representation makes equality between charts to be simply equality between functions. Although, this approach was not explored further.

6.2 Proving with Cubical Agda

Cubical Agda provides some very useful features, such as:

- Identifying isomorphism and equality,
- Transporting equalities between types and vice versa,

- Function extensionality,
- Wealth of lemmas and properties in its standard library,
- Substitution of types via `subst`,
- Higher Inductive Types.

However, some aspects of it are not as smooth. No pattern matching on `refl` sometimes complicates proofs, constant transports are not always equal to `refl` for subtle technical reasons, nested `subst` and `transport` are very difficult to deal with, introducing interval variables in proof definitions leads to warnings that make for a noisy IDE experience, and the names of lemmas and their documentation in the standard library are not informative enough. The issue of nested transports is a particularly thorny one: very often, an equality type states something that a human can verify informally quite easily, but doing the needed plumbing to get a proof to compute is difficult and results in extremely long proofs that are actually simple in idea.

6.3 Challenges in implementing dynamical systems

The implementation of a couple of dynamical systems has led us, the authors of the thesis, to an impression of how it is to use polynomial functors in a practical implementation. A few different aspects stand out when it comes to writing dynamical systems in Agda using polynomial functors.

6.3.1 Datatypes in Agda

When creating dynamical systems, many data types need to be created, making it easy to get lost in the details. There need to be data types for the state of each system, for the positions and output of each system, as well as for each input at each output. This is easier to handle on paper by abusing notation and leaving details out, but in Agda, everything is explicit, making it very verbose. Also, using polynomials created by the operators defined, such as addition or parallel product, is annoying. In these cases, the data types used behind the scenes are coproduct and product, with general constructor names that are terrible to work with when implementing dynamical systems, or wiring the wiring diagrams.

There is also the issue of handling the empty type $\perp$ when programming, especially when it is a direction at some position. It is of course impossible to provide a value of this type, so while it in theory nicely represents a state where "no input can be provided", in practice this means that a lot of care and thought need to be given to how to either avoid getting in such a state, or never attempt updating the system when there.

6.3.2 Delayed timesteps

When designing and implementing a dynamical system, it is important to consider timing in the flow of information. For example, given an input to a system, the output for that input is not received until the next time step. This creates a delay, which is in fact used by the Fibonacci example, where the identity system simply delayed the input by one frame. However, taking into account these delays in more extensive systems creates a lot of complexity to keep track of. The reservoir system suffered from this as well, needing state transition counters to account for delays in the inputs. In a setting like Python + NumPy, all the states can always be accessed as indices in a matrix that keeps track of the history if needed.

6.3.3 Printing

When the systems did not work as expected, a lot of effort was put into figuring out what was going wrong. For example, a first attempt at debugging the reservoir system was by manually changing the output type of all systems to include their state, which led to a stream of a massive product type to be analyzed by the caller in `Plot.agda`. When this became unwieldy, the idea to use the Haskell FFI to call out to `Debug.Trace.trace` was introduced, but this comes at the expense of some flexibility since it cannot be used in the `readout` functions, as the `update` function depends on that function's output. A tool similar to debuggers for imperative languages would have been useful for stepping through the dynamical systems and looking into their internal state, or perhaps some abstraction that plugs into dynamical systems in a nice way that we could not think of.

6.3.4 Freezing system states

Sometimes, we want a system to be unable to change some aspect of its state, but want them to "carry their state around" and transition its contents to some other state. An example of this is the reservoir's `outputWeights`. Recalling what that datatype looks like:

```
data ReservoirState (numNodes : ℕ) (systemDim : ℕ) : Set where
  Coll : (nodeStates : Vec ℝ numNodes)
         (counter : ℕ)
         (statesHistory : Vec (Vec ℝ numNodes) counter)
         (systemHistory : Vec (Vec ℝ systemDim) counter) →
         ReservoirState numNodes systemDim
  Touch : (nodeStates : Vec ℝ numNodes)
          (counter : ℕ)
          (outputWeights : OutputWeights numNodes systemDim) -- should be
                                                              -- immutable!

          →
          ReservoirState numNodes systemDim
  Go : (nodeStates : Vec ℝ numNodes)
       (outputWeights : OutputWeights numNodes systemDim)     -- should be
```

```
                                                    -- immutable!
       →
       ReservoirState numNodes systemDim
```

There is no straightforward way, as far as we can tell, to prevent the `outputWeights` component of the reservoir from being updated in the type level. An attempt was made by wrapping the contents of the `Touch` and `Go` states in a `Reader` monad, and then having the readout function unroll the computation contained in it, but this was quite hard to work with due to how `Reader` works in Agda - it is designed to be capable of working in categories other than **Set**, so more general than Haskell's version, but this causes the need for more manual tinkering to get it right. And even if it worked, still nothing would prevent the state update function from simply replacing the computation with something else. Perhaps linear types could be of help here too, by ensuring that particular component of the state is only used once when in the *touching* stage.

