[redacted]

CATEGORIES *in* CONTROL SYSTEMS SOFTWARE

TOWARD A UNIFIED THEORY *of* PROGRAMMING & CONTROL

by TIMOTHY A.V. TEATRO

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Doctor of Philosophy *in*
Electrical and Computer Engineering

Department *of* Electrical,
Computer & Software Engineering

University of Ontario
INSTITUTE OF TECHNOLOGY

Oshawa, Ontario, Canada
December 2023

THESIS EXAMINATION INFORMATION

Submitted By: **Timothy A. V. Teatro**

Doctor of Philosophy *in* **Electrical and Computer Engineering**

Thesis title:

Categories In Control Systems Software:
Toward a Unified Theory of Programming & Control

An oral defense of this thesis took place on December 8, 2023 in front of the following examining committee.

Examining Committee:

Chair of Examining Committee	Dr. Haoxiang Lang
Research Supervisor	Dr. J. Mikael Eklund
Co-research supervisor	Dr. Ruth Milman
Examining Committee Member	Dr. Lixuan Lu
Examining Committee Member	Dr. Ramiro Liscano
University Examiner	Dr. Ken Pu
External Examiner	Dr. Fabrice Colin, Laurentian University

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

ABSTRACT

Category theory is applied to the design and modeling of control systems application software with emphasis on feedback control. The language of application is ISO standard C++17, though the design is abstract and can be gainfully applied in any language expressive enough to embed domain specific languages for event stream processing with sufficient structure. The design is derived in a category, ***Cpp***, of a subset of C++ programs where types are modelled as sets and programs/routines are modelled as functions. This gives a forgetful functor from ***Cpp*** to $**et**, the category of sets which, in theory, facilitates broader compatibility with theories of dynamical systems in concrete categories.

A library of abstract datatypes (`struct` templates) and natural transformations (parametrically polymorphic function templates) is developed to demonstrate that (1) ***Cpp*** carries a bicartesian closed structure and (2) this structure has representation as standard compliant code. The axioms of this structure are encoded as unit-tests. And from this structure we specialise "machines" in the sense of Goguen (or more generally, Arbib & Manes), which actualise in ***Cpp*** as Moore machines. These Moore machines are then used as a basic model for the I/S/O structure of a control program.

Categorical Moore machines can be cast in terms of algebra and coalgebra which give natural mechanism to the input-driven evolution of internal state of the control programs, and infinite records of behaviour. The internal language of that model is consonant with sufficiently structured domain specific event-stream processing languages. The core examples and a case study use Rx, but FRP is a stated ideal and avenue for future work for modeling of interconnected and hybrid systems with computer controlled components.

The architecture is applied in two examples: (1) a simulated spring-mass-damper system with PID-force control, where comparison is made to analytical results, and (2) NMPC path tracking of a mobile robot with obstacle avoidance through soft constraint.

Keywords: Category theory; Control systems software; Functional Programming; Software Engineering; Coalgebra

AUTHOR'S DECLARATION

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Tim Teatro

Timothy A. V. Teatro

STATEMENT *of* CONTRIBUTIONS

I hereby certify that I am the sole author of this thesis. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

Part of this thesis was published in [184] and [135] and coauthored with my supervisors.

ACKNOWLEDGEMENTS

First, I feel profoundly grateful to my supervisors, Professors J. Mikael Eklund and Ruth Milman, for their unwavering patience and faultless direction. From the bottom of my heart, thank you both.

I am thankful to the Natural Sciences and Engineering Research Council and Defence Research and Development Canada for their generous funding.

Thanks to Nelson Niu, Ivan Čukić, Stephen Blackheath, Ben Deane and Sy Brand for insightful and encouraging conversations. Also, thanks to Bartosz Milewski, David I. Spivak, Brendan Fong, Michael Barr, Charles Wells and Edward Kmett for creating a vast wellspring of educational material on the application of category theory to computer science and donating it freely to the world.

I want to express my deepest appreciation to my incredible wife, Lindsay. Your dedication, encouragement and faith sustained me to the end. I have been truly fortunate to have you by my side.

Miles and Julia, my precious children, your joyous presence brought light to my darkest moments. You are yet to young to understand how I was moved by your bitter-sweet longing when Daddy had to spend time with his papers and books.

To my parents, Vera and Bill, I am grateful your love and sacrifices to support my academic journey.

A special thanks to Wendy Orton for her expert editing and proofreading in the 11th hour.

A warm thanks to my kind friend and lab mate Munachiso (Chisom) Ilokah for his faith, encouragement and reliable perspective.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS & INITIALISMS

ACT	Applied Category Theory
AI	artificial intelligence
AKA	Also Known As
AMS	American Mathematical Society
API	Application Programming Interface
biCCC	Bicartesian Closed Category
C++11	ISO/IEC 14882:2011 (See reference [116].)
C++17	ISO/IEC 14882:2017, [158], compliant C++ (See reference [158].)
C++20	ISO/IEC 14882:2020 compliant CPP (See reference [171].)
C++23	ISO/IEC 14882:2023
CA.	ca. (Circa.)
CCC	Cartesian Closed Category
CCW	counter-clockwise
CF.	cf. (Confer.)
CPU	Central Processing Unit
CT	Category Theory
DCPO	Directed-Complete Partial Order
DSL	Domain Specific Language
FP	Functional Programming
FRP	Functional Reactive Programming
GFP	greatest fixpoint
GNU	GNU (A recursive acronym: Gnu 's Not Unix)
GoF	Gang *of* Four (Named in reference to the four authors of the book *Design Patterns: Elements of Reusable Object-Oriented Software* [69].)
I/O	input/output
I/S	input/state
I/S/O	input/state/output
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organisation *for* Standardisation
IVP	Initial Value Problem
JSON	JavaScript Object Notation
LFP	least fixpoint
LQR	Linear Quadratic Regulator

MCU	microcontroller unit
MPC	model predictive control
NMPC	Nonlinear Model Predictive Control
NRVO	Named Return Value Optimisation
OO	Object-Oriented
P	Proportional (Controller)
PD	Proportional-Derivative (Controller)
PI	Proportional-Integral (Controller)
PID	Proportional-Integral-Derivative (Controller)
POD	“plain ol’ data”
RAII	Resource Acquisition Is Initialization
RMSD	Root-Mean-Square Deviation
ROS	Robot Operating System
Rx	ReactiveX
SLOC	Software Lines of Code
SSE	Steady State Error
STL	Standard Template Library
TCO	Tail Call Optimization
UML	Unified Modeling Language
UMP	“universal mapping property”
YAML	Yet Another Markup Language is a terse, human-readable data-serialization language, similar in spirit (but not in syntax) to JSON.

PREFACE

SINCE STARTING MY DOCTORAL WORK I had become deeply interested in the questions of how to best design control software. With the help of my supervisors, I wrote my first Nonlinear Model Predictive Control (NMPC) program for an autonomous mobile robot in 2013 [125], and another for a nuclear reactor in 2015 [138]. That experience led me to design a general Object-Oriented (OO) architecture for NMPC software [151], but I never found that design satisfying. It was clunky and covered in human fingerprints. As a computational physicist, I see control systems as a very natural process of information exchange among components of a system. Of course, programming languages themselves are artificial structures, but then again, so is pure mathematics. Yet there is something of mathematics that comports with the natural order and transcends its human origins. Language is a relational structure: a way of assigning meaning to symbols. I saw no reason that a sufficiently expressive computer programming language should be unable to reflect systems theoretical models. There ought to be something of the natural order that could be better reflected in the design of control systems software.

Around the time I began asking these questions, I became aware of a circulation of literature among a cadre of computer scientists applying a branch of abstract mathematics, Category Theory (CT), to program structure. There, CT is typically used to lend theoretical support to the paradigm of Functional Programming (FP). FP leads the programmer to structure programs in terms of *expressions* instead of *statements*. This seemed pertinent because mathematics is an expression oriented language. Perhaps expression orientedness is a necessary condition to capture natural order—it makes intuitive sense to me. After all, every quantitative relationship I have ever studied was couched in sets and relations as its fundament. If CT enables one to resolve programs into mathematical expressions, it may be a key to the essential nature of control systems software.

With all of this profundity and mystery on my mind, one day I read this tweet from an American physicist and software engineer, who at that time, was working on autonomous aerial vehicles at Amazon Prime Air.

> **Dr. Brian Beckman**
> @lorentzframe
>
> *Controllers have type signature state ⟶ (state, action). They are thus instances of the state monad and dual to Bayesian filters.*
>
> https://twitter.com/lorentzframe/status/954786729617051648

Monads are a fairly ordinary structure in CT but are particularly key in FP. I was aware of the intriguing mathematical duality between Linear Quadratic Regulators and Kálmán filters, and I wondered if that was what Beckman was alluding to in his tweet. Could such a deep mathematical connection be so plainly reflected in program structure? If so, then the analysis supported by CT could shed light on the fundamental questions of software for computer controlled systems, and I had found the right tooling to approach the problem. This document and its mixture of theory and applications is an artefact of my ongoing grappling with those questions.

STYLE & NOTATION

MAY WE? SHALL I? I quote from the preface of Hindley and Seldin's book:

> ..., a note about 'we' in this book: 'we' will almost always mean the reader and the authors together, not the authors alone. –[102]

As this thesis has one author, I am left saying 'I'. On the best advice of cognitive neuroscientists and linguists, who study effective styles of written technical communication [132], I favour *classic style.* In so keeping, I do not shy away from self-references with the first-person pronoun.

MATHEMATICAL NOTATION The mathematical notation in this thesis is the result of many hours disjoining overlaps, accommodating and harmonising conventions, balancing costs/benefits and, anticipating confusion. This is to be expected when a work such as this sits astride so many bodies of academic literature: pure category theory, automata theory, theoretical computer science, control systems theory and engineering, software engineering and, dynamical systems theory. It is become such a behemoth task that it fills its own chapter, so I encourage the reader to use Appendix A as a companion.

TYPOGRAPHICAL STYLE The thesis is set in my own flavour of Robert Bringhurst's signature typographical style, exhibited in his venerable book [118], which has become known as the *Typographers' Bible.* I hope he would approve of my modifications, which are an attempt to harmonise somewhat with IEEE-style publication.

INTRODUCTION

1

> Mathematical reasoning does play an essential role in all areas of computer science which have developed or are developing from an art to a science. Where such reasoning plays little or no role in an area of computer science, that portion of our discipline is still in its infancy and needs the support of mathematical thinking if it is to mature.
>
> — Anthony Ralston *&* Mary Shaw
>
> *Is Computer Science Really That Unmathematical?* [43]

THIS THESIS is committed to the province of control systems engineering; but it trespasses the borders of several academic disciplines. It borrows from the overlap in automata theory and formal methods in computer science unified under CT. A model of a control program is presented in a category of a subset of standards compliant[1] C++17 programs and implemented using an ReactiveX (Rx) pipeline. The model is both algebraic [52, 55] and coalgebraic [170]. The primary contribution of the thesis is conceptual. It is about deepening our understanding of control systems application software by mathematical reasoning and using the resulting model to inform software design. The thesis does not lead to difficult new theorems—its merit lies in the fact that it organises a mass of fine detail that is usually left open to interpretation of the programmer who has been handed a mathematical specification from a controls engineer and left to implement it. This organisation takes the form of an algebraic structure that should, one day soon, prove susceptible to formal verification and novel analysis techniques in composite with continuous-time dynamical systems coming from the literature of Applied Category Theory

[1] ISO/IEC 14882:2017, [158].

(ACT). The abstract nature of the model means that it can be grafted into a variety of programming languages. This gives flexibility to the engineer in his praxis, while also giving theoretical purchase to the computer scientist wielding languages for formal proof.

A control program runs on a computer connected to a machine forming a composite dynamical system. The goal of the program is to generate instructions for the computer to intervene in the natural dynamics of the machine in order to drive or constrain its behaviour. Common examples include

- thermostats for controlling a room's temperature,
- electronic speed controllers which manage turn-rate of a rotor in an electric motor
- automatic pilot for driving autonomous vehicles,
- chemical reactors which modulate chemical concentrations, agitation and temperature of a live chemical reaction,
- distillation columns which fractionate mixtures of liquids with different volatilities,

and so on.

So how does CT relate to programming? Categories consist of a collection (read *set*) of objects and collections of directed arrows between the objects, with an associative, unital composition operator. If you can define a *type of thing* such that we can collect all the things in a set and draw some sort of arrows representing a structured relationship among the things, then you might be able to create a category of the things. But the focus is on the arrows. The objects are viewed as mere points with no internal structure. This means the category theorist is a sophisticated sort of astronomer, deriving meaning from constellations (which sounds suspiciously like astrology, so let us not stress the analogy). Instead of peering into the objects with a microscope, the CT-orist gazes outward with a telescope to identify objects by their relationships to all other objects. The internal properties of the objects are not erased, but are reflected as a unique existence of certain arrows within constellations. If we indulge

A set is a gathering together into a whole of definite, distinct objects of our perception or of our thought–which are called elements of the set.

— Georg Cantor

Translated from *Beiträge zur Begründung der transfiniten Mengenlehre* in a little philosophical reflection, we might phrase the previous idea as "a thing is not given by its private internal properties, but in how it relates to all other such things". This leads to a notion of "universal things"—objects and arrows that fulfill existence and uniqueness properties that make them distinguished parts of these constellations (commutative diagrams) which then serve as "universal properties".

Returning to the point, theoretical computer scientists can express programs in categories with "types" as objects and programs as arrows transforming input to output. This yielding a algebraic, compositional, approach to programming. Programs can be *derived*, manipulated and simplified with all assumptions, identities and axioms made explicit and testable, and implementation is a matter of detail, forming types and functions which conform to the semantics. But more than this, CT gives a unique view into programming because it directs the programmer's mind away from operation and toward expression. That is, the invisible internal structure of a type determines the sensible operations (programs) that operate on it. If we know what we want to capture about the internals, we can figure out how that manifests as a commutative diagram of programs from which we can formally judge a "best fit".

In the present thesis, we will define a platonic category ***Cpp*** of a subset of C++17 programs where objects are types and arrows are programs. It is therefore trivial to identify programs with **Set**-functions and types with sets, so we have a forgetful functor ***Cpp*** → **Set**. In our case, a control program will be built from "universal" arrows arising from functor-algebras and -coalgebras capturing notions of input, state, output, behaviour and co/recursion. These will exist simultaneously in ***Cpp***, where it takes meaning as a C++ encoded program, and in **Set** where it takes meaning as a dynamical system. Of course, there is work involved in demonstrating that the structure in ***Cpp*** has some relationship to ISO C++17 compliant code. In order to build a universal control program within ***Cpp***, ***Cpp*** must be shown to admit the structure of a Bicartesian Closed Category (biccc).

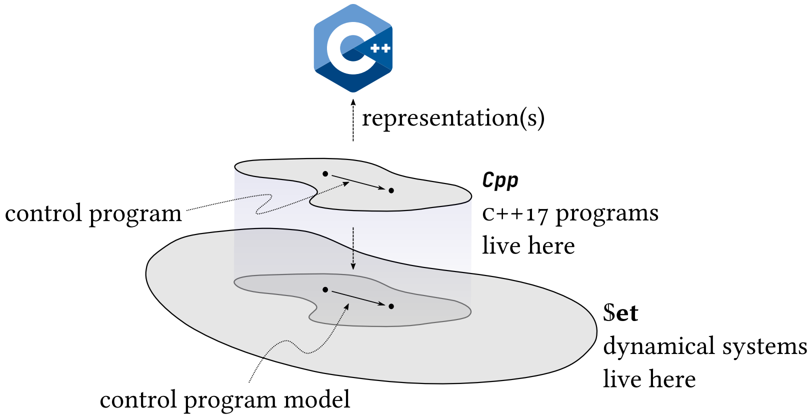


FIGURE 1.1: The category ***Cpp*** is at the centre of the illustrated relationships, and contains a subset of C++17 programs including control programs. A forgetful functor (translucent blue gradient) injects ***Cpp*** into the category of sets. The internal structure of ***Cpp*** has (non-unique) representations as actual ISO C++17 compliant code.

This means it has finite products, finite coproducts, initial and terminal objects, and exponential objects. A particular representation of the biccc structure is developed in CH. 4 giving a programming model collectively referred to as ***Cpp***/C++. These relationships are illustrated in FIG. 1.1.

Why Category Theory?

Category theory is infamous, even among mathematicians, for being abstract. For practitioners of software and control systems engineering, there is a deep allure—there are many aspects of our practice that remain unjustified on any formal basis, but nonetheless *feel true* to experienced minds. CT offers not only to ground these truths and bring justification to our day-to-day work, but to make control software specification much more explicit.

Any structure we define in CT will necessarily entail both objects *and* the arrows. Thus begins the marriage of structure and behaviour. In set theory, we can define the Cartesian product of two sets, $A \times B$, in terms of the elements of A and B: $A \times B = \{ (a, b) \mid a \in A \text{ and } b \in B \}$. Very little imagination is required to find the projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ that take $(a, b) \mapsto a$ and $(a, b) \mapsto b$ respectively. But the fundamental thing from the point of view of set theory is the elements that make up $A \times B$. We can derive the Cartesian product using CT (see §B.9.1) where we obtain the projections as part of the derivation. The CT-ical derivation captures the fundamental notion of what it means for a single set to reflect the relationship *A and B*. Because CT is blind to the inner-life of its objects, it cannot uniquely define a product. We only get the Cartesian product *up to isomorphism* So there is a trade-off. What do we gain for that trade-off? Generality!

Categorical thinking gives us a recipe for finding products in whatever category we wish. Products in ***Cpp*** might be represented as `std::pair` with `std::get` as projections (or members `.fst`/`.snd`), or simply `struct {A a; B b}` with member access for projections. Generality allows us to unite concepts across fields that are often nontrivial and not at all obvious. And because arrows are necessary parts of categorical questions and answers, we get a fusion of structure and behaviour.

That is why ACT is an attractive and growing area of research. Two of the most active sub-areas are dynamical systems and computer science. This thesis collects ideas from both and applies them to the practice of programming control application software with a view toward its modeling and design.

1.1 CONTROL PROGRAMS

Control application software may be a composite of interacting control programs. What I have in mind when I say *control program*, is a piece of software running on the control computer that intervenes in the time-evolution of a targeted dynamical system. This is achieved by computing a series of commands to be carried out by attached electromechanical devices. The targeted dynamical system, called *the plant*, usually progresses along continuous time governed by laws of physics. The whole arrangement is called *feedback control* if the computation of the control program is informed by data from a measurement apparatus. I aim at feedback control programs where non-feedback control is just a special case. The goal of the program is to manipulate the behaviour of the plant. Plants are often complex assemblies where subcomponents can be individually controlled, leading to a hierarchy of control.

An introductory text on digital control may illustrate a controlled system as a block diagram such as FIG. 1.2. In this figure, a plant is the target of some control effort exerted by a set of actuators integrated into the plant. The essence of the digital control problem is to engineer the dynamics of the controller (hardware and software) to ensure that the plant *behaviour*[2] conforms to some constraints on the set of all possible behaviours.

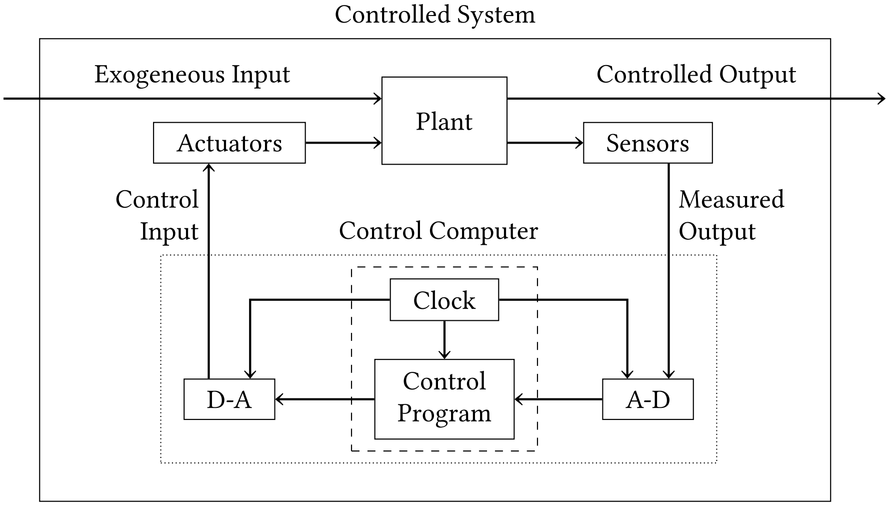


FIGURE 1.2: A block diagram illustrating a general architecture of a computer controlled system. Since some sensors and actuators contain their A–D and D–A converters, the boundaries of the control computer may not be clear (two posibilities are drown with dashed and dotted lines).

[2] The term *behaviour* has technical meaning, but for now, the colloquial use gives the right idea.

Mathematical modelling is a core part of the control system design process. Perhaps more than any other engineering discipline, control systems engineering is rooted in physics, dynamical systems and analysis. All too often this is at odds with the fact that a majority of controllers are implemented as programs running on digital devices. Despite the great care taken to model a given plant, the decision is almost universally made to merely identify the software component with the engineered control law, ignoring the dynamics of the program itself.

While introducing the vision of *Software Enabled Control*, Helen Gill and John Bay emphasize this point with admonition:

> … we must avoid the temptation to think of software as simply the language of the implementation. Control code … is a dynamic system. It has an internal state, responds to inputs, and produces outputs. It has time scales, transients, and saturation points. It can also be adaptive and distributed. … if we take this software dynamic system and couple it to the plant dynamics through the sensor and actuator dynamics, we have a composite system whose properties cannot be decided from the subsystems in isolation. [92, p. 4]

That was in 2003, and progress has been slow because there is a fundamental divide between software engineering and its siblings: software has no fundamental theory. Electrical and mechanical devices are modelled in physics and analysed in the language of mathematics providing a foundation for a formal design process. Ordinary computer code lacks algebraic structure and is correspondingly resistant to analysis.

In elementary algebra if it is written that $a = b$ and later that $a = c$ then we can infer that $b = c$. In a program:

```
int a = b;
// …
a = c;
if (b == c)
  printf("What a coincidence!");
```

there is no reason to expect $b = c$ because in C++, the equals (=) operator (almost always[3]) means assignment. The algebraic version enables clear reasoning while the assignment is a statement of action and introduces an discrete concept of time. Each assignment pushes program's execution, a tick marking a split of before and after. It is important to attend to this tick and behold it for what it is: dynamics.

1.2 GOALS *&* CONTRIBUTIONS

The main goals are

- a deeper understanding of control systems software that organises the fine detail of these programs in a way that eases specification and implementation,
- a mathematically motivated design of a control program written in C++,
- a model of a control program that bears interpretation in the same categorical setting as ordinary dynamical systems, and
- demonstration of the above by means of simulation and by means of implementation on a real electromechanical device designed for a non-academic purpose.

I feel each has been achieved, and that will be demonstrated in the pages to come. I hope that the reader will benefit from these contributions and find them useful to their own work. In the end, my only real goal is to be of use to the community of academics and practitioners who strive for a constant renewal of their understanding of the world around them.

1.3 OVERVIEW *of* THE THESIS

I have attempted to organise the thesis to demarcate pre-existing theory and my contribution. I have been told that my collection, organisation,

[3] In the special case of initialisation, an equals sign can be used to initialise without assignment, but this is an unimportant technicality.

adaptation and exposition of the pre-existing theory across domains is a contribution in and of itself. Nonetheless, Chapters 2 and 3 are from my reading of the literature while Chapters 4 and 5 are contributory descriptions of my research. Here is a synopsis of each chapter.

1.3.1 Chapter Synopses

CHAPTER 2 provides the reader with a review of relevant literature written in a non-mathematical and more historical voice. It tells a story of how many of the mathematical and computer scientific concepts I rely on arose as a cross-disciplinary collaborative enterprise. Topics include the rapprochement of automata theory and control, the categorification of automata and systems theory, functional programming, applied category theory and polynomial functors providing a general theory of interaction.

CHAPTER 3 If Chapter 2 is history then Chapter 3 is archaeology. This section excavates the key mathematical tools from the literature that will later be used to form the fundament of the thesis. Key topics include: categorical I/S/O systems, Co/algebra and fixpoints, snoc-lists, and introduction to Rx and the iterator/observer duality and Moore machines as systems in the theory of universal coalgebra.

CHAPTER 4 presents ***Cpp*** and its bicartesian closed structure. This chapter is where the mathematics meets the code. To demonstrate biccc structure, a support library is written to demonstrate that praxis tracks with theory and limitations are discussed. This all justifies the use of C++17 code aimed at modelling a category theoretical model of a program. Each structure and relationship in the biccc is given a representation in code, and each axiom turned into a unit test.

CHAPTER 5 draws the lines from a categorical model of Moore machines to a general design for control programs in C++17, and its coalgebraic systems model. A control program is reduced to an expression in an algebra of event streams that simultaneously has meaning as a Goguen or Moore machine in ***Cpp*** and implementation in Rx. The chapter finishes with an example of application to a classic Proportional-Integral-Derivative (PID) control exercise where we can run discrete-time simulations for comparison to continuum time analytical results.

CHAPTER 6 presents a case study of the Bishop robot developed as part of a multi-disciplinary project for the Ministry of Defence through a multi-university and industry partnership. Prof. Eklund's team was responsible for navigation and tracking systems in the robot which used `catnav`, an implementation of my architecture for Robot Operating System (ROS). The chapter closes with an anecdotal account of the east of development and extension of `catnav` over the 3-year duration of the project.

2 BACKGROUND LITERATURE

We cannot solve the problems we have created with the same thinking that created them.
— Unknown
(*Possibly* Ram Daas but commonly misattributed to Einstein)

The theory presented in this dissertation applies concepts from several major and convergent lines of research collaboration in mathematics, automata theory, logic and computer science. This chapter is intended to deliver a broad and historical view of the developments, describing some of the motivations and contributions in the context of their time. I also describe how the developments relate to the present thesis. The next chapter, *Theoretical Preliminaries*, will detail the mathematical concepts, extracted from this literature and streamlined for the thesis.

Though I developed a lot of my ideas independently, I came to learn late in my PhD research that, to my delight, I was in agreement with a great history of academic development. Moreover, that development came at the hands of skilled mathematicians who proved a great many lemmas and theorems. Consequently, I could see the way toward my goal of bringing categorical insight to the engineering of control systems software.

2.1 INTRODUCTION

In the early 1960s, analogies between the theories of automata and control had been drawn. It was a particularly opportune moment in history to make that leap: most of automata theory focused on understanding digital computers and language; and computer controlled systems were emerging as prominent in research and practice of controls engineering. (The space race was a strong driving force here, with the need to computerise space guidance and navigation.)

In 1965–66, Michael Arbib gives a mathematical rapprochement of automata theory and control [5, 10].mm[-61] (The details also appear in Part III

[-61] Michael Arbib (1940–) is a prominent computational neuroscientist. When he began

of the 1969 textbook *Topics in mathematical system theory* [20] coauthored by Rudolf Kálmán, Peter Falb and Arbib.) In 1968, Arbib would begin to conceptualise a general class of arbitrary sequential "input processing" systems within the context of category theory, wherein standard algebraic automata and finite automata are special cases. The resulting model harmonised well with parallel developments in theoretical computer science, which is why they are of particular importance to the thesis.

This section traces this history, extending to more recent generalisations and their place in computer science and *functional programming*. The story to unfold in the sections to follow.

2.2 CATEGORICAL SYSTEMS & CONTROL

2.2.1 The First Automata in Categories

In the year between 1965–66, at least two branches of the U.S. Military funded research in the area of formal language and automata theory at the University of Michigan, led by Burks [11]. In particular, they investigated the application of logic and mathematics to computing automata. A PhD student at the time, Yehoshafat Give'on studied the structure of categories of abstract automata which were disseminated in a series of technical reports, [6, 7, 12], the later becoming Give'on's PhD thesis. These are perhaps the earliest literature demonstrating the application of category theory to automata.

Give'on, building on the contemporary view of finite automata as monoidal algebras, studied a category he denoted $\mathbf{Set}^W$ of transition systems over a monoid, W, of inputs. Give'on gives a fairly thorough account of structure in that category, which is all very revealing of the intricate alge-

his undergraduate work he had a deep appreciation for the beauty of pure mathematics, and consequently intended to study it. He was encouraged to read Norbert Wiener's *Cybernetics*, causing a chain reaction of literary exploration of cybernetics, automata, theoretical foundations of computer science, information theory, neural networks, neuropsychology and ultimately neuroscience.

braic relationships, such as the isomorphism between the automorphism group of W and the semigroup of W acting on itself. This first categorisation of automata is an historically critical starting point but is very specific to automata viewed as monoidal algebras in the category of sets—which is a point of view that is eventually abstracted away.

NOTE. Around this time, Günter Hotz, a German mathematician, was writing about automata using *string diagrams* in $\mathbb{F}\mathbf{inSet}$ with a Cartesian monoidal structure [8, 9]. These papers are in German (except for the translated abstracts). Qualitatively it resembles later work on general notions of interconnectivity and interaction, CF. [139].

2.2.2 Rapprochement *of* Automata *&* Control

Starting around the mid 1960s one can find various suggestions and tendencies in literature to notice connections between the theories of automata and control. One of the early and most direct efforts toward a formal rapprochement was penned by Arbib in 1964 with “A common framework for automata theory and control theory,” [5]. (See also [10].) Leaving aside the fact that the motivations and methods of automata and control theories are different—automata theory leans on combinatorial techniques while control theory is based in analysis and optimisation—both rest upon on an input/state/output (I/S/O) apparatus that processes matter, energy and/or information.[1] There are clear analogies in the notions of behaviour and realization for both theories. Arbib’s thesis was based on an analysis of Kálmán’s paradigm shifting results for minimal realisation and duality of observability and controllability in state-based linear systems theory, [20, PT. I & IV].

Then in 1969, during their mutual occupation at MIT, Arbib and Zeiger deepened Arbib’s rapprochement [18]. They,

[1] It was not unnoticed in the literature that computers and languages could also be encompassed by such a view.

> … present the startling thesis that linearity has little to do with linear machines, and that certain peculiarities of the category of real vector spaces have acted to obscure the conventional view of the way linear machines run!

The key result in this paper is the startling generalisation of Kálmán duality to what is now called *Arbib-Zeiger duality* in sequential machines.

It is important to note at this point (the late 1960s) that no formal attempt has been made by Arbib to apply CT to automata or systems. But it was clear to him that that was the approach he had to take [161].

2.2.3 Lawvere Algebraic Automata

In the late 1960s, the most abstract approaches to automata theory are set in the algebra of modules, monoids, semigroups, actions on state spaces, and so on. In the 1967 paper, [13], Eilenberg and Wright[2] use Lawvere's categorisation of universal algebra [4] as the setting to describe the semigroup and monoid (action) pictures of automata. Lawvere theories are categories that capture the *platonic idea* of an algebraic structure. Take the theory $\boldsymbol{T}$ of semigroups or monoids or such, then a model of that algebraic structure internal to a category $\boldsymbol{C}$ as a product preserving (Cartesian) functor $\boldsymbol{T} \to \boldsymbol{C}$. The idea here is that the functor takes formal expressions in the theory (morphisms in $\boldsymbol{T}$) to their *meaning* (morphisms in $\boldsymbol{C}$). A language theorist might say that this is a relationship taking syntax to semantics. This is a powerful generalisation of automata that subtends a broader class of I/S/O systems.

[2] Eilenberg was Lawvere's PhD supervisor, and admittedly did not read Lawvere's thesis until years later, [99, §2]. I presume [13] resulted once he did.

2.2.4 Arbib and categories

In 1968, Arbib and Give'on would cross paths at Stanford as research associates of Kálmán.[3] They collaborated on twin papers, [15, 16], which further develop Give'on's categorical view of automata and Arbib's consolidation of automata theory and control theory. The first paper provides a theory of algebras regarded as automata—in prelude for its younger twin. It takes a view that is simultaneously computer scientific and universal-algebraic. Algebra automata are seen not as stateful processors of input sequences but as processors of trees—a set of rules for processing an instruction set viewed as something like a syntax tree with leaves of data. The second builds on that groundwork, recapitulating Eilenberg and Wright's paper, specifying it to encompass a notion of algebra automata that could be viewed as symbolic dynamics of systems.

2.2.5 Goguen's Machines in a Category

By the late 1970s, Arbib had sketched a "general theory of machines in a category" with automata resulting when construction is carried out in $\mathbb{S}\mathbf{et}$, and linear systems when construction is in $\mathbb{V}\mathbf{ect}_{\mathbb{C}}$. In [18], Arbib and Zeiger avoid identifying their methods as categorical, they promise a forthcoming paper *The Category Theory of System Realization*, which did not materialize as such. At that time, Arbib's knowledge of CT what insufficient to complete his sketch [161, §4.1]. Arbib discussed his sketch with Joseph Goguen who located Arbib's missing pieces in the concept of *adjoint functors*.

NOTE. Arbib and Manes later wrote a paper in 1974, "Foundations of system theory" [27] that deliberately and carefully draws the generalising line from Kálmán's module theoretic approach using the full power of category theory. I suspect it is the paper Arbib and Zeiger would like to

[3] Arbib had first met Give'on in 1963, when he visited the University of Michigan [161].

have written, except that they intentionally made their paper accessible to control theorists with little algebraic background beyond matrices and vectors.

Goguen borrowed inspiration from Arbib and in 1971[4] submitted an article, [33], [5] detailing the role of adjunctions in a categorical realisation theory of machines. Section 3.6 is largely a modernisation of the core of Goguen's categorical machine.

Aside from categorifying the definition of (a subclass of) automata, the key contribution of Goguen's paper is a theory of realisation delivered by a full categorification of "behaviour." By this, the definition of reachability is extended, but not observability or controllability. This is perhaps a missed opportunity.

Because Goguen's machines are based on tensor algebras, their scope is limited. They can not, for example, encompass linear systems or other more complex input structures because the tensor functor is not going to carry these models. So Arbib's intuitions for an abstract sequential systems as "machines in a category" had not been fully answered by Goguen.

2.2.6 General Machines in a Category

Sometime CA. 1971 at the University of Massachusetts, Arbib meets a gifted young algebraist named Ernest Manes who received a PhD in 1968 for his work on monadic algebras [14]. In [161], Arbib writes

> Ernie [Manes] had what I had been missing in my search for a theory of machines in a category, and it was given by his thesis work....

In April 1972, Manes presented a preliminary report on his work with Arbib at the 693rd meeting of the American Mathematical Society (AMS)

[4] The paper would not be published until 1975.

[5] Trivia: although Goguen gives Arbib ample credit for his intellectual contribution in [33], Arbib had imagined a collaborative effort and was somewhat disappointed when Goguen published independently.

titled *Machines in a Category* [23]. The details would come to print two years later [28]. Their approach was to generalise the tensor algebra of Goguen's approach to any endofunctor that has free-forgetful adjunction and an initial algebra. In that case, the endofunctor is called an *input process* (later, *recursion processes* [40] or *varietors* [45]). It is not common nomenclature, but for convenience I will call Arbib and Manes' model of automata *Arbib-Manes machines*. Arbib-Manes machines are demonstrated to capture input processes of several kinds: linear systems, stochastic systems, algebra automata and tree automata which are all demonstrated in the paper. And for all these situations, we have notions of reachability, observability and realization appropriate to the context of their domains. Nerode's algorithm is further generalised to suit the abstraction.

Goguen's automata can be seen as the special case of Arbib-Manes machines when the input process is a tensor product of state and input objects.

It was shown in [37] that the definition of Arbib-Manes machines is not "too wide" in the sense that the conditions of an endofunctor being an input process is sufficient to expect that minimal realizations of input/output (i/o)-relationships exist.

In later work, Arbib and Manes study input processes of "state-behaviour machines" where the free-forgetful adjunction of the input process has a cofree right adjoint: a free-forgetful-cofree adjunction [31]. (They did not use this word, but they were dealing in *coalgebra*. Cofree coalgebra, actually.)

Around 1975, a group of Czech mathematicians began applying insights from pure category theory to Arbib and Manes' work. The group was originally led by Věra Trnková and consisted of her graduate students Jirí Adámek, Václav Koubek and Jan Reiterman. The thread of research started with [25, 36] as a followup to [30], re-framing Arbib-Manes machines in slightly more general mathematical terms of *free algebra*, giving a construction algorithm with existence conditions and relating them to monads through the work of Michael Barr [21].[6] In the mathematical literature, there is a distinction drawn between "algebra in the sense of Lambek [7]" and "algebra in the sense of Eilenberg-Moore", the monadic flavour being the later. Eilenberg-Moore co/algebras are a full sub-category of the category of co/algebras of an endofunctor [66]. Input processes are predicated on free-forgetful adjunction and all adjunctions induce a monad [81, §VI.1 (p. 138)].

2.2-1 Goguen machines are a special case where the free-functor is specifically the *free monoid functor* which induces a very important monad for computer scientists: the *list monad*.

Adámek's doctoral (RNDr) thesis was on categorical automata theory and its relationship to universal algebra. The trajectory of Adámek's academic career would become interwoven along the seam between pure category theory and computer science. His most notable collaborators include theoretical computer scientists Stefan Milius and Lawrence S. Moss. His body of research is exemplary of the close relationship between categorical systems theory, automata theory and computer science. Fixpoints of functors are fundamental to data and codata types, but are also at the root of categorical systems models. This is because they systematise induction and coinduction within a category. Because input processes have initial algebras, and initial algebras have invertible structure maps, those initial algebras are also coalgebras. (Not necessarily *terminal coalgebras!*)

Toward a general theory of fixpoints, Adámek and collaborators produce a series of literature on the topic: [38, 39, 70, 140]; and ultimately, the modern survey [160].

[6] *Trivia:* Michael Barr (1937–) is the Peter Redpath Emeritus Professor of Pure Mathematics at McGill University. He coauthored [82] with Charles Wells, which is on my list of recommended sources for CT. My Canadian pride compels me to mention his place of tenure.

[7] *Trivia:* Joachim Lambek (1922–2014) was a Canadian mathematician and the Peter Redpath Emeritus Professor of Pure Mathematics at McGill University. He was the first to study fixpoints of functors, and I will reference his famous lemma several times in the thesis. This is yet another indulgence of Canadian pride.

2.2-2 The paper [39] deserves special mention as it is the first to give a canonical construction of the least fixpoint of an endofunctor which is now a fundamental tool in algebraic data structures. Adámek brilliantly extends the Kleene fixed-point theorem which, by the ascending Kleene chain starting from the bottom element, gives the least fixpoint of a (well behaved) function. Adámek had the intuition of constructing analogous chains for endofunctors on categories, starting from an initial object and yielding a least fixpoint as the colimit of the chain. (See REM. 3.7-4.) The colimit construction provides an algebra structure and thus fixpoints are initial algebras (see §3.7).

Least fixpoints are in the cluster of ideas including terms like free algebras, initial algebras, induction iteration and recursion. The dual cluster is greatest fixpoints, cofree coalgebras, terminal coalgebras, coinduction, coiteration and corecursion. Both play a part in observable sequential processes and studied by Adámek an colleagues: [37, 41, 44, 45, 55].

2.2-3 Dually to **2.2-2**, in 1993, Barr flips the arrows around and gives a construction of the greatest fixpoint as the limit of a descending chain from the terminal object [66]. This gives terminal coalgebras.

2.2.7 Systems & Coalgebra

Coalgebra seems to be the mathematics of computational dynamics. As a mathematical topic,

> ... is still in its infancy but promises a perspective on unifying, say, the theory of differential equations with automata and process theory and with biological and quantum computing, by providing and appropriate semantical basis with associated logic. — Jacobs [159]

This is of course, a very broad claim! But it is born out in the modern literature of applied category theory. (Visit the Topos Institute website [201] and see their colloquia [200] for a stunning variety of applications—all of them unified by CT.)

The term "universal coalgebra" seems to originate from Robert Davis, who produced a series of papers starting in 1970 with [22] and ending with [53] in 1986. Arbib and Manes include an example in their textbook on program semantics, in the section on fixpoints of functors [52, §10.2, EG. 18]. (In essence, the example is a simplification of Davis' developments, but seems to have come directly from their insight in [31].)

Research into universal coalgebra really built momentum when Dutch researcher Jan J. M. M. Rutten began a systematic investigation. His first publication on the topic was in 1995 [71], which lead to a number of subsequent studies too numerous to list here. It influenced the thinking of important researchers: Lawrence S. Moss [79, 83], Goguen [85], Grant Malcolm [74] and Bart Jacobs [73, 80, 87]. But the landmark publication on the topic is the venerable 2000 article "Universal coalgebra: a theory of systems" [88]. More recently, there is also the 2019 textbook by Rutten [170], *The Method of Coalgebra*, written just prior to his retirement. Excellent companions to these sources are the 2017 textbook, [159] by Jacobs and the survey paper [160] by Adámek *et al.* which usefully collects and rehearses key results.