6.4 Mixing Cubical Agda with `agda-categories`

A big consequence of mixing Cubical Agda with `agda-categories` is that it hinders the formalization from being merged into either of the repositories. The choice was to use Cubical Agda for its usefulness in proofs and `agda-categories` for its richness of categorical constructs. However, Cubical Agda has its own category theory library and would not want any code using `agda-categories` to be merged into it. At the same time, `agda-categories` would not want any code that depends on Cubical Agda. This issue hinders the code from being integrated into any of these repositories.

The code could be modified to be mergeable into the Cubical Agda library. All the categorical constructs used from `agda-categories` could be rewritten to use Cubical Agda's constructs. The constructs only available in `agda-categories` could be defined and added to the Cubical library. At the core, all the proofs should remain the same, although they might need some slight modifications. An important detail that must be considered when adapting the code only to use Cubical Agda is the notion of wild polynomials.

6.5 Wild polynomials

Some requirements differ between providing an instance of a category in `agda-categories` versus in the Cubical Agda library. One crucial difference is that Cubical Agda requires the lenses to be sets, using the cubical notion `isSet`. To provide a proof that lenses are sets, it is required that the type of positions and all directions of polynomials are sets. This difference gives rise to two different variants of polynomials. The normal polynomials, also called wild polynomials, without any requirement of the positions or directions being sets, are used to define a category in `agda-categories`. As well as the more strict variant of polynomials, called `SetPolynomial`, used to define an instance of a category in Cubical Agda library.

Both variants of polynomials are provided in the code, but wild polynomials are mostly used as they are more general. Wild polynomials also avoid the use of `isSet`, which would add noise to the code. However, set polynomials together with an instance of a category for Cubical agda library can be found in the code. In fact, some constructs and proofs require polynomials to be sets, such as the proof showing that **Poly** has equalizers.

6.6 Future work

There are many areas for future work formalizing **Poly**.

6.6.1 Comonoids and small categories

Finishing this isomorphism would be interesting - the existing attempt fails when translating the comonoid laws into category laws, mostly due to how difficult it is to work with the associativity proof. This suggests that a different formulation of comonoids, that does not need `transport`, might be fruitful.

6.6.2 Bicomodules are parametric right adjoints

Another construct covered in the poly-book is bicomodules, which relate to comonoids. Bicomodules have a practical use as data migration for databases [45] as well as effect handlers. Therefore, implementing and using bicomodules should be of particular interest to software developers.

6.6.3 Composition of squares between charts and lenses

The commuting squares between charts and lenses have been defined as `LensChartCommute`. Showing that these squares compose both horizontally and vertically is an important step in finalizing the formalization of the double category between charts and lenses. The proof that the map on positions of the squares composes is done, but the proof for map on directions remains undone, due to an explosion of subst's.

6.6.4 Consistency of distributed data types

There is some speculation that polynomial functors can be a good model of consistent distributed data storage, like Conflict-Free Replicated Data Types, so the categorical setting could be fruitful for proving the consistency of CRDTs.

6.6.5 Other categorical properties

The categorical constructs and properties formalized in this thesis comprise only a handful of all the properties that **Poly** exhibits. A natural extension is to formalize more constructs. For instance, the coequalizer, described informally in natural language in the poly-book, is an important construct as it shows that **Poly** has all small colimits. The poly-book contains plenty of more proofs and examples that could be formalized.

6.6.6 Dynamical systems

This thesis implements several dynamical systems, but it is a large field. Systems come in various flavors: time-delayed, open, closed, chaotic, stable, unstable, discrete, continuous, stochastic, deterministic, hybrid, etc. It would be interesting to continue to explore this landscape while leveraging the properties of **Poly** in the dynamical systems. For instance, our use of mode-dependence, in which we believe lies the true potential of **Poly**, is minor. Implementing more types of dynamical systems could also lead to common patterns or problems arising, whose solutions could lead to a better ecosystem in using **Poly** for modeling dynamics.

6.6.7 Interaction between different fields that Poly models

Poly is good at modeling databases, dynamical systems, and information theory, three fields that are loosely connected. **Poly** could be a setting in which to deepen these connections. **Poly** and coalgebras on it have also shown some promise in deep learning.

6.6.8 Building programming languages on top of Poly

As we've seen, **Poly** is cartesian closed, meaning it can model the Simply-Typed Lambda Calculus. One could pursue the development of programming languages with the primitives of **Poly** that have access to the modeling power of the category natively. **Poly** also naturally models Turing machines and is suited to imperative programming - since the STLC is equivalent in expressive power to Universal Turing machines, **Poly** can also remain the setting where the equivalence between imperative and declarative programs is formally proved.

7

Conclusion

This thesis has successfully formalized a significant amount of the theory of **Poly** and implemented several dynamical systems relying on the theory. The theory chapters have formalized the categories themselves, initial objects, terminal objects, products, coproducts, composition and parallel product as monoidal structures, exponential object, charts, quadruple adjunction with sets, as well as many more various proofs, examples, and lemmas needed. The implementation part has shown how lenses can represent dynamics in dynamical systems, how polynomials act as interfaces, how systems are wired together, how to implement systems, how to install behavior into wiring diagrams, and how to run systems arriving at concrete programs. Lenses between special polynomials have been shown to correspond to different concepts, such as Moore or Mealy machines, which shows the power polynomials and lenses possess as abstractions. Some programs implemented are Fibonacci sequence generators, the Lorenz system, the Hodgkin-Huxley model, and reservoir computers. Combining theory and practice, commuting squares between lenses and charts have been used to show how one dynamical system can simulate another.

The work of this thesis has met its purpose of contributing to the polynomial functors ecosystem, showing the feasibility of formalizing **Poly** and associated categories, as well as providing a solid ground to build up more advanced concepts, both theoretical and practical.

Bibliography

[1] M. Abbott, T. Altenkirch, and N. Ghani, "Categories of containers," in *Foundations of Software Science and Computation Structures*, A. D. Gordon, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 23–38, ISBN: 978-3-540-36576-1.

[2] N. GAMBINO and J. KOCK, "Polynomial functors and polynomial monads," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 154, no. 1, pp. 153–192, Sep. 2012. DOI: 10.1017/s0305004112000394. [Online]. Available: https://doi.org/10.1017/s0305004112000394.

[3] D. I. S. Nelson Niu, *Polynomial Functors: A Mathematical Theory of Interaction*. https://topos.site/poly-book.pdf, 2023.

[4] M. Scheffer, S. Carpenter, J. A. Foley, *et al.*, "Catastrophic shifts in ecosystems," *Nature*, vol. 413, pp. 591–596, 2001. DOI: 10.1038/35098000. [Online]. Available: https://doi.org/10.1038/35098000.

[5] E. Zafiris, "Complex systems from the perspective of category theory: I. functioning of the adjunction concept," *Axiomathes*, vol. 15, pp. 147–158, 2005. DOI: 10.1007/s10516-005-3790-0. [Online]. Available: https://doi.org/10.1007/s10516-005-3790-0.

[6] N. Ghani, J. Hedges, V. Winschel, and P. Zahn, *Compositional game theory*, 2018. arXiv: 1603.04641 [cs.GT].

[7] P. Perrone, *Markov categories and entropy*, 2022. arXiv: 2212.11719 [cs.IT].

[8] J. C. Baez and B. S. Pollard, "A compositional framework for reaction networks," *Reviews in Mathematical Physics*, vol. 29, no. 09, p. 1 750 028, 2017.

[9] S. Libkind, A. Baas, E. Patterson, and J. Fairbanks, "Operadic modeling of dynamical systems: Mathematics and computation," *Electronic Proceedings in Theoretical Computer Science*, vol. 372, pp. 192–206, Nov. 2022. DOI: 10.4204/eptcs.372.14. [Online]. Available: https://doi.org/10.4204/eptcs.372.14.

[10] [Online]. Available: https://agda.readthedocs.io/en/v2.6.3/getting-started/what-is-agda.html.

[11] W. A. Howard, "The formulae-as-types notion of construction," *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, vol. 2, pp. 479–490, 1980.

[12] A. Bove and P. Dybjer, "Dependent types at work," in *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, Eds. Berlin,

Heidelberg: Springer Berlin Heidelberg, 2009, pp. 57–99, ISBN: 978-3-642-03153-3. DOI: 10.1007/978-3-642-03153-3_2. [Online]. Available: https://doi.org/10.1007/978-3-642-03153-3_2.

[13] *Glasgow haskell compiler: Gadts*, Accessed: 2023-05-17. [Online]. Available: https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/gadt.html.