Having not been very mathematically explicit in this chapter, let me provide some intuition. Initial algebras are things like term-algebras, finite trees, things that are built up from below: starting from a null or empty base and iteratively layering the structure a countable number of times. Construction of the natural numbers (or ordinals) are a routine example of an algebraic construction from a simple $\mathbf{Set}$-functor: $N \coloneqq X \mapsto 1 + X$. Folding over an algebra gives something like recursive or inductive condensation of a structure into an element. Final coalgebras are more like counting down from infinity, starting with a potentially infinite structure and deconstructing layers corecursively. Unfolding them from an element gives something like structures or records of infinite behaviour. The terminal coalgebra of N is the set of natural numbers, including infinity: $\mathbb{N} \sqcup \{\omega\}$.

NOTE (personal aside). A surprising amount of the literature in the development of a categorical theory of systems and automata was carried out by people interested in cybernetics and neurology. Kleene was one of the great logicians of the 20th century, but his early work in automata was a byproduct of his grappling to understand the animal nervous system [2]. Though Arbib's collaborative efforts are a central focus of this section, fundamental systems theory was only a small part of his career as a computational neuroscientist. Automata theory is also at the core of many models of language, both artificial and natural [46]. It is evident that categorical automata theory organises deep relationships that pervade models of information processing and interaction. Currently, David I. Spivak, Nelson Niu and colleagues in the ACT domain are relocating automata theory as a special case in a general theory of interaction [182]. This more general theory encompasses Bayesian learning and entropy, leading recently to some of humanity's most comprehensive models of the brain [189]. I admit that I have become intellectually preoccupied by the application of CT to models of brains and theories of mind. It appears to me that applied category theory is like a wonderful mental virus—a *meme* in the true sense of Richard Dawkins—that is utterly incurable and fortunately produces long term sequela.

2.3 FUNCTIONAL PROGRAMMING

> One of the things that's fun about functional programming is that theory and practice come particularly close together. So, often you find things that arise in a somewhat of a pointy-headed theoretical kind of way that actually have immediate practical utility.
>
> — Simon Peyton Jones
>
> *Compiling without Continuations*

Functional Programming (FP) is an open, active and fruitful area of research [61]. A concise way of describing FP is "programming with λ-calculus.” Lambda calculus was introduced by the logician Alonzo Church in the 1930s as an alternative structure for the foundations of mathematics—based on operations instead of sets. It formalises the process of building functions from variables and constants and expressing application, abstraction and reduction of terms. Λ-calculus and combinatory logic are two systems which serve us as (equivalent) abstract programming languages. An excellent introduction and reference companion to both is *Lambda-calculus and combinators* by J. Roger Hindley and Jonathan Seldin [102]. What is usually called λ-calculus is actually a collection of several formal systems with varying structures of types, terms and equations, so one tends to speak of “λ-calculi”.

In theoretical computer science, a language is often given (denotational) semantics by identifying *programs* with λ-terms.

Equivalences lie at the heart of any practical calculus.
— Edsger Dijkstra
Et al. [50]

Alternatively, instead of thinking of programs in terms of λ-calculi, we are free instead to interpret them equivalently as categories. As Steven Awody puts it:

> CCC $\sim$ λ-calculus. —[109, §6.6]

In words, *simply typed* λ-calculus (one of the λ-calculi) is the internal language of a CCC, meaning that an expression in a λ-calculus is equivalent to and expression in terms of the structure of a CCCs. (And *vise versa*!) So I prefer to look at FP as “programming with category theory.”

Less mathematically and more toward the keyboard, FP is a *paradigm* of programming standing in contrast to more common paradigms of OO and procedural programming. I make this point to emphasise that a programmer can base their entire career on writing functional programs without ever knowing how they are wielding CT. Functional languages are identifiable by the style they encourage through syntax, defaults. There are a variety of modern programming languages aimed at FP: Haskell, O’Caml, F$^{\sharp}$, Erlang, Clojure, Scala, to name a few. Python has *Coconut*[8] which is a superset syntax to facilitate FP in python. For an excellent his-

[8] `https://coconut-lang.org/`

tory of functional programming languages, see David Turner's survey [126][9]. Many popular language are adding features to facilitate functional style, a fact I will capitalise on in this thesis by attempting functional style in C++17.

This section will focus on literature in FP (and theoretical computer science more broadly) that are important to expressing categorical automata. This means we are concerned with structural recursion (where recursion is a result of canonical manipulations of data structures reflecting recursive construction) and corecursion, and manipulation of asynchronous streams.

For a more general discussion of FP, a classic introduction is John Hughes' "Why Functional Programming Matters" [56] and the more modern retrospective "How functional programming mattered" [137] by Hu *et al.* On programming with CT, I recommend Jan-Willem Buurlage's [162] or Bartosz Milewski's [169]. For a non-mathematical perspective on FP in C++, recommend Ivan Čukić's [163], which I reviewed for the publisher[10].

NOTE. In the 1960s and 70s, while researchers on the side of systems theory were on a campaign to understand the mathematical essence of sequential systems, the parallel campaign was being carried out as *algebraic semantics*, nucleated at the intersection of mathematical logic and computer science and language theory. The research crossed over at many points. Goguen himself was not a pure mathematician, but a computer scientist. My favourite example of this sort of crossover is when computer scientist Robin Milner rediscovered the work of logician J. Roger Hindley [19] to develop what is now called the *Hindley-Milner type system*, used in many functional languages (particularly those descended from ML).

[9] Turner masterfully presents the history at Lambda Days 2017, `https://youtu.be/QVwm9jlBTik?si=Y6n7QINOI9Y2hk97`.

[10] Which I note for transparency's sake.

2.3.1 Would That It Were Loopless

The basic idea of a *category of types* or *program category* is to have objects representing the data types of the language and arrows representing as programs as I/O relationships with input in the domain and output in the codomain. Because the arrows of a category have compositional properties, programming is viewed as composing the arrows of the category to write the program in terms of sub-programs (functions). In order to support FP, a program category should at least be Cartesian closed. (But we can ask for much more, at some peril.)

The categories $\mathbb{S}\mathbf{et}$ and $\mathbb{F}\mathbf{inSet}$ are both Cartesian closed, and furthermore bicartesian closed. They are appealing as programming categories because they are extremely well understood; but they are overly permissive (that is, lacking structure that ensures programs are sensible), particularly where general recursion is concerned. We are able to construct programs with the arrows of those categories that do not terminate. (One quickly finds mischief with the so-called *fixpoint combinator*, since not all $\mathbb{S}\mathbf{et}$-functions have fixpoints.) There are more sophisticated choices for program categories that infuse topology and lattice theory so that all programs are well formed. Dana Scott's *domain theory* [48] is a theory of types and computability that accommodates recursive definitions of both functions and of types. Types are lattices and functions are *Scott-continuous*, meaning they satisfy properties including monotonicity. In brief and rough terms, Scott continuity means that finite information about function values is determined by finite information about the inputs. Therefore we expect the output information volume to scale sensibly with the that of the input. This handles both issues of partial data and infinite data.

Categorical functional programmers derive data types such as lists or trees as (greatest or least) fixpoints of polynomial endofunctors in program categories. These constructions also appear to in categorical theories of automata as free-monoids [33] or more generally as input processes [42, 55]. Subject to their existence and other caveats, it is generally the case that least fixpoint constructions yield finite structures (such as lists and trees) while greatest fixpoints yield possibly infinite structures (such as streams or infinite trees).

In this thesis, they are used both ways: as semantics for data structures woven through control programs, and also seen as tools to analyse behaviour of the control program in a systems theoretic sense.

I have avoided being very mathematical in this chapter, but I suspect an example of what a data type as a fixpoint looks like, at this point, would be clarifying. In the thesis I make heavy use of the model of lists (of the *snoc* variety, growing from the right) as a fixpoint of the parameterised (family of) $\mathbb{S}\mathbf{et}$-endofunctors $\hat{A}_{\bullet} \coloneqq X \mapsto 1 + X \times A$. The least fixpoint (cf. [39]) of this functor is the smallest set (or type) X that satisfies the relation

$$\hat{A}_{\bullet}\, L_A \cong L_A. \tag{2.3.1}$$

The witness of (2.3.1) in forward is an algebra $\kappa : \hat{A}_{\bullet}\, L_A \to L_A$ called a *constructor* in computer science circles. This is makes some sense: expanding the signature gives $1 + L \times A \to L$ which can be read as:

> *Give me no data (*1*), and I can give you an (empty) list, or give me a list and an A-element and I'll give you a new list, with your element appended to the right.*

The inverse of (2.3.1) is is a coalgebra $\kappa^{-1} : L \to \hat{A}_{\bullet}\, L_A$ called a *destructor*, which expands as $L_A \to 1 + L_A \times A$ and read as

> *Give me an empty list and I'll give you no data (*1*). Give me a non-empty list and I'll give you a pair containing its head and tail.*

The smallest solution L_A is the set of all finite lists of As, which are naturally regarded as a sort of binary tree with κ on the nodes and A elements on the leaves. The terminating leaf is the empty list []. The conventional notation for a least fixpoint is the operator μ, so $\mu\hat{A}_{\bullet}$ is the solution to (2.3.1). The notation can be overloaded to mean either the set L_A, or the algebra (L_A, κ) when context permits. The set $L_A \coloneqq \mu\hat{A}_{\bullet}$ is isomorphic to the Kleene-closure A^*. This is the monoid at the basis of Goguen's machines, but for a general input process F, the fixpoint construction μF is how Arbib, Manes and Trnková/Adámek's group generalise to the broader class of automata [36]. The greatest solution to (2.3.1) is the superset of $\mu\hat{A}_\bullet$ that includes infinite lists: $A^* + \{A^\omega\}$. This is denoted with the greatest fixpoint operator as $\nu\hat{A}_\bullet$.

Fixpoint solutions become most useful in the category theoretical context because of initiality/terminality, when the fixpoint is either an initial object $(\mu F, \kappa_F) \in F$-**Alg** or a terminal object $(\nu F, \kappa_F^{-1}) \in F$-**coAlg** (or simultaneously both, which happens in some categories). The unique morphisms from/to these structures provides a rigorous notion of co/recursion or co/induction [170]. The unique morphisms from μF are called *catamorphisms* generally, and are often called *fold*s in functional programming. Dually, the unique morphisms to νF are called anamorphisms generally and are often called *unfold*s in functional programming. This has a relationship to free-forgetful adjunction. Under the right conditions, free F-algebras are also initial, and the dual statement for forgetful-cofree adjunction and coalgebras. Bear in mind that all of this is subject to the existence and other sufficiencies of rigour which are not taken for granted.

Scott's domain theory was largely designed to make rigorous the construction of these sorts of types based on recursive construction in ways that guarantees program termination. Michael B. Smyth[11] and Gordon D. Plotkin categorify the construction in Scott domains [49]. This bridges the functional semantics of dependent types with the pure category theoretical construction. Because codata types (those based on terminal coalgebras) potentially contain infinite values, care must be taken within set-theoretic semantics. Barr shows that the existence of terminal coalgebras for a $\mathbf{Set}$ endofunctor in well-founded set theory [66], and that the terminal coalgebra is the Cauchy-completion of the initial algebra of the functor [66, THM. 3.2]. These arguments require constraints that do not work well for

[11] *Trivia:* Smyth's doctoral advisor was Dana Scott.

computer scientific ends. Barr then points out that, although he cannot prove it generally, specifically polynomial functors and others that arise in practice will satisfy the properties. (Polynomial functors are the key for this thesis.) As an example, he specifically uses the functor $X \mapsto 1 + A \times X$ (isomorphic to $\hat{A}_\bullet$ by the associator of a ccc.)

An issue in domain theory with particular relevance to the thesis is that for many conceptions of domains, bicccs are logically at odds with the structure required for lawful general recursion [54, §3]. Perhaps unfortunately for me, I intend to define control programs in a biccc of c++ programs, ***Cpp***, using recursively defined types!

It's not wise to violate rules until you know how to observe them
— T.S. Eliot

Learn the rules like a pro so you can break them like an artist.
— Pablo Picasso

2.3-1 So how will I justify it? If I were to press the issue, I suspect I would find success in Gunter's category of profinite domains [51]. This level of analysis is not quite right for the present thesis. The choice of program domain amounts to a type system of a language, and the only type system I aim to satisfy is c++'s which is not based on a grand or beautiful theory, and is itself overly permissive, just like $**et**. All I want is for ***Cpp*** to faithfully map to $**et** and have representations in c++17 that comport with the c++ type checker.[12] Co/recursive data types built from polynomial endofunctors in biccs comes from the landmark paper of Arbib and Manes, "Parametrized data types do not need highly constrained parameters" [47]. To further justify that my construction of control programs are valid in a more practical sense, I will invoke the idiom that "fast and loose reasoning is morally correct". This phrase comes from the 2006 paper, another landmark, of Nils Danielsson, John Hughes, Patrik Jansson and Jeremy Gibbons titled after the phrase [97]. In that paper, the authors define a language with two different denotational semantics: one set-theoretic (total) and the other domain-theoretic (partial). They establish a partial equivalence relation between the two showing that if two

[12] And beyond the type checker, c++17 does not support general recursion since it does not enforce Tail Call Optimization (tco): recursive functions are always in danger of overflowing the stack.

closed terms have the same semantics in the total language then they have related semantics in the partial language. A biccc type category follows, where they disallow all fixpoints except well-behaved ones; those that can be expressed as folds and unfolds.[13] This approach serves us well in the construction of control programs, as I shall define them almost entirely in those terms. Interpreting the conclusions of that paper in the context of c++ is more complex since c++ is not a lazy language (which has implications for infinite data) nor does it contain a formal "undefined value", $\perp$. (Later, in ch. 4, I do approximate an initial object type, `Never`, serving as the unit of the coproduct in ***Cpp***, which leads to the same sort of misery when equational reasoning about recursion.) The point remains that if we are only interested in processing finite and total values then the control programs we derive in ***Cpp*** should be sensible, terminating programs.

Grant Malcolm's PhD work [57, 58] shows that some very important properties and laws related to folds and unfolds that come for free, for any data type, when one defines a data type to be an initial algebra, or dually, a terminal coalgebra. His PhD thesis was my introduction to the concept of these types and operations. The more common reference is Erik Meijer's *et al.*[14] whimsically titled "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire" [62], but as an introduction I found it too terse and abstract.

2.3.2 Asynchronous Lists

When one iterates through an array or list using a loop, they tacitly presume the next value is waiting in memory to be used; this is called *pull-based access* because the data is figuratively pulled from memory at the behest of the consuming code. When data are distributed not in memory,

[13] Alternatively, we could disallow recursion on coproducts, limiting us to monomial recursion, but this is incompatible with the aims of the thesis.

[14] This is the same Meijer that invented the Rx interface to observable streams that is used later, and described in §3.10.

but in *time*, what should the interface to the consumer look like? A duality lurks here: *push* vs. *pull*. If one inverts the flow, the calculation does not pull at awaiting data, but instead, data is fed or *pushed* to an awaiting calculation.

If we manipulate lists as a whole, with operations that abstract away element-wise processing then we obtain expressions as a sort of pipeline for data manipulation. For example, in **Set** let

$$a \coloneqq [\,1,\ 2,\ 3\,] :\ A^* \quad \text{and} \quad f \coloneqq x \mapsto x^2 :\ \mathbb{Z} \to \mathbb{Z}.$$

Then we can express the element-wise application of f to a using the functor part of the list-monad (mentioned in passing in **2.2-1**) to lift f to $f^* :\ \mathbb{Z}^* \to \mathbb{Z}^*$ so

$$f^*(a) = [\,f(1),\ f(2),\ f(3)\,] = [\,1,\ 4,\ 9\,].$$

The ISO/IEC 14882:2020 compliant CPP (C++20) spelling is much less terse. Using the ranges library:

```
const auto f = [](int x) -> int { return x * x; };
const auto a = std::list<int>{1, 2, 3};
const auto a_sqr = a | std::views::transform(f);
```

Here, the pipe operator, `|`, feeds the expression on the left to the expression on the right. The `transform` operator implements the element-wise function lifting. Now, if we wanted to see the output,

```
auto show_int = [](int x) { std::printf("%d ", x); };
std::ranges::for_each(a_sqr, show_int);
```

prints "`1, 4, 9`" as you might expect.[15] But now, the calculation from input to output has a life of its own: it is defined algebraically by composition of operations, like arrow composition in a category. The list can be pushed through the calculation regardless of its quality as an iterable quantity in memory or an observable quantity in time. The control program

[15] See the code on Compiler Explorer: `https://godbolt.org/z/sW3f6bvWh`.

application we will develop in ***Cpp***/C++ will be similarly agnostic, which is good for testing!

Two non-equivalent Domain Specific Languages (DSLs) for asynchronous lists pervade current programming practises,

1. Functional Reactive Programming (FRP), developed by Conal Elliot in the early 1990s and eventually with Paul Hudak from 1996–2014, and
2. ReactiveX (Rx) developed by Erik Meijer and others at Microsoft in the early 2000s.

Note that there is a wide spectrum of *reactive programming* models [121]. I highlight FRP as being perhaps the most theoretically pure and the predecessor of many others, and Rx as being the most widely applied. Ultimately, my primary C++ representation of abstract control programs, demonstrated in CH.5 and 6, uses Rx. But Rx was developed industrially, so academic literature is comparatively sparse. A demonstration of a C++ control program underpinned by FRP (instead of Rx) can be found in one of the demo repositories accompanying the thesis [183]. I mention this to point out that FRP and Rx provide roughly equivalent bases for the thesis' payload, despite their principle differences. Section 3.10 contains a reasonably thorough account of the important literature on Rx, so I reserve the remainder of the section for a brief account of FRP as a future direction.

FRP is actually much more than a model for asynchronous data. It is a DSL for expressing computation on the time continuum, that has with simple and precise denotational semantics. As Elliott says,

> [FRP] lays it[s values] out in one dimension … that we relate to as time. —https://youtu.be/rfmkzp76M4M?si=USnRbN9v9Cp_Nsxf&t=560

Elliott's ideas for this DSL evolved over many years as he worked on different graphics projects at Sun Microsystems and Microsoft Research. And it went by several names during its evolution: TBAG [68], RBML (based on the language ML), ActiveVRML, DirectAnimation, RBMH (written when Elliott found Haskell), Fran (**f**unctional **r**eactive **an**imation) [78] and finally Functional Reactive Programming (FRP). In 1996, Hudak was a professor at Yale who studied functional programming. He was invited to talk at Microsoft where Elliott was working at the time. Elliot introduced him to the RBML incarnation, and he was excited by the potential. The two collaborated extending the implementations and applications of FRP, until Hudak's untimely death in 2015.[16] In that time, it has been been given new context in Hughes calculus of arrows [86], as *arrowised* FRP [100], extended to real-time systems [91], then to real-time robotic applications [93], small embedded applications [150] and given push-based implementation [103].

Computer science is not about machines, in the same way that astronomy is not about telescopes. There is an essential unity of mathematics and computer science.
— M. R. Fellows [60]

FRP is based around two abstract data types [78, 90], *behaviours* and *events*:

$$\texttt{behaviour}\langle A\rangle \cong \mathbb{R} \to A \quad \text{and} \quad \texttt{event}\langle B\rangle \cong (\mathbb{R} \times B)^*.$$

Behaviours model time-signals of a value type A, and events are sequences of monotonically non-decreasing pairs of a time stamp and event type B (for any types A and B). The two compose, so we can have "events of behaviours" or *reactive behaviours.* It is actually this interaction that makes FRP *reactive*, which is why other DSLs for reactive programming tend to focus on events. In FRP, this all has simple and immaculate denotational semantics and is surely a theoretical masterpiece. Unfortunately it is notoriously difficult to implement with fidelity to the denotational semantics. The issues revolve around object lifetime: potential space- and time-leaks that take various forms depending on the language's evaluation strategy (lazy or not) and garbage collection.

C++ has a strict evaluation policy (unless one uses indirection to circumvent it) and no garbage collection (unless one implements it). This means C++ implementations of FRP have to manually handle object lifetime and so the problems do not go away—they get uglier. C++ program-

[16] The names Fran and FRP are due to Hudak. Although graphics was originally the target application of FRP, Hudak and Elliott were well aware that the DSL had nothing specific to graphics, and Hudak proposed the ultimate tile of Functional Reactive Programming (FRP).

mers often delight in pointing out that the language bestows an arsenal of sharp and heavy tools prone to plummet unexpectedly onto toes, from a great height.[17] Manual memory and object lifetime management definitely require the keys to the armory.

The leading C++ implementation of FRP is Sodium [195] by Stephen Blackheath, along with the accompanying book [147]. It is known to suffer these issues and as Blackheath politely puts it: *the memory management is not quite right yet.* I briefly worked with Blackheath on Sodium-C++, but the collaboration was put on hold when I chose Rx for the thesis. I do plan to return to work on Sodium-C++. I think a good future direction for the present research is implementation in FRP. It was shown in [150] that FRP is viable in targeting small, real-time embedded systems. Most interestingly, one cannot fail to notice that a language of continuous functions and discrete events should harmonise beautifully with hybrid systems theory. (This was well noted in [93], so I take no credit for the thought.) But from my point of view, the denotational semantics is a road to categorification where undiscovered insight may lurk in the intersection of the computer science and systems theories.

NOTE. At the closing of this section I do not want to leave the reader with the impression that Rx, in comparison to FRP, is just a muddy workhorse. Rx is still mathematically beautiful. It was never based on FRP so it had no commitment to denotational semantics and continuum time. It was conceived following the observation that, with a little bit of squinting, the Gang *of* Four (GoF) *iterator* and *observer* design patterns were roughly dual. By some formality of dualising the iterator pattern, one derives the rudiments of the Rx DSL (see §3.10). Despite arising from an OO design pattern, what comes out of the derivation looks very much like FRP's events and it can be used functionally. Toward that point, note that Rx has official implementations in statedly functional languages: RxScala, RxClojure and RxElixer (Elixer is based on Erlang); along with several unofficial imple-

[17] One might wonder about steel-toed boots in this analogy. Static analysers perhaps?

mentations in Haskell[18] See the full list of current official implementations at

`reactivex.io/languages.html`.

NOTE. There is more current research on event streams from a sheaf theoretic perspective [176]. There is also a generalisation of monoidal streams to arbitrary monoidal categories [177].

2.4 APPLIED CATEGORY THEORY AS A FIELD

Functional programming in particular and automata/language theory in general are widely regarded as some of the first applications of CT. But as its own entitled discipline, Applied Category Theory is a relatively young and emerging area of mathematics [168]. As the name suggests, it is the study of CT in application to a variety of domains including language theory, engineering design, economics, chemistry and reaction kinetics, game theory, information theory, databases, programming language theory (FP and domain theory), mechanics, quantum field theory, control theory, neuroscience and many more in a quickly growing list. A brisk introduction to CT is provided in Appendix B, where some literary sources are recommended. At the time of writing, I am only aware of a single introductory textbook on ACT: [168] by Brendan Fong and David I. Spivak. See also, a favourite reference of mine, "Physics, Topology, Logic and Computation: A Rosetta Stone" [115] by American mathematical physicist John C. Baez. (Fair warning: as a theoretical physicist, I admit some bias in this recommendation.) The Topos Institute regularly hosts colloquia [200] showcasing a stunning variety of applications of CT dispersed across science, art, mathematics and engineering.

[18] EG., see `hackage.haskell.org/package/RxHaskell`.

2.4.1 Machines in $\mathbb{P}\mathbf{oly}$

Many applications of CT are growingly regarded as instances of a general theory of interaction provided by a calculus of polynomial functors. The stage for this rich calculus is the category $\mathbb{P}\mathbf{oly}$, where objects are the full subcategory of endofunctors on $\mathbb{S}\mathbf{et}$ spanned by coproducts of representable functors and where arrows are the natural transformations. (You can build categories of polynomial functors on other categories to obtain more specific interactions [188].)

A representable functor (in the sense of Yoneda) on $\boldsymbol{C}$ is any functor $F : \boldsymbol{C} \to \mathbb{S}\mathbf{et}$ such that F is isomorphic to a hom-functor $\boldsymbol{C}(X, -) \to \mathbb{S}\mathbf{et}$. The representing object X is seen as embodying the structure of the functor. Let $\boldsymbol{C} = \mathbb{S}\mathbf{et}$, and further consider the functor $y^X := \mathbb{S}\mathbf{et}(X, -)$ that sends

- each set Z to the hom-set $Z^X = \mathbb{S}\mathbf{et}(X, Z) = X \multimap Z$, and
- each function $f : A \to B$ to $X \multimap f : (X \multimap A) \to (X \multimap B)$.

These are the functors represented by X, and for all $X \in \mathbb{S}\mathbf{et}$ collectively, *the representables.* Coproducts (sums) of these representables are polynomial functors, or simply *polynomials.*

Polynomial functors were well studied in the 2010s by computer scientists Daniel Ahman, Tarmo Uustalu, Nicola Gambino and Joachim Kock [122, 127, 141, 149]. Then around 2020 David I. Spivak, following conversations with Kock, discovered that $\mathbb{P}\mathbf{oly}$ embodied many of the structures and relationships he had been studying for many years! This led to a series of articles [172, 175, 190]. Later, Nelson Niu and Spivak began writing the textbook (still a work in progress) [182]. Further on, this has led to application in computer science [185] and logic [174], pure mathematics [181, 186], database theory [191], deep learning and prediction markets [187], collectives in economics [180], information theory [192] and neuroscience [189]. David Jaz Myers in particular studies dynamical systems (including the ones normally thought of in terms of differential equations or integral manifolds and associated machinery) [178, 179], which I expect to be important to follow on work for this thesis.

The study and application of $\mathbb{P}$**oly** has been rapidly developing as I have been writing this document. Everything in the theory of $\mathbb{P}$**oly** is compatible with what I will present in this thesis: in fact, $\mathbb{P}$**oly** *subsumes* it. It will be exciting in the future to recast the present thesis in the context of $\mathbb{P}$**oly**, I could not rewrite my thesis to keep pace with the development. However, a few remarks are certainly in order to show how my thesis ties in.

Consider the simple cases of monomials, $p := S\,y^S$ and $q := O\,y^I$, where $I, S, O \in \mathbb{S}\mathbf{et}$. (In this context, it is customary to denote the Cartesian product by juxtaposition.) A morphism $\varphi : p \to q$ can be regarded as a pair $(\varphi_1, \varphi^\#)$ where

$$\varphi_1 : S \to O \quad \text{and} \quad \varphi^\# : S \times I \to S.$$

(See marginalium [19]) If I, S, O are regarded as the input, state and output sets of a machine, then φ_1 is a state readout map and $\varphi^\#$ is a state transition map!

This leads to the notion of a Moore machine in $\mathbb{P}$**oly** as particular kind of map.

LEMMA 2.4-1 ([175, Prop. 2.12]). An (I, O)-Moore machine with states S can be identified with a map of polynomial functors $S\,y^S \to O\,y^I$ and hence, with a Oy^I-coalgebra $S \to O\,y^I \circ S$. [20]

Machines in $\mathbb{P}$**oly** offer a great deal more flexibility than this. For examples, interconnection can be dynamic (composite systems can require themselves) and input can depend on state (like in the case of control bundles).

[19] Morphisms between polynomials can be regarded as dependent lenses, of which this is a simple case due to the fact that p and q are monomial. More generally, φ_1 would be a function $p(1) \to q(1)$ and $\varphi^\#$ would be a family of functions a little too involved to describe here.

[20] Composition of a functor with a set, in this context, is functor application

As I mentioned, the present work is subsumed by this business of polynomial functors. A key point of ingress is given in Spivak's following lemma:

LEMMA 2.4-2 ([175, Prop. 2.11]). A p-coalgebra (S, ς) can be identified with the lens $Sy^S \to p$ by the isomorphism

$$\mathbb{P}\mathbf{oly}(S, p(S)) \cong \mathbb{P}\mathbf{oly}(S \times (S \multimap -), P).$$

In this case, the category p-**coAlg** is identified as the category of dynamical systems with interface (input/output) given by p.

Later on, the reader can compare the contents of §3.9 and CH. 5 to the previous lemma to see the relationship in fuller view.

I want the reader to leave this section with some perspective on the sort of unification that is going on in ACT at the moment, in the words of Jaz Myers [178]:

> Moore machines, differential equations, and Markov decision processes are each dynamical systems understood in a different theory.
>
> 1. A Moore machine is a dynamical system in a discrete and deterministic systems theory.
> 2. A system of differential equations is a dynamical system in a differential systems theory.
> 3. A Markov decision process is a dynamical system in a stochastic systems theory.

If you can provisionally take that at face value, then perhaps you can see why I think designing control systems programs as categorical Moore machines is the right direction for a unified theory of systems and software.

* * *

There are simply too many giants, and each of them with two shoulders to stand upon. This chapter has necessarily been incomplete. I feel poorly for the authors left unmentioned and their works left un-cited. (I suspect the reader knows this feeling and can sympathise.) However, it is time to be more specific in moving toward the theoretical payload of the thesis.

The next chapter excavates and details the mathematical technology from these references which I will use in Chapters 4 and 5 to erect a categorical model of control application software.

THEORETICAL PRELIMINARIES

3

3.1 INTRODUCTION

THIS CHAPTER explicates the key mathematical technology that was sketched in historical context in the previous chapter. The first of two goals is to lay foundation for a definition of control programs as a particular notion of *dynamical system* that arises from the categorical rapprochement of algebra automata and control systems theory. The second goal is to describe the notions of computer programming with which the mathematical notion of a control program can be realised.

CONTRIBUTIONS All of the definitions, theorems and propositions in this chapter are assembled from preexisting literature. There is obviously some art in the selection, exposition and harmonisation of these topics (especially where notation is involved). So while I do add unique insight, especially in the intermediating text, this chapter is not to be generally regarded as contributory—except insofar as it brings together and makes accessible the mathematical development of others and tailors them to the present thesis.

3.1.1 Section Synopses

§3.2 POLYNOMIAL FUNCTORS The familiar concept of a polynomial functions from school math is extended to functors in bicccs. While some intuition from polynomial functions is useful for understanding these counterparts, there are other useful ways of thinking about them, as trees or generalised bundles, especially in the case of polynomial functors on $\mathbb{S}\mathbf{et}$. But these intuitions carry straightforwardly to concrete categories such as ***Cpp***.

§3.3 ALGEBRAS & COALGEBRAS For an arbitrary endofunctor F on a category $\boldsymbol{C}$, F-algebras, F-algebra homomorphisms and the induced category of F-algebras, F-**Alg** are given, along with their duals (coalgebra and friends). Some notation is given, especially μF and νF for the initial algebra and terminal coalgebra, respectively.

§3.6 CATEGORIFICATION OF CLASSICAL AUTOMATA is largely a rehearsal of the 1975 of Goguen [33] with notation and presentation more consistent with modern literature (and more specifically, the notation of the thesis) and much more explanatory discussion. The notion of *Goguen machines* are defined for a biCCC, $\boldsymbol{C}$. Goguen machines are a categorification of the notion of sequential automata ordinarily modelled as right-actions of a monoid of input values on a set of states (which, depending on the structure of the states, can give a module structure.) Generalisation of Kleene closure, $(-)^*$, to cartesian monoidal categories facilitates abstraction. The result is a structurally recursive, input driven calculation for state evolution: a discrete I/S/O system. A similar construction on pure I/O gives behaviours, of which an I/S/O system can be regarded as a representation. Definitions for reachability and equivalence follows.

§3.7 FIXPOINTS OF ENDOFUNCTORS describes the mathematical technology required to further generalise Goguen machines. In §3.6, the input driven state propagation comes from algebras of the functor $\hat{I}_\bullet := (1+-\otimes I)$ for an input object I. That functor is a specific instance of an *input process.* The categorical progression from $\hat{I}_\bullet$ to machine, including the free monoid construction are specific instances of fixpoint construction on the input process functor. When fixpoints are coincident with initial algebra, there are unique algebra homomorphisms called *catamorphisms* which give recursion in the context of input processes the meaning of state evolution. The dual construction on coalgebra leads to anamorphisms which generalise behaviour. Toward a fuller understanding of this more general notion of machines and behaviour, this section presents the notions of fixpoints of functors and their relationships to initial algebras and terminal coalgebras. In computer science, these are used to define data and codata structures, especially lists (data) and streams (codata). This is where automata theory and functional programming overlap, which is the motivation for the section.

§3.8 snoc lists as fixpoints gives a category theoretical description of snoc-lists as fixpoints of **Set**-functors. In fact, this is the same functor as $\hat{I}_\bullet$ above! But in **Set** it is denoted $\hat{A}_\bullet := 1 + - \times A$ and the carrier of $\mu\hat{A}_\bullet$ gives A^*, the Kleene-closure: the set of all finite lists of elements of A. Catamorphisms of $\mu\hat{A}_\bullet$ are the computer scientific *right fold*, a useful recursion scheme for algebraic program calculation. The section closes with the mention of *scans* which are very much like folds, but lack a category theoretical universal property. In ch. 5, I will demonstrate that scans are obtained by an endofunctor on the category $\hat{A}_\bullet$-**Alg**, since we will want it to describe control programs which output intermediate values of their recursive state evolution process.

§3.9 moore about dynamical systems extends the reasoning about machines, using the generalised input process and fixpoint construction to define the evolution and behaviour of Moore machines, which can be seen as Goguen machines in **Set**.

§3.10 async lists & observer-iterator duality A slightly informal presentation of the ReactiveX (Rx) interface for derived by dualising the classical *iterator* design pattern. This yields a mathematical interface specification for asynchronous collections of values which can be readily implemented in most programming languages.

3.1.2 Prerequisites

The fundament of my thesis rests upon the body of literature I suspect the reader to be least familiar with: Category Theory (CT). Appendix B gives a brisk introduction to the theory. The 1945 paper of Eilenberg and MacLane [1] is widely regarded as the foundational document of CT. The definitive textbook on the subject is Mac Lane's *Categories for the Working Mathematician* [81], but this is unlikely to be the ideal introduction for non-mathematicians (as the title hints). A more approachable introduction than Mac Lane's, but still aimed at a mathematical audience, is Awodey's [109], which I cite many times in the thesis. Arbib and Manes wrote [32], with the express intent of "make[ing] category theory [...] accessible to computer scientists and control theorists." A briefer and more classical algebraic approach along these lines is "Basic Concepts of Category Theory Applicable to Computation and Control" [26] by the same authors. To the reader inclined toward computer science and programming, I usually recommend any or all of

- Barr and Wells' [82],
- Buurlage's [162], and
- Milewski's [169].

Buurlage and Milewski are particularly accessible texts, not only in pedagogical approach but also because they are freely available online.[1] More specifically, prerequisites for this chapter include,

Co/limits: See §B.8; externally, see [109, CH. 5], [32, §2.4], [81, CH. III].

Categorical products and coproducts: §B.9 and §B.10; externally, see [32, §1.2], [82, CH. 5] [109, §2.4, §3.2], [162, CH. 3].

Monoidal closed categories: monoidal §B.11; closed §B.15; externally, see [81, §VII.7], [82, §16.1].

[1] For which I owe the authors a special thanks for their generosity. I would also thank Igal Tabachnik for this effort, collecting Milewski's works and moulding them into a book.

- Cartesian Closed Categories (cccs): DEF. B.15-8; which is a category monoidal closed with respect to the categorical product; externally, see [81, §IV.6], [109, §6.2], [82, CH. 5].
- Bicartesian Closed Categories (bicccs): DEF. B.15-10; externally, see [67] for a thorough definition, but bicccs seem to be understood well enough as composite cartesian closed and cocartesian monoidal categories that they are taken for granted.
- Adjoint functors: §B.16; externally, see [81, CH. IV], [109, CH. 9], [162, CH. 6], [82, CH. 13].

3.1.3 Categorical setting

For this chapter, the category $\boldsymbol{C}$ always has a symmetric monoidal closed structure. The constituents of that structure will be denoted

$$\boldsymbol{C},\ \otimes,\ \multimap,\ 1_{\boldsymbol{C}},\ \alpha,\ \ell,\ \varrho,\ \gamma,$$

with product unit $1_{\boldsymbol{C}}$, associator α, left/right unitors ℓ/ϱ and symmetric braiding γ. Exponentials are denoted $(\multimap) :\ \boldsymbol{C}^{\mathrm{op}} \times \boldsymbol{C} \to \boldsymbol{C}$, which forms adjunctions

$$- \otimes X \dashv X \multimap -,$$

for any $X \in \boldsymbol{C}$, with hom-transposition λ witnessing

$$\hom(A \otimes B, C) \xrightarrow{\sim} \hom(A, B \multimap C).$$

Evaluation is denoted

$$ev_{A,B} :\ A \otimes (A \multimap B) \to B.$$

When required, $\boldsymbol{C}$ will also have cocartesian monoidal structure[2] *denumerable coproducts*. By that I mean colimits over discrete finite diagrams $\dot{\boldsymbol{n}} \to \boldsymbol{C}$ for arbitrary $n \in \mathbb{N}$. They will be denoted by the bifunctor $+$ with

[2] DEF. B.11-11.

unit/initial object $0_{\boldsymbol{C}}$. Coproducts have an associator, unitors and braiding as well, but their notation is not needed for this section. The product must distribute over the coproduct. We will make use of the distributor, denoted d, in various forms:

$$\begin{aligned}
A \otimes (B + C) &\xrightarrow{\sim} (A \otimes B) + (A \otimes C) \\
A \otimes \sum_m B_n &\xrightarrow{\sim} \sum_n (A \otimes B_n) \\
\Big(\sum_n A_n\Big) \otimes B &\xrightarrow{\sim} \sum_n (A_n \otimes B) \\
\Big(\sum_m A_m\Big) \otimes \Big(\sum_n B_n\Big) &\xrightarrow{\sim} \sum_{(m,n)} (A_m \otimes B_m).
\end{aligned} \tag{3.1.1}$$

(see [33, PROP. 2.2] or [81, §VII.3, THM. 2].) These distributors exist by virtue of the closure on $\otimes$. It is a basic fact of CT that *left-adjoints preserve colimits* and right-adjoints preserve limits (CF. [109, §9.6]). Since the product is closed, there are adjunctions $- \otimes X \dashv X \multimap -$ and so $- \otimes X$ preserves colimits (and thereby coproducts). Since $\otimes$ is symmetric, $- \otimes X \overset{\gamma}{\xrightarrow{\sim}} - \otimes X$ and the product preserves colimits in both arguments. See also [24].

Despite the fact that we are endeavouring to assume nothing about the inner lives of the objects in $\boldsymbol{C}$ (since CT provides no tools of direct introspection) I will occasionally use

$$\boldsymbol{C} = \mathbf{Set}, \quad \otimes = \times, \quad + = \sqcup, \quad 1_{\boldsymbol{C}} = 1, \quad 0_{\boldsymbol{C}} = 0, \quad \multimap = \hom$$

where we can regress to set theory for visual intuition. The associators, unitors, braidings and distributor for the bicartesian closure of $\mathbf{Set}$ are detailed in §B.12. As I write the thesis, this is always my mental model.

3.2 POLYNOMIAL FUNCTORS

The category theoretical concept of a polynomial functor varies slightly by author and application. In fact, the same concept has been reinvented several times in different domains. Herein, a "polynomial functor" is roughly identical to the notions of container functors in theoretical computer science.

Polynomial functors are so called because they are a categorification of the concept of polynomial functions. In other words, they are recognisable as combinations of sums, products and exponents. For example,

$$P\,X \coloneqq 3\,X^4 + X^2 + 1. \tag{3.2.2}$$

Here the variable X and each of the constants are sets: $3 \cong \{0,1,2\}$, and so on. The sum, product and exponentials are from the category's structure (requiring biccc-structure).

NOTATION 3.2-1 (Enumeration set)**.** Given a non-negative integer $n \in \mathbb{N}$, the sans serif symbol n denotes the set $\{\,0,\ldots,n-1\,\}$. Likewise with literal digits: $\mathsf{0} = \{\ \}$, $\mathsf{1} = \{\,0\,\}$, $\mathsf{2} = \{\,0,1\,\}$ and so on. Such sets may be called *enumeration sets.*

Consider that a $\mathbb{S}\mathbf{et}$-function $p : E \to B$ can be thought of as inducing a family of sets $(E_b)_{b\in B}$ regarded of as fibres over the base point b. That is,

$$E_b = \{\,e : E \mid p(b) = e\,\}.$$

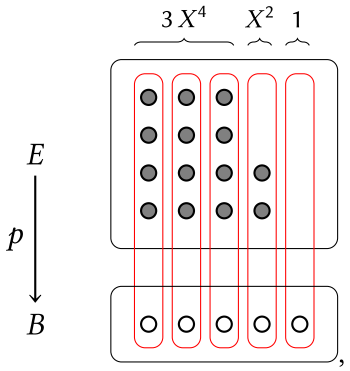
We can make this more concise as a preimage $E_b = p^{-1}(b)$, but bear in mind that the -1 superscript does not mean that p is bijective.

DEFINITION 3.2-2. An endofunctor $P : \mathbb{S}\mathbf{et} \to \mathbb{S}\mathbf{et}$ is a polynomial functor in $\mathbb{S}\mathbf{et}$ if there exists a morphism $p : E \to B$ inducing family of sets $(B_i)_{e\in E}$ and such that

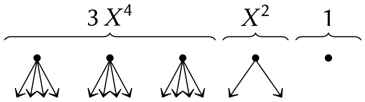
$$P\,X \cong \sum_{b\in B} X^{E_b} \equiv \sum_{b\in B} E_b \multimap X.$$

More generally, a robust calculus of polynomial functors can be constructed in any locally Cartesian closed category [122]. For the thesis, we can work comfortably in $\mathbb{S}\mathbf{et}$.