[14] [Online]. Available: https://agda.readthedocs.io/en/v2.6.0/language/cubical.html.

[15] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[16] Agda, *Agda/cubical: An experimental library for cubical agda*. [Online]. Available: https://github.com/agda/cubical.

[17] A. Vezzosi, A. Mörtberg, and A. Abel, "Cubical agda: A dependently typed programming language with univalence and higher inductive types," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. DOI: 10.1145/3341691. [Online]. Available: https://doi.org/10.1145/3341691.

[18] B. Fong and D. I. Spivak, *Seven sketches in compositionality: An invitation to applied category theory*, 2018. arXiv: 1803.05316 [math.CT].

[19] *1lab - a formalised, cross-linked reference resource for cubical methods in homotopy type theory.* Accessed: 2023-05-17, 2023. [Online]. Available: https://1lab.dev.

[20] Agda, *Agda/cubical: An experimental library for cubical agda*. [Online]. Available: https://github.com/agda/cubical.

[21] J. Z. S. Hu and J. Carette, "Formalizing category theory in agda," in *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ACM, Jan. 2021. DOI: 10.1145/3437992.3439922. [Online]. Available: https://doi.org/10.1145/3437992.3439922.

[22] J. Kock, *Polynomial functors and trees*, 2009. arXiv: 0807.2874 [math.CT].

[23] D. I. Spivak, *Functorial aggregation*, 2023. arXiv: 2111.10968 [math.CT].

[24] Amuricys, *Amuricys/polynomial-functors*. [Online]. Available: https://github.com/amuricys/polynomial-functors.

[25] Iceland_jack, $Iceland_jack'sattemptatpolynomialfunctorsinhaskell$, https://web.archive.org/web/20230523131853/https://twitter.com/Iceland_jack/status/1646315831402323970, Accessed: 2023-05-17, 2023.

[26] S. Visscher, *Sjoerd visscher's attempt at polynomial functors in haskell*, https://web.archive.org/web/20230523131533/https://gist.github.com/sjoerdvisscher/0fd871b1c2df4f4750945eb08378fb4a, Accessed: 2023-05-17, 2023.

[27] daig, *Daig/polynomial-functors*. [Online]. Available: https://github.com/daig/polynomial-functors.

[28] D. J. Myers, "Categorical systems theory," Manuscript in preparation, 2022.

[29] D. Vagner, D. I. Spivak, and E. Lerman, *Algebras of open dynamical systems on the operad of wiring diagrams*, 2015. arXiv: 1408.1598 [math.CT].

[30] e. a. Urs Schreiber Mike Shulman, *Concept with an attitude*, Online; accessed 17-May-2023, 2023. [Online]. Available: `https://ncatlab.org/nlab/show/concept+with+an+attitude`.

[31] [Online]. Available: `https://hackage.haskell.org/package/optparse-applicative`.

[32] [Online]. Available: `https://hackage.haskell.org/package/Chart`.

[33] *Hmatrix: A haskell library of linear algebra, numerical computation, and general utilities for scientific computation*, Accessed: 2023-05-17. [Online]. Available: `https://hackage.haskell.org/package/hmatrix`.

[34] [Online]. Available: `https://hackage.haskell.org/package/blas`.

[35] J. D. Murray, *Mathematical Biology: I. An Introduction*, 3rd. Springer, 2002.

[36] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of the Atmospheric Sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[37] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, p. 500, 1952.

[38] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal Dynamics: From single neurons to networks and models of cognition.* Cambridge University Press, 2014. [Online]. Available: `https://neuronaldynamics.epfl.ch/index.html`.

[39] B. Schrauwen, D. Verstraeten, and J. Campenhout, "An overview of reservoir computing: Theory, applications and implementations," Jan. 2007, pp. 471–482.

[40] A. Ferreira, T. Ludermir, R. Aquino, M. Lira, and O. Neto, "Investigating the use of reservoir computing for forecasting the hourly wind speed in short-term," Jun. 2008, pp. 1649–1656. DOI: `10.1109/IJCNN.2008.4634019`.

[41] H. Jaeger, "Adaptive nonlinear system identification with echo state networks," in *Proceedings of the 2002 45th IEEE Conference on Decision and Control*, IEEE, vol. 3, 2002, pp. 2696–2701.

[42] M. Cucchi, S. Abreu, G. Ciccone, D. Brunner, and H. Kleemann, "Hands-on reservoir computing: A tutorial for practical implementation," *Neuromorphic Computing and Engineering*, vol. 2, no. 3, p. 032 002, Aug. 2022. DOI: `10.1088/2634-4386/ac7db7`. [Online]. Available: `https://dx.doi.org/10.1088/2634-4386/ac7db7`.