There are at least two other useful descriptions: as bundles and as trees [182]. Recalling (3.2.2), the polynomial $3\,X^4 + X^2 + 1$ might be visualised as a bundle like this,



or as trees like this,



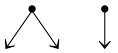
In the bundle picture, the base points, collected from the coefficient sets, cast fibres which are the exponent sets. In the tree picture, the base-points are interpreted as nodes from which fibre elements emanate as edges.

Two identities to keep in mind are,

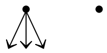
$$\sum_{b\in B} X \cong B \times X \quad \text{and} \quad \prod_{e\in E} X \cong E \multimap X.$$

The first is why the monomial $3\,X^4 = \sum_{b\in 3} X^4$ appears as three terms apparent in the bundle and tree pictures.

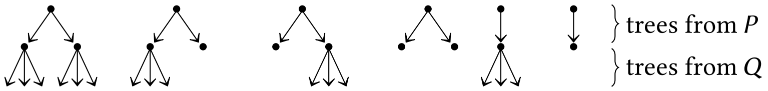
Just like polynomial functions, polynomial functors can be composed: $P \circ Q$. Moreover, the result is polynomial, so polynomials are closed under composition [190]. We will be particularly concerned with expressions like $P \circ P \circ \cdots = P^n$ where a polynomial is composed with itself some n-times. The tree picture is particularly useful for understanding what the composite looks like. When two polynomial functors are composed, the nodes and edges of the second are placed onto the edges of the first, in every possible combination, and condensed downward. Let

$$P \coloneqq X \mapsto X^2 + X \quad \text{or}$$ 

and

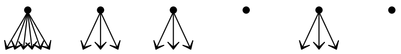
$$Q \coloneqq X \mapsto X^3 + 1 \quad \text{or}$$ 

then we could intuit by school algebra that $P \circ Q \cong X \mapsto X^6 + 3\,X^3 + 2$. Indeed, we can arrive at that conclusion graphically. The trees from Q are affixed to the edges of trees from P in every combination:



(3.2.3)

The two-tier structure is then condensed into a single layer by contracting along the P-edges to give

$$P \circ Q \cong X \mapsto X^6 + 3\,X^3 + 2 \quad \text{or}$$ 

The two-tier structure of (3.2.3) makes clearer how to interpret the edges. Though all base-points in P and Q were drawn with bullets, the base point of P with two edges is different than the base point with one. The bases are sets with distinct elements, and the data of the polynomial associates the edges (collectively a fibre) to the individual points in the base set. P and Q have two different base sets, so the association of the elements of each P-fibre with a Q-base element is important data in the composite. That is, each fibre element in the condensed picture is actually a choice of P-base element, P-fibre element, Q-basepoint and Q-fibre element. While the condensed picture is technically correct, it obscures that fact.

REMARK 3.2-3. So far, the examples have had finite set coefficients and exponets, but there is nothing to preclude terms like $\mathbb{R}\,X^{\mathbb{Z}}$, except that they are difficult to illustrate.

3.3 ALGEBRAS & COALGEBRAS

Algebras and Coalgebras of endofunctors on an arbitrary category $\boldsymbol{C}$ are essential instruments in the construction of the thesis' model. Functor algebras, in certain contexts, can be thought of as maps for evaluation or interpretation of structure [47]. They are used in the categorification of universal algebra to encode algebras without laws or *anarchic algebras* [64, §2.2]. Functor algebras form categories, and when those categories have initial objects they become the essence of process, recursion and induction [40], [169, CH. 24] Dually with coalgebras we get behaviour, corecursion and coinduction [170].

With all of that promise, here are the definitions that deliver it; the triplet: functor algebra, algebra homomorphism, category of algebras of a functor; and then the duals.

DEFINITION 3.3-1. Given an endofunctor F on a category $\boldsymbol{C}$, an ***F-algebra*** is a pair (A, α) consisting of,

- an object $A \in \operatorname{Ob} \boldsymbol{C}$ called the ***carrier object***, and
- an arrow $\alpha : F\,A \to A$, called the structure map of the algebra.

DEFINITION 3.3-2. For an endofunctor F on $\boldsymbol{C}$, an ***F-algebra homomorphism***, $h : (A, \alpha) \to (B, \beta)$, is a $\boldsymbol{C}$-arrow $|h| : A \to B$ such that the following diagram commutes:

$$\begin{tikzcd}
FA \arrow[r, "Fh"] \arrow[d, "\alpha"'] & FB \arrow[d, "\beta"] \\
A \arrow[r, "h"'] & B
\end{tikzcd} , \qquad (3.3.4)$$

thereby preserving the F-algebra structure.

DEFINITION 3.3-3. An endofunctor F on $\boldsymbol{C}$ induces a category of its algebras, F-$\mathbf{Alg}(\boldsymbol{C})$, with

objects: all F-algebras,

$$\operatorname{Ob} F\text{-}\mathbf{Alg} \coloneqq \sum_{A\,:\,\boldsymbol{C}} \{FA \to A\},$$

and

arrows: all algebra homomorphisms among them,

$$F\text{-}\mathbf{Alg}\big((A,\ \alpha),\ (B,\ \beta)\big) \coloneqq \big\{h \in \boldsymbol{C}(A,\ B) \mid h \circ \alpha = \beta \circ Fh\big\}.$$

When clarity permits, the category is omitted from the notation: F-**Alg**.

NOTATION 3.3-4. If F-**Alg** for $F : \boldsymbol{C} \to \boldsymbol{C}$ has an initial object then it is denoted in whole as μF or in components $(\mu F,\ in_F)$. The subscript on in can be omitted when clarity permits.

And now the duals.

DEFINITION 3.3-5. Given an endofunctor F on a category $\boldsymbol{C}$, an ***F-coalgebra*** is a pair (A, α) consisting of

- an object $A \in \mathrm{Ob}\,\boldsymbol{C}$ called the ***carrier object***, and
- an arrow $\alpha : A \to F\,A$, called the *costructure map* of the algebra.

DEFINITION 3.3-6. For an endofunctor F on $\boldsymbol{C}$, a ***F-coalgebra homomorphism***, $h : (A, \alpha) \to (B, \beta)$, is a $\boldsymbol{C}$-arrow $|h| : A \to B$ preserving the F-coalgebra structure by making the following diagram commute:

$$\begin{tikzcd}
FA \arrow[r, "Fh"] & FB \\
A \arrow[u, "\alpha"] \arrow[r, "h"] & B \arrow[u, "\beta"']
\end{tikzcd} .$$

DEFINITION 3.3-7. An endofunctor F on $\boldsymbol{C}$ induces a category of its coalgebras, F-**coAlg**($\boldsymbol{C}$) with

objects: all F-coalgebras:

$$\mathrm{Ob}\,F\text{-}\mathbf{coAlg} \coloneqq \sum_{A\,:\,\boldsymbol{C}} \big\{A \to FA\big\}$$

arrows: all algebra homomorphisms among them:

$$F\text{-}\mathbf{coAlg}\big((A,\ \alpha),\ (B,\ \beta)\big) \coloneqq \big\{h \in \boldsymbol{C}(A,\ B) \mid \beta \circ h = Fh \circ \alpha\big\}$$

When clarity permits, the category is omitted from the notation: F-**coAlg**.

NOTATION 3.3-8. If F-**coAlg** for $F : \boldsymbol{C} \to \boldsymbol{C}$ has an terminal object then it is denoted in whole as νF or in components $(\nu F,\ out_F)$. The subscript on *out* can be omitted, clarity permitting.

ALGEBRAS VS. COALGEBRAS Although algebras and coalgebras are dual structures, it is not the case that F-**Alg** and F-**coAlg** are dual categories, [108]. In fact,

$$(F\text{-}\mathbf{Alg})^{\mathrm{op}} = (F^{\mathrm{op}})\text{-}\mathbf{coAlg}.$$

The consequences of this relationship are deep and interesting, especially in the context of dynamical systems and control. But more on that later.

3.4 ALGEBRAS FOR A MONAD & FREE MONOIDS

As I mentioned in the last chapter, there is a distinction drawn between “algebra in the sense of Lambek” and “algebra in the sense of Eilenberg-Moore”. The afore definitions are algebras in the Lambek style. When the functor F^L additionally carries the structure of a monad (§B.14) then everything above specifies to *Eilenberg-Moore algebras* simply *monad algebras* [81, CH. VI].

This illustrates the well-known principle of universal algebra: all information is in the free algebras
— Arbib & Manes [32, p. 171]

DEFINITION 3.4-1. Given a monad $T = (T, \eta, \mu)$ in a category $\boldsymbol{C}$, a ***T-algebra*** is a pair (A, α) consisting of,

- an object $A \in \mathrm{Ob}\, \boldsymbol{C}$ called the ***carrier object***, and
- an arrow $\alpha : F^L A \to A$, called the structure map, that preserves the monadic structure by satisfying

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[dr, equal] & TA \arrow[d, "\alpha"] \\ & A \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} T^2 A \arrow[r, "\mu_A"] \arrow[d, "T\alpha"'] & TA \arrow[d, "\alpha"] \\ TA \arrow[r, "\alpha"'] & A \end{tikzcd} .$$

Homomorphisms of T-algebras are the same as for functor algebras because (3.3.4) will respect whatever monadic structure T carries.

The category of T-algebras and their homomorphisms over a category $\boldsymbol{C}$ is called the Eilenberg-Moore category of the monad, and is denoted $\boldsymbol{C}$.

THEOREM 3.4-2 ([109, PROP. 10.3])**.** Every adjunction

$$\left(\begin{tikzcd} \boldsymbol{C} \arrow[r, bend right, "F"'] \arrow[r, phantom, "\perp"] & \boldsymbol{D} \arrow[l, bend right, "U"'] \end{tikzcd}, \ \eta, \ \varepsilon \right)$$

with unit $\eta : \ \mathsf{id}_D \Rightarrow UF$ and counit $\varepsilon : \ FU \Rightarrow \mathsf{id}_{\boldsymbol{C}}$, induces a monad $(T, \ \eta, \ \mu)$ on $\boldsymbol{C}$ where

$$\begin{aligned} T\,C &= UF\,X : \ \boldsymbol{C} \to \boldsymbol{C} \\ \eta_C &= \eta_C \quad \text{(the unit of the adjunction)} \\ \mu_C &= U\varepsilon_{FC}. \end{aligned}$$

for $C \in \boldsymbol{C}$.

CONSTRUCTION 3.4-3 (Free $\$\mathbf{et}$-monoids)**.** Recall from DEF. B.11-4, that $\mathbf{Mon}(\$\mathbf{et})$ is the category of all algebraic monoids (sets with unital associative operation) with a forgetful functor $U_{\mathbf{Mon}} : \ \mathbf{Mon}(\$\mathbf{et}) \to \$\mathbf{et}$.

Let A be an arbitrary set. A *word* or *list* over A is a finite sequence of its elements, $[\,a_1, \, a_2, \, \dots\,]$. The set of all such lists is denoted and defined $A^* := \sum_{n \in \mathbb{N}} A^{\times n}$ (called the Kleene closure or Kleene-star), which includes the empty word $[]$. This set is closed under the operation of concatenation, denoted $+\!\!+$, where two lists are concatenated as

$$[a, b, c] +\!\!+ [c, b, a] = [a, b, c, c, b, a].$$

This forms an algebraic monoid since, for any words $w, w'^{\cdots} \in A^*$,

- $w +\!\!+ w' \in A^*$, giving closure
- $w +\!\!+ (w' +\!\!+ w'') = (w +\!\!+ w') +\!\!+ w''$ giving associativity, and
- $w +\!\!+ [] = [] +\!\!+ w = w$ giving left and right identity.

Therefore objects $(X^*,\ +\!\!+,\ [])$ are algebraic monoids.

We will sketch a routine construction of $(A^*,\ +\!\!+,\ [])$ for $A \in \mathbf{\$et}$ as free objects in $\mathbf{Mon}(\mathbf{\$et})$ by defining a "free" functor F^L as a left adjoint to the forgetful $U^L := U_{\mathbf{Mon}}$.

The free functor takes each set to our proclaimed free monoid as

$$\begin{array}{rcl} F^L :\ \mathbf{\$et} & \to & \mathbf{Mon}(\mathbf{\$et}) \\ X & \mapsto & (X^*,\ +\!\!+,\ \vec{[]}) \end{array}$$

$$\begin{tikzcd} X \arrow[dd, "f"'] & & (X^*,\ +\!\!+,\ \vec{[]}) \arrow[dd, "F^L f"] \\ & \mapsto & \\ Y & & (Y^*,\ +\!\!+,\ \vec{[]}) \end{tikzcd},$$

where $F^L f$ is defined by pointwise application,

$$F^L f :\ X^* \to Y^*;\ \ [\,x_1,\ x_2,\ \ldots\,] \mapsto [\,f(x_1),\ f(x_2),\ \ldots\,]$$

and of course $F^L f([]) = []$.

We pose the free-forgetful adjunction $F^L \dashv U^L$ by defining the unit and counit. The unit is a natural "insertion of generators",

$$\eta :\ \mathrm{id}_{\mathbf{\$et}} \Rightarrow U^L F^L \ \text{ with components }\ \eta_A :\ A \to A^*;\ x \mapsto [x].$$

For any given element in the generating set A, η_A simply pics out the singleton list in A^* containing that element. Comparatively, the counit does a lot more work. Given any monoid $M = (|M|,\ \text{Ⓜ},\ e_M) \in \mathbf{Mon}(\mathbf{\$et})$, the counit

$$\varepsilon :\ F^L U^L \Rightarrow \mathrm{id}_{\mathbf{Mon}}$$

maps the free-monoid on $|M|$ back into M. The only data on hand to perform such a procedure is Ⓜ and e_M, so we pass straightforwardly to the mapping

$$\begin{aligned} &\varepsilon_M :\ (M^*,\ +\!\!+,\ \vec{[]}) \to (M,\ \text{Ⓜ},\ e_M) \\ &\quad \begin{cases} [] & \mapsto\ e_M \\ [\,m_1,\ m_2,\ m_3,\ \ldots,\ m_n\,] & \mapsto\ m_1 \text{ Ⓜ } m_2 \text{ Ⓜ } m_3 \text{ Ⓜ } \cdots \text{ Ⓜ } m_n \end{cases} \end{aligned} \tag{3.4.5}$$

In words, we take the underlying set of M and generate from it the free monad $(|M|^*,\ \mathbin{+\!\!+},\ \vec{[]})$ of all finite strings of M-elements; then each of those strings, which may be regarded as $[m_1] \mathbin{+\!\!+} [m_2] \mathbin{+\!\!+} \cdots$ maps to the same expression but with the brackets removed and $\mathbin{+\!\!+}$ replaced by Ⓜ: m_1 Ⓜ m_2 Ⓜ $\cdots$.

If unfamiliar, I shall leave it to the reader to convince themselves that this co/unit satisfy the triangle identities, (B.16.21), or perhaps consult a more generous presentation: [109, eg. 10.7], [82, ch. 13], [169, ch. 13]. In short,

$$U^L\varepsilon \circ \eta_{U^L} = \mathrm{id}_{U^L}$$

is satisfied because, starting from a monoid $(M,$ Ⓜ$, e_M)$,

- forgetting the structure gives $M = \{e_M,\ m_1,\ m_2,\ \ldots\}$, then
- inserting the generators into M^* gives the image $\{[e_M],\ [m_1],\ [m_2],\ \ldots\}$, then
- applying to that the forgotten ε, has the effect of simply removing the list-brackets, leaving us where we started.

And

$$\varepsilon_{F^L} \circ F^L\eta = \mathrm{id}_{F^L}$$

is satisfied because, starting from a set A,

- applying F^L gives $(A^*,\ \mathbin{+\!\!+},\ [])$, then
- applying $F^L\eta$ identifies the lists of A^* as generators in A^{**}, giving $(A^{**},\ \mathbin{+\!\!+},\ [])$, all lists of lists of A-elements, then
- applying ε_{F^L} concatenates the outer layer of lists giving back the set of all lists of A, still in the monoid $(A^*,\ \mathbin{+\!\!+},\ [])$.

CONSTRUCTION 3.4-4 (The list monad)**.** By THM. 3.4-2, we know the adjunction $F^L \dashv U^L$ gives rise to a monad $L = (U^LF^L,\ \eta,\ \mu)$ where

$$\eta = \eta :\ \mathrm{id} \Rightarrow L \ \ \text{(i.e., the monad unit is the adjunction unit)}$$

and

$$\mu = U^L\varepsilon_F :\ L^2 \Rightarrow L,$$

taking “lists of lists” to lists by concatenation. (See [81, p. IV.4].) This “free monoid monad” is the *list-monad*, ubiquitous in the community of FP-programmers.

CONSTRUCTION 3.4-5 (*L*-algebras)**.** Consider the L-algebras, $(A, \alpha : LA \to A)$. The structure map α must be equivalent to a collection of maps to distill each word in A^* to a single A-value. But all the structure of the adjunction allows us to be more concise. Substituting α into the “universal mapping property” (UMP) of the adjunctive unit, (B.16.22), we get

$$\begin{tikzcd}
(A^*, +\!\!+, []) = F^L A \arrow[r, dashed, "\varepsilon\circ F^L\alpha"] & (A, \text{Ⓐ}, a_0)
\end{tikzcd}$$

$$\begin{tikzcd}
U^L F^L A = L A \arrow[r, "\alpha"] & A = U^L(A, \text{Ⓐ}, a_0) \\
A \arrow[u, "\eta_A"] \arrow[ur, equal] &
\end{tikzcd}$$

which shows that the mapping of α is tantamount to the choice of a monoid $(A, \text{Ⓐ}, \vec{a}_0)$ as

$$\alpha : \begin{cases} [] & \mapsto \; a_0 \\ [\,a_1,\, a_2,\, a_3,\, \ldots,\, a_n\,] & \mapsto \; a_1 \,\text{Ⓐ}\, a_2 \,\text{Ⓐ}\, a_3 \,\text{Ⓐ} \cdots \text{Ⓐ}\, a_n \end{cases}$$

Notice then that L-algebras have inbuilt, the monoid axioms suggesting something interesting about the relationship between the Eilenberg-Moore category $\mathbf{\$et}^L$ and $\mathbf{Mon}(\mathbf{\$et})$. (I leave that to the reader’s imagination, as it is something I shall not use.)

3.5 CATEGORICAL MONOIDS *&* MONOID ACTIONS

In this section we see how algebras over a monad generalise “actions of a monoid” for monoids in an arbitrary category. The slogan I borrow from Arbib and Manes is “sets are functors”, but paraphrased to *objects are functors*; this view being facilitated by a monoidal product. Let $\boldsymbol{C}$ be monoidal closed, as per §3.1.3.

DEFINITION 3.5-1. The functor P assigns, to each $\boldsymbol{C}$-object, a functor; and to each arrow, a natural transformation as

$$\begin{aligned} P :\ & \boldsymbol{C} \to \boldsymbol{C}^{\boldsymbol{C}} \\ & X \mapsto - \otimes X \end{aligned}$$

$$\begin{tikzcd} A \arrow[d, "f"'] & \mapsto & \mathrm{id}_{\_} \otimes A \arrow[d, Rightarrow, "\mathrm{id}_{\_} \otimes f"] \\ B & & \mathrm{id}_{\_} \otimes B \end{tikzcd} \cdot$$

Let us embrace the notion of objects as functors and denote in short $\hat{A} = P\,A$, $\hat{B} = P\,B$ and so on. Likewise, any $f :\ A \to B$ becomes $\hat{f} :\ \hat{A} \Rightarrow \hat{B}$, satisfying the naturality square

$$\begin{tikzcd} \hat{A}\,X \arrow[r, "\hat{A}\,g"] \arrow[d, "\hat{f}"'] & \hat{A}\,Y \arrow[d, "\hat{f}"] \\ \hat{B}\,X \arrow[r, "\hat{B}\,g"] & \hat{B}\,Y \end{tikzcd}$$

for any $g :\ X \to Y$.

Since $\mathbb{C}$**at** is Cartesian closed, we can regard $\otimes :\ \boldsymbol{C} \times \boldsymbol{C} \to \boldsymbol{C}$ as curried: $\boldsymbol{C} \to \boldsymbol{C}^{\boldsymbol{C}}$, and P just lambda-lifts the second position instead of the first.

THEOREM 3.5-2. The natural transformations $\hat{f}$ are unique, and in bijection with arrows hom(A, B), so P is fully faithful, and it is additionally injective on objects

Proof. Cf. [32, §10.2]. □

In other words, this construction embeds $\boldsymbol{C}$ as a full subcategory of $\boldsymbol{C}^{\boldsymbol{C}}$. This is alluding to the fact that monoids in $\boldsymbol{C}$ will also be monoids in $\boldsymbol{C}^{\boldsymbol{C}}$, as we shall later see.

OBSERVATION 3.5-3. Composition of these left-tensor functors mirrors the monoidal composition of the product:

$$\hat{B} \circ \hat{A} = (- \otimes A) \otimes B$$

OBSERVATION 3.5-4. Iterated products can be constructed as iterated composites:

$$\begin{aligned}
\hat{A}^0\, 1_{\boldsymbol{C}} &= A^{\otimes 0} = 1_{\boldsymbol{C}} \\
\hat{A}^1\, 1_{\boldsymbol{C}} &\cong A^{\otimes 1} = A \ \ (\text{since } \ell_A : 1_{\boldsymbol{C}} \otimes A \xrightarrow{\sim} A) \\
\hat{A}^2\, 1_{\boldsymbol{C}} &\cong A^{\otimes 2} \\
\hat{A}^3\, 1_{\boldsymbol{C}} &\cong A^{\otimes 3} \\
&\ \vdots \\
\hat{A}^n\, 1_{\boldsymbol{C}} &\cong A^{\otimes n}
\end{aligned}$$

This way, abstract lists (that take their precise meaning depending on ⊗) can be built as chains

$$\begin{tikzcd}
1_{\boldsymbol{C}} \arrow[r, "\hat{A}"] \arrow[d, equal] & \hat{A}\, 1_{\boldsymbol{C}} \arrow[r, "\hat{A}"] \arrow[d, "\ell_A", "\wr"'] & \hat{A}^2\, 1_{\boldsymbol{C}} \arrow[r, "\hat{A}"] \arrow[d, "\ell \otimes \mathrm{id}", "\wr"'] & \cdots \\
A^{\otimes 0} \arrow[r, "- \otimes A"] & A^{\otimes 1} \arrow[r, "- \otimes A"] & A^{\otimes 2} \arrow[r, "- \otimes A"] & \cdots
\end{tikzcd}$$

This foreshadows a more rigorous and general construction later: fixpoints of endofunctors.

NOTATION 3.5-5. For each $A \in \boldsymbol{C}$, the functor $\hat{A}$ will have algebras of the form $(B, \beta : \hat{A}\, B \to B)$. Since we have identified these functors with objects, let us call them ***object algebras***.

We now position ourselves to describe monoids internal to $\boldsymbol{C}$, which will have an interesting relationship to object algebras.

DEFINITION 3.5-6 (monoid object)**.** In a monoidal category $(\boldsymbol{C}, \otimes, 1_{\boldsymbol{C}})$, a ***monoid object in $\boldsymbol{C}$***, AKA an *internal monoid*, AKA a ***$\boldsymbol{C}$-monoid***, is a diagram of shape

$$1_{\boldsymbol{C}} \xrightarrow{\vec{e}_M} M \xleftarrow{\text{Ⓜ}} M \otimes M\,, \tag{3.5.6}$$

specified as a tuple $(M, \text{Ⓜ}, \vec{e}_M)$ where,

- M, a $\boldsymbol{C}$-object (the *monoid object*),

- $\textcircled{M} : M \otimes M \to M$, a morphism called the ***multiplication*** of the monoid,
- $\vec{e}_M : 1_{\boldsymbol{C}} \to M$, and a global element determining a ***monoidal unit***;

all subject to the commutativity of the association pentagon[3]

$$\begin{tikzcd}
 & M & \\
M \otimes M \arrow[ur, "\textcircled{M}"] & & M \otimes M \arrow[ul, "\textcircled{M}"'] \\
M \otimes (M \otimes M) \arrow[u, "\mathrm{id}_M \otimes \textcircled{M}"] \arrow[rr, "\alpha"'] & & (M \otimes M) \otimes M \arrow[u, "\textcircled{M} \otimes \mathrm{id}_M"'] ,
\end{tikzcd}$$

and the unitor diagrams

$$\begin{tikzcd}
1_{\boldsymbol{C}} \otimes M \arrow[r, "\vec{e}_M \otimes \mathrm{id}_M"] \arrow[dr, "\ell_M"'] & M \otimes M \arrow[d, "\textcircled{M}"] & M \otimes 1_{\boldsymbol{C}} . \arrow[l, "\mathrm{id}_M \otimes \vec{e}_M"'] \arrow[dl, "\varrho_M"] \\
 & M &
\end{tikzcd}$$

NOTE. It is often useful to denote a monoid's operation with an infix symbol, $m_1 \textcircled{M} m_2$, as above. When it is of benefit to clarity, an infix symbol may be put between parentheses and treated as an prefix morphism $m_1 \textcircled{M} m_2 \equiv (\textcircled{M})(m_1, m_2)$, as per NTN. A.1-10.

Notice that algebraic monoids are exactly $(\mathbb{S}\mathbf{et}, \times, 1)$-monoids; but categorical monoids generalise to any monoidal category.

Monoid homomorphisms are also manifest at the internal level of analysis.

DEFINITION 3.5-7. A $\boldsymbol{C}$-arrow $h : M \to M'$ is a ***$\boldsymbol{C}$-monoid homomorphism*** if it preserves the associativity and unitor diagrams by commuta-

[3] Trivia: sometimes called "Mac Lane's pentagon" after Saunders Mac Lane who wrote [81]

tivity of the diagrams

$$\begin{tikzcd}
M \otimes M \arrow[r, "h \otimes h"] \arrow[d, "\mu"'] & M' \otimes M' \arrow[d, "\mu'"] \\
M \arrow[r, "h"] & M'
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
 & 1_{\boldsymbol{C}} \arrow[dl, "\vec{1}_M"'] \arrow[dr, "\vec{1}_{M'}"] & \\
M \arrow[rr, "h"] & & M'
\end{tikzcd}
.$$

This leads to a category!

DEFINITION 3.5-8. The category of $\boldsymbol{C}$-monoids, $\mathbf{Mon}(\boldsymbol{C})$ is comprised of

objects: all monoid objects in $\boldsymbol{C}$ and

arrows: all monoid object homomorphisms.

When $\boldsymbol{C}$ is clear from context, we just write **Mon**.

OBSERVATION 3.5-9. Every $\boldsymbol{C}$-monoid $(M,\ \textcircled{M},\ \vec{e}_M)$ induces a monad $(\hat{M},\ \mu,\ \eta)$ with

$\mu \coloneqq (\mathrm{id} \times \textcircled{M}) \circ \alpha^{-1} :\ \hat{M}^2 - \Rightarrow \hat{M}-$, where

- α^{-1} is the (inverse) product associator: $(- \otimes M) \otimes M \xrightarrow{\sim} - \otimes (M \otimes M)$

and

$\eta \coloneqq (\mathrm{id} \otimes \vec{e}_M) \circ \varrho^{-1} :\ - \Rightarrow \hat{M}-$, where

- ϱ^{-1} is the (inverse) right unitor: $- \to - \otimes 1$.

These μ, η observe the monad laws,

$$\begin{tikzcd}
\hat{M}^3 \arrow[r, "\hat{A}\mu"] \arrow[d, "\mu\hat{A}"'] & \hat{M}^2 \arrow[d, "\mu"] \\
\hat{M}^2 \arrow[r, "\mu"'] & \hat{M}
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\hat{M} \arrow[r, "\hat{M}\eta"] \arrow[dr, equal] & \hat{M}^2 \arrow[d, "\mu"] & \hat{M} \arrow[l, "\hat{\eta}\hat{M}"'] \arrow[dl, equal] \\
 & \hat{M} &
\end{tikzcd}$$

So, for any monoid $M \in \mathbf{Mon}(\boldsymbol{C})$ we have a monad in $\boldsymbol{C}^{\boldsymbol{C}}$ given by $(\hat{M},\ \mu,\ \eta)$ above!

This observation if surprising, may at least have been predictable. The functor P maps $\boldsymbol{C}$ into the functor category $\boldsymbol{C}^{\boldsymbol{C}}$. There is a now famous slogan within FP lore that "monads are just monoids in the category of endofunctors". So we can reason just on a semantic level that, through P, the image of monoids in $\boldsymbol{C}$ will be monoids $\boldsymbol{C}^{\boldsymbol{C}}$ and thus are monads.

There is a forgetful functor $U : \mathbf{Mon}(\boldsymbol{C}) \to \boldsymbol{C}$ which has a 'free' left adjoint, taking each $\boldsymbol{C}$-object to the free monoid it generates.

CONSTRUCTION 3.5-10 (Free $\boldsymbol{C}$-monoids)**.** This construction parallels exactly the construction of $\mathbf{Set}$-monoids from CONST. 3.4-3, only now we lack the insight of element-wise thinking. But because multi-products are countably iterated applications of $\otimes$, there is a natural ordering that was illustrated in OBS. 3.5-4. Thereby, iterated products inherit the ordering of the natural numbers making them amenable to inductive reasoning.

First, we define the functor $(-)^*$ (re-using the Kleene-star) as follows:

$$\begin{tikzcd}
(-)^* : \boldsymbol{C} \arrow[r] & \boldsymbol{C} & \\
X \arrow[r, mapsto] & \sum_{n\in\mathbb{N}} X^{\otimes n} & \\
X \arrow[dd, "f"'] & & X^* \arrow[dd, "h"] \\
\arrow[r, phantom, "\mapsto"] & \sum_{n\in\mathbb{N}} f^{\otimes n} : & \\
Y & & Y^*
\end{tikzcd} .$$

Objects X^* are the coproduct of all finite $\otimes$-powers of X.[4] Likewise, f^* is f applied to each factor in a given power of X: for $X^{\otimes n}$ we have

$$f^* \xleftarrow{\iota_n} f^{\otimes n} = \underbrace{f \otimes f \otimes \cdots \otimes f}_{n-\text{times}} : \; X^{\otimes n} \to Y^{\otimes n}.$$

In $\mathbf{Set}$ with $\otimes = \times$ and $+ = \sqcup$, then $(-)^*$ is the Kleene start and f^* is just the pointwise application of f to a tuple of Xs producing a tuple of Ys. One can swiftly verify that $(-)^*$ obeys the functor axioms: $(g \circ f)^* = g^* \circ f^*$ and $\mathrm{id}_X^* = \mathrm{id}_{X^*}$. We can then construct free monoids from the free-forgetful

[4] If the reader is familiar with *graded* structures, this definition makes it apparent that $(X^*, +\!\!+, \iota_0)$ is a graded monoid with the coproduct preserving the degree. Then we have $X^{\otimes m} \otimes X^{\otimes n} \cong X^{\otimes(m+n)}$, $\forall\, m, n \in \mathbb{N}$.

adjunction $F^L \dashv U^L$ with

$$\begin{tikzcd}[row sep=small]
F^L : \boldsymbol{C} \arrow[r] & \mathbf{Mon}(\boldsymbol{C}) & & U^L : \mathbf{Mon}(\boldsymbol{C}) \arrow[r] & \boldsymbol{C} \\
X \arrow[r, mapsto] & (X^*,\, +\!\!+,\, \iota_0) & & (X,\, \mu_X,\, \vec{1}_X) \arrow[r, mapsto] & X \\
X \arrow[dd, "h"'] & (X^*,\, +\!\!+,\, \iota_0) \arrow[dd, "h^*"] & \dashv & (X,\, \mu_X,\, \vec{1}_X) \arrow[dd, "h"'] & X \arrow[dd, "h"] \\
\arrow[r, phantom, "\mapsto"] & & & \arrow[r, phantom, "\mapsto"] & \\
Y & (Y^*,\, +\!\!+,\, \iota_0) & & (Y,\, \mu_Y,\, \vec{1}_Y) & Y
\end{tikzcd}$$

where the unit $\iota_0 : 1_{\boldsymbol{C}} \to X^*$ is the 0-th coproduct injection sending the $1_{\boldsymbol{C}}$ to $X^{\otimes 0} = 1_{\boldsymbol{C}}$: the empty product. Multiplication $+\!\!+ : X^* \otimes X^* \to X^*$ is a generalised concatenation: $X^{\otimes m} \otimes X^{\otimes n} \cong X^{\otimes m+n}$ defined by the calculation

$$X^* \otimes X^*$$

$=$ {by definition}

$$\left(\sum\nolimits_m X^{\otimes m}\right) \otimes \left(\sum\nolimits_n X^{\otimes n}\right)$$

$\overset{\sim}{\to}$ {applying the distributor (3.1.1)}

$$\sum\nolimits_{m,n} X^{\otimes m} \otimes X^{\otimes n}$$

$\overset{\sim}{\to}$ {consolidating with $\varpi_X^{m,n}$, PROP. B.9-6}

$$\sum\nolimits_{m,n} X^{\otimes(m+n)}$$

$\to$ {injection of the coproduct X^* by ι_{m+n}}

$$X^*$$

As a composite,

$$\begin{aligned} +\!\!+ := X^* \otimes X^* &\xrightarrow{\varpi_X^{m,n} \circ d} \sum_{m,n} X^{\otimes(m+n)} \xrightarrow{\sum_{m,n} \iota_{n+m}} X^* \\ &= \left(\sum_{m,n} \iota_{n+m}\right) \circ \varpi_X^{m,n} \circ d. \end{aligned}$$

This means for $x_m \in X^{\otimes m}$ and $x_n \in X^{\otimes n}$, $x_m +\!\!+ x_n \in X^{\otimes(m+n)} \in X^*$ is formed by simply regrouping x_m and x_n into a single product object $x_{m,n} \in$

$X^{\otimes(m+n)}$ which has the m factors of x_m with the n factors of x_n appended to the right. Those data are then injected to X^*.

Mac Lane shows the adjunction $F^L \dashv U^L$ by defining η and demonstrating that it fulfills the universal mapping property of the adjunction unit [81, §VII.3, THM. 2]. Instead, I will briefly describe the unit and counit in analogy to the ones defined for $(\mathbf{\$et}, \times)$ in CONST. 3.4-3. The unit of the adjunction, $\eta = \iota_1$, an abstract insertion of generators provided by the natural coproduct injection. The counit extends a monoidal operation to evaluate iterated products by recursive use of the monoidal unit, analogous to (3.4.5), but now without the point-wise internals.

Given a monoid $A = (A, Ⓐ, \vec{a}_0) \in \mathbf{Mon}(\boldsymbol{C})$ and the free monoid $F^L U^L\, A = (A^*, \mathbin{+\!\!+}, \iota_0)$ the counit at coordinate A is

$$\varepsilon_A := \sum_{n\in\mathbb{N}} Ⓐ^{(n)}$$

where

$$Ⓐ^{(0)} := \vec{a}_0, \quad Ⓐ^{(1)} := \mathrm{id}_A, \quad Ⓐ^{(2)} := Ⓐ, \quad Ⓐ^{(n+1)} := (Ⓐ) \circ (Ⓐ^{n} \otimes \mathrm{id}_A).$$

The inductive definition of $Ⓐ^{n}$ is a bit ad-hoc and the whole operation will be naturally systematised later; and OBS. 3.5-4 pointing out chains of iterated functor application was an early hint at the process.

3.5.1 Monoid Actions

DEFINITION 3.5-11. Given a $\boldsymbol{C}$-monoid $M = (M, Ⓜ, e_M)$, a ***right*** M***-action*** is a tuple

$$\left(S,\ M,\ S \otimes M \xrightarrow{\rho} S\right)$$

where $(S,\ \rho)$ is an M-algebra that additionally preserves the monadic structure of M by observing commutativity of the diagrams

$$\begin{tikzcd}
S \otimes (M \otimes M) \arrow[r, "\mathrm{id}_S \otimes \textcircled{\scriptsize M}"] \arrow[d, "\alpha"'] & S \otimes M \arrow[dd, "\rho"] \\
(S \otimes M) \otimes M \arrow[d, "\rho \otimes \mathrm{id}_M"'] & \\
S \otimes M \arrow[r, "\rho"] & S
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
S \otimes 1_{\boldsymbol{C}} \arrow[r, "\mathrm{id}_S \otimes e_M"] \arrow[d, "\varrho"'] & S \otimes M \arrow[dl, "\rho"] \\
S &
\end{tikzcd}
.$$

When S can be safely inferred, we simply identify the right M-action with ρ.

NOTE. Since we only discuss right actions, let it be understood that an M-action is a right M-action.

OBSERVATION 3.5-12 ([32, §10.2, OBS. 8])**.** Let $(M,\ \textcircled{\scriptsize M},\ e_M)$ be a monoid with corresponding monad $(\hat{M},\ \mu,\ \eta)$ as per OBS. 3.5-9. Any $\boldsymbol{C}$-arrow $\rho :\ S\otimes M \to A$ is a monoid action if and only if

$$\begin{tikzcd}
\hat{M}^2 S \arrow[r, "\mu"] \arrow[d, "\hat{M}\rho"'] & \hat{M} S \arrow[d, "\rho"] \\
\hat{M} S \arrow[r, "\rho"] & S
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
S \arrow[r, "\eta"] \arrow[dr, equal] & \hat{M} S \arrow[d, "\rho"] \\
& S
\end{tikzcd}$$

commute. And, *these are exactly the conditions for ρ to be an $\hat{M}$-algebra.* Namely, associativity and unital conditions.

DEFINITION 3.5-13. An M-action homomorphism is a $\boldsymbol{C}$-arrow $h :\ S \to S'$ such that

$$\begin{tikzcd}
S \otimes M \arrow[r, "h \otimes \mathrm{id}_M"] \arrow[d, "\rho"'] & S' \otimes M \arrow[d, "\rho'"] \\
S \arrow[r, "h"] & S'
\end{tikzcd} \tag{3.5.7}$$

commutes.

OBSERVATION 3.5-14. Rehearsing (3.5.7) with the domains expressed through $\hat{M}$ gives

$$\begin{tikzcd}
\hat{M}S \arrow[r, "\hat{M}h"] \arrow[d, "\rho"'] & \hat{M}S' \arrow[d, "\rho'"] \\
S \arrow[r, "h"] & S'
\end{tikzcd}$$

it immediately follows from comparison to (3.3.4) that M-action homomorphisms are $\hat{M}$-algebra homomorphisms as well.

DEFINITION 3.5-15. The category of right M-actions in $\boldsymbol{C}$, $\mathbf{Act}^M(\boldsymbol{C})$ is comprised of

objects: all right M-actions in $\boldsymbol{C}$ and

arrows: all right M-action homomorphisms.

OBSERVATION 3.5-16. Monadic $\hat{M}$-algebras are exactly M-actions. From the point of view of systems theory, we are particularly interested in an adjoint situation between $\mathbf{Act}^{M^*}$ and the Lambek algebra category $\hat{M}$-$\mathbf{Alg}$, to be discussed in §3.6 where it will be more tangibly described in the context of automata.

3.6 CATEGORIFICATION OF CLASSICAL AUTOMATA

The definitions, calculations and results of this section are meant to form an interface from ordinary algebraic reasoning about automata, which I hope will be familiar to the reader, to the more abstract setting of CT. It is largely a modernised summary of a 1975 paper by Goguen [33], with a view toward abstracting further in later sections.

With that said, let us start with something familiar: a fairly typical definition of an automaton/machine/discrete system:

DEFINITION 3.6-1. A discrete, sequential ***input/state/output (I/S/O) system*** is specified as a tuple

$$\left(I,\ S,\ O,\ S\times I \xrightarrow{\delta} S,\ S \xrightarrow{r} O,\ s_0\right)$$

where

- I, S, O are sets of values for input, state and output, respectively;
- $\delta : S \times I \to S$ is the ***state transition function***, which encodes the input-evolution of the systems state, sometimes viewed with the additional structure of a *right action* of I on S;
- $r : S \to Y$ is the ***readout function***, mapping internal states to externally observable variables; and
- $s_0 \in S$ is the ***initial state*** describing the starting configuration of the system prior to any input driven transitions.

In the literature, this base definition reflects some commonalities but are often embellished with data and axioms based on the topic of discussion. For examples, sometimes the sets are required to be finite.

DEFINITION 3.6-2 (Preliminary)**.** Given a discrete sequential I/S/O system $\Sigma = (I, S, O, \delta, r, s_0)$, if S is a finite set and I, O are non-empty finite sets, then Σ is a ***Moore machine***.

Sometimes the sets are replaced with vector spaces and δ, r are linear maps giving discrete linear machines. Sometimes time-dynamics are included by appending some notion of time and replacing δ with difference or differential equations.

NOTE. Many definitions *do not* include an initial state, most prominently, Kálmán's [20, §1.4] and Arbib's [28] definitions of machines. This has deep algebraic (and philosophical) repercussions that will be exposed in this section and laid bare in the next and in CH. 5. It is related to the algebra/coalgebra duality, which extends to a duality of an I/S-system with initial state and an I/S/O-system without initial state.

Categorification[5] of discrete sequential I/S/O systems proceeds along the following lines,

[5] Trivia: the term *categorified* was coined by American mathematician and theoretical physicist Louis Crane, who is perhaps best known (at least to me) for his contributions in topological quantum field theory. Web-search "Barrett–Crane model" for some particulars.

- the sets I, S, O become objects in a biccc $\boldsymbol{C}$;
- δ and r become arrows in this category; and
- iteration of the state transitions is induced naturally as the action of the free monoid generated by the input object.

This will then lead to the generalisation of *reachability* and *behaviours.*

3.6.1 Transition domain functors and their algebras

Let $\boldsymbol{C}$ be a category, as described in §3.1.3, with closed monoidal structure and denumerable coproducts. Also let

$$\Sigma = \left(I,\ \ S,\ \ O,\ \ S \otimes I \xrightarrow{\delta} S,\ \ S \xrightarrow{r} O,\ \ \vec{s}_0 \right)$$

be an I/S/O system like DEF. 3.6-1, but with the sets I, S, O replaced by $\boldsymbol{C}$-objects and δ, r by $\boldsymbol{C}$-arrows. The initial state s_0 becomes a global element, $\vec{s}_0 : 1_{\boldsymbol{C}} \to S$.

OBSERVATION 3.6-3. We can immediately spot that $(S, \delta : \hat{I}S \to S)$ in Σ is an $\hat{I}$-algebra. The functor $\hat{I}$ should be regarded as "transforming the state space S into a new object, $\hat{I}S$ on which the dynamics act [31, p. 315].

3.6.2 State Stepping *&* Monoid Actions

An ordinary algebraic treatment of sequential machine theory may include the formulation of the "right-action" of the free monoid of input on the set of states. This extends the machinery in Σ to operate on sequences of input. This idea terms of sets and functions is this: when given a sequence of input $w \in [\, i_0, i_1, \ldots\,]$, we have to hand-crank the iteration of Σ by accumulating state transitions in the first argument of δ:

$$s' = \cdots\delta(\cdots\delta(\delta(s_0, i_0), i_1), \ldots)\cdots \tag{3.6.8}$$

at arbitrary depth. Organised as a right action

$$\big(S, (I^*, +\!\!+, \iota_0), \cdot : \hat{I}^* S \to S) \big),$$

we can write this more simply as

$$s' = s_0\,[\,i_0,\, i_1,\, \dots\,].$$

Iteration still underpins this cleaner notation as we work through the list:

$$s' = (s_0\, i_0)\,[\,i_1,\, \dots\,] = ((s_0\, i_0)\, i_2)\,[\dots],$$

but it is intentionally obscured as an automaticity by the juxtaposition of s_0 with a word.

Section 3.5.1 gives a categorification of right monoid actions. By extension of OBS. 3.5-12, each monoid $(I^*,\ +\!\!+,\ \iota_0)$ gives an equivalent monad $(\hat{I}^*,\ \mu,\ \eta)$, implying an equivalence between $\mathbf{Act}^{I^*}$ and the Eilenberg-Moore category $\boldsymbol{C}^{(\hat{I}^*,\,\mu,\,\eta)}$. But that section ends with an unsupported claim that even the categories $\mathbf{Act}^{I^*}$ and $\hat{I}$-$\mathbf{Alg}$ (the category of I-object-algebras) are adjoint. Here is the place we elucidate that situation in earnest.

THEOREM 3.6-4 ([33, THM. 3.2])**.** If $\boldsymbol{C}$ is a closed symmetric monoidal category with denumerable coproducts, then

$$\mathbf{Act}^{I^*} \cong \hat{I}\text{-}\mathbf{Alg}$$

for all $I \in \boldsymbol{C}$.

Proof. CF. [33][6]. □

3.6-5 The proof of this theorem is in simply witnessing the adjunction with a pair of functors and then showing they are mutually inverse. We shall not herein demonstrate the inverse relationship those functors (thought it is straightforward to do so) but we will go into the details of the functors themselves as they relate to automata theory. Let us denote them,

$$(\bar{-}) :\ \mathbf{Act}^{I^*} \to \hat{I}\text{-}\mathbf{Alg} \quad \text{and} \quad (-)^+ :\ \hat{I}\text{-}\mathbf{Alg} \to \mathbf{Act}^{I^*}.$$

[6] Goguen's [33] spelled $\hat{I}$-$\mathbf{Alg}$ as $\mathbf{Mond}^I$. What I call object-algebras, Goguen called "I-monadic algebras", but in that time he used term "monad" differently. At that point in history, the more common term for monad was "triple", though the term "monad" was introduced by Bénabou in 1967.

In the first, we can pre-compose a given ρ with $\mathrm{id}_S \otimes \iota_1$ which adapts the domain, restricting from $S \otimes I^*$ to $S \otimes I$ as

$$S \otimes I \xrightarrow{\mathrm{id}\,\otimes\,\iota_1} S \otimes I^* \xrightarrow{\;\rho\;} S. \tag{3.6.9}$$

More explicitly we define,

$$\begin{aligned} (\bar{-}) :\ & \mathbf{Act}^{I^*} \to \hat{I}\text{-}\mathbf{Alg} \\ & \rho \ \mapsto\ \bar{\rho} \quad := \quad \rho \circ (\mathrm{id}_S \otimes \iota_1) \end{aligned}$$

$$\begin{tikzcd} \rho \arrow[d, "h"'] & \mapsto & \bar{\rho} \arrow[d, "\bar{h}"] & \text{which are both just} & S \arrow[d, "h"'] & \in \boldsymbol{C}. \\ \rho' & & \bar{\rho}' & & S' & \end{tikzcd}$$

If we think concretely of sets, then (3.6.9) maps thusly: $(s,\, i) \mapsto (s,\, [i]) \mapsto \rho(s, [i])$. The action on $\mathbf{Act}^{I^*}$-arrows is practically identity since they are both just $\boldsymbol{C}$-arrows between state objects.

The second functor, $(-)^+ : \hat{I}\text{-}\mathbf{Alg} \to \mathbf{Act}^{I^*}$ is slightly more involved because we need an inductively defined family of morphisms to enact iteration for every $I^{\otimes n}$ in I^*.

$$\begin{aligned} (-)^+ :\ & \hat{I}\text{-}\mathbf{Alg} \to \mathbf{Act}^{I^*} \\ & \delta \ \mapsto\ \delta^+ \end{aligned}$$

$$\begin{tikzcd} \delta \arrow[d, "h"'] & \mapsto & \delta^+ \arrow[d, "h^+"] & \text{which are both just} & S \arrow[d, "h"'] & \in \boldsymbol{C}. \\ \delta' & & (\delta')^+ & & S' & \end{tikzcd}$$

The maps on arrows is nearly identity since h is just a $\boldsymbol{C}$-arrow from S to S'.

The mapping from $\delta \mapsto \delta^+$ is defined inductively. Let

$$c_0 : S \otimes 1_{\boldsymbol{C}} \to S := \varrho \ \text{ (the right unitor),}$$

as the base case of the induction. Then c_{n+1} is composed from the calculation

$$\begin{aligned} & S \otimes I^{\otimes(n+1)} \\ = \quad & \{\text{applying } \mathrm{id}_S \otimes \left(\varpi_I^{n,1}\right)^{-1}\text{, with } \varpi \text{ from } \textsc{prop. } \text{B.9-6}\} \end{aligned}$$

$$
\begin{array}{ll}
 & S \otimes \left(I^{\otimes n} \otimes I\right) \\
\xrightarrow{\sim} & \quad \{\text{application of the associator, } \alpha\} \\
 & \left(S \otimes I^{\otimes n}\right) \otimes I \\
\to & \quad \{\text{application of } c_n \otimes \mathrm{id}_I\} \\
 & S \otimes I \\
\to & \quad \{\text{application of } \delta\} \\
 & S
\end{array}
$$

Thereby,

$$c_{n+1} : S \otimes I^{\otimes(n+1)} \to S := \delta \circ (c_n \otimes \mathrm{id}_I) \circ \alpha \circ (\mathrm{id}_S \otimes \left(\varpi_I^{n,1}\right)^{-1}),$$

and thus we define

$$\delta^+ := \sum_{n \in \mathbb{N}} c_n : S \otimes I^* \to S.$$

The inductive definition applied to functional structure equates to recursion. This recursion provides a mathematical "call stack" that holds the state of the system as the state of the calculation, as in (3.6.8).

NOTE. This pattern will be a leitmotif, rehearsed thought the thesis at different levels of abstraction.

3.6.3 Equipping initial state

Many mathematical definitions of a state system (as in DEF. 3.6-1) parcel an initial state with the dynamical machinery. To have an equivalence between those classical systems and the categorical definition, we need to enshrine initial state within the categorical data. This is done in two parts: (1) shifting to pointed object algebras, $\hat{I}_\bullet$-algebras by appending a global elements $\vec{s}_0 \in \boldsymbol{C}(1_{\boldsymbol{C}}, S)$ and requiring the homomorphisms preserve them, yielding the category $\hat{I}_\bullet$-**Alg**; and (2) likewise embellishing the category **Act**$^-$ with a pointed structure: **Act**$^-_\bullet$; This will once again lead to an isomorphism of categories formalising the equivalence of single-step and iterative state progression, but now bringing along the initial state.

DEFINITION 3.6-6. For any $I \in \boldsymbol{C}$ let $\hat{I}_{\bullet}$ be the functor $\hat{I}_{\bullet} - := 1_{\boldsymbol{C}} + \hat{I}\, -$. An algebra of this functor satisfies

$$\begin{tikzcd}
1_{\boldsymbol{C}} + \hat{I}\, S \arrow[r, "1_{\boldsymbol{C}} + \hat{I}\, h"] \arrow[d, "\vec{s}_0 \triangledown \delta"] & 1_{\boldsymbol{C}} + \hat{I}\, S' \arrow[d, "\vec{s}'_0 \triangledown \delta'"'] \\
S \arrow[r, "h"] & S'
\end{tikzcd}$$

where $\vec{s}_0 : 1_{\boldsymbol{C}} \to S$ and $\vec{s}'_0 : 1_{\boldsymbol{C}} \to S'$ are global elements. This is equivalent to the simultaneous commutativity of the pair of diagrams:

$$\begin{tikzcd}
 & 1_{\boldsymbol{C}} \arrow[dl, "\vec{s}_0"'] \arrow[dr, "\vec{s}'_0"] & \\
S \arrow[rr, "h"] & & S'
\end{tikzcd} \quad \text{and} \quad \begin{tikzcd}
\hat{I}\, S \arrow[r, "\hat{I}\, h"] \arrow[d, "\delta"] & \hat{I}\, S' \arrow[d, "\delta'"'] \\
S \arrow[r, "h"] & S'
\end{tikzcd}$$

Notice that this gives a means to appoint a state transition map in Σ with its initial state. We can also append basepoints to the trio of definitions for M-actions.

DEFINITION 3.6-7. Given a $\boldsymbol{C}$-monoid M, a ***pointed right*** M***-action*** is a tuple

$$\left(S,\ S \otimes M \xrightarrow{\rho} S,\ \vec{s}_0 \right),$$

where (S, ρ) is a right M-action and $\vec{s}_0$ is a global element of S called the ***base point***.

DEFINITION 3.6-8. A pointed M-action homomorphism is an M-action homomorphism which preserves base points by additionally observing the commutativity of

$$\begin{tikzcd}
 & 1_{\boldsymbol{C}} \arrow[dl, "\vec{s}_0"'] \arrow[dr, "\vec{s}'_0"] & \\
S \arrow[rr, "h"] & & S'
\end{tikzcd} .$$

DEFINITION 3.6-9. The category $\mathbf{Act}_{\bullet}^{M}(\boldsymbol{C})$ of *pointed* right M-actions in $\boldsymbol{C}$, is comprised of

objects: all *pointed* right M-actions in $\boldsymbol{C}$ and
arrows: all *pointed* right M-action homomorphisms.

The pointed equivalent of THM. 3.6-4 follows.

THEOREM 3.6-10 ([33, THM. 3.3][7]). If $\boldsymbol{C}$ is a closed symmetric monoidal category with countable coproducts then

$$\mathbf{Act}_{\bullet}^{I^*} \cong \hat{I}_{\bullet}\text{-}\mathbf{Alg}$$

for any $I \in \boldsymbol{C}$.

The isomorphism relating $\hat{I}_{\bullet}$-**Alg** and $\mathbf{Act}_{\bullet}^{I^*}$, is substantially the same as THM. 3.6-4, but with base-points carried along. We can immediately recognise that, by definition, $(I^*,\ \bar{\mathbin{+\!\!+}},\ \iota_0)$ is a pointed I^*-action and that $(I^*,\ \iota_0 \triangledown \bar{\mathbin{+\!\!+}})$ is a pointed object algebra. Denote $\bar{\mathbin{+\!\!+}}_{\bullet} := \iota_0 \triangledown \bar{\mathbin{+\!\!+}}$.

THEOREM 3.6-11 ([33, THM. 3.4]). In the category $\hat{I}_{\bullet}$-$\mathbf{Alg}(\boldsymbol{C})$, the algebra $(I^*,\ \bar{\mathbin{+\!\!+}}_{\bullet})$ is an initial object.

Proof. By definition, $(I^*,\ \bar{\mathbin{+\!\!+}}_{\bullet})$ is initial in $\hat{I}_{\bullet}$-**Alg** if an only if there exist unique $\hat{I}$-algebra homomorphisms

$$(I^*,\ \bar{\mathbin{+\!\!+}}_{\bullet}) \to A \quad \text{for all} \quad A = (S,\ \vec{s}_0 \triangledown \delta) \in \hat{I}_{\bullet}\text{-}\mathbf{Alg}\ .$$

The nature of this arrow can only depend on the behaviour of $\vec{s}_0 \triangledown \delta$, so the notation will centre on that: call it $(\!|\vec{s}_0 \triangledown \delta|\!)$. [8]

Before describing $(\!|\vec{s}_0 \triangledown \delta|\!)$, note that we can provide an inverse for $\bar{\mathbin{+\!\!+}}$ as

$$\bar{\mathbin{+\!\!+}}_{\bullet}^{-1} : I^* \to 1_{\boldsymbol{C}} + I^* \otimes I$$

[7] Goguen's [33] spells $\hat{I}_{\bullet}$-**Alg** as $\mathbf{Mod}^{I}$.

[8] I will later define $(\!|-|\!)$ more generally as a universal arrow called a *catamorphism*, which are the unique algebra homomorphisms from any initial functor-algebra.

defined piecewise as follows. Let

$$c_0 \coloneqq (\iota_0)_\bullet : I^{\otimes 0} \to 1_{\boldsymbol{C}} + I^* \otimes I,$$

where $(\iota_0)_\bullet$ is the o-th coproduct injection into the sum $1_{\boldsymbol{C}} + I^* \otimes I$. This merely maps the empty product to the term $1_{\boldsymbol{C}}$. Let

$$c_1 \coloneqq I^{\otimes 1} = I \xrightarrow{\ell_I^{-1}} 1_{\boldsymbol{C}} \otimes I \xrightarrow{(\iota_0)_{I^*} \otimes \mathrm{id}} I^* \otimes I \xrightarrow{(\iota_1)_\bullet} 1_{\boldsymbol{C}} + I^* \otimes I$$

where $(\iota_0)_I^*$ is the unit of $(I^*,\ +\!\!+,\ \iota_0)$ with the subscript to disambiguate from $(\iota)_\bullet$. Then for $n \in \mathbb{N}_{\geq 2}$ let

$$c_n \coloneqq I^{\otimes n} = I^{\otimes n-1} \otimes I \xrightarrow{(\iota_{n-1})_{I^*} \otimes \mathrm{id}} I^* \otimes I \xrightarrow{(\iota_1)_\bullet} 1_{\boldsymbol{C}} + I^* \otimes I.$$

Finally,

$$\bar{+\!\!+}_\bullet^{-1} \coloneqq \sum_{n \in \mathbb{N}} c_n.$$

(This inverse is a generalisation of the *safe head/tail* list deconstruction in functional programming.) Specialising (3.3-2) to our particular case,

$$\begin{tikzcd}
\hat{I}_\bullet I^* \ = \ 1_{\boldsymbol{C}} + I^* \otimes I \arrow[r, "\hat{I}_\bullet (\!| \vec{s}_0 \triangledown \delta |\!)"] \arrow[d, shift left, "\bar{+\!\!+}_\bullet"] & 1_{\boldsymbol{C}} + S \otimes I \ = \ \hat{I}_\bullet S \arrow[d, "\vec{s}_0 \triangledown \delta"] \\
I^* \arrow[r, "(\!| \vec{s}_0 \triangledown \delta |\!)"] \arrow[u, shift left, "\bar{+\!\!+}_\bullet^{-1}"] & S
\end{tikzcd},$$

the addition of the inverse $\bar{+\!\!+}_\bullet^{-1}$, allows us to chase through a definition of $(\!| \vec{s}_0 \triangledown \delta |\!)$ as

$$\begin{aligned}
(\!| \vec{s}_0 \triangledown \delta |\!) &= \delta \circ \hat{I}\, (\!| \vec{s}_0 \triangledown \delta |\!) \circ \bar{+\!\!+}_\bullet^{-1} \\
&= \delta \circ \big(1_{\boldsymbol{C}} + (\!| \vec{s}_0 \triangledown \delta |\!) \otimes \mathrm{id}_I\big) \circ \bar{+\!\!+}_\bullet^{-1}.
\end{aligned}$$

This unambiguously prescribes $(\!| \vec{s}_0 \triangledown \delta |\!)$, which is only enabled by the isomorphism $\hat{I}_\bullet \cong I^*$ mediated by $\bar{+\!\!+}_\bullet$ and its inverse. (We will later recognise this more generally as an application of *Lambek's lemma*.) Conclude therefore that $(\!| \vec{s}_0 \triangledown \delta |\!)$ is determined uniquely and hence, $(I^*,\ \bar{+\!\!+}_\bullet)$ is initial. □

The intuition here is that these unique $\hat{I}_{\bullet}$-algebra homomorphisms, call them ***$\hat{I}_{\bullet}$-folds*** because they *fold* a structure map over a product of input data in I^*, recursively applying it starting from s_0. Using $\boldsymbol{C} = \$\mathbf{et}$, much more intuition will be provided in §3.8. But for now, if we take a list $[\, i_0, i_1, i_2 \,] \in I^*$ and then

$$(\!|\vec{s}_0 \triangledown \delta|\!)[\, i_0, i_1, i_2 \,] = \delta(\delta(\delta(s_0, i_0), i_1), i_2), \tag{3.6.10}$$

is just the iteration of δ carrying the state of the calculation in the call-stack along the first argument of δ. More generally, given a machine Σ in $\boldsymbol{C}$, these catamorphisms give an arrow $I^* \to S$, relating sequenced input to the state object by recursive composition of δ with $\vec{s}_0$ as a base case.

DEFINITION 3.6-12. A $\hat{I}_{\bullet}$-algebra $(S, \vec{s}_0 \triangledown \delta)$ is called ***reachable*** just in case $(\!|\vec{s}_0 \triangledown \delta|\!)$ is an epimorphism in $\boldsymbol{C}$.

In $\$\mathbf{et}$, epimorphisms are exactly surjections. So one can connect their intuitions of *reachability* with the notion that a system is reachable if all states are reachable by at least one input sequence. But it is not that case that epimorphisms are the analogue of surjections in every category. Many papers on this topic start with structuring $\boldsymbol{C}$ with a formal epi/mono factorisation system, which is equipment I have avoided.

COROLLARY 3.6-13. Considering the theorems

- 3.6-10 showing that $\hat{I}_{\bullet}$-algebras (and their homomorphisms) are in isomorphism with pointed I-actions (and their homomorphisms); and
- 3.6-11 sowing that I^* is initial in $\hat{I}$-**Alg**;

together, conclude that $(I^*, \mathbin{+\!\!+}, \iota_0)$ is also initial in $\mathbf{Act}^{I}_{\bullet}$.

3.6.4 Goguen machines & behaviour

The system Σ from the previous chapter is now formally defined as a Goguen machine.

DEFINITION 3.6-14. A ***Goguen machine*** is an I/S/O-system in $\boldsymbol{C}$ specified as a tuple

$$\left(I,\ S,\ O,\ S \otimes I \xrightarrow{\delta} S,\ S \xrightarrow{r} O,\ \vec{s}_0\right)$$

where

- I, S, O are $\boldsymbol{C}$-objects marking input, state and output, respectively;
- $(S, \vec{s}_0 \triangledown \delta)$ is a $\hat{I}_\bullet$-algebra in $\boldsymbol{C}$;
- $r : S \to O$ is a $\boldsymbol{C}$-arrow called the *readout* arrow.

Such a system is called *reachable* just in case its $\hat{I}_\bullet$-algebra is reachable.

For the remainder of this section, Goguen machines will just be called *machines.*

Morphisms of machines will involve arrows between the input, state and output objects preserving the basepoint, $\hat{I}$-algebra structure and the readout maps.

DEFINITION 3.6-15. Given $\boldsymbol{C}$-machines

$$\Sigma = (I, S, O, \delta, r, \vec{s}_0)$$

and

$$\Sigma' = (I', S', O', \delta', r', \vec{s}_0'),$$

a ***homomorphism of machines*** $\Sigma \to \Sigma'$ is a triple of $\boldsymbol{C}$-arrows

$$\left(I \xrightarrow{f} I',\ S \xrightarrow{g} S',\ O \xrightarrow{h} O'\right)$$

such that

$$\begin{tikzcd} & 1_{\boldsymbol{C}} \arrow[dl, "\vec{s}_0"'] \arrow[dr, "\vec{s}_0'"] & \\ S \arrow[rr, "g"] & & S' \end{tikzcd} \quad , \quad \begin{tikzcd} S \otimes I \arrow[r, "g \otimes f"] \arrow[d, "\delta"'] & S' \otimes I \arrow[d, "\delta'"] \\ S \arrow[r, "g"] & S' \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} S \arrow[r, "g"] \arrow[d, "r"'] & S' \arrow[d, "r'"] \\ O \arrow[r, "h"] & O' \end{tikzcd}$$

commute.

DEFINITION 3.6-16. The category of Goguen machines in $\boldsymbol{C}$, $\mathbf{Mach}(\boldsymbol{C})$ is comprised of

objects: all Goguen machines in $\boldsymbol{C}$, and

arrows: all machine homomorphisms.

There are also the full subcategories $\mathbf{M}^I \hookrightarrow \mathbf{Mach}$ of reachable I-machines, with $f : I \to I' = \mathrm{id}_I$ in all homomorphisms.

I/O-behaviour exists independently in $\boldsymbol{C}$: after all, state is just an artefact of modelling—but input/output relationships are empirically observable. From that philosophical point comes the notion of representation of an I/O-behaviour by an I/S/O-system.

DEFINITION 3.6-17. A ***I/O-behaviour***, or simply ***behaviour*** in $\boldsymbol{C}$ is a $\boldsymbol{C}$-arrow $y : I^* \to O$.

DEFINITION 3.6-18. A homomorphism of behaviours in $\boldsymbol{C}$ is a pair of arrows (f, h) such that

$$\begin{tikzcd}
I^* \arrow[r, "f^*"] \arrow[d, "y"'] & (I')^* \arrow[d, "y'"] \\
O \arrow[r, "h"] & O'
\end{tikzcd}$$

commutes.

DEFINITION 3.6-19. The category of behaviours in $\boldsymbol{C}$, $\mathbf{Beh}(\boldsymbol{C})$ is comprised of

objects: all behaviours in $\boldsymbol{C}$ and

arrows: all behaviour homomorphisms.

There is also the full subcategory $\mathbf{B}^I(\boldsymbol{C}) \hookrightarrow \mathbf{Beh}(\boldsymbol{C})$ of behaviours with domain I^*.

Given a Goguen machine we can straightforwardly assemble its external behaviour from its constituents. This fact leads to a functor from $\mathbf{Mach}$ to $\mathbf{Beh}$.

DEFINITION 3.6-20. Given a $\boldsymbol{C}$-machine Σ, the functor

$$\begin{aligned} B :\ \mathbf{Mach}(\boldsymbol{C}) &\to \mathbf{Beh}(\boldsymbol{C}) \\ \Sigma &\mapsto r \circ (\!|\vec{s}_0 \triangledown \delta|\!) :\ I^* \to O \\ (f,\, g,\, h) &\mapsto (f,\, h) \end{aligned}$$

is the ***external behaviour*** functor. In the case of reachable machines, the functor restricts to $B :\ \mathbf{M}^I \to \mathbf{B}^I$.

The inner workings of the calculation are mostly carried out by the catamorphism $(\!|\vec{s}_0 \triangledown \delta|\!)$. Recall, most crucially that the catamorphism internally encodes the state transition *and* the initial state. Thinking in **\$et**, the function $(\!|\vec{s}_0 \triangledown \delta|\!)$ iterates state transitions along a given input sequence $w \in I^*$ so that $(\!|\vec{s}_0 \triangledown \delta|\!)(w) \in S$ is the resultant state. The readout function then produces the output. Since the state transitions are all performed internally to the expression, no state information is ever externally available and $B\,\Sigma$ only produces external observables.

DEFINITION 3.6-21. Given a behaviour y, it is said that a machine Σ is a ***realisation of*** y just in case $B\Sigma = y$.

We then come to the questions of identification and realisation we know from elementary systems theory that state representations are non-unique. So if we are handed a behaviour y, we can expect there may be many machines that represent it equivalently.

3.6.5 $\hat{I}_\bullet$-algebra structure on $I^* \multimap O$

Since $\boldsymbol{C}$ is Cartesian closed, it has objects internally that encode behaviours. Using the intuition of **\$et**, these objects are function spaces. We can therefore think of the objects of $\mathbf{B}^I$ as *elements* of $I^* \multimap O$. More generally,

$$\mathrm{Ob}\,\mathbf{B}^I \cong \boldsymbol{C}(1_{\boldsymbol{C}},\, I^* \multimap O)$$

with

$$\mathbf{B}^I(y,\, y') \cong \boldsymbol{C}(I^* \multimap O,\, I^* \multimap O').$$

Diagrammatically, given $(\mathrm{id}_{I^*}, h) \in \mathbf{B}^I(y, y')$ we have commutativity of

$$\begin{tikzcd}
y \arrow[dd, "{(\mathrm{id}, h)}"] & & & I^* \multimap Y \arrow[dd, "\mathrm{id} \multimap h"] \\
& 1_{\boldsymbol{C}} \arrow[urr, "\vec{y}"] \arrow[drr, "\vec{y}'"'] & & \\
y' & & & I^* \multimap Y'
\end{tikzcd}$$

But much more is true: we have pointed algebras in $\boldsymbol{C}$ so we have important structure playing out in $I^*_\bullet$-**Alg**. Given a $y \in \mathbf{B}^I$, we can always get the output resulting from null input. Define ev_B by the composite

$$ev_B := (I^* \multimap O) \xrightarrow{\varrho^{-1}} (I^* \multimap O) \otimes 1_{\boldsymbol{C}} \xrightarrow{\mathrm{id} \otimes \iota_0} (I^* \multimap O) \otimes I^* \xrightarrow{ev} O.$$

If there were an underlying machine $y = BM$ then $ev_B(BM) \in O$ is the output of the internal initial state. But there need not be an underlying machine and behaviours on their own have some notion of a starting configuration.

Thinking in $\boldsymbol{C} = \mathbf{Set}$, we might imagine that starting with a chosen behaviour y and input $i \in I$, then there is a behaviour y' such that

$$y'([] \mathbin{+\!\!+} w) = y([i] \mathbin{+\!\!+} w) \qquad \text{for all} \qquad w \in I^*. \tag{3.6.11}$$

The idea here is that there is some notion of a *continuation* of evolutionary behaviour. For two $\mathbf{Set}$-systems

$$y = B(I, S, O, \delta, r, \vec{s}_0) \ \text{ and } \ y' = B(I, S, O, \delta, r, \vec{s}_0'),$$

if (3.6.11) holds it means $s_0' = \delta(s_0, i)$. This is suggestive of an $\hat{I}_\bullet$-algebra structure on behaviours, which allows a behaviour to take input and produce a continuation of the behaviour.

A base point for a behaviour is given as a global element (a point)

$$\vec{y} : 1_{\boldsymbol{C}} \to (I^* \multimap O),$$

representing the starting configuration which can be evolved successively by input through an $\hat{I}$-algebra

$$\varphi : \ (I^* \multimap O) \otimes I \to (I^* \multimap O).$$

with a $\hat{I}_\bullet$ algebra arising from the combination $\vec{y} \triangledown \varphi$.

We can specialise the commuting diagrams for $\hat{I}_\bullet$-algebra homomorphisms to the present case where we are only transforming along O:

$$\begin{tikzcd}
\hat{I}_\bullet (I^* \multimap O) \arrow[r, "\hat{I}_\bullet(I^* \multimap h)"] \arrow[d, "\vec{y} \triangledown \varphi"'] & \hat{I}_\bullet (I^* \multimap O') \arrow[d, "\vec{y}' \triangledown \varphi'"] \\
I^* \multimap O \arrow[r, "I^* \multimap h"] & I^* \multimap O'
\end{tikzcd} \tag{3.6.12}$$

The structure map φ can be constructed from the arrows we have identified in $\boldsymbol{C}$. That said, the following proposition gives a tool to ease the definition.

PROPOSITION 3.6-22.[9] Since $X \multimap Y \cong \boldsymbol{C}(X, Y)$ for all X, $Y \in \boldsymbol{C}$, and since a category axiomatically contains all composite arrows, there is an arrow for ***composition of internal homs***,

$$(B \multimap C) \otimes (A \multimap B) \xrightarrow{(\diamond)} (A \multimap C)$$

that makes the following triangle commute:

$$\begin{tikzcd}
(B \multimap C) \otimes (A \multimap B) \otimes A \arrow[r, "(\diamond) \otimes \mathrm{id}"] \arrow[dr, "ev \circ (\mathrm{id} \otimes ev)"'] & (A \multimap C) \otimes A \arrow[d, "ev"] \\
& C
\end{tikzcd}$$

(This is evident if $\boldsymbol{C}$ is regarded as enriched in $\mathbb{S}$**et**.)

Now we define φ by the calculation

$$\begin{aligned}
&\mathrm{dom}\,\varphi = (I^* \multimap O) \otimes I \\
\to \quad &\{\text{injecting } I \text{ into } I^* \text{ with } (\mathrm{id} \otimes \iota_1)\}
\end{aligned}$$

[9] Thanks to FShrike for helping me work this through. See `https://math.stackexchange.com/a/4749826/49718`.

$$\begin{aligned}
& (I^* \multimap O) \otimes I^* \\
\to \quad & \{\text{lambda-lifting } I^* \text{ with curried } \mathbin{+\!\!+}\text{: } (\mathrm{id} \otimes \lambda\mathbin{+\!\!+})\} \\
& (I^* \multimap O) \otimes (I^* \multimap I^*) \\
\to \quad & \{\text{composing hom-objects with } (\diamond)\} \\
& I^* \multimap O.
\end{aligned}$$

So in point-free style we write

$$\varphi \coloneqq (\diamond) \circ (\mathrm{id} \otimes \lambda\mathbin{+\!\!+}) \circ (\mathrm{id} \otimes \iota_1) = (\diamond) \circ \big(\mathrm{id} \otimes (\lambda\mathbin{+\!\!+} \circ \iota_1)\big).$$

This φ gives a means of stepping forward a behaviour on a per-input basis, and when appointed with a base, $\vec{y} \triangledown \varphi$ gives a $\hat{I}_{\bullet}$-algebra as per (3.6.12).

PROPOSITION 3.6-23 ([33, PROP. 4.1])**.** The assignment of an $\hat{I}_{\bullet}$-algebra structure on $I^* \multimap O$ to a behaviour $\boldsymbol{C}(I^*, O)$ is functorial:

$$\begin{aligned}
\mathbf{B}^I &\to \hat{I}_{\bullet}\text{-}\mathbf{Alg} \\
y &\mapsto (I^* \multimap O,\ \vec{y} \triangledown \varphi) \\
(\mathrm{id}, h) &\mapsto I^* \multimap h
\end{aligned}$$

The proof can be found in [33], though the current presentation departs from Goguen, who (probably for the sake of ease) decided to define φ^+ instead of φ implicitly relying on THM. 3.6-10 to carry over his results. The claim is the same: a given $(I,\ h :\ O \to O') :\ \mathbf{B}^I(y,\ y')$ there is a corresponding $\hat{I}_{\bullet}$-algebra homomorphism, where the diagram (3.6.12) commutes. This can be done from some straightforward diagram chasing facilitated by naturality conditions.

This is a fact that will become interesting later on. Using THM. 3.6-10, the extension of the functor into the category of right-actions,

$$\mathbf{B}^I \to \hat{I}_{\bullet}\text{-}\mathbf{Alg} \xrightarrow{(\bar{-})} \mathbf{Act}_{\bullet}^{I^*},$$

is a right adjoint to the forgetful $\mathbf{Act}^{I^*} \to \boldsymbol{C}$. That is to say $I^* \multimap O$ is *cofree*, generated by O.

OBSERVATION 3.6-24. Since, for a given monoid M, M-actions are exactly monadic $\hat{M}$-algebras, then conclude that $M^* \multimap O$ carries the cofree coalgebra. In later chapters this will play out as *initial algebra* vs. *terminal coalgebra*, but that is a reflection of the duality of free algebra vs. cofree coalgebra. An earlier remark pointed out that this duality relates to a discrete system having initial state vs. having output, and this is the first hint of the output side of the dual.

3.6.6 Behaviour & Running Machines

Recall (from THM. 3.6-11) that $(I^*,\ \iota_0 \triangledown \bar{+\!\!\!+})$ initial in $\hat{I}_{\bullet}$-**Alg**, and that catamorphisms (the unique arrows) give a way of processing sequential input by recursion of the algebra of the codomain, called *folding*. These catamorphisms may be regarded as *running* the machine. Endowing behaviours with an $\hat{I}_{\bullet}$-algebra structure makes them amenable to this mechanism as well. But catamorphisms on behaviours are morphisms from a free object to a cofree object, so some additional profundity is to be expected. We will obtain a beautiful set of relationships that relate machines and their behaviour.

Given a behaviour $y \in \mathbf{B}^I$, a behaviour catamorphism has the form

$$(\!|\vec{y} \triangledown \varphi|\!) :\ I^* \to (I^* \multimap O).$$

Let us relate this to the internals of a given a machine $\Sigma = (I,\ S,\ O,\ \delta,\ r,\ \vec{s}_0)$ where

$$y_\Sigma = B\Sigma = r \circ (\!|\vec{s}_0 \triangledown \delta|\!) :\ I^* \multimap O,$$

is the external behaviour of Σ.

First, given a machine Σ we can map the state object into the space of behaviours by currying the arrow

$$r \circ \delta^+ :\ S \otimes I^* \to O$$

to obtain

$$\check{r} \coloneqq \lambda(r \circ \delta^+) :\ S \to (I^* \multimap O).$$

It is immediately apparent that $\check{r}$ is a $\hat{I}_\bullet$-**Alg** homomorphism from the diagrams

$$\begin{tikzcd} & 1_{\boldsymbol{C}} \arrow[dl, "\vec{s}_0"'] \arrow[dr, "\vec{y}_\Sigma"] & \\ S \arrow[rr, "\check{r}"] & & I^* \multimap O \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} S \otimes I \arrow[r, "\check{r}\otimes\mathrm{id}"] \arrow[d, "\delta"] & (I^* \multimap O) \otimes I \arrow[d, "\varphi"] \\ S \arrow[r, "\check{r}"] & I^* \multimap O \end{tikzcd} .$$

(The commutativity of these diagrams can be proven straightforwardly from substitution of definitions, as per [33, p. 32].) I include these diagrams separately for additional clarity, but per the observation in DEF. 3.6-6, they are equivalent to the single diagram with algebras $\vec{s}_0 \triangledown \delta$ and $\vec{y}_\Sigma \triangledown \varphi$:

$$\begin{tikzcd} \hat{I}_\bullet\, S \arrow[r, "\hat{I}_\bullet\, \check{r}"] \arrow[d, "\vec{s}_0 \triangledown \delta"] & \hat{I}_\bullet\, (I^* \multimap O) \arrow[d, "\vec{y}_\Sigma \triangledown \varphi"] \\ S \arrow[r, "\check{r}"] & I^* \multimap O \end{tikzcd} . \tag{3.6.13}$$

To further elucidate the relationships including catamorphisms, we rewrite (3.6.13) to include the initial algebra $(I^*, \iota_0 \triangledown \text{ǂ}_\bullet)$ and behaviour evaluation morphism:

$$\begin{tikzcd} \hat{I}_\bullet\, I^* \arrow[r, "\hat{I}\, \llparenthesis \vec{s}_0 \triangledown \delta \rrparenthesis"] \arrow[d, shift left, "\text{ǂ}_\bullet"] \arrow[rr, bend left, "\hat{I}\, \llparenthesis \vec{y}_\Sigma \triangledown \varphi \rrparenthesis"] & \hat{I}_\bullet\, S \arrow[r, "\hat{I}\, \check{r}"] \arrow[d, "\vec{s} \triangledown \delta"] & \hat{I}_\bullet\, (I^* \multimap O) \arrow[d, "\vec{y}_\Sigma \triangledown \varphi"] & \\ I^* \arrow[r, "\llparenthesis \vec{s}_0 \triangledown \delta \rrparenthesis"] \arrow[u, shift left, "\text{ǂ}_\bullet^{-1}"] \arrow[rr, bend right, "\llparenthesis \vec{y}_\Sigma \triangledown \varphi \rrparenthesis"'] & S \arrow[r, "\check{r}"] & I^* \multimap O \arrow[r, "ev_B"] & O \end{tikzcd} . \tag{3.6.14}$$

Two particularly contrasting equations for $\llparenthesis \vec{y}_M \triangledown \varphi \rrparenthesis$ can be read from the squares:

$$\llparenthesis \vec{y}_\Sigma \triangledown \varphi \rrparenthesis = \varphi \circ \hat{I}_\bullet\, \llparenthesis \vec{y}_\Sigma \triangledown \varphi \rrparenthesis \circ \text{ǂ}_\bullet^{-1} \tag{3.6.15}$$

$$= \check{r} \circ \llparenthesis \vec{s}_0 \triangledown \delta \rrparenthesis \tag{3.6.16}$$

This is the natural connection between the intrinsic and the extrinsic evolution of I/O-behaviour. The first, (3.6.15) does not thread through a machine's internal transitions—it uses only the arrows $I^* \to O$ (encoded in $I^* \multimap O$) to fold over an input sequence—and hence transcending representation. The second, (3.6.16), conversely runs through every facet of the inner workings of Σ and shows that $(\!|\vec{y}_\Sigma \triangledown \varphi|\!)$ can be factored through $(\!|\vec{s}_0 \triangledown \delta|\!)$. In fact, the full composition along the bottom:

$$B\,\Sigma = I^* \xrightarrow{(\!|\vec{s}_0 \triangledown \delta|\!)} S \xrightarrow{\check{r}} I^* \multimap O \xrightarrow{ev_B} O$$

is called the intrinsic factorisation of $B\,\Sigma$ [33, THM. 4.5]. Most importantly, this gives a tangible notion of "running" an I/S/O machine.

At the beginning of the section, I pointed out that many definitions of "machine" that one finds in literature does not bundle internal state with the dynamical equipment. This has implications on what exactly one means by "behaviour". The arrow $\check{r}$ maps internal machine states to behaviour by our definition. If the system did not have an initial state installed by definition then $\check{r}$ would be the more fundamental notion.

Personally, I am partial to Willems' notion of behaviour [101], of which this section's definition is a simulacrum. Willems definition has an intrinsic notion of time, but we are presently using input to drive state transformation, not a time arrow.

3.7 FIXPOINTS OF ENDOFUNCTORS

The previous section builds a notion of I/S/O-systems using object algebras, pointed object algebras ($\hat{I}$-**Alg** and $\hat{I}_\bullet$-**Alg**) and monoid actions ($\mathbf{Act}_\bullet^{I^*}$). This section generalises some of those foundations. We use any polynomial functor to provide a generalisation of the tensor algebras $S \otimes I \to S$, which were the bedrock of Goguen machines. Arbib & Manes and others went to exquisite lengths to generalise machines in this way to capture a wider variety of sequential state processing machines [29, 40]. However, my motivation for viewing machines more broadly is that the underlying tooling of Arbib-Manes machines is fixpoints of functors, which Adámek was keen to point out [37]. This is an important point of crossover for us because fixpoints of functors are also used in theoretical computer science to model abstract datatypes [49], and codata-types [110]. At that level of analysis we are able to reconcile with computer scientific notions that lead to a semantics for processing linear data structures (like iterated products), leading to the next section, §3.8.