[43] H. Jaeger, "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note," Bonn, Germany: German National Research Center for Information Technology GMD Technical Report, Tech. Rep. 148, 2001.

[44] S. H. Strogatz, *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering.* Westview press, 2001.

[45] [Online]. Available: `https://topos.site/blog/2022/08/imagining-bicomodules-with-type-theory/`.

[46] *Nix: The purely functional package manager*, Accessed: 2023-05-17. [Online]. Available: `https://nixos.org/nix/`.

[47] *Chart: A haskell library for producing 2d charts and plots*, Accessed: 2023-05-17. [Online]. Available: `https://hackage.haskell.org/package/Chart`.

[48] *Optparse-applicative: Utilities and combinators for parsing command line options*, Accessed: 2023-05-17. [Online]. Available: `https://hackage.haskell.org/package/optparse-applicative`.

[49] *Glasgow haskell compiler: Datakinds*, Accessed: 2023-05-17. [Online]. Available: `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/data_kinds.html#extension-DataKinds`.

A

Appendix 1: Haskell supporting code

The Agda FFI is used to interact with Haskell and its ecosystem for three main purposes: fast matrix inversion, plotting graphs, and a convenient command-line interface for running and plotting the continuous dynamical systems. The Nix package manager [46] is used to plug the Haskell code and its ecosystem with the Agda code.

A.1 Matrix inversion

In the reservoir example 5.3.2.4, many matrix operations are used. Many of these operations are implemented directly in Agda - see the `Dynamical/Reservoir/Matrix` folder - using facilities from the standard library's `Data.Vec` type. However, inverting a matrix is an expensive operation that requires an algorithm that is quite performance-intensive. For this reason, an existing implementation in the Haskell ecosystem from the HMatrix [33] library is used. This is done by declaring a postulate and using the `FOREIGN` pragma in the Agda module, and accessing the Haskell module by importing it in the generated Haskell code:

```
...
{-# FOREIGN GHC
import HsMatrix
#-}

postulate
  invertMatrixAsListTrusted : List (List ℝ) → List (List ℝ)
{-# COMPILE GHC invertMatrixAsListTrusted = invertMatrixAsList #-}
...
```

The matrix is temporarily represented as a list of lists, where each list is a row, to simplify translating the matrix dimensions to the Agda level. In this way, only the type conversions directly in Agda have to be taken care of. The code in the Haskell module is simply this:

```
...
import Numeric.LinearAlgebra qualified as HMat
```

```
...
invertMatrixAsList :: [[Double]] -> [[Double]]
invertMatrixAsList = HMat.toLists . HMat.pinv . HMat.fromLists
```

The matrix is then converted back to the Agda representation - a length-indexed vector of length-indexed vectors, by using the `trustMe` construct from the standard library to convince Agda that the matrix has the right dimensions:

```
...
open import Relation.Binary.PropositionalEquality.TrustMe
...
fl : ∀ {A : Set} {n} → (l : List A) → {l : L.length l ≡ n} → Vec A n
fl l {refl} = fromList l

fromListOfLists : ∀ {n m} →
    (l : List (List ℝ)) →
    {p₁ : L.length l ≡ n} →
    {p₂ : M.map L.length (L.head l) ≡ M.just m}
    → Matrix ℝ n m
fromListOfLists [] {refl} = 𝕄 []
fromListOfLists (x :: xs) {refl} {p} =
    𝕄 (fl x {maybepr p} ::
            (V.map (\x → fl x {trustMe}) $ fl xs {trustMe}))
    where maybepr : ∀ {m n} → M.just m ≡ M.just n → m ≡ n
          maybepr refl = refl

_⁻¹ : ∀ {n : ℕ} → Matrix ℝ n n → Matrix ℝ n n
_⁻¹ {n} (𝕄 m) =
  let asList = toList $ V.map toList m
      inverted = invertMatrixAsListTrusted asList
  in fromListOfLists inverted {trustMe} {trustMe}
infixl 40 _⁻¹
```

In this way, the $\_^{-1}$ operation can be made to take in square matrices and return square matrices of the same dimension.