In THM. 3.6-11 the algebra $(I^*,\ \bar{\mathbin{+\!\!+}}_{\bullet})$ was shown to be initial in $\hat{I}_{\bullet}$-**Alg**. Along the way, it was shown that the structure map $\bar{\mathbin{+\!\!+}}_{\bullet}$ was only a rearranger of data, and therefore admitted an inverse. That fact was already enough to identify initiality, by revealing that algebra as a *fixpoint*.

The term "fixpoint" is evocative of the notion of the fixpoints of a function where the sequence

$$\left\{ f(z),\ f(f(z)),\ f(f(f(z))),\ \ldots \right\}$$

approaches a value x. In other words, there is some special value $x \in \operatorname{dom} f$ such that $f(x) = x$.

For an endofunctor F, what would it mean to have $FX = X$? Equality is too strong in the categorical context: we have no tooling to judge two objects as equal, but we can see isomorphism in the arrows. The fixpoint of a functor is an object X together with a pair of functions underwriting an isomorphism: $FX \cong X$. So, quite incredibly, it is both an F-algebra and F-coalgebra. Lambek's lemma connects fixpoints and initial algebras (or in dual, terminal coalgebras).

LEMMA 3.7-1 (Lambek [109, §10.5]). Given an endofunctor F if

$$(I, i :\ F\,I \to I)$$

is an initial object in F-**Alg** then then i is an isomorphism:

$$F\,I \cong I.$$

Using fixpoint operators introduced in NTN. 3.3-4, if an initial F-algebra is a fixpoint:

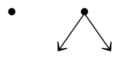
$$F(\mu F) \cong \mu F.$$

Dually, a terminal coalgebra is also a fixpoint. But even though a fixpoint is both an algebra and a coalgebra, initial algebras and terminal coalgebras do not necessarily coincide on the same fixpoint, but an adjunction may sit between them.

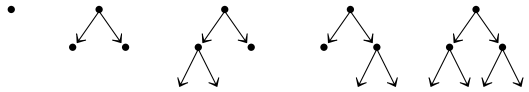
The thesis leans heavily on fixpoints of polynomial functors, because polynomial combinations of sets encode structure in data (products and coproducts) and transformations (exponentials). Recall from DEF. 3.5-6 that a monoid $M = (|M|, \text{Ⓜ}, e)$ in a monoidal category, let us use $\mathbb{S}\mathbf{et}$ for now, can be represented as a diagram (3.5.6) reprinted here:

$$\begin{tikzcd} 1_{\boldsymbol{C}} \arrow[r, "\vec{e}_M"] & M & M \otimes M \arrow[l, "\text{Ⓜ}"'] \end{tikzcd} , .$$

This admits a simple representation as an algebra of a nearly familiar polynomial functor $\mathsf{M} \coloneqq X \mapsto 1 + X^2$. Forget about the "algebra" part for now and focus only on M. The tree view[10] of M looks like this:



A value $m \in \mathsf{M}M$ is either () or a node with two M-elements as leaves (a pair of M-elements). Now consider $\mathsf{M}^2 = \mathsf{M} \circ \mathsf{M}$; the tree view looks like this:

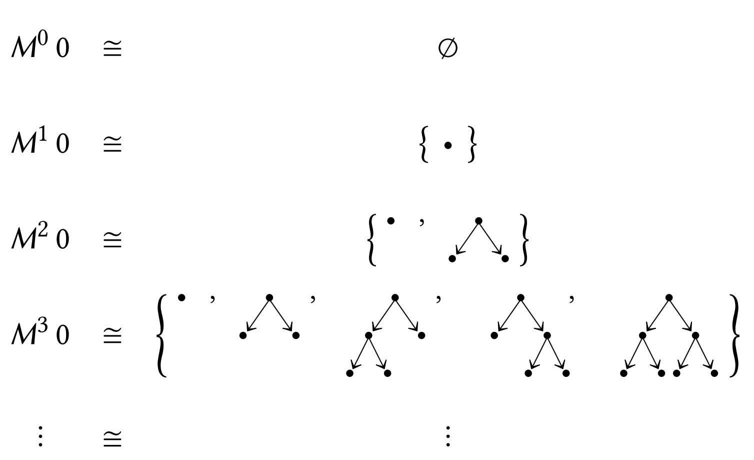


This is the abstract set of all depth-2 monoid expressions we can build. So $\mathsf{M}^2 M$ includes expressions like $(m_1 \bigcirc m_2) \bigcirc e$. Fixpoints of polynomial endofunctors capture (possibly infinite) repetitions of structure.

Returning to the notion of fixpoints, what would a fixpoint of M encode? The intuition from function fixpoints says that it may have something to do with the sequence $\{\mathsf{M}\,X, \mathsf{M}^2\,X, \mathsf{M}^3\,X, \ldots\}$, similar to the sequence from OBS. 3.5-4. We are not concerned with how elements of any

[10] See §3.2.

particular set interact with M. We want to collect the trees that M encodes: combinations of leaves ($\in 1$) and binary nodes. So we may well look at how M interacts with the empty set 0, initial in $\mathbf{Set}$, meaning it has exactly one (trivial) arrow in $\hom(0, A)$ for any set A. This will be important later. The sequence $\{M^0\,0,\ M^1\,0,\ M^2\,0,\ M^3\,0,\ \ldots\}$ collects the trees:



One might get the impression that the fixpoint of M is something like $S = \sum_{n\in\mathbb{N}} M^n\,0$, but this is not quite right. The elements of $M^{n-1}\,0$ are also in $M^n\,0$, which is not very parsimonious. That means that this S will not fulfil a universal property requiring that there is no simpler representation[11]. Set coproducts are a colimit of the diagram $\dot{\mathbf{2}} \to \mathbf{Set}$.[12] A colimit is the right approach here too, but with a more interesting diagram.

DEFINITION 3.7-2. Consider the linear order of integers, $\omega = (\mathbb{N}, \leq)$. It can be regarded as a category[13], $\boldsymbol{\omega}$, with $\mathrm{Ob}\,\boldsymbol{\omega} = \mathbb{N}$ and exactly one arrow in $\mathrm{Hom}(n_1, n_2)$ if and only if $n_1 \leq n_2$. Taken as the index for a diagram,

[11] Universal properties for a construction have a requirement that a solution comes along with morphisms so that all other candidates uniquely factor through it. This means the solution is optimal in terms of information content and all other candidates have either too much or too little information. Moreover, the factorising morphisms are often extremely useful in their own right, as will be the case here.

[12] See §B.9, p. 288.

[13] See §B.1-8, p. 267.

$D : \boldsymbol{\omega} \to \boldsymbol{C}$, the image of the diagram is a sequence of $\boldsymbol{C}$-objects and $\boldsymbol{C}$-arrows:

$$X_0 \xrightarrow{x_0} X_1 \xrightarrow{x_1} X_2 \xrightarrow{x_3} \cdots$$

called an ***ω-chain***.

DEFINITION 3.7-3. In a category with an initial object the ***initial sequence*** of an endofunctor F is a particular $\boldsymbol{\omega}$-chain:

$$0 \xrightarrow{!_{F\,0}} F\,0 \xrightarrow{F\,!_{F\,0}} FF\,0 \xrightarrow{FF\,!_{F\,0}} F^3\,0 \xrightarrow{F^3\,!_{F\,0}} \cdots$$

where $!_{F\,0} : 0 \to F\,0$ denotes the unique arrow from the initial object to $F\,0$.

The colimit of F's initial sequence is a terminal cocone[14] with μF at its coapex. That is

$$\mu F = \operatorname*{colim}_{n<\omega} F^n\,0,$$

and by Lambek's lemma we know it is a fixpoint. Denote the fixpoint's

$$\begin{aligned} \text{algebra}\ \ &(\mu F,\, \kappa : F\,\mu F \to \mu F) \\ \text{and coalgebra}\ \ &(\mu F,\, \kappa^{-1} : \mu F \to F\,\mu F), \end{aligned}$$

where the co/structure maps[15] are mutually inverse: $\kappa \circ \kappa^{-1} = \kappa^{-1} \circ \kappa = \mathrm{id}$. The structure map provides a way to trace back through the initial chain, forming the cocone faces:

$$\begin{tikzcd}
0 \arrow[r, "!_{F\,0}"] \arrow[d, "!_{\mu F}"] & F\,0 \arrow[r, "F!_{F\,0}"] \arrow[d, "F!_{\mu F}"] & FF\,0 \arrow[r, "FF!_{F\,0}"] \arrow[d, "FF!_{\mu F}"] & \cdots \arrow[r, "F^{n-1}!_{F\,0}"] \arrow[d, phantom, "\vdots"] & F^n\,0 \arrow[r, "F^n!_{F\,0}"] \arrow[d, "F^n!_{\mu F}"] & \cdots \\
\mu F & F\mu F \arrow[l, "\kappa"'] & FF\mu F \arrow[l, "F\kappa"'] & \cdots \arrow[l, "FF\kappa"'] & F^n\mu F \arrow[l, "F^{n-1}\kappa"'] & \cdots \arrow[l, "F^n\kappa"']
\end{tikzcd} \tag{3.7.17}$$

[14] DEF. B.8-11, p. 286.

[15] The pneumonic behind κ is the Greek word for *constructor*, which Google Translate tells me is "κατασκευαστής".

The edges of the cone faces $k_n : F^n\,0 \to \mu F$ are found by chasing (3.7.17) from any $F^n\,0$ down and back to μF:

$$\begin{cases} k_0 = !_{\mu F} \\ k_n = \kappa \circ F\kappa \circ F^2\kappa \circ \cdots \circ F^{n-1}\kappa \circ F^n !_{F\,0} \end{cases}$$

Since any and all squares or rectangles in (3.7.17) commute, the tuple $(\mu F,\, k_n)$ indeed constitutes a cocone.

We can construct cocones under the initial sequence for arbitrary F-algebras. Take $(X,\, \chi)$ which produces the cocone $(X,\, x_n)$, constructed as above. What makes $(\mu F,\, k_n)$ the initial cocone is the existence of a unique cocone morphism $(\mu F,\, k_n) \to (X,\, x_n)$. This unique map is determined entirely by χ so its notation emphasises it: $(\!|\chi|\!)$.[16] The cocone homomorphism makes the diagrams:

$$\begin{tikzcd}
 & F^n\,0 \arrow[dl, "k_n"'] \arrow[dr, "x_n"] & \\
\mu F \arrow[rr, dashed, "\exists!", "(\!|\chi|\!)"'] & & X
\end{tikzcd} \tag{3.7.18}$$

commute for all $n \in \boldsymbol{\omega}$.

REMARK 3.7-4. A fixpoint that gives an initial algebra is called a ***least fixpoint (LFP)*** [39]. The term comes from order theory. In fact, Lambek's lemma is a categorification of Tarski's fixpoint theorem [17]. Adámek's canonical construction of the lest fixpoint [39] is then a categorified application of the Kleene fixed-point theorem. That is, if you have a DCPO with a bottom element and a continuous (in the sense of Scott), monotone endofunction on the DCPO, then the function has a LFP which is the supremum of the ascending Kleene chain.

As a final note, we can show the isomorphism $F\,\mu F \cong \mu F$ in the current setup without the direct insight of Lambek. The colimit operation commutes with application of F (because we require that F is $\boldsymbol{\omega}$-cocontinuous)[17].

[16] The banana brackets notation $(\!|\chi|\!)$ comes from [62].

[17] Explained later, in §3.7.3.

That enables the following identifications:

$$F\,\mu F = F\left(\lim_{n<\omega} F^n\,0\right) \quad\cong\quad \lim_{n<\omega} F\,F^n 0 = \lim_{n<\omega} F^n\,0 = \mu F.$$

Another way to look at this is we can apply F to the entire initial chain:

$$0 \xrightarrow{!_{F\,0}} F\,0 \xrightarrow{F\,!_{F\,0}} FF\,0 \xrightarrow{FF\,!_{F\,0}} F^3\,0 \xrightarrow{F^3\,!_{F\,0}} \cdots$$

becomes

$$F\,0 \xrightarrow{!_{F\,0}} F^2\,0 \xrightarrow{F\,!_{F\,0}} F^3\,0 \xrightarrow{F\,F\,!_{F\,0}} F^4\,0 \xrightarrow{F^3\,!_{F\,0}} \cdots \quad .$$

Since the morphism $!_{F0}$ is unique, there is exactly one way we can prepend $(0 \rightarrow)$ to the front of the new chain, thereby reverting it back to the initial chain. The whole process can be cycled yet again: applying F to the initial chain and undone by prepending $(0 \rightarrow)$. We thus conclude the chains are isomorphic. Since colimits are unique up to isomorphism, we conclude the witnesses of $F\,\mu F \cong \mu F$ are unique.

PROPOSITION 3.7-5 ([109, PROP. 10.14])**.** If $\boldsymbol{C}$ has finite coproducts then given an endofunctor $F : \boldsymbol{C} \rightarrow \boldsymbol{C}$ the following conditions are equivalent:

1. F-algebras are algebras for a monad $(T,\ \eta,\ \mu)$ and there is and equivalence F-**Alg** $\cong \boldsymbol{C}$ that preserves the respective forgetful functors.
2. The forgetful functor F-**Alg** $\rightarrow \boldsymbol{C}$ has a left adjoint (a free-functor).
3. For each object $A \in \boldsymbol{C}$ the endofunctor $A + F -$ has an initial algebra.

REMARK 3.7-6. Recall the previous section $\hat{l}_\bullet$-**Alg** was shown to have an initial object (THM. 3.6-11). By implication, there is a corresponding monad and its Eilenberg-Moore category is isomorphic to $\hat{l}_\bullet$-**Alg**. Indeed, CONST. 3.4-4 shows that $\hat{l}_\bullet$ is the basis for the list monad which will be reiterated for the case of `snoc`-lists in §3.8.

3.7.1 Catamorphisms

The special morphism in (3.7.18) can be straightforwardly derived. To (3.7.18), affix the algebras:

$$\begin{tikzcd}
& F^n\,0 \arrow[dl, "k_n"'] \arrow[dr, "x_n"] & \\
\mu F \arrow[rr, dashed, "\exists!", "(\!|\chi|\!)"'] \arrow[d, shift left, "\kappa^{-1}"] & & X \\
F\,\mu F \arrow[u, shift left, "\kappa"] \arrow[rr, "F(\!|\chi|\!)"'] & & FX \arrow[u, "\chi"]
\end{tikzcd}$$

Chasing from μF to X gives the equation:

$$(\!|\chi|\!) = \chi \circ F(\!|\chi|\!) \circ \kappa^{-1}. \qquad (3.7.19)$$

The morphism $(\!|\chi|\!)$ is called a catamorphism [62], [77, §2.6].

DEFINITION 3.7-7. Given an endofunctor F with fixpoint $(\mu F, \kappa)$, An F-***catamorphism*** is the unique F-algebra homomorphism making the following diagram commute:

$$\begin{tikzcd}
F\,\mu F \arrow[r, "F(\!|\chi|\!)"] \arrow[d, shift right, "\kappa"'] & FX \arrow[d, "\chi"] \\
\mu F \arrow[u, shift right, "\kappa^{-1}"'] \arrow[r, dashed, "\exists!", "(\!|\chi|\!)"'] & X
\end{tikzcd}$$

What does a catamorphism do? That depends entirely on the meaning of F. But we can give concrete meaning to the running example of the monoid functor, $M \coloneqq X \mapsto 1 + X^2$. A monoid algebra, $(X, \vec{e} \triangledown \text{Ⓜ} : MX \to X)$, gives interpretation to the unit and binary operation and could be used to evaluate a single term in the monoid. (Note that, at this point, there is nothing to enforce that $\vec{e}$ and Ⓜ obey the monoid laws.) An element of μM is an expression tree and

$$(\!|\vec{e} \triangledown \text{Ⓜ}|\!) : \mu M \to X$$

evaluates expressions, reducing them to a single value of the monoid.

REMARK 3.7-8. Recall from the last section (in THM. 3.6-11) that I prematurely introduced the notation $(\!|\vec{s}_0 \triangledown \delta|\!)$ for $\hat{l}_\bullet$-folds. These were catamorphisms.

REMARK 3.7-9. Its chain construction gives a natural ordering to layers of structure and catamorphisms give a way of recursively processing the structure. On polynomials, this gives a way to structurally represent induction, since the algebras will contain a notion of evaluation of a "base case" on the leaves and inductive steps ascending through the layers of structure.

3.7.2 Anamorphisms

The whole construction from $\boldsymbol{\omega}$-chain to catamorphism dualises.

DEFINITION 3.7-10. The ***terminal sequence*** of an endofunctor F requires the existence of a terminal object and is the $\boldsymbol{\omega}^{\mathrm{op}}$-chain that runs contralaterally to the initial sequence:

$$\begin{tikzcd} 1 & F\,1 \arrow[l, "!_{F\,1}"'] & F\,F\,1 \arrow[l, "F\,!_{F\,1}"'] & F^3\,1 \arrow[l, "F\,F\,!_{F\,1}"'] & \cdots \arrow[l, "F^3\,!_{F\,1}"'] \end{tikzcd}$$

where now, $!_{F\,0} : F1 \to 1$ denotes the unique arrow to the terminal object.

In dual, the colimit under the initial chain that gave the initial algebra becomes a limit over the terminal chain:

$$\nu F = \lim_{n \in \boldsymbol{\omega}^{\mathrm{op}}} F^n.$$

By the dual of Lambek's lemma, we know that νF carries a coalgebra that is also an algebra. The coalgebra, $(\nu F, \kappa^{-1})$, is a terminal object in F-**coAlg**: a *terminal coalgebra.*

Finally, the catamorphism dualises to anamorphism.

DEFINITION 3.7-11. Given an endofunctor F with terminal coalgebra $(\nu F, \kappa)$, an F-***anamorphism*** is the unique F-coalgebra homomorphism making the following diagram commute, for any given coalgebra (Z, ζ):

$$\begin{tikzcd}
FZ \arrow[r, "F \llparenthesis \zeta \rrparenthesis"] & F \nu F \arrow[d, shift right, "\kappa"'] \\
Z \arrow[u, "\zeta"] \arrow[r, dashed, "\exists!", "\llparenthesis \zeta \rrparenthesis"'] & \nu F \arrow[u, shift right, "\kappa^{-1}"']
\end{tikzcd}$$

This gives the equation:

$$\llparenthesis \zeta \rrparenthesis = \kappa \circ F \llparenthesis \zeta \rrparenthesis \circ \zeta .$$

REMARK 3.7-12. Because the initial algebra construction is based on the initial chain, "counting up" the ordinals from 0, it gave the LFP. The $\boldsymbol{\omega}^{\mathrm{op}}$ category is more like "counting down" from infinity and potentially contains infinite structures. So a fixpoint that gives an terminal coalgebra is called a ***greatest fixpoint (GFP)*** [70]. In computer science, it is often taken for granted that initial algebras and terminal coalgebras are coincident at the same fixpoints (with different carriers), but this is not generally the case. It is worth noting however that it does happen often with many functors worth studying.

REMARK 3.7-13. In dual to REM. 3.7-9 Terminal coalgebras on polynomials and anamorphisms gives a way to structurally represent *coinduction* and corecursion.

3.7.3 Polynomial Functors Have Initial Algebras & Terminal Coalgebras

A functor $F : \boldsymbol{C} \to \boldsymbol{D}$ is called $\boldsymbol{J}$-continuous if

$$F(\lim G) \cong \lim FG.$$

for all diagrams $G \in \boldsymbol{C}^{\boldsymbol{J}}$. It is $\boldsymbol{J}$-cocontinuous if the same is true for colimits.

In our case, we are concerned with whether or not a functor preserves $\boldsymbol{\omega}$-chains or $\boldsymbol{\omega}^{\mathrm{op}}$-chains. This is a prerequisite for having a fixpoint. We require the fixpoint because it is a prerequisite for the existence of an initial algebra.

LEMMA 3.7-14 ([36, Thm. II.4]). A $\mathbb{S}\mathbf{et}$ functor has an initial algebra if and only if it has a fixpoint.

The dual is also true of coalgebra. The thesis only relies on initial algebras and terminal coalgebras of polynomial functors. So any construction of catamorphisms and anamorphisms of polynomial functors is underwritten in the following lemmas.

LEMMA 3.7-15 ([162, Lem. 8.13 & Prop. 8.14]). Every polynomial functor on $\mathbb{S}\mathbf{et}$ is $\boldsymbol{\omega}$-cocontinuous and hence has initial algebras.

LEMMA 3.7-16 ([88, Thm. 10.1]). Every polynomial functor on $\mathbb{S}\mathbf{et}$ is $\boldsymbol{\omega}$-continuous and hence has terminal coalgebras.

This justifies the constructions in the coming chapters.

3.8 SNOC LISTS AS FIXPOINTS

In computer science, a “cons” list, [72, p. 115], is a fundamental data structure used in many functional programming languages to represent sequences of data. The name “cons” originated from the Lisp programming language and is a truncation of the word “*cons*tructor”. A cons list is built by “consing” a new value onto an existing list (starting from the empty list) resulting in a new list that begins with the newly added element. That is to say, the list grows to the left. For example, in Lisp we would build a list of integers $[\,1,\ 2,\ 3\,]$ as

```
(cons 1 (cons 2 (cons 3 nil)))
```

with `nil` representing the empty list.

A “snoc” list, on the other hand, is a variant of the cons list which grows to the right:

```
(snoc (snoc (snoc nil 1) 2) 3)
```

The name "snoc" is cons spelled backward.

We have already seen snoc lists in the previous section as arising out of the free forgetful adjunction $F^L \dashv U^L$: CONST. 3.4-3. In this section, we further elucidate a mathematical model of snoc lists as fixpoints of a polynomial and understand the resulting catamorphisms and anamorphisms using the tree-view of polynomials.

3.8-1 SNOC CONSTRUCTOR, MATHEMATICALLY Recall the "objects are functors in monoidal categories" view, DEF. 3.5-1, now in the context of $\mathbb{S}\mathbf{et}$. The value of the functor P at any $A \in \mathbb{S}\mathbf{et}$ is denoted

$$\begin{aligned} \hat{A} :\ \mathbb{S}\mathbf{et} &\to \mathbb{S}\mathbf{et} \\ X &\mapsto X \times A \\ f &\mapsto f \times \mathrm{id}_A. \end{aligned}$$

Also recall the free $\mathbb{S}\mathbf{et}$-monoids, CONST. 3.4-3, that for all words in the Kleene-closure,

$$A^* = \sum_{n \in \mathbb{N}} A^n,$$

we have the natural concatenation operator

$$+\!\!+ :\ A^* \times A^* \to A^*$$

which is associative, closed and unital with the empty list, []. There is also a natural inclusion of generators,

$$\eta^L_A :\ A \to A^*;\ a \mapsto [a].$$

From **3.6-5** we can modify $+\!\!+$ to act as an $\hat{A}$-algebra by the composite:

$$\overline{+\!\!+} := A^* \times A \xrightarrow{\mathrm{id} \otimes \eta^L_A} A^* \otimes A^* \xrightarrow{+\!\!+} A^*,$$

which maps like this:

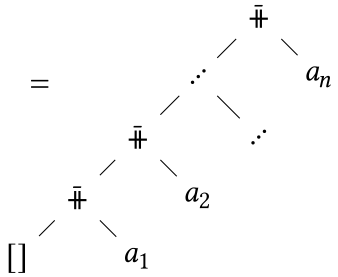
$$([\, a_0, a_1, a_2, \dots, a_n \,], a) \mapsto [\, a_0, a_1, a_2, \dots, a_n, a \,].$$

So a snoc list of As, $w \in A^*$, can be constructed as an expression:

$$w = [\, a_1,\, a_2,\, a_3,\, \dots,\, a_n \,] = [] \mathbin{\bar{+\!\!+}} a_1 \mathbin{\bar{+\!\!+}} a_2 \mathbin{\bar{+\!\!+}} \cdots \mathbin{\bar{+\!\!+}} a_n. \tag{3.8.20}$$

which can be represented visually as a syntax tree like so:

$$w \quad = \tag{3.8.21}$$



This tree notion bears non-coincidental resemblance to the depiction of polynomial functors as trees.

3.8-2 SNOC LISTS ARE FIXPOINTS The tree construction in the previous paragraph is suggestive of a polynomial functor. The LFP of this functor should be carried by A^*. Perhaps a first guess is that $\hat{A}$ is the functor in question, but on inspection, this functor has no LFP: $\hat{A}^n\, 0 \;=\; 0$ for any n.[18] The empty list is a hint to the solution: in the last section, pointed functors allowed us to model monoids with the unit as the distinguished point, and $(A^*, +\!\!+, [])$ is a monoid. As in the previous section, denote by $\hat{A}_\bullet$ the functor

$$\begin{aligned} \hat{A}_\bullet :\; \mathbb{S}\mathbf{et} &\to \mathbb{S}\mathbf{et} \\ X &\mapsto 1 + X \times A \\ f &\mapsto \mathrm{id}_1 + (f \times \mathrm{id}_A). \end{aligned}$$

A computer scientist might call this the "maybe-pair" functor. We can give $\bar{+\!\!+}$ a pointed structure, making it an $\hat{A}_\bullet$ algebra:

$$\bar{+\!\!+}_\bullet := \vec{[]} \triangledown \bar{+\!\!+} :\; 1_{\mathcal{C}} + A^* \times A \to A^*,$$

[18] It does, however, have a GFP: A^ω which are infinite streams.

where $\vec{[]}$ is the global element mapping $() \mapsto []$. The functor $\hat{A}_\bullet$ is polynomial, has a LFP—the colimit under the chain

$$\begin{tikzcd} 0 \arrow[r, "!"] & \hat{A}_\bullet\, 0 \arrow[r, "\hat{A}_\bullet\, !"] & \hat{A}_\bullet^2 0 \arrow[r, "(\hat{A}_\bullet)^2\, !"] & \cdots \end{tikzcd} \quad ,$$

that is,

$$\mu \hat{A}_\bullet = \operatorname*{colim}_{n<\omega} \hat{A}_\bullet^n\, 0.$$

Consider a pictographic representation of $\hat{A}_\bullet$ as a pair of an A-value on the right with an empty box on the left, awaiting an argument:

$$\hat{A}_\bullet\, \boxed{} \ni (\boxed{}, a) \quad \text{where} \quad a \in A$$

Forget for a moment that the asterisk indicates that $\hat{A}_\bullet -$ can also be 1, a null value, instead of a pair. An unbounded recursion would then look as if we were staring into an infinity-mirror:

$$\mu \hat{A}_\bullet \cong \hat{A}_\bullet \hat{A}_\bullet \hat{A}_\bullet \hat{A}_\bullet \cdots \ni (\boxed{(\boxed{(\boxed{(\boxed{(\boxed{(\boxed{\cdots}, a_{n-5})}, a_{n-4})}, a_{n-3})}, a_{n-2})}, a_{n-1})}, a_n)$$

with $n \to$ ω. The null value can terminate our decent into the infinity mirror at any level, fixing the length at the null iteration. The set $\mu \hat{A}$ is therefore the set of all lists we could make, of any denumerable length:

$$\mu \hat{A}_\bullet \cong \sum_{n \in \mathbb{N}} A^n = A^*,$$

as desired. This is all mediated by the isomorphism $\bar{+\!\!+}_\bullet$:

$$\begin{gathered} \bar{+\!\!+}_\bullet := \vec{[]} \triangledown (\bar{+\!\!+}) \;\; : \; \hat{A}_\bullet\, A^* \to A^* \\ \begin{cases} () \mapsto [] \\ ([\, a_0, a_1, \ldots], a) \mapsto [\, a_0, a_1, \ldots, a\,] \end{cases} . \end{gathered} \tag{3.8.22}$$

with its inverse defined (more generally) in the proof of THM. 3.6-11, but can be simplified in $\mathbf{Set}$ to reflect (3.8.22):

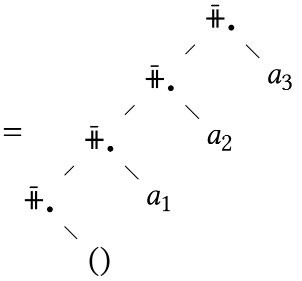
$$\begin{gathered} \bar{+\!\!+}_\bullet^{-1} : \; A^* \to \hat{A}_\bullet\, A^* \\ \begin{cases} [] \mapsto () \\ [\, a_0, a_1, \ldots, a\,] \mapsto ([\, a_0, a_1, \ldots], a) \end{cases} . \end{gathered}$$

We can still build lists as in (3.8.20),

$$[\,a_1,\, a_2,\, a_3,\, \ldots,\, a_n\,] = \big(\, (\bar{+\!\!+}_\bullet)\,()\,\big)\; \bar{+\!\!+}_\bullet\; a_1\; \bar{+\!\!+}_\bullet\; a_2\; \bar{+\!\!+}_\bullet\; \cdots\; \bar{+\!\!+}_\bullet\; a_n.$$

and view it as a tree,

$$[\,a_1,\, a_2,\, a_3\,] \quad = $$



.

3.8.1 Snoc List Catamorphism

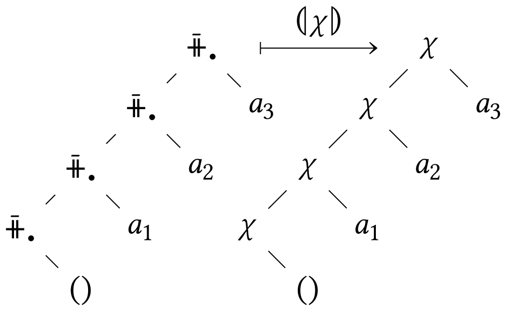
Since $\mu\hat{A}_\bullet \cong A^*$ and $\hat{A}_\bullet A^* \cong A^*$, as witnessed by the bijection $\bar{+\!\!+}_\bullet$, then we know that $(\mu\hat{A}_\bullet,\ \bar{+\!\!+}_\bullet)$ is an initial algebra. This means that there is a unique algebra homomorphism from it to any other $\hat{A}_\bullet$-algebra, say, $(X,\ \chi)$:

$$\begin{tikzcd}
\hat{A}_\bullet\, A^* \;=\; 1 + A^* \times A \arrow[r, "\hat{A}_\bullet\, (\!|\chi|\!)"] \arrow[d, shift left, "\bar{+\!\!+}_\bullet"] & 1 + X \times A \;=\; \hat{A}_\bullet\, X \arrow[d, "\chi"] \\
A^* \arrow[u, shift left, "\bar{+\!\!+}_\bullet^{-1}"] \arrow[r, "(\!|\chi|\!)"] & X
\end{tikzcd} \quad , \qquad (3.8.23)$$

The equation for the catamorphism can be read directly from (3.8.23), and is of course the same as (3.7.19),

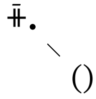
$$(\!|\chi|\!) = \chi \circ P_A\, (\!|\chi|\!) \circ \bar{+\!\!+}_\bullet^{-1},$$

which is a recursive definition since $(\!|\chi|\!)$ appears on the right and left hand side of the equality operator. The structure of $\hat{A}_\bullet$ determines the structure of its algebras. In functional terms, all $\hat{A}_\bullet$-algebras can be written in the form $\vec{x}_0 \triangledown f$ where $f :\ X \times A \rightarrow A$ is a $\hat{A}$-algebra. When $(\!|\chi|\!)$ is applied to a list of A-elements, $\bar{+\!\!+}_\bullet^{-1}$ deconstructs the list into a pair containing the right-most element and the remainder of the list. This exposes the right-most item for the present level of calculation, while next, $P_A\,(\!|\chi|\!)$ applies the catamorphism to the remaining portion of the list. Assuming we started with a list with more than one element, χ cannot evaluate anything at any layer until we have a fully unpacked (by deconstruction) the entire list finally yielding an empty list which is evaluated by χ as x_0. Then the call tree generated by the recursive expansion can be collapsed upward from x_0 by applying χ which will use f to accumulate a final result. This is perhaps more easily seen by illustration. Given a word $w = [\,a_1,\ a_2,\ a_3\,]$ regarded as an expression tree as in (3.8.21), we may view the catamorphism as transforming the expression tree by substitution of the operations at each node:



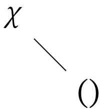
(3.8.24)

Keeping in mind



$$= (\bar{+\!\!\!+}\bullet)\,() = \vec{[\,]}\,() = [\,],$$

and



$$= \chi\,() = \vec{x}_0\,() = x_0$$

Using $\chi \coloneqq \vec{x}_0 \triangledown f$, previous tree then expresses the mapping

$$[a_1,\ a_2,\ a_3] \xmapsto{(\!|\vec{x}\triangledown f|\!)} f\,(f\,(f\,(\vec{x}_0(),\ a_1),\ a_2),\ a_3).$$

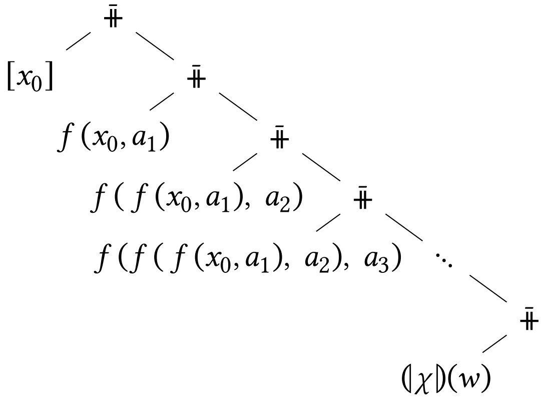
The essential nature of a catamorphism on polynomials is simply replacement—we transform the tree in the initial algebra, built from construction operations, by replacing the constructors with evaluating operations. This is true of algebra homomorphisms in general, but what is special here is the initial algebra collects values into a structure to produce tree structures.

In functional programming this pattern of recursion on lists is called a *fold* [77]. These are prefixed with a direction, left- and right-fold depending on the direction it processes the input list.[19] Specifically, we have described a left-fold on snoc-lists.

Snoc list catamorphisms models a notion of state: as the calculation is performed on a list, element by element, a value of an arbitrary type is passed up through the call chain allowing one level of calculation (See (3.8.24)) to depend on previous ones.

3.8.2 List scans

A possible deficit of folds is that they deny access to intermediate values in the calculation. Given an $\hat{A}_{\bullet}$-algebra $(X,\ \chi := \vec{x}_0 \triangledown f)$ and given a word $w = [a_1, a_2, a_3, ..., a_n] \in A^*$, $(\!|\chi|\!)(w)$ is a value in X. A *scan* will perform the same computation but will produce a list of intermediate values as a word in X^*. In terms of our tree picture, a scan produces



or more compactly,

$$[\,x_0,\ f(x_0,\ a_1),\ f(f(x_0,\ a_1),\ a_2),\ \ldots,\ (\!|\chi|\!)(w)\,].$$

[19] While left-fold is a catamorphism on snoc-lists, right-fold is catamorphism on cons-lists.

This interpretation of a scan will be important later. As we shall see, it represents exactly the pattern of iteration upon which a control program operates, and when combined with a *push based* list interface, gives us the semantics we want for writing control programs.

Some readers may notice that, while catamorphisms seem well grounded in CT, scans are defined *ad hoc*, with an obvious resemblance to catamorphisms, but without categorical backing. In a later (contributory) chapter of the thesis, I will show that there is a functor in the category of $\hat{A}_\bullet$-algebras that maps each catamorphism into a scan.

3.9 MOORE ABOUT DYNAMICAL SYSTEMS

According to Rutten, "coalgebra = system" Rutten, for coalgebras of some endofunctor that encodes the structure of iterated evolution over some state dynamics. Initial algebras of input processes capture a notion of machines evolving forward an arbitrary but finite number of steps. Terminal coalgebras capture the records of potentially infinite behaviour. In this section, we are concerned with a particular type of system: *Moore machines.*

The preliminary definition DEF. 3.6-2 is the traditional one, of an I/S/O system with finite sets I, S, O and I, O non-empty. Goguen machines generalise I/S/O systems to arbitrary monoidal categories with finite coproducts and adds categorical machinery to support behaviours in general, external behaviour of machines and an intrinsic factorisation. Based on that, we now renew the definition of a Moore machine in terms of Goguen's.

DEFINITION 3.9-1. A Moore machine is a Goguen machine in $\mathbf{Set}$ (or in ***Cpp***) with the Cartesian closed monoidal structure $(\mathbf{Set}, \times, 1)$ and finite coproducts furnished by the cocartesian monoidal structure $(\mathbf{Set}, +, 0)$.

Let $\Sigma = (I, S, O, \delta, r, \vec{s}_0)$ be a Moore machine and recall the diagram (3.6.14) summarising key relationships in Goguen's model, modified slightly in light of the past few sections:

$$\begin{tikzcd}
\hat{I}_\bullet I^* \arrow[r, "\hat{I}\,(\!|\vec{s}_0 \triangledown \delta|\!)"] \arrow[d, shift left, "\overline{+\!\!+}_\bullet"] \arrow[rr, bend left, "\hat{I}\,(\!|\vec{y}_\Sigma \triangledown \varphi|\!)"] & \hat{I}_\bullet S \arrow[r, "\hat{I}\,[\!(\grave{\delta} \vartriangle r)\!]_M"] \arrow[d, "\vec{s} \triangledown \delta"] & \hat{I}_\bullet (I^* \multimap O) \arrow[d, "\vec{y}_\Sigma \triangledown \varphi"] & \\
I^* \arrow[u, shift left, "\overline{+\!\!+}_\bullet^{-1}"] \arrow[r, "(\!|\vec{s}_0 \triangledown \delta|\!)"] \arrow[rr, bend right, "(\!|\vec{y}_\Sigma \triangledown \varphi|\!)"'] & S \arrow[r, "{[\!(\grave{\delta} \vartriangle r)\!]_M}"] & I^* \multimap O \arrow[r, "ev_B"] & O
\end{tikzcd} \quad . \tag{3.9.25}$$

The main difference is that $\check{r}$ is replaced by an anamorphism $[\!(\grave{\delta} \vartriangle r)\!]_M$ where $\grave{(-)}$ denotes the exponential transpose (i.e., currying, so $\grave{\delta} = \lambda\delta$) and the functor M is defined in the next subheading.

OBSERVATION 3.9-2. The arrow $\check{r}$ from §3.6, was introduced *ad-hoc* a useful tool to connect machine internals to behaviour. Now it has a deeper categorical root:

$$\check{r} = [\!(\grave{\delta} \vartriangle r)\!]_M = \lambda\big(r \circ (\!|\vec{s}_0 \triangledown \delta|\!)\big) : \; S \to I^* \multimap O.$$

The equality above is true from the standpoint of value, but the calculation of $[\!(\grave{\delta} \vartriangle r)\!]_M$ is quite different from $\check{r}$, which has implications for the software to come.

3.9.1 Terminal Moore Machines

NOTATION 3.9-3. let E_A be an alias for the covariant internal hom-functor

$$E_A\, X = A \multimap X.$$

NOTATION 3.9-4. Let $M_O^I := \hat{O} \circ E_I = (I \multimap -) \times O$. The obvious mnemonic here is "*M*oore". Since O and I will be consistently used for the output and input spaces, the scripts will be omitted: $M = M_O^I$.

Given a Moore machine $\Sigma = (I,\, S,\, O,\, \delta,\, r,\, \vec{s}_0)$, an M-coalgebra (S, ς) is a set of states and a costructure map of form:

$$\varsigma = \grave{\delta} \vartriangle r : \; S \to (I \multimap S) \times O. \tag{3.9.26}$$

Pointwise, (3.9.26) reads

$$\varsigma(s) = \big(\, i \mapsto \delta(s,i),\ r(s)\,\big),$$

A terminal coalgebra of M will be carried by an object νM and, as a fixpoint, will have an isomorphism

$$\kappa_M :\ M(\nu M) \xrightarrow{\sim} \nu M$$

and, by definition, for any M-coalgebra (S, ς), there is a unique coalgebra homomorphism

$$⦇\varsigma⦈_M :\ (S, \varsigma) \to (\nu M, \kappa_M^{-1})$$

satisfying

$$\begin{tikzcd} MS \arrow[r, "M\,⦇\varsigma⦈_M"] & M(\nu M) \arrow[d, shift right, "\kappa_M"'] \\ S \arrow[u, "\varsigma"] \arrow[r, dashed, "\exists!", "⦇\varsigma⦈_M"'] & \nu M \arrow[u, shift right, "\kappa_M^{-1}"'] \end{tikzcd} . \tag{3.9.27}$$

We can read off the corecursive equation

$$⦇\varsigma⦈ = \kappa \circ M\,⦇\varsigma⦈ \circ \varsigma.$$

We already know that $I^* \multimap O$ must carry the terminal coalgebra of M because it is the codomain of $⦇\grave{\delta} \vartriangle r⦈_M$. We also know from OBS. 3.6-24 that the forgetful functor on $\mathbf{Set}^{\hat{I}}$ has a cofree (right) adjoint and that $I^* \multimap O$ is cofree generated by O. Yet playful curiosity may lead one to casually write out νM as an iterated application of M (similarly to **3.8-2** for the LFP of $\hat{A}$). This gives:

$$\begin{aligned} \cdots MMM \cdots \;\;&\cong \\ &\underbrace{\cdots (I \multimap (I \multimap (I}_{\text{input}} \multimap (\cdots) \times \underbrace{O) \times O) \times O) \times O \cdots}_{\text{output}} . \end{aligned} \tag{3.9.28}$$

Though informal, it does illustrate how νM structurally encodes the process of exchanging input for output—nowhere in (3.9.28) is the appearance of state—it is obscured in the "call stack" of nested function spaces[20]. Using $\nu M = I^* \multimap O$, the diagram (3.9.27) elaborates to:

$$\begin{tikzcd}
(I \multimap S) \times O \arrow[r, "M \llparenthesis \varsigma \rrparenthesis_M"] & \big(I \multimap (I^* \multimap O)\big) \times O \arrow[d, shift right, "\kappa_M"'] \\
S \arrow[u, "\varsigma"] \arrow[r, dashed, "\exists!", "\llparenthesis \varsigma \rrparenthesis_M"'] & I^* \multimap O \arrow[u, shift right, "\kappa_M^{-1}"']
\end{tikzcd} .$$

So the terminal object in M-**coAlg** is the coalgebra $(I^* \multimap O,\ \kappa_M^{-1})$ where the costructure map is

$$\begin{aligned}
\kappa_M^{-1} : I^* \multimap O &\to \big(I \multimap (I^* \multimap O)\big) \times O; \\
y &\mapsto \Big(i \mapsto \big(w \mapsto y\,([i] \mathbin{+\!\!+} w)\big),\ y[\,]\Big).
\end{aligned}$$

(One can consult [88, eg. 10.2] or [52, §10.2, eg. 18] for detailed proof.)

3.9-5 The unique $\llparenthesis \varsigma \rrparenthesis_M$ assigns to each state a function $I^* \multimap O$, that consumes a list of I-values and produces the output that results following all of the state transitions induced by each input. Consider a behaviour $y_\Sigma \in I^* \multimap O$ associated by $\llparenthesis \varsigma \rrparenthesis$ to s_0. Such y_Σ works internally by propagating the state transitions associated with each input in a word $w \in I^*$, so that $y_\Sigma\, w \in O$ is the output of the Moore machine at the terminus of the induced state-trajectory:

$$w = [i_0,\ i_1,\ i_2,\ \ldots i_n] \overset{y_\Sigma}{\longmapsto} r \circ \delta(\cdots \delta\,(\delta\,(\delta\,(s_0,\ i_0),\ i_1),\ i_2) \cdots, i_n).$$

OBSERVATION 3.9-6. The above displayed equation looks very similar to the pattern of recursion in (3.6.10), which shows how the catamorphism $\llparenthesis \vec{s}_0 \triangledown \delta \rrparenthesis$ automates state-stepping. But recall from §3.8.1 that catamorphisms on snoc lists compute by traversing the list to its root, building a call stack before hitting the recursive "base case" (the empty list) and evaluating from the there up. M-anamorphisms are corecursive and traverse the other way around. In the case of state-stepping, they can start from

[20] If r is a bijection, then we can recover the internal state. So the state is obscured to the extent that it can be determined by output.

an initial state and iterate *ad infinitum*, driven one input at a time. This has consequences for the thesis' model because feedback control systems software gets input from the external world one datum at a time: not all up front as is required by catamorphisms. Keep this in mind when reading the next section on asynchronous lists.

The picture of Moore machines as coalgebra has been known since at least 1975 [32]. For more a full reading of "systems as coalgebras", see the landmark paper [88], the textbook [170], or, in the context of ACT, the see [182, §3.1] or the draft textbook [178].

3.10 ASYNC LISTS & OBSERVER-ITERATOR DUALITY

> **John A De Goes,**
> American Author and Software Engineer
> @jdegoes
>
> *Functional reactive programming 'failed' because it is leaky atop the procedural foundations of computing.*
>
> *The more practical alternative to FRP is potentially infinite streams (subscriptions), combined with functional effects.*
>
> *It's not as beautiful as FRP but it works.*
>
> `https://twitter.com/jdegoes/status/1153642841601691648`

The previous sections have given a mathematical model of lists and the operations that come along with category theoretical machinery. But the mathematics consider neither memory nor time.

In standard list-like collections like `std::vector` or `std::list`, the consumer of the list is in charge of the traversal, *pulling* data as needed. Feedback control systems and estimators are more naturally reactive. Rather than pulling data from a structure, data values are *pushed* though a computation pipeline as they become available from sensors or clocks, driving output of dependent quantities. In other words, we want to compose the calculation as an expression in a calculus of operations on sequences of values—distributed not in the computer's memory, but through time. From the point of view of the math, that distinction is only a matter of interface.

In this section, an interface for push-based "observable" collections is derived as the categorical dual to the pull-based "iterable" collections. The derivation is attributed to Meijer, Dyer and others at Microsoft in the development of *Rx.NET* which eventually became the popular cross-language ReactiveX (Rx) libraries [113].

The derivation of the asynchronous collection interface consists merely of transcribing the Unified Modeling Language (UML) [96] diagram for the GoF iterator pattern, [69], into a category theory like diagram of arrows and then reversing the arrows to get the dual interface. There are several informal/non peer reviewed references demonstrating the process: [112, 117, 120] and a conference presentation, [131]. It is also described in the many books on the topic of Rx. Perhaps the best academic reference is the Masters thesis of Bertoluzzo [154].

* * *

Iterators are commonplace in mainstream programming, which is why it may be the most well known of the so called *Gang* of *Four (GoF)* patterns presented by Gamma *et al.* in [69].[21] The GoF say that iterators…

> provide a way to access the elements of a aggregate object sequentially without exposing its underlying representation.

This is achieved using two abstract interfaces: an `Aggregate`, which produces iterator objects and `Iterators` which provide an interface for advancing through the collection and retrieving values. Those interfaces are illustrated in the UML diagram, FIG. 3.1.

It is important to distinguish between *internal* and *external* iterators. When the client controls the iteration by advancing the traversal and requesting data, then this is an **external** iterator. If the client provides a

[21] The GoF book is widely regarded as a seminal on the topic of OO software design patterns.

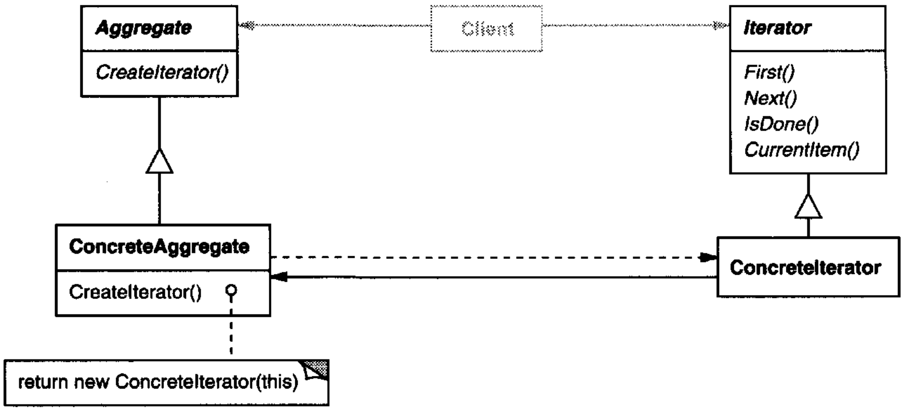


FIGURE 3.1: Iterator pattern class diagram from [69, p. 259].

function to the iterator and the iterator handles the traversal and application then it is an **internal** iterator. Internal iterators have very nice properties, and have deep categorical underpinnings [104]. However, the GoF Iterator Pattern describes an external iterator, and this is the starting point of our derivation.

NOTATION 3.10-1. This notation is exclusive to only this section. First, let () represent the singleton set. This gives routines with codomain 1 the appearance of a nullary function call. Let $a : \mathtt{A}\langle T\rangle$ be an object, an "instance" of type $\mathtt{A}\langle T\rangle$.[22] Furthermore, let $\mathtt{A}\langle T\rangle$ be a class with a member function `foo`. Member functions implicitly take instances as an argument, so $a.\mathtt{foo}()$ is morally equivalent[23] to $\mathtt{foo}(a)$. To simplify type arrow specification for interfaces, let

$$\underbrace{() \xrightarrow{\mathtt{foo}} ()}_{\mathtt{A}\langle T\rangle}$$

denote that the member function `foo` is nullary (except for the implicit argument), allowing us to avoid naming an instance of $\mathtt{A}\langle T\rangle$. Since `foo` returns nothing you might guess that its reason for being is to mutate its

[22] The angle-bracket syntax for type arguments is borrowed from languages like C++ or C#. The thesis will eventually use formal ISO C++, so this is a natural choice.

[23] This equivalence was proposed as a "unified call syntax" for C++ [134].

instance. Member functions that mutate their instance are indicated by a harpoon:

$$\underbrace{() \xrightleftharpoons{\texttt{foo}} B}_{A\langle T\rangle}$$

If we have another member function, `bar`, in the interface, we stack the member functions:

$$\underbrace{\begin{array}{c} () \xrightleftharpoons{\texttt{foo}} () \\[2ex] B \xrightarrow{\texttt{bar}} C \end{array}}_{A\langle T\rangle} \qquad \text{or} \qquad \underbrace{\begin{array}{l} \texttt{foo} : () \rightleftharpoons () \\[1ex] \texttt{bar} : B \rightarrow C \end{array}}_{A\langle T\rangle} .$$

Here, `bar` still takes its implicit argument but additionally takes an argument of type B. Finally, a (possibly mutating) member function that takes no (non-implicit) arguments and returns any type of value, say $() \rightarrow D$, is a generalised notion of a ***getter***. "Getter" is a programmer's argot for member functions that retrieve values from an instance. Its dual, $D \rightleftharpoons ()$, is a generalised notion of a ***setter***. . "Setter" is a programmer's argot for a member function that sets values in its instance. All setters are presumed to mutate their instance, since that is the point of them in the first place.

Returning to the derivation of an interface for asynchronous collections, first consider the abstract `Aggregate` from FIG. 3.1. It facilitates the creation of an iterator for a collection which can be sequentially traversed. It consists of a single method:

$$\underbrace{() \xrightarrow{\texttt{CreateIterator}} \texttt{Iterator}\langle T\rangle}_{\texttt{Aggregate}\langle T\rangle} \tag{3.10.29}$$

where T is the element type of the collection.

The `Iterator` further consists of several member functions. We focus on the following subset:

$$\underbrace{\begin{aligned} \texttt{Next}:&\ () \rightleftharpoons () \\ \texttt{IsDone}:&\ () \to \texttt{bool} \\ \texttt{CurrentItem}:&\ () \to T \end{aligned}}_{\texttt{Iterator}\langle T\rangle}$$

`Next` advances the iterator which mutates it so that it points to the next value in the aggregate sequence. It may be viewed as a setter. `IsDone` method allows the consumer to detect the end of the iteration, when the iterator has advanced to the end of the aggregate. Both `IsDone` and `CurrentItem` can be seen as getters. `CurrentItem` can take on the functionality of `IsDone` if it is modified to return an optional value (modelled by the disjoint union $() + T$):

$$\underbrace{\begin{aligned} \texttt{Next}:&\ () \rightleftharpoons () \\ \texttt{CurrentItem}:&\ () \to () + T \end{aligned}}_{\texttt{Iterator}\langle T\rangle}$$

This version of `CurrentItem` returns a null value, (), just in case the end of the aggregate is reached and the traversal is over. Now the remaining two member functions can be merged into a single one that advances the iterator and returns the (optional) current value:

$$\underbrace{\texttt{NextItem}:\ () \rightleftharpoons () + T}_{\texttt{Iterator}\langle T\rangle}$$

Substituting this into (3.10.29) gives a composite description of how one obtains an iterator and uses it to interface with an aggregate:

$$\underbrace{() \xrightarrow{\texttt{CreateIterator}}}_{\texttt{Aggregate}\langle T\rangle} \underbrace{\big(() \xrightleftharpoons{\texttt{NextItem}} () + T\big)}_{\texttt{Iterator}\langle T\rangle}$$

This diagram illustrates a getter which retrieves another getter: a getter of values that returns something new each time it is called (until it does not). Of course, we have to keep in mind that these getter arrows have side effects, but we may now reverse the arrows (and change the names) to dualise the diagram:

$$\underbrace{\left(() + T \xrightarrow{\texttt{Update}} ()\right)}_{\texttt{Observer}} \underbrace{\xrightarrow{\texttt{Attach}} ()}_{\texttt{Subject}} . \tag{3.10.30}$$

The *getters* have now become *setters*: effectful functions with () as codomain. The *Iterator*, an arrow producing values becomes the `Observer`, an arrow which accepts values. `Iterators` were retrieved through the `Aggregate` interface. `Observers` are subscribed (`Attach`ed) through the `Subject` interface. `Iterators` allowed values to be pulled from an `Aggregate`, `Observers` accept values pushed from `Subjects`.

A client holding an iterator is in charge of traversing the `Aggregate`, so the data must be all simultaneously be present in memory (lest the iterator advancement block the thread). Because `Subjects` push data, they can promulgate values leaving their `Observers` to react. An observer is a glorified callback.

The GoF book describes the *Observer Pattern* [69, p. 293] which:

> [defines] a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated automatically.

The observer pattern is summarised in the class diagram FIG. 3.2. The resemblance between the GoF Observer Pattern (with `Subject`/`Observer`) to the dualised form of the `Aggregate`/`Iterator` is no mere coincidence.[24]

The canonical GoF observer pattern has several issues which are well documented by Blackheath and Jones in [147, Ch. 1] and by Pai and Abraham in [165, Ch. 5]. Those issues are solved in reactive models such as Meijer's Rx and Elliott and Hudak's *Functional Reactive Programming (FRP)* [78, 147], so we shall not dwell on the GoF version.

[24] Though, at the time Gamma *et al.* wrote GoF, the observer-iterator duality was not known. (At least, not to anyone who wrote about it.)

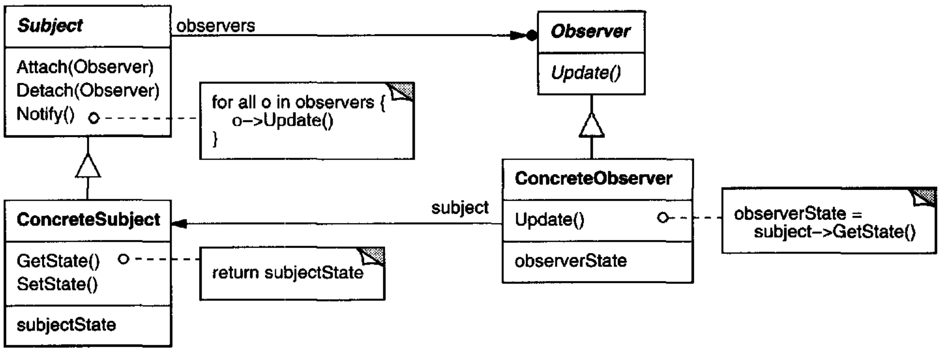


FIGURE 3.2: Observer pattern class diagram from [69, p. 294].

The thesis focuses on Rx as implemented by Shoop in [198]. There is also some attention given to FRP as implemented by Blackheath in [195] as an alternative.

3.10.1 The Rx Observable Interface

The `Observer` interface in (3.10.30) can be split back into two member functions:

$$\underbrace{\underbrace{\begin{pmatrix} T \xrightarrow{\texttt{OnNext}} () \\ () \xrightarrow{\texttt{OnComplete}} () \end{pmatrix}}_{\texttt{Observer}\langle T\rangle} \underbrace{\xrightarrow{\texttt{Subscribe}} ()}_{\texttt{Observable}\langle T\rangle}}_{\texttt{Subscription}\langle T\rangle} \tag{3.10.31}$$

The `Notify` method has also been renamed to `Subscribe` in keeping with the Rx convention [113]. The `OnNext` method is a channel from an observable to each of its subscribers. It is the dual of the iterator's `NextValue` member and is called each time a new value is to be published to the subscribers. The `OnComplete` method is the dual of the iterator's `IsDone` and is used by the observable to signal the end of the collection to each of the subscribers.

There are two mechanisms present in Rx that are missing from the model in (3.10.31):

1. the observer should have an `OnError` method that accepts an exception object, and
2. the observable (the subject in the GoF pattern) should have a way of *unsubscribing*.

So (3.10.31) should look like this:

$$\underbrace{\left(\underbrace{\begin{pmatrix} T \xrightarrow{\texttt{OnNext}} () \\ () \xrightarrow{\texttt{OnComplete}} () \\ () \xrightarrow{\texttt{OnError}} () \end{pmatrix}}_{\texttt{Observer}\langle T\rangle} \underbrace{\xrightarrow{\texttt{Subscribe}} ()}_{\texttt{Observable}\langle T\rangle}\right) \xrightarrow{\texttt{Unsubscribe}} ()}_{\texttt{Subscription}\langle T\rangle} \tag{3.10.32}$$

In practice, the `OnComplete` and `OnError` functions are optional, but should be provided for any stream that terminates during the program lifetime, or has any possibility of throwing an exception. You will see this in the upcoming example. When we apply this to control programs, we will design `T` to optionally contain error information so that errors can be handled in course of normal operation. We will also assume that incoming data does not “complete” since we expect our feedback sources (sensors and operator input) do not terminate before controller shutdown.

Meijer *et al.* derive the Rx subscription from C$^{\#}$’s `IEnumerable`/`IEnumerator` interface (not the GoF iterator pattern). `IEnumerator` is a subtype of `IDisposable`, which provides an interface for releasing the iterator acquired from an `IEnumerable` instance. As a result, `Subscribe` returns a disposable resource handle. This is consonant with the GoF observer pattern because the `Subject` has a `Detach` member function. In RxCpp, there is a corresponding `unsubscribe` member function on a `subscription`. In control systems programs, we will identify the program (as a dynamical system) with the subscription, so we do not in any case manually unsubscribe. (That will all be handled by C++’s Resource Acquisition Is Initialization (RAII) mechanism at program termination when the subscription goes out of scope.) So we do not retain a handle (a named variable) for the subscription object.

3.10.2 RxCpp: a Brief Tutorial

The idea behind Rx programming is that we build a computation pipeline as an expression in a DSL providing an algebra of observables. Rx implementations come with a battery of operators for manipulating observables. The website `https://reactivex.io/documentation/operators.html` provides detailed descriptions of the standard operators and tabulations of their implementations in each supported language's Rx library. They are too many operators to list here and, moreover, different implementations may be more or less complete. For an account of RxCpp's supported operators, you can consult the documentation [198]. The book *C++ Reactive Programming* by Pai and Abraham [165] covers key operators and the fundamentals of building reactive applications with RxCpp.

An example of a simple Rx operator is `map`, which takes a function as an argument and applies it elementwise to the observable sequence. It has category theoretical backing. Recalling the conventional list-functor which maps each $A \in \mathbb{S}\mathbf{et}$ to A^*, the conventional action on arrows is pointwise application: given $f : A \to B$,

$$f^* : A^* \to B^*;\;\; [\,a_1, a_2, \ldots\,] \mapsto [\,f(a_1), f(a_2), \ldots\,]$$

and $f^*[] = []$. In functional languages, this operation is often called `map`, as it is in Rx.

EXAMPLE 3.10-2. Our program is written in C++17 with the RxCpp library.

SCENARIO We have a problem with a noisy sensor and we want to smooth the signal with a simple box filter of width N. Our sensor `source` is of type `observable⟨double⟩`.

SOLUTION We can then write the box filtered `output` signal as a composite of the Rx built-in `buffer` and `map` operators.

The `buffer(`N`, skip)` operator stores N values as a `std::vector`, with `skip` elements between consecutive buffers. Choosing `skip` = 1 makes a standard ring-buffer of size N. Rx operators can be illustrated in a nice visual style called *marble diagrams*[25]. A marble diagram illustrating `buffer(3, 1)` is shown in FIG. 3.3. In these diagrams, time runs from left to right and each stream (observable) is represented by a line. Data is transformed from top to bottom with a datum illustrated as a bead one one of the stream lines. Beads can be distinguished by shape, colour or content (though bead shape sometimes indicates type). Operators are illustrated as boxes instead of lines, and data sequencing and alignment is indicated by dashed arrows.

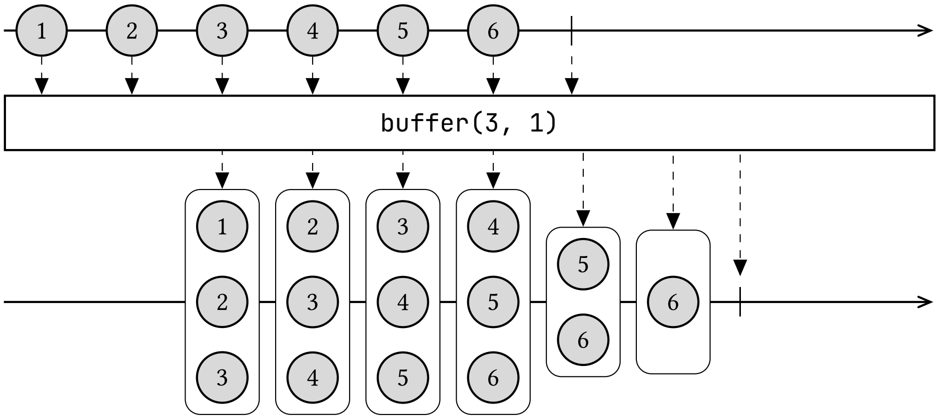


FIGURE 3.3: Marble diagram for RxCpp's `buffer` operator with count (that is, `buffer_with_count`). The vertical stacks of data represent `std::vectors` that the buffer operator emits.

Begin by declaring the source from the sensor:

```
rxcpp::observable<double> source = …
```

[25] I cannot be sure from whom marble diagrams originated but they appear in primitive form in the US patent applications for the Rx interface [117].

Since we do not have a specific device or driver in mind, the ellipsis is as specific as we can get. When `source` emits its values, we collect them with `buffer`. When the `buffer` emits its payload as a `std::vector`, we then average the values using `map`. We will need to provide `map` with a function that averages the values in a vector of doubles:

```
auto avg_buffer =
  [](const std::vector<double> &buffer) -> double {
    auto sum = std::accumulate(buffer.begin(), buffer.end(), 0.);
    return sum / buffer.size();
  };
```

Now we can write an expression defining `output` in terms of operations on the `source`:

```
auto output = source | rx::buffer(N, 1) | rx::map(avg_buffer);
```

Output is also an `observable`⟨`double`⟩ and we need to subscribe to it to have access to the values. Recall from (3.10.32) that an observer is a pair of functions/routines that handle the `OnNext` and `OnComplete` cases (or optionally as a triple with `OnError`). We will subscribe a pair of lambda functions that print the `OnNext` and `OnComplete` cases:

```
output.subscribe(
    [](double x) { printf("OnNext: %.2f\n", x); },
    []() { printf("OnComple\n"); });
```

This is a pair of lambdas instead of a trio because we decline to handle the `OnError` case. If we gave only a single lambda, we would also be declining to handle the `OnComplete` case.

As a test, if we let $N = 5$ and

```
auto source =
    rxcpp::observable<>::range(1, 10)
      | rx::map([](auto x) -> double { return (double) x; });
```

(that is, the integers $1, \ldots 10$ converted to `double`). This setup gives output:

```
  OnNext: 3.00
  OnNext: 4.00
  OnNext: 5.00
```

```
OnNext: 6.00
OnNext: 7.00
OnNext: 8.00
OnNext: 8.50
OnNext: 9.00
OnNext: 9.50
OnNext: 10.00
OnComplete
```

The output after "`OnNext:␣8.00`" is not quite what we might expect from a typical box filter. This is because, as you can see from FIG. 3.3, there is a "wind-down" effect where the buffer narrows after the input sequence is exhausted. If we want to avoid the wined-down, we can terminate the stream as soon as the buffer begins to narrow. We do this with the `take_while` operator which accepts a predicate and calls `on_completed` the first time it fails:

```
constexpr auto bufw = 5;
auto output =
  source
    | rx::buffer(bufw, 1)
    | rx::take_while([](auto& v){return v.size() == bufw;})
    | rx::map(avg_buffer);
```

The preceding snippets are included (in full, with unit test) in the demo code accompanying the thesis [194] and in LST. C.1.1 of Appendix C, p. 317.

3.10.3 Kálmán Filtering Examples

A more advanced demonstration using Kálmán filtering is provided in my demo repository[26] [199]. The scenarios include: estimating the altitude of a free-falling object with noisy radar readings[27] and; estimating the coefficients of an unknown cubic polynomial from noisy samples. The

[26] `https://github.com/timtro/kalman-folding`

[27] Borrowed from the book *Fundamentals of Kalman Filtering* by Zarchan and Musoff [107, pp. 159–171].

demos are thrice rehearsed: in FRP using SodiumFRP [195], in RxCpp, and using ordinary list catamorphism (folds)[28]. The examples are based on a series of papers by Beckman: [142–146]. Also see his conference talk [152] based partly on those papers.

[28] See §3.8.1.

4 A PLATONIC BICCC OF C++ PROGRAMS

Always design a thing by considering it in its larger context—a chair in a room, a room in a house, a house in an environment, an environment in a city plan.
— Eliel Saarinen

4.1 CHAPTER SYNOPSIS

THE aim of this chapter is to present a (platonic) category, ***Cpp***, of C++ programs with a cartesian monoidal, a cocartesian monoidal and ultimately, a bicartesian closed structure (DEF. B.15-10). And alongside that, C++17 standard compliant code that models the structure. This code is a particular representation of ***Cpp***, similar to how column vectors are representations of vector spaces. I denote this representation ***Cpp***/C++.

The purpose of ***Cpp***/C++ is not to create a useful C++ sublanguage. The purpose is,

- to demonstrate that the structure of ***Cpp*** can be represented in a way that a standards compliant C++ compiler can implement,
- to make vividly clear where and how the axioms of ***Cpp*** constrain the design of C++ programs so that, when we must, we can break those constraints deliberately and artfully,
- to demonstrate that, with some amount of squinting, we can treat appropriately constrained C++ programs as reflections of constellations in **$et**, and
- apply some "fast and loose reasoning" that is morally correct, in the sense of [97].

Table 4.1 summarises the key structures of a biccc and the C++ types upon which they will be modelled.

MOTIVATION This structure is what will allow us to model *Moore machines* in the category of C++ programs. Compatible Moore machines can be composed in series or parallel and such a composite is used to model control programs. This chapter is also important because in the literature table 4.1: Summary of biccc structures and c++ types upon which they can be modelled.

Biccc structure	Definition	C++ structure
Product	B.9-3	POD-struct or std::pair
Coproduct	B.10-2	std::variant
Initiator/termintor	B.3-1	Singleton types/void
Exponentials	B.15-3	std::function lambda functions

of CT applied to programming there are many choices for modeling. Some of those choices are determined by the language of application and others by the aspects of programs we wish to capture. So this chapter makes clear what we are and are not modeling in the C++ language.

CONTRIBUTION This chapter is the first I have seen that establishes foundations for a platonic category of C++ programs. Conversations about CT applied to C++ are rare but not unheard of. In the book *Category Theory for Programmers*, [169], Milewski often uses C++ in explorations of category theoretical programming patterns. And though it is not a mathematical book, Čukić has a nice book on FP in C++, [163] [1]. Since C++ is a very pragmatic industrial language, it is probably counter-cultural to have an abstract mathematical model.

I have made the code listed in this section publicly available in a git repository [194]. The repository also contains a suite of unit tests that demonstrate how the provided code can be used and how it interacts with C++ mechanisms such as copy and move semantics. Similar code also appears in the public repository for my experimental library tfunc [166] where you can additionally find benchmarks.

[1] I count [163] even though it is not steeped in CT because functional programming is the oldest and most mature example of applied CT.

MIXED C++ AND MATH NOTATION In mathematical expressions ***Cpp*** objects and arrows appear in the same typewriter text as the code listings while also following the ISO math convention of italicising variables. Moreover, these entities are given C++17-standard compliant names. Function application is indicated with parentheses $f(x)$, despite the fact that this is unconventional in functional programming and CT. We will come to the notion of functors and natural transformations as particular types of C++ templates where the language notation is to use angle-brackets for template arguments. This will carry into the mathematical notation; so a functor F acts on a type variable T and the notation is $F\langle T\rangle$. Outside of these standards, we adopt the convention that types will be named in PascalCase while variables and functions are named in snake_case (except when the name of a type appears in the name of a variable).

4.1-1 CODE PREAMBLE The code in the Chatper's examples use some simple equality-comparable singleton types,

```
struct A {bool operator==(const A) const { return true; }};
struct B {bool operator==(const B) const { return true; }};
struct C {bool operator==(const C) const { return true; }};
struct D {bool operator==(const D) const { return true; }};
```

and as some functions among them,

```
// f : A → B
const auto f = [](A) -> B { return {}; };
// g : B → C
const auto g = [](B) -> C { return {}; };
// h : C → D
const auto h = [](C) -> D { return {}; };
```

They are defined in the header file `include/test-tools.hpp` from the thesis demo repository [194].

4.2 A CATEGORY OF C++ PROGRAMS

In a category of programs, objects are structural representations of variable/object types (in the type-theoretical sense, like floating point values, 32-bit integers, arrays and alike) and arrows are functional routines that map arguments to return values.

In the simplest cases, the objects of these program categories are sets. But type theory offers more sophisticated options that allow us to model some aspects of the runtime environment. For example, it is common in type theory to have a *bottom type* which is a subtype of all other types and with a *bottom value* denoted $\perp$. If an arrow has some subset of its domain that causes undefined behavior, infinite recursion, exceptions or otherwise unrecoverable errors, then the image of that subset is $\perp$.

Any program is a model of a model within a theory of a model of an abstraction of some portion of the world or of some universe of discourse
— Meir M. Lehman
Programs, Life Cycles, and Laws of Software Evolution

For this thesis, we follow the approach of Malcolm, [57, 58], and use sets as our types, and mathematical functions as our arrows.

DEFINITION 4.2-1. The category ***Cpp*** has

OBJECTS for each C++ type, a set of that type's values.

ARROWS *pure* functions mapping types to other types.

For examples, the types `int32_t`, `bool` and `std::vector⟨double⟩` are all in ***Cpp***. A simple example of an arrow in ***Cpp***(`int32_t, bool`) would be the closure,

```
[](int32_t x) -> bool { return x ⩾ 5; }
```

The domain and codomain of the arrows are the argument and return types, respectively. Named functions are also, of course, perfectly good ***Cpp***-arrows. Member functions can be regarded as functions with their first argument (the object upon which they are called) is curried in. Definition B.1-1 introduces the notation hom(T, U) as the collection of arrows between objects A and B of a category. The C++ Standard Template Library (STL) provides `std::function`[2] that can wrap callable entities such as named function, function pointers, closures, callable objects, pointers-to-member-functions. This provides a uniform interface to callable objects. It is known to incur a performance penalty, but for the exemplary code of this chapter it is a favourable trade. The expression `std::function⟨U(T)⟩`,

[2] in the `<functional>` header.

as a notation, is verbose and does not harmonise with mathematical notation. The code in LST. C.2.1 defines the templates `Hom` and `Dom` which are a thin layer of template metaprogram around `std::function` allowing us to write

$$\texttt{Hom}\langle T,\ U\rangle \quad \text{instead of} \quad \texttt{std::function}\langle U(T)\rangle.$$

For functions of more than one variable, we have an additional notational device, `Doms`, to mark the domain,

$$\texttt{Hom}\langle\texttt{Doms}\langle T,\ U\rangle,\ V\rangle \quad \text{instead of} \quad \texttt{std::function}\langle V(T,\ U)\rangle.$$

FUNCTION PURITY The emphasised term *pure* in the former definition refers to the small minority of C++ functions behave like mathematical ones. To the C++ programmer, a function is a callable block of instructions that may or may not take arguments and may or may not return a value. To the mathematician, a function is a set relation that is left-total and right-unique. Mathematical functions produce output consistent with their arguments.

Take a mathematical function f and a value $x \in \operatorname{dom} f$, then $y = f\,x$ is a value in $\operatorname{cod} f$; and everywhere we write $f\,x$ we can replace it with y. This is called *referential transparency*.

DEFINITION 4.2-2. Let us call a general C++ "function" a ***routine***. A routine that is left-total, right-unique and referentially transparent, is called a C++ function. To emphasise adherence to these properties, it may be called a ***pure*** function.

Some examples of impurities are: reading from a random number generator, polling system time, printing to the screen and taking input from the keyboard (or anywhere other than arguments). Milewski popularised a useful litmus for purity: a routine that can be *memoised*[3] behaves purely enough to call a function.

[3] Memoisation is a way to improve the performance of expensive function at the expense of memory. A memoised function is wrapped in a routine that keeps a cache of

Purity is important because it allows us to discuss values, expressions and behaviour in equivalent terms. So the arrows of ***Cpp*** are pure functions and any C++ routine that is not pure is not in ***Cpp***.

4.2.1 Category axioms of ***Cpp***

To give ***Cpp*** the structure of a category we must identify structures in the C++ language that model the data and axioms of a category as per DEF. B.1-1. Namely, we need to demonstrate

1. closed function composition, Appendix C-1,
2. associativity of functions composition, Appendix C-2 and
3. identity functions, Appendix C-3.

*4.2.1.1 1. Functional Closure in **Cpp***

Nested call expressions such as `f(g(x))` are of course valid C++, and does what we might think. If `f` and `g` are pure, and their signatures are compatible, then this will compile into something behaving as f ∘ g. But there is no builtin function composition in C++17. We can create lambdas on the fly:

```
  auto fog = [=](auto x){ return f(g(x)); }
```

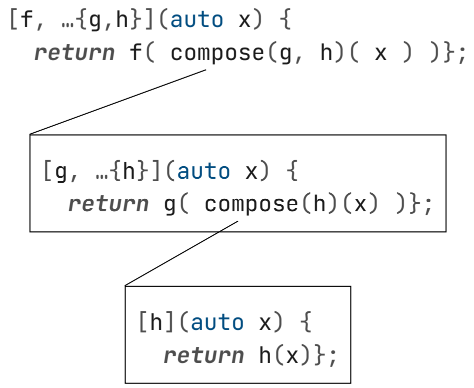
but to satisfy Appendix 1, we have an operation for composing arbitrary functions,

```
  auto fogoh = compose(f, g, h);
```

The following implementation is used in parts of the thesis demo repository [194].

```
template <typename F, typename... Fs>
constexpr decltype(auto) compose(F f, Fs... fs) {
  if constexpr (sizeof...(fs) < 1)
    return [f](auto &&x) -> decltype(auto) {
```

prior results that are keyed/indexed by the arguments. When a memoised function is called with previously seen arguments, the cached value is returned instead of recomputing it.



```
[f, …{g,h}](auto x) {
  return f( compose(g, h)( x ) )};
```

```
[g, …{h}](auto x) {
  return g( compose(h)(x) )};
```

```
[h](auto x) {
  return h(x)};
```

FIGURE 4.1: Pseudo C++ expansion of `compose(f,g,h)`. The `...{}` notation indicates the contents of the variadic parameter pack. The distracting noise from argument forwarding and `std::invoke` are removed. The boxes and lines indicate expansion of `comose` in each iteration.

```
      return std::invoke(f, std::forward<decltype(x)>(x));
    };
  else
    return [f, fs...](auto &&x) -> decltype(auto) {
      return std::invoke(
          f, compose(fs...)(std::forward<decltype(x)>(x)));
    };
}
```

This `compose` routine is recursively defined on a variadic list of arguments. Those arguments are the functions for composition. In the base case, where only one argument is given (and `sizeof...(fs) < 1`), the call to `compose` merely wraps the argument function in a lambda that perfectly forwards its argument to the given function. Successive iterations produce a nested structure of lambdas that take (by capture) copies of the functions provided in the argument list. As a simple example, Figure 4.1 illustrates in a sort of simplified C++ pseudocode how the compiler will expand `compose(f, g, h)`. While this implementation of `compose` is relatively economical in terms of Software Lines of Code (SLOC), it does suffer inefficiencies. Each layer contains copies of the functions in the layers below. Anecdotally, I have crudely benchmarked this implementation against others and there is a measurable squander of memory and CPU time. For composites of 4 functions or less, the **Cpp**/C++ code in this chapter, uses a less general, more verbose yet more straightforward (and explicitly typed) solution that can be found in LST. C.2.3.

In general, I recommended as alternatives alternatives the `compose` or `demux` functions from Louis Dionne's Boost.Hana [156], or `compose` from Paul Fultz's Fit [196] (which is now Boost.HOF [164]). Both of those alternatives are high quality implementations which have passed a peer review process (as all Boost libraries have).

4.2.1.2 2. Associativity of Composition

The C++ standards prescribe a strict evaluation strategy: expressions are evaluated as soon as they are bound to a variable or function argument. In a call such as `f(g(·), h(·), j(·))`, the arguments of `f` are evaluated ahead of the function body but the sequence in which the arguments are evaluated is explicitly unspecified by the standard [158, §8.2.2/5]. The order is known to differ among compilers and even architecture. If the functions `f`, `g`, and `h` are referentially transparent, the order is unobservable. It cannot change the behaviour of the program. So, referentially transparent functions compose associatively, satisfying C-2.

4.2.1.3 3. Compositional Identity

The identity axiom C-3 of a category requires, for each object, an identity arrow such that for all arrows f,

$$f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f.$$

In C++ we might approximate this as a template function:

```
template <typename T>
constexpr decltype(auto) id(T &&x) {
  return std::forward<T>(x);
}
```

This definition has some edge cases straining fidelity to C-3. Consider the following examples.

- The initialisations

  ```
  int adamsMeaning = id({41});
  const char kernighansGreeting[] = id("hello, world\n");
  ```

  will fail to compile because braced-initialisers or string literals are special cases in the context of initialisers as described in [158, §11.6] and wrapping them in a function removes them from that context.
- When `id` is applied to a returned expression (such as `return id(foo);`) it will interfere with Named Return Value Optimisation (NRVO), potentially forcing unnecessary copies. Those copies are not observable in a pure context[4], but may have an undesirable performance impact.
- When a temporary value is passed to `id` which is then used in an initializer

  ```
    Foo myFoo = id(Foo(x));
  ```

  `id` will interfere with lifetime extension of temporaries [158, §15.2/5] which may lead to unexpected access violations.

There are certainly more exceptions, so identity in ***Cpp*** is imperfectly witnessed by these proposed (or possibly any) identity functions. Within the context of the present research, those issues do not present practical problems.

In the demo repository [194], you can find unit test cases such as:

```
TEST_CASE("f == f ∘ id_A == id_B ∘ f.") {
  REQUIRE(f(A{}) == compose(f, id<A>)(A{}));
  REQUIRE(f(A{}) == compose(id<B>, f)(A{}));
  REQUIRE(compose(f, id<A>)(A{}) == compose(id<B>, f)(A{}));
}
```

[4] Impurities in the copy constructor can observe the copy. For example, if the copy constructor writes to a log.

REMARK 4.2-3. Since writing this section, an identity function, `std::identity`, was proposed in [167, §19.14.10] and added to the `<functional>` header in C++20.

As an aid to the C++ type system, we wrap the identity function in a `std::function`

```
template <typename T>
auto id = Hom<T, T>{tf::id<T>};
```

(where the afore-listed `id` function template originates in the `tf` namespace.)

4.3 ENDOFUNCTORS ON ***Cpp***

In C++ programming, the term functor is often used for an object with a call operator defined. Here we obviously use the term in the sense of CT defined in §B.6. Furthermore, an edofunctor on ***Cpp*** is called a *type functor*. That is, it is a map from types to types with corresponding action on functions. In C++ the "types-to-types" part is achievable through the template system. For example, `std::vector` allows us to map any type to a random-access list of elements of that type. But this has no intrinsic action on functions. For that, we need to define a map on arrows, usually called *fmap* in functional programming.

DEFINITION 4.3-1. In C++ a ***type functor*** consists of

- a map of ***Cpp*** objects manifest as a class/struct template of a single type variable: `template<typename> F`,
- a map of arrows—a function of type $(T \to U) \to (F\langle T\rangle \to F\langle U\rangle)$, that preserves composition and identity.