A.2 Graph plotting

The figures in the continuous systems are all generated via the Haskell library Chart [47]. This case is similar to the above, with the simplification that there is no need to use `trustMe`. Instead, the Agda postulate for the plotting function runs in IO. A minor issue is solved by implementing a custom product type to guarantee the FFI that no dependent typing is involved. The postulate code is as follows:

```
open import Data.Product as P hiding (_×_) renaming (_,_ to _,p_)
record _×_ (A B : Set) : Set where
  constructor _,_
  field
```

```
    fst : A
    snd : B

fromSigma : {A B : Set} → A P.× B → A × B
fromSigma ( a ,p b ) = a , b

postulate
  plotDynamics : String →
                 String →
                 String →
                 Float →
                 List (String × List Float) → IO ⊤

{-# FOREIGN GHC import HsPlot #-}
{-# COMPILE GHC plotDynamics = plotToFile #-}
{-# COMPILE GHC _×_ = data (,) ((,)) #-}
```

The corresponding Haskell code is small enough that the entire module is included:

```
-- Dynamical/Plot/src/HsPlot.hs
{-# LANGUAGE BlockArguments #-}
{-# LANGUAGE ImportQualifiedPost #-}
{-# LANGUAGE OverloadedStrings #-}

module HsPlot where

import Graphics.Rendering.Chart.Easy
import Graphics.Rendering.Chart.Backend.Cairo
import Control.Monad (forM_)
import Data.Text qualified as T

plotToFile :: T.Text ->
              T.Text ->
              T.Text ->
              Double ->
              [(T.Text, [Double])] ->
              IO ()
plotToFile xaxisTitle yaxisTitle title dt lines =
    toFile def (T.unpack . T.replace " " "_" $ T.toLower title <> ".png") $
    do
        layout_title .= T.unpack title
        layout_x_axis . laxis_title .= T.unpack xaxisTitle
        layout_y_axis . laxis_title .= T.unpack yaxisTitle
        setColors . fmap opaque $
            [purple, sienna, plum,
             powderblue, salmon, sandybrown,
             cornflowerblue, blanchedalmond,
```

```
             firebrick, gainsboro, honeydew]
    forM_ lines \(name, l) ->
        plot (line (T.unpack name) [zip [0, dt..] l ])
```

The string arguments are given as `Data.Text.Text`, even though `Chart` handles `Strings`, because that is what Agda's `Data.String.String` corresponds to; see the table in the official documentation.

A.3 Command-line interface

Finally, there is the issue of conveniently running a system of choice, with a given selection of parameters. This is solved by using optparse-applicative [48], a well-established Haskell library. The FFI code looks similar to above, but the main interest is now to convert between types that are pattern-matched on in the main Agda program. The Agda code, with the type containing the paramaters of the different systems is as follows:

```
{-# FOREIGN GHC
import OptparseHs
#-}

data SystemParams : Set where
  ReservoirParams : (rdim : ℕ) →
                    (trainSteps touchSteps outputLength : ℕ) →
                    (lorinitx lorinity lorinitz dt variance : Float) →
                    SystemParams
  LorenzParams    : (lorinitx lorinity lorinitz dt : Float) → SystemParams
  HodgkinHuxleyParams : (dt : Float) → SystemParams
  LotkaVolterraParams : (α β δ γ r0 f0 dt : Float) → SystemParams
{-# COMPILE GHC SystemParams = data SystemParams
    (ReservoirParams |
     LorenzParams |
     HodgkinHuxleyParams |
     LotkaVolterraParams) #-}

record Options : Set where
  constructor mkopt
  field
    system   : SystemParams

{-# COMPILE GHC Options = data Options (Options) #-}

postulate
  parseOptions : IO Options
{-# COMPILE GHC parseOptions = parseOptions #-}
```

And its corresponding Haskell type is:

```
data SystemParams where
  ReservoirParams ::
    { rdimf          :: Integer
    , trainStepsf    :: Integer
    , touchStepsf    :: Integer
    , outputLengthf :: Integer
    , lorinitx       :: Double
    , lorinity       :: Double
    , lorinitz       :: Double
    , dt             :: Double
    , variance         :: Double
    } -> SystemParams
  LorenzParams ::
    { lorinitx' :: Double
    , lorinity' :: Double
    , lorinitz' :: Double
    , dt'       :: Double
    } -> SystemParams
  HodgkinHuxleyParams ::
    { dt'' :: Double }
    -> SystemParams
  LotkaVolterraParams ::
    { alpha :: Double
    , beta :: Double
    , delta :: Double
    , gamma :: Double
    , r0 :: Double
    , f0 :: Double
    , dt''' :: Double }
    -> SystemParams
deriving instance Show SystemParams
```