Preserving composition and identity means that the image of the diagram,

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend right, "g\circ f"'] & B \arrow[r, "g"] & C \end{tikzcd},$$

through the functor, commutes. For the purpose of discussion we can organise the former definition in a `namespace` [5] that groups the relevant object-map as an alias template, `Of`, and the arrow-map as a function template, `fmap`.

```
namespace F {
  template <typename T>
  using Of = …;

  template <typename Fn>
  auto fmap(Fn f) -> Hom<Of<Dom<Fn>>, Of<Cod<Fn>>>;
}; // namespace F
```

Now we can be clear about what it means to preserve composition and identity. If we draw the arrow closure diagram in ***Cpp***, its image through a type functor F should commute as well:

$$\begin{tikzcd}
\boldsymbol{Cpp} & F::\mathtt{Of}\langle T\rangle \arrow[r, "F::\mathtt{fmap}(f)"'] \arrow[rr, bend left, "F::\mathtt{fmap}(\mathtt{compose}(g,f))"] \arrow[loop above, "\mathtt{id}\langle F::\mathtt{Of}\langle T\rangle\rangle"] & F::\mathtt{Of}\langle U\rangle \arrow[r, "F::\mathtt{fmap}(g)"'] \arrow[loop above, "\mathtt{id}\langle F::\mathtt{Of}\langle U\rangle\rangle"] & F::\mathtt{Of}\langle V\rangle \arrow[loop above, "\mathtt{id}\langle F::\mathtt{Of}\langle V\rangle\rangle"] \\
\boldsymbol{Cpp} \arrow[u, "F"] & A \arrow[r, "f"'] \arrow[rr, bend right, "\mathtt{compose}(g,f)"'] \arrow[loop below, "\mathtt{id}\langle T\rangle"'] & B \arrow[r, "g"'] \arrow[loop below, "\mathtt{id}\langle U\rangle"'] & C\,. \arrow[loop below, "\mathtt{id}\langle V\rangle"']
\end{tikzcd} \tag{4.3.1}$$

The bottom commutes axiomatically since ***Cpp*** is a category. The top will commute axiomatically if F is a functor. Using the tools described in **4.1-1**, we can write unit-test `REQUIRE`ments based on the commutativity conditions of (4.3.1).

```
TEST_CASE("Check the functor laws for F") {
  auto fa = F::Of<A>{…};

  // F::fmap(g) ∘ F::fmap(f) = F::fmap(g ∘ f)
  REQUIRE(
```

[5] … or, similarly, a `struct`.

```
    compose(F::fmap(g), F::fmap(f))(fa)
        ==
            F::fmap(compose(g, f))(fa)
  );

  // F::fmap(id⟨−⟩) = id⟨f::Of⟨−⟩⟩
  REQUIRE(F::fmap(id<A>)(fa) == id<F::Of<A>>(fa));
}
```

EXAMPLE 4.3-2 (`std::vector` as a List-functor). We can regard `std::vector⟨T⟩` as an object map and define an arrow map to complete a functor. The typical arrow map on linear (list- or array-like) structures behaves ask

$$\texttt{fmap}\,(f)\left([\,a_1,\, a_2,\, \dots,\, a_N\,]\right) = [\,f(a_1),\, f(a_2),\, \dots,\, f(a_N)\,]$$

The underlying algorithm for this exists in the STL `algorithm` header as `std::transform`. We need only provide the `fmap` interface to it:

```
namespace Vector {
  template <typename T>
  using Of = std::vector<T>;

  template <typename Fn>
  auto fmap(Fn fn) -> Hom<Of<Dom<Fn>>, Of<Cod<Fn>>> {
    using T = Dom<Fn>;
    using U = Cod<Fn>;
    return [fn](Of<T> t_s) {
      Of<U> u_s;
      u_s.reserve(t_s.size());

      std::transform(
          cbegin(t_s), cend(t_s), std::back_inserter(u_s), fn);
      return u_s;
    };
  };
}; // namespace Vector
```

And now check that the functor preserves composition and identity in ***Cpp***:

```
TEST_CASE("Check the functor laws for Vector::fmap") {
  //        Alias for std::vector<A>
  //                 ↓
  auto a_s = Vector::Of<A>{A{}, A{}, A{}};
  REQUIRE(
    compose(Vector::fmap(g), Vector::fmap(f))(a_s)
        ==
            Vector::fmap(compose(g, f))(a_s));

  REQUIRE(Vector::fmap(id<A>)(a_s) == id<Vector::Of<A>>(a_s));
}
```

This next example will be a little different than `Vector` because we will be creating a family of functors—one for each type in ***Cpp***. It is an important example because it is a prelude to two important concepts: 1. natural transformations, and 2. bifunctors.

EXAMPLE 4.3-3 (Constant functor)**.** In introductory CT texts, the *constant functor* is often one of the first examples. In ***Cpp***, a constant type functor maps all ***Cpp***-objects to a single type, *T*, and all ***Cpp***-arrows to id⟨*T*⟩.

One such functor exists for each object in ***Cpp***, but each one behaves the same way. It is natural to write code that will produce such a functor over any given type. That is, for any type *T* we create a constant functor with the object map Of⟨*U*⟩ = *T*, for all *U*, and fmap(*f*) = id⟨*T*⟩. The most straightforward way (that I know of) to make such a thing is to nest templates

```
template <typename T>
struct Always {
  template <typename>
  using Given = T;
};
```

So Always⟨*T*⟩::Given⟨*U*⟩ *always* evaluates to the typename *T* for all *U* ∈ ***Cpp***. Now we would define the functor Const⟨*T*⟩ that, for any *T*. In the previous example, we organised `Vector` in a namespace; but now that we have a family of functors, we need to template the functor on *T*. So instead of using a namespace, we switch to a `struct` and make the `fmap` a `static` member

```
template <typename T>
```

```
struct Const {

  template <typename U>
  using Of = typename Always<T>::template Given<U>;

  template <typename Fn>
  static auto fmap(Fn) -> Hom<T, T> {
    return id<T>;
  };
};
```

Now confirm that `Const` is functorial, (at least for the case of `Const⟨A⟩`)

```
TEST_CASE("Functor axioms of Const<A>.") {
  REQUIRE(
    compose(Const<A>::fmap(g), Const<A>::fmap(f))(A{})
      ==
        Const<A>::fmap(compose(g, f))(A{})
  );

  REQUIRE(Const<A>::fmap(id<A>)(A{}) == A{});
  // Interestingly, id<B> will be mapped to id<A> too:
  REQUIRE(Const<A>::fmap(id<B>)(A{}) == A{});
  // as will id<C>:
  REQUIRE(Const<A>::fmap(id<C>)(A{}) == A{});
}
```

EXAMPLE 4.3-4 (`sptr` non-functor). C++ (via its C heritage) has a system by which we can obtain and manipulate the addresses of various objects and entities. *Pointers* hold these addresses and come with the *dereferencing* operation, denoted with the unary-star operator $\ast(-)$, by which we can retrieve the value at the end of a pointer. Pointers are typed so that we know how to interpret those values. (Never mind the nasty business of void-pointers.) ISO/IEC 14882:2011 (C++11) [116] introduced *smart pointers*, which hold manifold benefits over working with raw C-style pointers.

It will later be useful to adapt a normal function in `Hom`⟨*T*, *U*⟩ to accept and return `std::shared_ptr`s: `Hom`⟨`std::shared_ptr`⟨*T*⟩, `std::shared_ptr`⟨*U*⟩⟩ This may smell of a functor, but not quite. Consider this implementation:

```
namespace sptr {
  template <typename T>
  using Of = std::shared_ptr<T>;

  template <typename Fn>
  auto map(Fn fn) -> Hom<Of<Dom<Fn>>, Of<Cod<Fn>>> {
    return [fn](Of<Dom<Fn>> x_ptr) -> Of<Cod<Fn>> {
      const auto result = fn(*x_ptr);
      using Result = std::remove_cv_t<decltype(result)>;

      return std::make_shared<Result>(result);
    };
  }
} // namespace sptr
```

Here, I have refrained from using the name `fmap`, instead using `map`. This is because `sptr` is not functorial: it does not preserve commuting triangles or identities. Rather, it is functorial *up to natural transformation*:

$$\begin{tikzcd}
\texttt{sptr::Of}\langle T\rangle \arrow[loop above, "\texttt{id}\langle\texttt{sptr::Of}\langle T\rangle\rangle"] \arrow[r, squiggly, "\texttt{sptr::fmap}(f)"'] \arrow[rr, squiggly, bend left, "\texttt{sptr::fmap}(\texttt{compose}(g,f))"] \arrow[d, "*(\texttt{-})"] & \texttt{sptr::Of}\langle U\rangle \arrow[loop above, "\texttt{id}\langle\texttt{sptr::Of}\langle U\rangle\rangle"] \arrow[r, squiggly, "\texttt{sptr::fmap}(g)"'] \arrow[d, "*(\texttt{-})"] & \texttt{sptr::Of}\langle V\rangle \arrow[loop above, "\texttt{id}\langle\texttt{sptr::Of}\langle V\rangle\rangle"] \arrow[d, "*(\texttt{-})"] \\
A \arrow[loop below, "\texttt{id}\langle T\rangle"] \arrow[r, "f"'] \arrow[rr, bend right, "\texttt{compose}(g,f)"'] & B \arrow[loop below, "\texttt{id}\langle U\rangle"] \arrow[r, "g"'] & C\,.\arrow[loop below, "\texttt{id}\langle V\rangle"]
\end{tikzcd}$$

In the diagram above, the squiggle-arrows indicate that those do not participate in the commutativity of their interior. Instead, we have to dereference the values (using the unary-star operator) to get commutativity. Given a pointer `p` of type `std::shared_ptr`$\langle T\rangle$, if we called `sptr::map(f)(p)` several times, none of the results would be equal because we would get a pointer to a new piece of memory each time. On pointers, equality comparison checks if two pointers point to the same memory address. So if the values at the ends of the pointers are the same, the addresses are different and therefore not equal. This is why we must dereference before we get any satisfying compositional properties.

Here is the usual functor unit-tests for `sptr`, with the subtle change:

```
TEST_CASE("ptr is functorial 'up to natural transformation'") {
  auto a_ptr = std::make_shared<A>();

  REQUIRE(
    *compose(sptr::map(g), sptr::map(f))(a_ptr)
        ==
            *sptr::map(compose(g, f))(a_ptr)
  );

  REQUIRE(id<sptr::Of<A>>(a_ptr) == a_ptr);

  REQUIRE(*sptr::map(id<A>)(a_ptr) == *id<sptr::Of<A>>(a_ptr));

  // Note that we have to dereference for the test.
  // This is not a functor without
}
```

Note well that the familiar functor laws are embellished with dereferencing operations. So we can build composite in the image of `sptr`, but commutativity does not hold until we drop back down into the preimage, through the dereferencing operation.

* * *

4.3-5 Grouping the object and arrow maps in a namespace or `struct` is simply an organisational convenience. In practice, it serves the programmer perfectly well to have `fmap` as a free function or member function—with a name that best reflects what `fmap` means in the context of that functor. For example, ISO proposal P0798 [155] proposes `transform` as a member function of `std::optional` which acts as an `fmap` on the so called *maybe functor* based on `std::optional`. Later in the chapter, we will define the categorical products and coproducts and it will be more convenient to have their arrow-maps as free functions with more semantically relevant names. From a mathematical point of view, the important things are that the object-map works for all template typename arguments, and that the arrow map preserves the structure of the category.

4.4 NATURAL TRANSFORMATIONS IN *Cpp*

Natural transformations, (§B.7, DEF. B.7-1) are functor-homomorphisms, describing how to map one functor into another. A natural transformation in ***Cpp*** between type functors F and G can be drawn as

$$\begin{tikzcd} \mathbf{Cpp} \arrow[r, bend left, "F", ""{name=U, below}] \arrow[r, bend right, "G"', ""{name=D}] & \mathbf{Cpp} \arrow[Rightarrow, from=U, to=D, "eta"] \end{tikzcd} \quad \text{or more compactly as} \quad eta : F \Rightarrow G.$$

It represents a family of functions,

$$\Big(F::\mathtt{Of}\langle T\rangle \xrightarrow{eta\langle T\rangle} G::\mathtt{Of}\langle T\rangle \Big)_{T \in \mathbf{Cpp}},$$

subject to naturality constraints. They are straightforwardly represented as a special kind of function template.

DEFINITION 4.4-1. Consider type functors F and G. In ***Cpp***/C++, a template function is a natural transformation if, for all T in ***Cpp***, it can be viewed as a member of

```
template<typename T> Hom<F<T>, G<T>>;
```

subject to the commutativity of the naturality square:

$$\begin{tikzcd} G::\mathtt{Of}\langle T\rangle \arrow[r, "G::\mathtt{fmap}(f)"] & G::\mathtt{Of}\langle U\rangle \\ F::\mathtt{Of}\langle T\rangle \arrow[u, "eta\langle T\rangle"] \arrow[r, "F::\mathtt{fmap}(f)"'] & F::\mathtt{Of}\langle U\rangle \arrow[u, "eta\langle U\rangle"'] \end{tikzcd} \tag{4.4.2}$$

Not all C++ function templates can be viewed as natural transformations. C++ function templates are *ad-hoc* polymorphic while the afore DEF. 4.4-1 implies that, in order to be a natural transformation, the function template must be *parametrically polymorphic*. This means that the function template must be well formed for any and all template type arguments $T \in$ ***Cpp***, without specialisation. The body of the function template can only deal in the structure of F and G and must not affect contained T values.

EXAMPLE 4.4-2. Consider the `Vector` functor from earlier. The only thing a `Vector::Of⟨int⟩` and a `Vector::Of⟨bool⟩` have in common is the `std::vector` interface. A function that tells you the length of a `std::vector` obviously has no regard the type of the contained elements, and therefore can be viewed as a natural transformation from `Vector` to the constant functor at `std::size_t`.

To demonstrate that, we can define the natural transformation,

$$\texttt{len} : \texttt{Vector} \Rightarrow \texttt{Const}\langle\texttt{std::size\_t}\rangle,$$

which simply returns the result from `std::vector`'s `size()` member function:

```
template <typename T>
auto len(Vector::Of<T> t_s) -> Const<std::size_t>::Of<T> {
  return t_s.size();
}
```

We can test against the naturality square (4.4.2):

```
TEST_CASE("Test naturality square for len.") {
  constexpr std::size_t actual_length = 5;

  auto a_s = Vector::Of<A>(actual_length);

  // Does what it is supposed to:
  REQUIRE(len(a_s) == actual_length);

  // Satisfies the naturality square:
  REQUIRE(
    compose(len<B>, Vector::fmap(f))(a_s)
        ==
          compose(Const<std::size_t>::fmap(f), len<A>)(a_s)
  );
}
```

This is not an exhaustive test since `actual_length` is fixed; but it makes the point.

*   *   *

In the afore sections of this chapter, I have defined the category ***Cpp***. For this category to have practical substance, it was developed in the text alongside a representation denoted ***Cpp***/C++. This representation demonstrates conformity to the category axioms and the ability to express type functors and their natural transformations. I would now augment ***Cpp*** with the structure of a biccc and further develop ***Cpp***/C++ to support it. This will enable us to model programs in terms of algebraic datatypes with operations of pairing, projection, disjointly uniting, choosing/visiting, function application and currying.

4.5 CARTESIAN MONOIDAL STRUCTURE IN *Cpp*

A natural approach to developing ***Cpp*** as a ccc[6] is to start by demonstrating the existence of a categorical product and then extend it with associativity, identity, self-inverse braiding (creating a symmetric monoidal structure) and currying (which closes the monoid). This approach is executed in this section and the next. In this section focuses on the development of a cartesian monoidal structure in ***Cpp*** as well as a representation in C++ code. In the next section, §4.6, the cartesian monoid is given closure by demonstrating internal homs, internal evaluation and currying with respect to the cartesian product.

THE CATEGORICAL PRODUCT The construction of products [7] in ***Cpp*** echos their construction in $\mathbf{Set}$ (§B.9.1, p. 290). Since ***Cpp*** is modelled on $\mathbf{Set}$ and the categorical product of $\mathbf{Set}$ is the familiar set-theoretical Cartesian product, we start from there.

Consider a type-bifunctor P. The forgetful functor $U_{\mathit{Cpp}} : \mathit{Cpp} \rightarrow \mathbf{Set}$ associates each ***Cpp*** type to its underlying set of values in $\mathbf{Set}$. The categorical product in $\mathbf{Set}$ is the cartesian product. Any T, $U \in \mathit{Cpp}$ are associated

[6] DEF. B.15-8, p. 309

[7] §B.9, p. 288.

through F to the sets T, U. Furthermore, se associate $T \times U$ to the image $\mathtt{P}\langle T,\ U\rangle$ through $U_{\mathbf{Cpp}}$.

A binary Cartesian product is well modelled by the STL class template, `std::pair`:

```
template <typename T, typename U>
struct P : std::pair<T, U> {
  using std::pair<T, U>::pair;
};
```

The pneumonic here is `P`-roduct, since pairs and tuples are usually called product types.

Recall that a (binary) categorical product is a limit of a discrete diagram with two objects. A cone in ***Cpp*** over this diagram is a span: a tuple of (C, f, g), related as: $T \xleftarrow{f} C \xrightarrow{g} U$. The limit of such is a terminal cone: a cone through which every other cone uniquely factors.

Now consider `P` at the apex of such a limit cone, through which any other cone factors:

$$\begin{tikzcd}
& C \arrow[ld, "f"'] \arrow[d, dashed, "\exists!\ u"'] \arrow[rd, "g"] & \\
T & \mathtt{P}\langle T,\ U\rangle \arrow[l, "\mathtt{proj\_l}"] \arrow[r, "\mathtt{proj\_r}"'] & U\ .
\end{tikzcd} \tag{4.5.3}$$

This prescribes a unique u that facilitates the commutativity of the left and right triangles, each subtended by the projections. The projects are natural transformations, though we need not focus on naturality: it is sufficient that they participate in the commutativity of the diagram. It does however mean that they will be represented as template functions in ***Cpp***/C++:

```
template <typename T, typename U>
auto proj_l(P<T, U> tu) -> T {
  return std::get<0>(tu);
}

template <typename T, typename U>
auto proj_r(P<T, U> tu) -> U {
  return std::get<1>(tu);
}
```

We can witness P at the apex of a terminal cone by defining the unique u, which is an operator parameterised by the functions f and g. Often called fanout, u maps $c \mapsto \{f(c), g(c)\}$. In ***Cpp***/C++:

```
template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename X = Cod<Fn>, typename Y = Cod<Gn>>
auto fanout(Fn fn, Gn gn) -> Hom<T, P<X, Y>> {
  static_assert(std::is_same_v<T, U>);
  return [fn, gn](auto t) -> P<X, Y> {
    static_assert(std::is_invocable_v<Fn, decltype(t)>);
    static_assert(std::is_invocable_v<Gn, decltype(t)>);

    return {fn(t), gn(t)};
  };
}
```

which we can see occasions the commutativity of (4.5.3):

```
TEST_CASE(
    "Commutativity of left and right triangles in (4.7.7)") {
  auto a_to_c = [](A) { return C{}; };
  auto b_to_c = [](B) { return C{}; };

  auto left_triangle_path =
      compose(fanin(a_to_c, b_to_c), inject_l<A, B>);
  REQUIRE(left_triangle_path(A{}) == a_to_c(A{}));

  auto right_triangle_path =
      compose(fanin(a_to_c, b_to_c), inject_r<A, B>);
  REQUIRE(right_triangle_path(B{}) == b_to_c(B{}));
}
```

when T, U, $V =$ A, B, C.

In C++, "plain ol' data" (POD) structures can also be regarded as types with projections provided by member access *via* the dot-operator. It is difficult to define the fan-out operation for these structures in general, but nonetheless, they are amply fit for most purposes of product types. Based on the std::pair datatype as a product, we turn our attention in pursuit of a braided monoidal structure DEF. B.11-7.

A braided monoidal structure on P is the tuple

$$(\texttt{P},\ I,\ \texttt{associator\_fd},\ \texttt{l\_unitor},\ \texttt{r\_unitor},\ \texttt{braid})$$

where

- P, as the product, must be shown as a bifunctor.
- I is a terminal ***Cpp***-object acting as the monoidal unit
- associator_fd, the associator, a natural isomorphism with with components

  $$\texttt{associator\_fd}\langle T,\ U,\ V\rangle :\ \texttt{P}\langle T,\ \texttt{P}\langle U,\ V\rangle\rangle \xrightarrow{\sim} \texttt{P}\langle \texttt{P}\langle T,\ U\rangle,\ V\rangle,$$

  for all $T, U, V \in$ ***Cpp***, with inverse associator_rv.
- l_unitor_fd and r_unitor_fd, natural isomorphisms called (respectively) the left- and right-unitors with components of the forms

  $$\texttt{l\_unitor\_fd}\langle T\rangle :\ \texttt{P}\langle I,\ T\rangle \xrightarrow{\sim} T$$
  $$\texttt{r\_unitor\_fd}\langle T\rangle :\ \texttt{P}\langle T,\ I\rangle \xrightarrow{\sim} T,$$

  for all $T \in$ ***Cpp***, with inverses postfixed _rv instead of _fd.
- braid, the braiding natural isomorphism with components

  $$\texttt{braid}\langle T,\ U\rangle :\ \texttt{P}\langle T,\ U\rangle \xrightarrow{\sim} \texttt{P}\langle U,\ T\rangle,$$

  which are self inverse.

These data are subject the commutativity of the associator (B.11.14), the unitor (B.11.15) and the braiding (B.11.16) diagrams. The following titled passages give ***Cpp***/C++ implementations along with unit tests demonstrating their facility to make those diagrams commute.

PAIR-BIFUNCTOR In order to construct the morphisms of the associator, unitor and braiding diagrams, we must be able to construct products of *functions*, and not just types. This means that P must be bifunctorial—functorial in both the left and right parameters: P : ***Cpp*** × ***Cpp*** → ***Cpp***. In the category of sets, the cartesian product of functions, denoted $f \times g$, is a map from $(\operatorname{dom} f) \times (\operatorname{dom} g)$ to $(\operatorname{cod} f) \times (\operatorname{cod} g)$, defined elementwise by $(x, y) \mapsto (f\,x, g\,y)$. This is a *bimap* (two-variable fmap) of a bifunctor. The ***`Cpp`***/C++ spelling is much less terse:

```
template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename X = Cod<Fn>, typename Y = Cod<Gn>>
auto prod(Fn fn, Gn gn) -> Hom<P<T, U>, P<X, Y>> {
  return [fn, gn](P<T, U> tu) -> P<X, Y> {
    auto [t, u] = tu;
    return {fn(t), gn(u)};
  };
}
```

Here, I abuse the default template argument system somewhat, using it to define *`T`*, *`U`*, *`X`* and *`Y`* before the function declarator in order to uncomplicate the function's signature.

As per **4.3-5**, `prod`, the bimap of `P`, is left as a free function (template) without structurally associating it to `P` using a namespace or `struct`.

We can confirm that `prod` of functions makes `P` functorial in the left and right factors individually, using the same `REQUIRE`ments as for `Functor`. We simply place `id` in the factor we are not testing:

```
TEST_CASE("P (via prod) is functorial in both factors.") {
  auto ab = P<A, B>{};

  REQUIRE(
    compose(prod(g, id<B>), prod(f, id<B>))(ab)
      ==
        prod(compose(g, f), id<B>)(ab)
  );

  REQUIRE(
    compose(prod(id<A>, h), prod(id<A>, g))(ab)
      ==
        prod(id<A>, compose(h, g))(ab)
  );

  REQUIRE(prod(id<A>, id<B>)(ab) == id<P<A, B>>(ab));
}
```

THE ASSOCIATOR The associator for `P` mediates an equivalence of various nested groupings of products. Specifically, it mediates the isomorphism:

$$\mathtt{P}\langle T,\ \mathtt{P}\langle U,\ V\rangle\rangle \cong \mathtt{P}\langle \mathtt{P}\langle T,\ U\rangle,\ V\rangle,$$

in forward (`_fd`) and reverse (`_rv`) directions. Note that the order of variables, $(T,\ U,\ V)$, is the same on the left and right sides of the above equation and the difference is whether the inner pairing is on the left or right factor of the outer pairing. The associator must satisfy commutativity of the associator diagram:

$$\begin{tikzcd}
\mathtt{P}\langle \mathtt{P}\langle T,\ U\rangle,\ \mathtt{P}\langle V,\ W\rangle\rangle \arrow[dr, bend left, "\mathtt{associator\_fd}\langle \mathtt{P}\langle T, U\rangle, V, W\rangle"] & \\
\mathtt{P}\langle T,\ \mathtt{P}\langle U,\ \mathtt{P}\langle V,\ W\rangle\rangle\rangle \arrow[u, "\mathtt{associator\_fd}\langle T, U, \mathtt{P}\langle V, W\rangle\rangle"] \arrow[d, "\mathtt{prod(id}\langle T\rangle\mathtt{, associator\_fd}\langle U, V, W\rangle\mathtt{)}"'] & \mathtt{P}\langle \mathtt{P}\langle \mathtt{P}\langle T,\ U\rangle,\ V\rangle,\ W\rangle \\
\mathtt{P}\langle T,\ \mathtt{P}\langle \mathtt{P}\langle U,\ V\rangle,\ W\rangle\rangle \arrow[r, "\mathtt{associator\_fd}\langle T, \mathtt{P}\langle U, V\rangle, W\rangle"'] & \mathtt{P}\langle \mathtt{P}\langle T,\ \mathtt{P}\langle U,\ V\rangle\rangle,\ W\rangle\ . \arrow[u, "\mathtt{prod(associator\_fd}\langle T, U, V\rangle\mathtt{,id}\langle W\rangle\mathtt{)}"']
\end{tikzcd}$$

The associator can be represented, in forward and reverse directions, by the template functions:

```
template <typename T, typename U, typename V>
auto associator_fd(P<T, P<U, V>> t_uv) -> P<P<T, U>, V> {
  auto [t, uv] = t_uv;
  auto [u, v] = uv;

  return {{t, u}, v};
}

template <typename T, typename U, typename V>
auto associator_rv(P<P<T, U>, V> tu_v) -> P<T, P<U, V>> {
  auto [tu, v] = tu_v;
  auto [t, u] = tu;

  return {t, {u, v}};
}
```

It may be obvious from the definitions, but we should confirm that these are mutually inverse:

```
TEST_CASE(
    "associator_fd and associator_rv are mutually "
    "inverse.") {
  auto associator_fd_rv = compose(
        associator_rv<A, B, C>,
        associator_fd<A, B, C>
      );
  auto associator_rv_fd = compose(
        associator_fd<A, B, C>,
        associator_rv<A, B, C>
      );

  auto a_bc = P<A, P<B, C>>{};
  REQUIRE(associator_fd_rv(a_bc) == id<P<A, P<B, C>>>(a_bc));
  auto ab_c = P<P<A, B>, C>{};
  REQUIRE(associator_rv_fd(ab_c) == id<P<P<A, B>, C>>(ab_c));
}
```

We should also test them against the associator diagram:

```
TEST_CASE("Associator diagram for P") {
  auto start = P<A, P<B, P<C, D>>>{};

  auto cw_path = compose(
        associator_fd<P<A, B>, C, D>,
        associator_fd<A, B, P<C, D>>
      );

  auto ccw_path = compose(
        prod(associator_fd<A, B, C>, id<D>),
        associator_fd<A, P<B, C>, D>,
        prod(id<A>, associator_fd<B, C, D>)
      );

  REQUIRE(ccw_path(start) == cw_path(start));
};
```

THE UNITOR In analogy to the school algebreic equiation $1{\cdot}x = x{\cdot}1 = x$, the unitor structure mediates the isomorphism

$$\mathtt{P}\langle \mathtt{I},\ A\rangle \cong \mathtt{P}\langle A,\ \mathtt{I}\rangle \cong A.$$

The identity element, `I`, can be any singleton type is suitable. as they are all terminal objects [8]of ***Cpp***. Since the C++ type system forces us to commit to a name let us define[9]

```
struct I { bool operator==(const I) const { return true; } };
```

which is a singleton type equipped with equality comparison. (This equality comparison is very simple because the *I*-value is unique and any two instances must be equal.)

Category theory does not afford us the tools to distinguish a singleton by peeking at its contents. As always, we look to the arrows without to tell us what must be within. A terminal object in a category is a terminus of a unique arrow from every other object. In ***Cpp***, there is exactly one left-total function from any *T* to `I`:

$$\texttt{Hom}\langle T,\ \texttt{I}\rangle \cong \left\{\texttt{[](T)\{ return I\{\}; \}}\right\} \cong \texttt{I}.$$

Terminal objects in ***Cpp*** are also interesting because the endow the category with an internal notion of cardinality and give us *global elements*[10]. More explicitly, the set of all functions from `I` to any *T* is isomorphic to *T* itself:

$$\texttt{Hom}\langle \texttt{I},\ T\rangle \cong \left\{\, \texttt{[](I)\{ return } t\texttt{; \}} \;\middle|\; t \in T \,\right\} \cong T.$$

The values of $\texttt{P}\langle \texttt{I},\ T\rangle$ can be thought of as $\{\, (i,\ t) \mid i = \texttt{I\{\}} \text{ and } t \in T \,\}$. Because there is no variety in the first factor, i, it can be discarded from the pair $(i,\ t)$ without data loss: we can simply add it back without needing to remember which *I*-value we discarded! The unitors, in forward and reverse directions, merely ablate or adjoin (respectively) the i:

```
template <typename T>
auto l_unitor_fw(P<I, T> it) -> T {
  return std::get<1>(it);
}
```

[8] DEF. B.3-1, p. 271

[9] This is identical to the definitions of `A`, `B`, `C` and `D` in the demo suite [194].

[10] §B.4, p. 272.

```
template <typename T>
auto l_unitor_rv(T t) -> P<I, T> {
  return {I{}, t};
}

template <typename T>
auto r_unitor_fw(P<T, I> ti) -> T {
  return std::get<0>(ti);
}

template <typename T>
auto r_unitor_rv(T t) -> P<T, I> {
  return {t, I{}};
}
```

We ensure that the `_fw` and `_rv` are mutually inverse:

```
TEST_CASE("_fw and _rv are mutual inverses for L-/R-unitor") {
  auto ia = P<I, A>{};
  auto ai = P<A, I>{};

  REQUIRE(
      compose(l_unitor_rv<A>, l_unitor_fw<A>)(ia) ==
          id<P<I, A>>(ia)
  );
  REQUIRE(
      compose(l_unitor_fw<A>, l_unitor_rv<A>)(A{}) ==
          id<A>(A{})
  );

  REQUIRE(
      compose(r_unitor_rv<A>, r_unitor_fw<A>)(ai) ==
          id<P<A, I>>(ai)
  );
  REQUIRE(compose(r_unitor_fw<A>, r_unitor_rv<A>)(A{}) ==
          id<A>(A{})
  );
}
```

The unitor triangle depicts the interplay between the left- and right-unitors with the associator:

$$\begin{tikzcd}
\mathtt{P}\langle T,\ \mathtt{P}\langle I,\ U\rangle\rangle \arrow[rr, "\mathtt{associator\_fd}\langle T, I, U\rangle"] \arrow[dr, "\mathtt{prod}(\mathtt{id}\langle T\rangle, \mathtt{l\_unitor\_fd}\langle U\rangle)"'] & & \mathtt{P}\langle \mathtt{P}\langle T,\ I\rangle,\ U\rangle\ . \arrow[dl, "\mathtt{prod}(\mathtt{r\_unitor\_fd}\langle T\rangle, \mathtt{id}\langle U\rangle)"] \\
& \mathtt{P}\langle T,\ U\rangle &
\end{tikzcd} \tag{4.5.4}$$

and we test the implementations against this diagram:

```
TEST_CASE("Commutativity of the Unitor diagram, (4.5.4).") {
  auto a_ib = P<A, P<I, B>>{};

  auto cw_path = compose(
        prod(r_unitor_fw<A>, id<B>),
        associator_fd<A, I, B>
      );
  auto ccw_path = prod(id<A>, l_unitor_fw<B>);

  REQUIRE(cw_path(a_ib) == ccw_path(a_ib));
}
```

THE BRAIDING While it may seem obvious that $\mathtt{P}\langle T,\ U\rangle \cong \mathtt{P}\langle U,\ T\rangle$ (since it is only a matter of swapping the factors), it is not generally the case in monoidal structures. The braiding is a pair of natural transformations that witness the isomorphism. In general, braiding must satisfy two hexagonal diagrams that structure appropriate interplay with the associator (DEF. B.11-7). For P, it is also true that the forward braiding will be self-inverse, since swapping two things and then swapping them again leaves them in their original positions. Happily, this means we are pursuing a symmetric monoidal structure; and one of the hexagons implies the other. (See [81, §XI.1, p. 253].) Choosing one arbitrarily, this is the braiding hexagon in ***Cpp*** we will test against:

$$\begin{tikzcd}
& \mathsf{P}\langle \mathsf{P}\langle T,\, U\rangle,\, V\rangle \arrow[r, "\texttt{braid}\langle \texttt{P}\langle T, U\rangle, V\rangle"] \arrow[dl, bend right, "\texttt{associator\_rv}\langle T, U, V\rangle"] & \mathsf{P}\langle V,\, \mathsf{P}\langle T,\, U\rangle\rangle \arrow[dr, bend left, "\texttt{associator\_fd}\langle V, T, U\rangle"'] & \\
\mathsf{P}\langle T,\, \mathsf{P}\langle U,\, V\rangle\rangle \arrow[dr, bend right, "\texttt{prod}(\texttt{id}\langle T\rangle, \texttt{braid}\langle U, V\rangle)"] & & & \mathsf{P}\langle \mathsf{P}\langle V,\, T\rangle,\, U\rangle \arrow[dl, bend left, "\texttt{prod}(\texttt{braid}\langle V, T\rangle, \texttt{id}\langle U\rangle)"'] \\
& \mathsf{P}\langle T,\, \mathsf{P}\langle V,\, U\rangle\rangle \arrow[r, "\texttt{associator\_fd}\langle T, V, U\rangle"'] & \mathsf{P}\langle \mathsf{P}\langle T,\, V\rangle,\, U\rangle\ , &
\end{tikzcd} \tag{4.5.5}$$

The braider can be implemented as the following template function:

```
template <typename T, typename U>
auto braid(P<T, U> tu) -> P<U, T> {
  auto [t, u] = tu;
  return {u, t};
}
```

and we can show it is self-inverse:

```
TEST_CASE("Braiding is self-inverse") {
  auto ab = P<A, B>{};
  REQUIRE(braid(braid(ab)) == id<P<A, B>>(ab));
}
```

With symmetry confirmed, we can check the chosen braiding hexagon:

```
TEST_CASE("Braiding diagram (4.5.5)") {
  auto ab_c = P<P<A, B>, C>{};

  auto cw_path = compose(
        prod(braid<C, A>, id<B>),
        associator_fd<C, A, B>,
        braid<P<A, B>, C>
      );
  auto ccw_path = compose(
        associator_fd<A, C, B>,
        prod(id<A>, braid<B, C>),
        associator_rv<A, B, C>
      );

  REQUIRE(cw_path(ab_c) == ccw_path(ab_c));
}
```

4.6 CARTESIAN CLOSURE ***Cpp***

A cartesian closed category is one in which the categorical product forms a monoidal structure (a cartesian monoidal category) and additionally has internal homs[11] or *exponentials*. That is to say that a certain constellation of objects and arrows indicate that the category has some internal reckoning of the structure of its own arrows. In the category of sets, we should not be surprised to find such an arrangement. After all, the set of all functions between two sets is itself a set—and is therefore a constituent among the category's objects. Since category theory affords no tools of introspection of the objects, it is the arrows, as always, that reveals the pattern.

In ***Cpp***, the notion of "sets of functions" will have something to do with representing functions as first-class citizens of the language; represented as values that can be assigned to variables, passed to or returned from other functions, and alike. We have *already* been doing is in ***Cpp***/C++, but I only now justify it.

Specifically, we have been using `Hom` as a shorthand for `std::function` which is a wrapper for any manner of internal representations of "callable" types. Even though `std::function` internally employs type erasure, its instances are type-safe and under the jurisdiction of the type system; it does therefore model internal homs.

Cartesian closure can be observed by the existence of exponentials, or equivalently, as adjunction between the functors $\mathtt{P}\langle -, T\rangle$ and $\mathtt{Hom}\langle T, -\rangle$. Specifically, in ***Cpp*** this means we want to prove an isomorphism of hom-sets:

$$\mathtt{Hom}\langle \mathtt{P}\langle T, U\rangle, V\rangle \cong \mathtt{Hom}\langle T, \mathtt{Hom}\langle U, V\rangle\rangle$$

meaning that $\mathtt{Hom}\langle T, -\rangle$ is right-adjoint[12] to $\mathtt{P}\langle -, T\rangle$, for T, U, V spanning the objects of ***Cpp***. In the forward direction, this morphism of hom-sets is

[11] DEF. B.15-2

[12] Adjunctions are covered in §B.16.

called *currying*. Currying allows us to wield functions with argument lists partially applied.

Though the exponential view and adjunction views are equivalent, they each bring slightly different supportive tooling, and we are going to use both. The two views share in common the notion of currying. The adjoint view requires that currying have an inverse *uncurry*. The exponential view requires currying and the existence of an *internal evaluator*, `ev`, making the following diagram commute for all T, U, V and $k : \mathtt{P}\langle T, U\rangle \to V$:

$$\begin{tikzcd}
\mathtt{Hom}\langle U, V\rangle & & \mathtt{P}\langle \mathtt{Hom}\langle U, V\rangle, U\rangle \arrow[r, "\mathtt{ev}\langle T, U\rangle"] & V , \\
T \arrow[u, "\mathtt{pcurry}(k)"] & & \mathtt{P}\langle T, U\rangle . \arrow[u, dashed, "\mathtt{prod}(\mathtt{pcurry}(k), \mathtt{id}\langle U\rangle)"] \arrow[ur, "k"'] &
\end{tikzcd} \tag{4.6.6}$$

The function template `pcurry` transforms functions on pairs into functions that can take arguments one at a time by judicious selection of internal homs. So $\mathtt{pcurry}(k)(t)(u)$ is the same as $k(t, u)$, where the curried version takes (t) and returns a function from U to V that simply applies *both* `t` and `u` to k.

These can be implemented as follows:

```
template <typename T, typename U>
auto ev(P<Hom<T, U>, T> fn_and_arg) {
  auto [fn, x] = fn_and_arg;
  return fn(x);
}

template <typename Fn, typename TU = Dom<Fn>,
          typename T = std::tuple_element_t<0, TU>,
          typename U = std::tuple_element_t<1, TU>,
          typename V = Cod<Fn>>
auto pcurry(Fn fn) -> Hom<T, Hom<U, V>> {
  return [fn](T t) -> Hom<U, V> {
    return [fn, t](U u) -> V { return fn({t, u}); };
  };
}
```

which can be tested against the exponential diagram:

```
TEST_CASE("Commutativity of (4.6.6)") {
  auto ab = P<A, B>{};
  auto k = [](P<A, B>) -> C { return {}; };

  auto cw = compose(ev<B, C>, prod(pcurry(k), id<B>));

  REQUIRE(cw(ab) == k(ab));
}
```

The function template `puncurry` is implemented as:

```
template <typename Fn,             typename T = Dom<Fn>,
          typename UtoV = Cod<Fn>, typename U = Dom<UtoV>,
                                   typename V = Cod<UtoV>>
auto puncurry(Fn fn) -> Hom<P<T, U>, V> {
  return [fn](P<T, U> p) -> V { return fn(p.first)(p.second); };
}
```

which is inverse to `pcurry`:

```
TEST_CASE("pcurry and puncurry are inverse") {
  auto ab = P<A, B>{};
  auto k = [](P<A, B>) -> C { return {}; };

  REQUIRE(puncurry(pcurry(k))(ab) == C{});
}
```

4.6.1 Arbitrary Finite Products

The binary product of types generalises to finite arbitrary products. We can do this in code, either by nesting `std::pairs` (that is, `P`) or directly using `std::tuple`.

For example, we can make the 3-tuple of T, U, V as

$$\mathtt{P}\langle \mathtt{P}\langle T, U\rangle, V\rangle,$$

where the prjections are

- `compose(proj_l, proj_l)`, in the first factor,
- `compose(proj_l, proj_r)`, in the second factor,
- `proj_r`, in the third factor,

The organisation of nesting does not matter since we have shown associativity, relating `P`⟨`P`⟨T, U⟩, V⟩ bijectively to `P`⟨T, `P`⟨U, V⟩⟩ through the associator. And since that organisation is irrelevant, it is overall much simpler to use `std::tuple`⟨T, U, V⟩, with `std::get`⟨0–2⟩ as projections.

4.6.2 Equivalence of C++ Argument Lists & Tuples

A reasonable person may be concerned, at this point, that we are limiting ourselves to functions of a single variable, even if that variable can be of product type. Afterall, C++ argument lists are not implicitly equivalent to tuples or pairs. Happily, an isomorphism exists between unary C++ functions of tuples and C++ functions of several arguments:

```
template <typename Fn>
auto to_unary(Fn &&f) {
  return [f = std::forward<Fn>(f)](auto &&args) mutable {
    return std::apply(f, std::forward<decltype(args)>(args));
  };
}

template <typename Fn>
auto to_n_ary(Fn &&f) {
  return [f = std::forward<Fn>(f)](auto &&...args) mutable {
    return f(
        std::make_tuple(std::forward<decltype(args)>(args)...));
  };
}
```

We can confirm these are an inverse pair:

```
TEST_CASE("f(T_1, T_2, T_3, ..., T_n) ≅ f(std::tuple⟨T_1, T_2, T_3, ..., T_n⟩)") {
  auto f = [](A, B, C) -> D { return D{}; };

  REQUIRE( // `to_unary` does as expected
      to_unary(f)(std::tuple<A, B, C>{}) == f(A{}, B{}, C{}));

  REQUIRE( // `to_n_ary` is inverse to `to_unary`
      to_n_ary(to_unary(f))(A{}, B{}, C{}) == f(A{}, B{}, C{}));
}
```

4.7 COPRODUCTS AND BICARTESIAN CLOSED STRUCTURE

Dual to the cartesian product is the cartesian coproduct. A monoidal structure based on this coproduct is called a *cocartesian monoidal structure.* A CCC can be made *bicartesian* by endowment with a cocartesian monoidal structure wherein the products distribute over the coproducts. Furthermore, it will be demonstrated that the cocartesian monoidal structure has a symmetric braiding.