The code for parsing this type looks like this:

```
systemParser :: Parser SystemParams
systemParser = hsubparser $
     command "Reservoir"
        (info reservoirParamsParser (progDesc "Reservoir system"))
  <> command "Lorenz"
        (info lorenzParamsParser (progDesc "Lorenz system"))
  <> command "HodgkinHuxley"
        (info hodgkinHuxleyParamsParser (progDesc "Hodgkin-Huxley system"))
  <> command "LotkaVolterra"
        (info lotkaVolterraParamsParser (progDesc "Lotka-Volterra system"))

reservoirParamsParser :: Parser SystemParams
```

```
reservoirParamsParser = ReservoirParams
  <$> option auto
    (long "numNodes" <> short 'n' <> metavar "NUMNODES" <>
        help "Number of nodes in the reservoir")
  <*> option auto
    (long "trainingSteps" <> short 't' <> metavar "TRAINSTEPS" <>
        help "Number of data points to train on")
  -- ... other params

lorenzParamsParser :: Parser SystemParams
lorenzParamsParser = LorenzParams <$> -- ... params
-- other systems with a similar type signature
```

It is worth remarking that Haskell and optparse-applicative allow for a concise expression of more type safety than a simple sum type, through the use of the language extensions for GADTs [13] and DataKinds [49], by having the `SystemParams` type be parametrized by the kind of a separate type called `ParamType`:

```
data ParamType
    = ReservoirParamsType
    | LorenzParamsType
    | HodgkinHuxleyParamsType
    | LotkaVolterraParamsType
  deriving Show
```

Which can then be given as an argument to `SystemParams`:

```
data SystemParams (p :: ParamType) where
  ReservoirParams ::
    { rdimf          :: Integer
    , trainStepsf    :: Integer
    , touchStepsf    :: Integer
    , outputLengthf  :: Integer
    , lorinitx       :: Double
    , lorinity       :: Double
    , lorinitz       :: Double
    , dt             :: Double
    , variance        :: Double
    } -> SystemParams ReservoirParamsType
    ... other constructors
```

This way, the individual parsers can be made to only parse specific sets of parameters. Forgetting an argument would give a compilation error:

```
lorenzParamsParser :: Parser (SystemParams LorenzParamsType)
lorenzParamsParser = LorenzParams
  <$> option auto (long "x0" <> short 'x' <> metavar "X0" <>
    help "Initial x value")
  <*> option auto (long "y0" <> short 'y' <> metavar "Y0" <>
```

```
  help "Initial y value")
-- <*> option auto (long "z0" <> short 'z' <> metavar "Z0" <>
--  help "Initial z value") error! (SystemParams LorenzParamsType)
-- expects this float
<*> option auto (long "dt" <> short 't' <> metavar "DT" <>
  help "Time step")
```

This is a preferable solution in the Haskell level, but converting the types given as arguments to a higher-kinded type in Agda to the DataKinds-produced types proved tricky, so we opted for the simpler solution of a pure sum type.

A.4 Building

The challenge was to get the Agda code that uses the Haskell code to have access to the larger Haskell ecosystem in an easily reproducible and user-friendly way. This is important because of the intensity of debugging in programming with polynomial functors - it is regular programming after all, just in a different framework.

A.4.1 Nix

The Nix[46] package manager is a good solution to this problem. It is widely used by the Haskell community, so it has good support and tooling for it, and provided a reliable way to build across the authors of the thesis, and for anyone wishing to use this code for themselves.

The most important primitive of Nix is the *derivation*. A derivation is an expression that describes how to build, package, install, run and in some cases deploy a piece of software. This can be an executable, a library, or a combination thereof. Derivations are meant to be deterministic, and will build only once unless the source code of the software or the derivation itself change. We tried to keep the Nix code well organized and encapsulating different responsibilities, though there's not much of it.

A.4.2 Building Agda with Nix

As mentioned, a Nix derivation includes a description of how to build a piece of software. In our case, this is the Agda executable that runs and plots the chosen system's dynamics, in `Dynamical/Plot.agda`. The actual building happens in a derivation's `buildPhase` attribute, and the dependencies go in `nativeBuildInputs` or `buildInputs`, depending on whether they're build or runtime dependencies. Nix provides a lot of flexibility on how to handle different types of depedencies, however, and there are other phases that can handle different parts of the build process, like `configurePhase`.

Our dependencies are the Agda libraries we use, which are `agda-categories`, `cubical` and `standard-library`, as well as the Haskell packages we have to use for each piece of Haskell code. Haskell has its own tooling regarding package management, the most conventional of which is Cabal, which we use to list the dependencies for each piece of Haskell code. We want to make these two ecosystems interact, and we can do this via the Agda compiler's provided method to give flags to GHC directly, the flag `-ghc-flag=`.

The question then is: how do we translate Haskell dependencies into the Nix environment, which we can then use to feed into the Agda compiler we invoke in the `buildPhase`? For this, we use a Nix tool called `cabal2nix`, which reads a *.cabal file and converts it to a Nix expression that can be manipulated like any other data structure in the language. We then collect these dependencies into a custom GHC environment, also via Nix: this creates a GHC executable that has access, in its own internal package database, to the Haskell packages provided.

There are also some more details to take care of, like overriding the Nix package repository's version of the Cubical library, configuring the build environment for Agda, and providing include flags for GHC. The code for all of this is in the expression below:

```
// agda.nix
  { pkgs ? import <nixpkgs> { } }:

  let
    agdaDepsNames = [ "standard-library" "agda-categories" "cubical" ];
    agdaDeps = builtins.map (n:
      if n == "cubical" then
        pkgs.agdaPackages.${n}.overrideAttrs (old: {
          version = "0.4";
          src = pkgs.fetchFromGitHub {
            repo = "cubical";
            owner = "agda";
            rev = "v0.4";
            sha256 = "0ca7s8vp8q4a04z5f9v1nx7k43kqxypvdynxcpspjrjpwvkg6wbf";
          };
        })
      else
        pkgs.agdaPackages.${n}) agdaDepsNames;
    getCabalStuff = name: path:
      (builtins.filter (d: !(isNull d))
        (pkgs.haskellPackages.callCabal2nix name path { }).propagatedBuildInputs);
    haskellDeps = (getCabalStuff "HsPlot" ./Dynamical/Plot/HsPlot.cabal);
    customGhc = pkgs.haskellPackages.ghcWithPackages (ps: haskellDeps);
    nativeBuildInputs = with pkgs; [
      customGhc
      (agda.withPackages (p: agdaDeps))
    ];
    commonConfigurePhase = ''
      export AGDA_DIR=$PWD/.agda
      mkdir -p $AGDA_DIR
```

```
      for dep in ${pkgs.lib.concatStringsSep " " agdaDepsNames}; do
        echo $dep >> $AGDA_DIR/defaults
      done
    '';
    buildCommand = haskellSourcePaths: ''
      agda -c \
        ${
          pkgs.lib.concatMapStrings (path: "--ghc-flag=-i${path} ")
          haskellSourcePaths
        } \
        ${
          pkgs.lib.concatMapStrings (dep: "--ghc-flag=-package=${dep.name} ")
          haskellDeps
        } \
        --ghc-flag=-package-db=${customGhc}/lib/ghc-${customGhc.version}/ \\
          package.conf.d \
        Dynamical/Plot/Plot.agda
    '';
  in { inherit commonConfigurePhase nativeBuildInputs buildCommand; }
```

Not to get into too much detail - this expression is a big `let`-block that binds the three variables `commonConfigurePhase`, `nativeBuildInputs` and `buildCommand` and exposes them. We then import these two variables in the files `default.nix` and `shell.nix`:

```
// default.nix
{ pkgs ? import <nixpkgs> { } }:

let
  agda = import ./agda.nix { inherit pkgs; };
  binName = "plot";
in pkgs.stdenv.mkDerivation {
  name = binName;
  src = ./.;
  nativeBuildInputs = agda.nativeBuildInputs;
  configurePhase = agda.commonConfigurePhase;
  buildPhase = agda.buildCommand
    [ "Dynamical/Plot/src" "Dynamical/Matrix/src" ] ;
  installPhase = ''
    mkdir -p $out/bin
    cp $TMP/poly/${binName} $out/bin
  '';
}
```

This file allows a user to invoke `nix-build` in the folder it's located to run it as almost a script. It ultimately runs the build command defined in `agda.nix` and puts it into a `result` symbolic link that points to the Nix store, where all Nix-evaluated expressions' results live.

There's also the file `shell.nix`, which is used with the command `nix-shell` instead. We use to expose a convenient command inside a shell environment that reuses the build artifacts in the current folder. This is unlike running `nix-build`, which always creates an isolated environment in which to build.

```
// shell.nix
{ pkgs ? import <nixpkgs> {} }:

let
  agda = import ./agda.nix { inherit pkgs; };
in pkgs.mkShell {
  nativeBuildInputs = agda.nativeBuildInputs;
  configurePhase = agda.commonConfigurePhase;
  shellHook = ''
    export build="${agda.buildCommand
      [ "Dynamical/Plot/src" "Dynamical/Matrix/src" ]}"
  '';
}
```

From within the Nix shell, we can run `$build` and get an Agda executable named `Plot` that runs our program.