The previous sections developed a product structure on the type constructor `P`. The projection, the `fanout`, and the `prod` bimap allow us compose expressions on `P`⟨*T*, *U*⟩ as if its values simultaneously contained *T and U* values (for any *T* and *U* in ***Cpp***).

So I emphasise that the product structure encodes an ***and*** relationship on two objects because, dual to that, the coproduct structure will encode an ***or*** relationship on two objects.

Consider a type constructor `S` as the categorical coproduct in ***Cpp***. (the pneumonic here is `S`um.) A braided monoidal structure on `S` is the tuple:

(`S`, `Never`, `associator_co_fd`, `l_unitor_co`, `r_unitor_co`, `braid_co`)

where

- `S`, the bifunctorial coproduct type constructor.
- *Never* is an initial ***Cpp***-object, an empty type, acting as the monoidal unit
- `associator_co_fd`, the associator, a natural isomorphism with with components

  `associator_co_fd`⟨*T*, *U*, *V*⟩ : `S`⟨*T*, `S`⟨*U*, *V*⟩⟩ ⥲ `S`⟨`S`⟨*T*, *U*⟩, *V*⟩,

  for all $T, U, V \in$ ***Cpp***, with inverse `associator_co_rv`.
- `l_unitor_co_fd` and `r_unitor_co_fd`, natural isomorphisms called (respectively) the left- and right-unitors with components of the

forms

$$\texttt{l\_unitor\_co\_fd}\langle T\rangle : \texttt{S}\langle\texttt{Never}, T\rangle \xrightarrow{\sim} T$$

$$\texttt{r\_unitor\_co\_fd}\langle T\rangle : \texttt{S}\langle T, \texttt{Never}\rangle \xrightarrow{\sim} T.$$

for all $T \in$ ***Cpp***, with inverses postfixed `_rv` instead of `_fd`.

- `braid_co`, the braiding natural isomorphism with components

$$\texttt{braid\_co}\langle T, U\rangle : \texttt{S}\langle T, U\rangle \xrightarrow{\sim} \texttt{S}\langle U, T\rangle,$$

which are self inverse.

THE CATEGORICAL COPRODUCT The construction of coproducts in ***Cpp*** echos their construction in $\mathbf{Set}$[13]. The forgetful functor

$$U_{\textit{Cpp}} : \textit{Cpp} \to \mathbf{Set},$$

associates each ***Cpp*** type to its underlying set of values in $\mathbf{Set}$. The categorical coproduct in $\mathbf{Set}$ is disjoint union. Any T, $U \in$ ***Cpp*** are associated through $U_{\textit{Cpp}}$ to the sets T, U. We associate $T \sqcup U$ to the image of $\texttt{S}\langle T, U\rangle$ through $U_{\textit{Cpp}}$.

Recall that a (binary) categorical coproduct is a colimit of a discrete diagram with two objects. A cocone over this diagram is a cospan; in ***Cpp***:

$$\begin{tikzcd} T \arrow[r, "f"'] & C & U \arrow[l, "g"] \end{tikzcd} .$$

A colimit of the diagram is an initial cocone. The coapex of the cocone, in set theoretical terms, will act as a disjoint union with the sides of the cocone providing canonical *injections*.

Two obvious candidates for implementing `S` are plain C-style *unions*, [158, §12.3], or the STL class template `std::variant` [158, §23.7.3]. Unions do not suit the situation because, being very well named, they model a union of types: $T_0 \cup T_1$. There is no way to identify the original membership

[13] Categorical coproducts: §B.10, p. 292; such coproducts in in $\mathbf{Set}$: §B.10.1, p. 294

of a value. Having $t \in T_0 \cup T_1$, we have no way of knowing if $t \in T_0$ or $t \in T_1$, and worse, it could be in both if the types intersect! We need a *disjoint* union $T_0 \sqcup T_1 \cong \{ (i, t) \mid t \in T_i \}$, where some sort of tag or index is kept to distinguish the parentage of a value in the union and precluding ambiguity in the case of intersection of the terms. The class template `std::variant` is well suited, but not perfect.

First, some nomenclature. We call `std::variant`$\langle T_0, T_1 \rangle$ a `sum` of types T_0 and T_1. Let us call T_0 and T_1 the *alternatives* or more generally, the *terms* of the *sum*. A value, `t`, from that sum can originate from either T_0 or T_1, and `t.index()` is the zero-based index of the term in the sum from which `t` originates. Unfortunately for us, a `std::variant`-value can also be in limbus state, "`valueless_by_exception`", where it holds no value at all [158, §12.7.3.5]. This situation arises when an exception is thrown during a type-changing assignment or emplacement. Since exceptions are beyond the scope of ***Cpp*** anyway[14], we disregard this contingency.

There are issues that will force us to be a little less direct in our use of `std::variant` as a model for coproducts. I will mention those when the become relevant. For now, consider the structure

```
template <typename T, typename U>
struct S : std::variant<T, U> {
  using std::variant<T, U>::variant;

  S() = delete;
};
```

Shirking the usual advice against inheriting from STL class templates, inheriting from `std::variant` does three things for us.

1. It allows us to limit the number of alternatives in the variant to two, since `std::variant` will allow an arbitrary list of terms.
2. It allows us to create template specialisations which will be important when we introduce the monoidal unit, `Never`.
3. It allows us to preclude default-construction by deleting the default constructor. In the case of products, it is fairly obvious that a default

[14] See *Limitations of the Model* §4.9.

construction of the product will involve default construction of each of its factors. It probably is not as obvious what default construction of a sum should do.[15]

Since we do nothing with virtual destruction, we can ignore the usual admonishments against inheriting from STL class templates.

Consider `S` at the apex of a colimit cocone, in the following diagram:

$$\begin{tikzcd}
T \arrow[r, "\texttt{inject\_l}"] \arrow[dr, "f"'] & \texttt{S}\langle T, U\rangle \arrow[d, dashed, "\exists!\, u"'] & U , \arrow[l, "\texttt{inject\_r}"'] \arrow[dl, "g"] \\
& V &
\end{tikzcd} \tag{4.7.7}$$

which prescribes a unique u that facilitates the commutativity of the left and right triangles, each subtended by the projection functions,

```
template <typename T, typename U>
auto inject_l(T t) -> S<T, U> {
  return S<T, U>(std::in_place_index<0>, t);
}

template <typename T, typename U>
auto inject_r(U t) -> S<T, U> {
  return S<T, U>(std::in_place_index<1>, t);
}
```

Much of the `std::variant` interface requires that a type `std::variant`$\langle \ldots, T, \ldots\rangle$ will only participate in overload resolution if any T appears *exactly once* in the template argument list. This means the diagonal `S`$\langle T, T\rangle$ is affected. This limitation arises in parts of the `std::variant` Application Programming Interface (API) that facilitate interaction with value types by way of type-based dispatch. For example, `std::get`$\langle T\rangle$ cannot be expected to perform consistently on a variant with multiple T alternatives. Perhaps most unfortunate among these exclusions is visitation. We must constrain ourselves to index-based interactions in order to cover the diagonal. This is why `inject_l/r` from the previous listing place their arguments in `S` by index.

[15] As a matter interest, the default constructor of `std::variant` calls the default constructor of its first alternative, if that exists, and is otherwise deleted.

We can witness `S` at the coapex of a terminal cocone by defining the unique u, which is an operator parameterised by the functions f and g. The operation mapping f and g to u is often called *fanin*. If the reader will forgive the mix of mathematical and computerised notation, a fanin of f and g maps a value v as

$$(index,\ v) \mapsto \begin{cases} f(v) & \text{if} \quad index = 0 \\ g(v) & \text{if} \quad index = 1 \end{cases}$$

The ***Cpp***/C++spelling is a little more verbose:

```
template <typename Fn,           typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename V = Cod<Fn>>
auto fanin(Fn fn, Gn gn) -> Hom<S<T, U>, V> {

  static_assert(std::is_same_v<V, Cod<Gn>>);

  return [fn, gn](S<T, U> t_or_u) -> V {
    static_assert(std::is_invocable_v<Fn, T>);
    static_assert(std::is_invocable_v<Gn, U>);

    if (t_or_u.index() == 0)
      return fn(std::get<0>(t_or_u));
    else
      return gn(std::get<1>(t_or_u));
  };
}
```

We can test that this occasions the commutativity of (4.7.7):

```
TEST_CASE(
    "Commutativity of left and right triangles in (4.7.7)") {
  auto a_to_c = [](A) { return C{}; };
  auto b_to_c = [](B) { return C{}; };

  auto left_triangle_path =
      compose(fanin(a_to_c, b_to_c), inject_l<A, B>);
  REQUIRE(left_triangle_path(A{}) == a_to_c(A{}));

  auto right_triangle_path =
      compose(fanin(a_to_c, b_to_c), inject_r<A, B>);
  REQUIRE(right_triangle_path(B{}) == b_to_c(B{}));
}
```

where T, U, $V =$ A, B, C

S-BIFUNCTOR As a bifunctor, S must be functorial in both the left and right template parameters. In the category of sets, the categorical coproduct of functions, denoted $f \sqcup g$, is a map from $(\operatorname{dom} f) \sqcup (\operatorname{dom} g)$ to $(\operatorname{cod} f) \sqcup (\operatorname{cod} g)$. Thinking in terms of ***Cpp***/C++, and again using a mixed mathematical notation, we can express the mapping as:

$$(index, v) \mapsto \begin{cases} \texttt{inject\_l}\langle \operatorname{cod} f, \operatorname{cod} g\rangle \circ f(v) & \text{if} \quad index = 0 \\ \texttt{inject\_r}\langle \operatorname{cod} f, \operatorname{cod} g\rangle \circ g(v) & \text{if} \quad index = 1 \end{cases}$$

Given functions, $f \in \mathtt{Hom}\langle T, X\rangle$, and $g \in \mathtt{Hom}\langle U, Y\rangle$, their sum $\mathtt{coprod}(f, g)$ is of type $\mathtt{Hom}\langle \mathtt{S}\langle T, U\rangle, \mathtt{S}\langle X, Y\rangle\rangle$. When coprod is given a value in T, it applies f returning a X; and when it is given a value in U it applies g returning a Y. In C++ this can be implemented as:

```
template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename X = Cod<Fn>, typename Y = Cod<Gn>>
auto coprod(Fn fn, Gn gn) -> Hom<S<T, U>, S<X, Y>> {
  using TorU = S<T, U>;
  using XorY = S<X, Y>;

  return [fn, gn](TorU t_or_u) -> XorY {
    if (t_or_u.index() == 0)
      return inject_l<X, Y>(fn(std::get<0>(t_or_u)));
    else
      return inject_r<X, Y>(gn(std::get<1>(t_or_u)));
  };
}
```

We ensure that the functor laws hold in both positions in the following test case. Because sum-types are involved, even when using the singletons A, B, and C we will have multiple values in $\mathtt{S}\langle \mathtt{A}, \mathtt{B}\rangle$: one each through left and right injections:

```
TEST_CASE(
    "(S, coprod) is functorial in the left- and "
    "right-position.") {
  auto actual_ab = std::vector<S<A, B>>{
      inject_l<A, B>(A{}),
      inject_r<A, B>(B{})
    };

  for (auto &x : actual_ab) {
    REQUIRE(
      compose(coprod(g, id<B>), coprod(f, id<B>))(x)
          ==
              coprod(compose(g, f), id<B>)(x)
    );

    REQUIRE(
      compose(coprod(id<A>, h), coprod(id<A>, g))(x)
          ==
              coprod(id<A>, compose(h, g))(x)
    );

    REQUIRE(coprod(id<A>, id<B>)(x) == id<S<A, B>>(x));
  }
}
```

The associator If we read the type $\texttt{S}\langle T,\ \texttt{S}\langle B,\ C\rangle\rangle$ as "a T or (a B or a C)" the parenthetical delimitation does nothing to change the overall meaning of the phrase. So it is equivalent to "(a T or a B) or a C". This is the nature of the associator on the coproduct, which is a pair of functions mediating the natural isomorphism:

$$\texttt{associator\_co\_fd}\langle T,\ U,\ V\rangle :\ \texttt{S}\langle T,\ \texttt{S}\langle U,\ V\rangle\rangle \xrightarrow{\sim} \texttt{S}\langle \texttt{S}\langle T,\ U\rangle,\ V\rangle,$$

such that the associator pentagon commutes:

$$\begin{tikzcd}
& \texttt{S}\langle \texttt{S}\langle T,\ U\rangle,\ \texttt{S}\langle V,\ W\rangle\rangle \arrow[dr, "\texttt{associator\_co\_fd}\langle \texttt{S}\langle T, U\rangle, V, W\rangle"] & \\
\texttt{S}\langle T,\ \texttt{S}\langle U,\ \texttt{S}\langle V,\ W\rangle\rangle\rangle \arrow[ur, swap, "\texttt{associator\_co\_fd}\langle T, U, \texttt{S}\langle V, W\rangle\rangle"] \arrow[d, "\texttt{prod}(\texttt{id}\langle T\rangle, \texttt{associator\_co\_fd}\langle U, V, W\rangle)"] & & \texttt{S}\langle \texttt{S}\langle \texttt{S}\langle T,\ U\rangle,\ V\rangle,\ W\rangle \\
\texttt{S}\langle T,\ \texttt{S}\langle \texttt{S}\langle U,\ V\rangle,\ W\rangle\rangle \arrow[rr, swap, "\texttt{associator\_co\_fd}\langle T, \texttt{S}\langle U, V\rangle, W\rangle"] & & \texttt{S}\langle \texttt{S}\langle T,\ \texttt{S}\langle U,\ V\rangle\rangle,\ W\rangle\ . \arrow[u, swap, "\texttt{prod}(\texttt{associator\_co\_fd}\langle T, U, V\rangle, \texttt{id}\langle W\rangle)"]
\end{tikzcd}$$

(4.7.8)

The code for the associator is unsightly, and carries a lot of complexities due to an issue we have not yet discussed. The monoidal unit of S, called Never will be implemented during the discussion of the unitor. For now, be aware that it has to be explicitly handled in the associator because it has a deleted constructor. This means that the code for the associator needs to statically check for the case of a Never in each template argument to avoid instantiating an injection function for Never, which would fail for lacking a constructor. Here is the code for the forward and reverse associator_co function templates:

```
template <typename T, typename U, typename V>
auto associator_co_fd(S<T, S<U, V>> t_uv) -> S<S<T, U>, V> {
  if (t_uv.index() == 0) {
    if constexpr (!std::is_same_v<T, Never>)
      return inject_l<S<T, U>, V>(std::get<0>(t_uv));
  } else {
    auto &uv = std::get<1>(t_uv);
    if (uv.index() == 0) {
      if constexpr (!std::is_same_v<U, Never>)
        return inject_l<S<T, U>, V>(std::get<0>(uv));
    } else {
      if constexpr (!std::is_same_v<V, Never>)
        return inject_r<S<T, U>, V>(std::get<1>(uv));
    }
  }
  throw std::domain_error("Recieved a variant with no value.");
}

template <typename T, typename U, typename V>
auto associator_co_rv(S<S<T, U>, V> tu_v) -> S<T, S<U, V>> {
  if (tu_v.index() == 0) {
    auto &tu = std::get<0>(tu_v);
    if (tu.index() == 0) {
      if constexpr (!std::is_same_v<T, Never>)
        return inject_l<T, S<U, V>>(std::get<0>(tu));
    } else {
      if constexpr (!std::is_same_v<U, Never>)
        return inject_r<T, S<U, V>>(std::get<1>(tu));
    }
  } else {
    if constexpr (!std::is_same_v<V, Never>)
      return inject_r<T, S<U, V>>(std::get<1>(tu_v));
  }
  throw std::domain_error("Recieved a variant with no value.");
}
```

These function templates check for a value in the T, U or V positions of the argument $\mathtt{S}\langle\mathtt{S}\langle T, U\rangle, V\rangle$ and returns the same value but injected appropriately into $\mathtt{S}\langle T, \mathtt{S}\langle U, V\rangle\rangle$ (and the opposite in the reverse case). It would be much more readable if it were not for the static `Never` checks. In the next subsection THE UNITOR, we will see more on `Never` and why such measures are necessary.

First, we check that the forward and reverse cases are mutually inverse:

```
TEST_CASE(
    "coassociator_fd and coassociator_rv are mutually "
    "inverse.") {
  auto associator_co_fd_rv = compose(
        associator_co_rv<A, B, C>,
        associator_co_fd<A, B, C>
      );
  auto associator_co_rv_fd = compose(
        associator_co_fd<A, B, C>,
        associator_co_rv<A, B, C>
      );

  auto a_bc = inject_l<A, S<B, C>>(A{});
  REQUIRE(associator_co_fd_rv(a_bc) == id<S<A, S<B, C>>>(a_bc));
  auto ab_c = inject_r<S<A, B>, C>(C{});
  REQUIRE(associator_co_rv_fd(ab_c) == id<S<S<A, B>, C>>(ab_c));
}
```

And then confirm that `associator_co_fd` sustains the commutativity of (4.7.8):

```
TEST_CASE("Associator diagram for coproduct") {
  // All four values in S<A, S<B, S<C, D>>>:
  auto start_vals = std::vector<S<A, S<B, S<C, D>>>>{
    inject_l<A, S<B, S<C, D>>>(
             A{}
    ),
    inject_r<A, S<B, S<C, D>>>(
         inject_l<B, S<C, D>>(
                  B{}
      )
    ),
    inject_r<A, S<B, S<C, D>>>(
         inject_r<B, S<C, D>>(
              inject_l<C, D>(
                       C{}
        )
      )
    ),
    inject_r<A, S<B, S<C, D>>>(
         inject_r<B, S<C, D>>(
              inject_r<C, D>(
                          D{}
        )
      )
    )
  };

  auto cw_path = compose(
        associator_co_fd<S<A, B>, C, D>,
        associator_co_fd<A, B, S<C, D>>
      );

  auto ccw_path = compose(
        coprod(associator_co_fd<A, B, C>, id<D>),
        associator_co_fd<A, S<B, C>, D>,
        coprod(id<A>, associator_co_fd<B, C, D>)
      );

  for (auto &each : start_vals)
    REQUIRE(ccw_path(each) == cw_path(each));
};
```

Here, we test the loop for all values in $\mathtt{S}\langle A, \mathtt{S}\langle B, \mathtt{S}\langle C, D\rangle\rangle\rangle$, of which there are 4: the sum of 4 singletons.

THE UNITOR In analogy to the school algebreic equation $0+x = x+0 = x$, the unitor structure mediates the isomorphism

$$\mathtt{S}\langle \mathtt{Never}, T\rangle \cong \mathtt{S}\langle T, \mathtt{Never}\rangle \cong T,$$

or namely,

$$\begin{aligned}\mathtt{l\_unitor\_co\_fd}\langle T\rangle &: \mathtt{S}\langle \mathtt{Never},\, T\rangle \xrightarrow{\sim} T\\ \mathtt{r\_unitor\_co\_fd}\langle T\rangle &: \mathtt{S}\langle T,\, \mathtt{Never}\rangle \xrightarrow{\sim} T.\end{aligned}$$

The relationship implies that `Never` is an empty type—a type that has no instances. When handed a value of type `S`⟨*T*, `Never`⟩ we know that the value is of the term *T*. In $**et** the analogue of `Never` is the empty set.

In category theory, there are no tools to inspect an isolated type and call it empty and we will have to discern that on the basis of the arrows. In the broader context of the category, we are looking for an initial object[16]: an object to which there are no arrows, and from which there is a unique arrow to every other object. The unique arrows are formalities without substance. A function from the empty set, to any other, is only technically a function because there are is no content in the domain to which the codomain can be related.

`Never` is implemented as a `struct` with deleted constructors:

```cpp
struct Never { // Monoidal unit for S
  Never() = delete;
  Never(const Never &) = delete;
  virtual ~Never();

  bool operator==(const Never &) const {
    throw std::domain_error(
        "`Never` instances should not exist, "
        "and someone must have done something perverse.");
  }
};
```

The default and copy constructors are deleted, and the destructor is made virtual so that it is impossible to construct a never by conventional means. The equality operator facilitates unit testing and throws a `domain_error` because you should never have a `Never`, much less two for comparison—it should never be possible to invoke it. Even though exceptions are not part

[16] DEF. B.3-1, p. 271

of the ***Cpp*** model; if you have by some extraordinary means conjured an instance of `Never` then you have already absconded.

The C++ langauge is very pragmatic and not designed for convient expression of what cannot be constructed. Having named such a thing, we ought to prepare ourselves for some measure of inconvenience when dealing with it. In defining the associator, we had to circumlocute instantiation of `inject_l`⟨`Never`, …⟩ or `inject_r`⟨…, `Never`⟩, even though it would never be called. Before progressing further, we have to deal with the fact that specifing `Nothing` as an alternative in `std::variant` implicitly deletes the constructors of the that type, effectively crippling it. Such a variant should be *simpler* owing to the fact that it will never hold a `Never`. To make that explicit, we partially specialise the `S` structure, adding back the constructors for the non-`Never` cases:

```
template <typename T>
struct S<T, Never> : std::variant<T, Never> {
  using std::variant<T, Never>::variant;

  S() : std::variant<T, Never>{inject_l<T, Never>(T{})} {}

  S(const S &other)
      : std::variant<T, Never>(
            std::in_place_type<T>, std::get<T>(other)) {}
};

template <typename T>
struct S<Never, T> : std::variant<Never, T> {
  using std::variant<Never, T>::variant;

  S() : std::variant<Never, T>{inject_r<Never, T>(T{})} {}

  S(const S &other)
      : std::variant<Never, T>(
            std::in_place_type<T>, std::get<T>(other)) {}
};
```

So now we are doubly safeguarded from observing `Never`-values. There is no reasonable way to construct a `Never`, and even if you did, you could not use it to construct an instance of `S`.

Now we are in a position to define the unitor satisfying the triangle:

$$\begin{tikzcd}
\mathtt{S}\langle T,\ \mathtt{S}\langle \mathtt{Never},\ U\rangle\rangle \arrow[rr, "\mathtt{associator\_co\_fd}\langle T, \mathtt{Never}, U\rangle"] \arrow[dr, bend left, "\mathtt{coprod}(\mathtt{id}\langle T\rangle, \mathtt{l\_unitor\_co\_fd}\langle U\rangle)"'] & & \mathtt{S}\langle \mathtt{S}\langle T,\ \mathtt{Never}\rangle,\ U\rangle\ . \arrow[dl, bend right, "\mathtt{coprod}(\mathtt{r\_unitor\_co\_fd}\langle T\rangle, \mathtt{id}\langle U\rangle)"] \\
& \mathtt{S}\langle T,\ U\rangle &
\end{tikzcd}$$

(4.7.9)

Since the unitor is bijective and has left/right parity, the unitor function templates are fourfold:

```
template <typename T>
auto l_unitor_co_fw(S<Never, T> just_t) -> T {
  return std::get<1>(just_t);
}

template <typename T>
auto l_unitor_co_rv(T t) -> S<Never, T> {
  return inject_r<Never, T>(t);
}

template <typename T>
auto r_unitor_co_fw(S<T, Never> just_t) -> T {
  return std::get<0>(just_t);
}

template <typename T>
auto r_unitor_co_rv(T t) -> S<T, Never> {
  return inject_l<T, Never>(t);
}
```

We check that the `_fw` and `_rv` transformations are mutually inverse:

```
TEST_CASE("_fw and _rv are mutual inverses for l/r-unitor") {
  auto ra = inject_r<Never, A>(A{});
  auto la = inject_l<A, Never>(A{});

  REQUIRE(
      compose(l_unitor_co_rv<A>, l_unitor_co_fw<A>)(ra) ==
          id<S<Never, A>>(ra)
  );
  REQUIRE(
      compose(l_unitor_co_fw<A>, l_unitor_co_rv<A>)(A{}) ==
          id<A>(A{})
  );

  REQUIRE(
      compose(r_unitor_co_rv<A>, r_unitor_co_fw<A>)(la) ==
          id<S<A, Never>>(la)
  );
  REQUIRE(compose(r_unitor_co_fw<A>, r_unitor_co_rv<A>)(A{}) ==
          id<A>(A{})
  );
}
```

And finally, check that the unitor satisfies (4.7.9):

```
TEST_CASE("Unitor diagram for coproduct") {
  auto a_or_rb = inject_l<A, S<Never, B>>(A{});

  auto cw_path = compose(
        coprod(r_unitor_co_fw<A>, id<B>),
        associator_co_fd<A, Never, B>
      );
  auto ccw_path = coprod(id<A>, l_unitor_co_fw<B>);

  associator_co_fd<A, Never, B>(a_or_rb);

  REQUIRE(cw_path(a_or_rb) == ccw_path(a_or_rb));
}
```

THE BRAIDING The statement that a thing is from sets "*A or B*" is logically equivalent to its transposition, that it is from "*B or A*". So it should be the case that

$$\mathtt{S}\langle T,\ U\rangle \cong \mathtt{S}\langle U,\ T\rangle.$$

Moreover, a value $(i, v) \in \mathtt{S}\langle T,\ U\rangle$ can be relocated to $\mathtt{S}\langle U,\ T\rangle$ by simply changing the index, i. This is the job of the braiding, and can be implemented as a function template:

```
template <typename T, typename U>
auto braid_co(S<T, U> t_or_u) -> S<U, T> {
  if (t_or_u.index() == 0)
    return inject_r<U, T>(std::get<0>(t_or_u));
  else
```

```
    return inject_l<U, T>(std::get<1>(t_or_u));
}
```

This is a little more complicated than changing the index. Part of that is the fault of C++, but it is more so a matter of internal consistency which demands that we use the injection functions.

This braiding is self-inverse:

```
TEST_CASE("Braiding of coproduct is self-inverse") {
  auto ab = std::vector<S<A, B>>{
    inject_l<A, B>(A{}),
    inject_r<A, B>(B{})
  };

  for (auto each: ab)
    REQUIRE(braid_co(braid_co(each)) == id<S<A, B>>(each));
}
```

In the presence of this symmetry, the two hexagonal coherence diagrams for the braiding[17] are made so that one implies the other. (See [81, §XI.1, p. 253].) Choosing the same as for the product braiding, we test `braid_co` against:

$$\begin{tikzcd}
 & \mathtt{S}\langle \mathtt{S}\langle T, U\rangle, V\rangle \arrow[r, "\mathtt{braid\_co}\langle \mathtt{S}\langle T, U\rangle, V\rangle"] \arrow[dl, bend right, "\mathtt{associator\_co\_rv}\langle T, U, V\rangle"] & \mathtt{S}\langle V, \mathtt{S}\langle T, U\rangle\rangle \arrow[dr, bend left, "\mathtt{associator\_co\_fd}\langle V, T, U\rangle"'] & \\
\mathtt{S}\langle T, \mathtt{S}\langle U, V\rangle\rangle \arrow[dr, bend right, "\mathtt{prod}(\mathtt{id}\langle T\rangle, \mathtt{braid\_co}\langle U, V\rangle)"] & & & \mathtt{S}\langle \mathtt{S}\langle V, T\rangle, U\rangle \arrow[dl, bend left, "\mathtt{prod}(\mathtt{braid\_co}\langle V, T\rangle, \mathtt{id}\langle U\rangle)"'] \\
 & \mathtt{S}\langle T, \mathtt{S}\langle V, U\rangle\rangle \arrow[r, "\mathtt{associator\_co\_fd}\langle T, V, U\rangle"'] & \mathtt{S}\langle \mathtt{S}\langle T, V\rangle, U\rangle\ , &
\end{tikzcd}$$

The unit test case for this diagram is:

```
TEST_CASE("Braiding diagram 1 for coproduct") {
  auto start_vals = std::vector<S<S<A, B>, C>>{
    inject_l<S<A, B>, C>(
      inject_l<A, B>(
```

[17] see DEF. B.11-7, p. 299.

```
               A{}
      )
    ),
    inject_l<S<A, B>, C>(
      inject_r<A, B>(
                  B{}
      )
    ),
    inject_r<S<A, B>, C>(
                      C{}
    ),
  };

  auto cw_path = compose(
        coprod(braid_co<C, A>, id<B>),
        associator_co_fd<C, A, B>,
        braid_co<S<A, B>, C>
      );
  auto ccw_path = compose(
        associator_co_fd<A, C, B>,
        coprod(id<A>, braid_co<B, C>),
        associator_co_rv<A, B, C>
      );

  for (auto each : start_vals)
    REQUIRE(cw_path(each) == ccw_path(each));
}
```

P DISTRIBUTS OVER S In a CCC with binary coproducts, the products necessarily distribute over the coproducts [109, §6.8-Q13]. This means that there is a natural isomorphism

$$\mathsf{S}\langle \mathsf{P}\langle T, Z\rangle, \mathsf{P}\langle U, Z\rangle\rangle \cong \mathsf{P}\langle \mathsf{S}\langle T, U\rangle, Z\rangle \tag{4.7.10}$$

The appendix on CT does not cover some of the topics required to explain why this is necessarily true.[18] The following construction will however make it apparent that it is true in ***Cpp*** because the distributor can be constructed by composition of the universal arrows of the product, coproduct

[18] For a more abstract explanation see [84, §5.5].

and exponentials. Namely, the forward direction of (4.7.10), `factorise`, will be written in terms of the universal arrows

- `fanin`, (the universal cocone/coproduct morphism),
- `inject_l/r` (the coproduct injections), and
- `prod` (the bimap of `P`),

which require the basic universal arrows of products and coproducts, but do not require the cartesian closure. The inverse direction, `expand`, does require cartesian closure and is expressed in terms of

- `compose` and `id` (the basics of the category),
- `fanin`,
- `inject_l/r` and
- `pcurry` and `puncurry` (the natural isomorphism of cartesian closure).

The existence of `factorise` and `expand` are guaranteed by the axioms of a biccc, as demonstrated in these definitions which are composed only from the elements intrinsic to that structure.

The factorisation transformation is the simplest of the two. To emphasise that it is constructed from universal arrows, I compose the arrow `universal_factorise`, the result of which is returned by the function:

```
template <typename T, typename U, typename X>
auto factorise(S<P<T, X>, P<U, X>> tx_ux) -> P<S<T, U>, X> {
  const auto universal_factorise = fanin(
        prod(inject_l<T, U>, id<X>),
        prod(inject_r<T, U>, id<X>)
      );

  return universal_factorise(tx_ux);
}
```

The `expand` transformation is much more intricate. I have placed type annotations (using a terser mathematical notation) to aid the eye:

```
template <typename T, typename U, typename Z>
auto expand(P<S<T, U>, Z> t_or_u_and_x) -> S<P<T, Z>, P<U, Z>> {
  // tz : T × Z → (T × Z) + (U × Z)
  const auto tz =
      pcurry(compose(
              inject_l<P<T, Z>, P<U, Z>>,
              id<P<T, Z>>
            ));
  // uz : U × Z → (T × Z) + (U × Z)
  const auto uz =
      pcurry(compose(
              inject_r<P<T, Z>, P<U, Z>>,
              id<P<U, Z>>
            ));

  // tz_uz : (T + U) → Z ⊸ (T × Z) + (U × Z)
  const auto tz_uz = fanin(tz, uz);
  // universal_expand : (T + U) × Z  ⊸  (T × Z) + (U × Z)
  const auto universal_expand = puncurry(tz_uz);

  return universal_expand(t_or_u_and_x);
}
```

Roughly speaking, in the code above

- we create a composite, `tz`, taking a pair $\mathtt{P}\langle T, Z\rangle$ and returning that value injected into the sum $\mathtt{S}\langle \mathtt{P}\langle T, Z\rangle, \mathtt{P}\langle U, Z\rangle\rangle$,
- a second morphism is created, `uz`, identical except that for U instead of T,
- `tz` and `uz` are curried and then fanned-into an arrow `tz_uz`.
- `tz_uz` takes a T *or* a U and returns a function that further takes a Z and returns the appropriate pair injected into the sum: $\mathtt{S}\langle \mathtt{P}\langle T, Z\rangle, \mathtt{P}\langle U, Z\rangle\rangle$, and
- the `expand` function is not curried, so we uncurry `tz_uz` to create `universal_expand`.

We test the isomorphism by showing that `expand` and `factorise` are mutually inverse:

```
TEST_CASE("expand and factorise are mutually inverse") {
  SECTION("in the forward then reverse direction") {
    auto values = std::vector<S<P<A, C>, P<B, C>>>{
        inject_l<P<A, C>, P<B, C>>({A{}, C{}}),
        inject_r<P<A, C>, P<B, C>>({B{}, C{}})};

    auto fw_rv = compose(expand<A, B, C>, factorise<A, B, C>);

    for (auto each : values) {
```

```
   REQUIRE(fw_rv(each) == id<S<P<A, C>, P<B, C>>>(each));
  }
 }

 SECTION("and in the opposite direction") {
  auto values = std::vector<P<S<A, B>, C>>{
    {inject_l<A, B>(A{}), C{}},
    {inject_r<A, B>(B{}), C{}}
   };

  auto rv_fw = compose(factorise<A, B, C>, expand<A, B, C>);

  for (auto each : values)
   REQUIRE(rv_fw(each) == id<P<S<A, B>, C>>(each));
 }
}
```

We have a ccc with a cocartesian monoidal structure and the product distributing over the coproduct. In short, ***Cpp*** is a biccc.

4.8 FIXPOINTS & SNOC-LISTS

A mathematical description of snoc lists was given in §3.8. The biccc structure developed for ***Cpp*** in the previous sections is sufficient for us to reason using a calculus of fixpoints (initial algebras and terminal coalgebras) [97].

Next, we will demonstrate this specifically for the case of *snoc*-lists, defining them at the level of the c++17 type system by approximating the fixpoint operator μ.

We will then demonstrate catamorphisms on these snoc lists.

4.8.1 μ to `Mu`

To begin modelling μP^*_T we need a c++ simulacrum of "μ" which, for some type functor, F, gives us an isomorphism between $F\langle\mu F\rangle$ and μF. Call it `Mu` and witness the isomorphism with `in` and `out`. The following definitions for those three are straightforward but I happily attribute them to Neibler [123]:

```
template <template <typename> class F>
struct Mu : F<Mu<F>> {
  explicit Mu(F<Mu<F>> f) : F<Mu<F>>(f) {}
};

template <template <typename> class F>
auto in(F<Mu<F>> f) -> Mu<F> {
  return Mu<F>{f};
}

template <template <typename> class F>
auto out(Mu<F> f) -> F<Mu<F>> {
  return f;
}
```

Here, the κ_F isomorphism is named `in` and `out` as an inverse pair.

4.8-1 Because an `F<Mu<F>>` "is a" `Mu<F>` (by class-inheritance) the transformations

$$\texttt{in} : \texttt{F<Mu<F>>} \to \texttt{Mu<F>}$$

and its inverse

$$\texttt{out} : \texttt{Mu<F>} \to \texttt{F<Mu<F>>}$$

are practically identity functions.

4.8.2 $\hat{T}_\bullet$ to `SnocList`$\langle T \rangle$

The next step to using `Mu` for building snoc-lists is to define a C++ analogue to $P_T^* U$:

```
template <typename T>
struct MP {
  template <typename U>
  using Left = S<I, P<std::shared_ptr<U>, T>>;

  using Right = T; // not really usable.
};
```

The pneumonic here is "`Maybe Product`", since this models a value that *maybe* a pair of values or *maybe* the cartesian unit `I`. But why a `std::shared_ptr` to U instead of a bare U? Because `Mu` inherits from `F<Mu<F>>`.

At the point of evaluating the body of a class or struct, it is an incomplete type[19] and the compiler cannot complete the it if it is defined in terms of itself. A pointer to a thing is not the same as the thing itself so `std::shared_ptr` provides a level of indirection we need to complete the type. This is the price we pay to the C++ type system for jumping up the level of abstraction.

But now we can define `Mu<OP<T>::template Left>`, which we will call `SnocList`:

```
template <typename T>
using SnocList = Mu<MP<T>::template Left>;
```

4.8.3 Building `SnocLists`

Now we can properly define `snoc`:

```
template <typename T>
auto snoc(SnocList<T> lst, T t) -> SnocList<T> {
  return in<MP<T>::template Left>(
      P{std::make_shared<SnocList<T>>(lst), t});
}
```

Notice that this is a thin wrapper around `in` (with lots of template argument noise) and is essentially an adaptor in the style of §4.6.2 for converting `snoc`'s function argument list into a `std::pair` to be passed to `in`. But to call the function `snoc`, we must be true to its widely understood interface, shown in `Lisp` at the beginning of the section. This means it must be a two-argument function and not one taking a `P` (AKA `std::pair`).

The Lisp code from earlier started with a `nil`, representing an empty list, and used `snoc` to build from that. We will need a `nil` too. Unfortunately, it will need to be an alias template so that the compiler can now the element type of the list we are building:

```
template <typename T>
auto nil = in<MP<T>::template Left>(I{});
```

[19] See [158, §6.9]

We can now build lists as we may expect to

```
TEST_CASE("Building `List`s") {

  auto list_ints = snoc(snoc(snoc(nil<int>, 1), 2), 3);
  auto another_list_ints =
      snoc(snoc(snoc(snoc(nil<int>, 1), 2), 3), 4);

  REQUIRE(std::is_same_v<SnocList<int>, decltype(list_ints)>);
  REQUIRE(
    std::is_same_v<snoclist_element_type<decltype(list_ints)>, int>);

  REQUIRE(list_ints == list_ints);
  // NB: require_FALSE:
  REQUIRE_FALSE(list_ints == another_list_ints);
}
```

The code for the helper trait `list_element_type` and the overload for `operator==` on `SnocLists` is provided in LST. C.2.8.

4.8.4 `SnocList` isomorphic to `std::vector`

This subsection's title name is broad, and requires qualification. `SnocLists` and its interface is not isomorphic to `std::vector` and its interface—they are very different beasts. But the underlying data structures are equivalent, because they are both linear datatypes. More specifically, there is a pair of functions that will translate back and forth between `SnocLists` and `std::vectors`. Namely these are `to_vector` and `to_snoclist`. They are very straightforward to implement but are distractingly verbose, so I have listen then in the appendix listing LST. C.2.8. Here is a (non-exhaustive) unit test case demonstrating that they are mutually inverse, and that they behave as one might expect:

```
TEST_CASE(
    "Arbitrary nested optional-pairs isomorphic to "
    "lists") {

  auto list_as = snoc(snoc(snoc(nil<A>, A{}), A{}), A{});
  auto vec_as = std::vector{A{}, A{}, A{}};
  auto list_ints = snoc(snoc(snoc(nil<int>, 1), 2), 3);
  auto vec_ints = std::vector{1, 2, 3};

  REQUIRE(to_vector(list_as) == vec_as);
  REQUIRE(to_vector(list_ints) == vec_ints);

  REQUIRE(list_as == to_snoclist(vec_as));
  REQUIRE(list_ints == to_snoclist(vec_ints));

  REQUIRE(to_vector(to_snoclist(vec_ints)) == vec_ints);
  REQUIRE(to_snoclist(to_vector(list_ints)) == list_ints);
}
```

4.8-2 The point of this demonstration was to show that we can reason about algebraic data types using the methods and tools of category theory, but that ***Cpp*** has internal representations of these structures that can be shown to be equivalent. This is important to the thesis because we will later reason using catamorphisms on linear data structures with more exotic semantics, and it is important to show that these are amenable to the same category theoretical analysis as simple `SnocList`s. Conversely, it is much easier to reason about and to work with `std::vector` than `SnocList` (because C++ was presumably not designed for representing abstractions built on algebraic data types) so it is nice to know we can reason about the simple type and have that reasoning extend upward to more abstract type algebra without having to deal with the challenging C++ syntax involved.

4.8.5 `SnocList`-catamorphisms

We now demonstrate that `SnocList`s have catamorphisms that can be defined abstractly in terms of `in` and `out`, as they are in CT.

Let F be an type functor. An F-algebra is a pair $(X, alg : F\langle X\rangle \to X)$. $\mathtt{Mu}\langle F\rangle$ is an "infinity-mirror" like structure closed by `in` and `out`. Owing to Lambek's famous lemma, the fact that `in`/`out` constitute an isomorphism, we know that $(\mathtt{Mu}\langle F\rangle, \mathtt{in})$ is initial in the induced category of F-algebras. That means that there is a unique F-algebra homomorphism underwriting the commutativity of this diagram:

$$\begin{tikzcd}
F\langle \mathtt{Mu}\langle F\rangle\rangle \arrow[r, "F::\mathtt{fmap}(\llparenthesis alg \rrparenthesis)"] \arrow[d, shift right, "\mathtt{in}"] & F\langle X\rangle \arrow[d, "alg"] \\
\mathtt{Mu}\langle F\rangle \arrow[u, shift right, "\mathtt{out}"] \arrow[r, dashed, "\exists!", "\llparenthesis alg \rrparenthesis"'] & X
\end{tikzcd}$$

Chasing the diagram around the outside we get an equation for the catamorphism:

$$\llparenthesis alg \rrparenthesis = alg \circ F::\mathtt{fmap}(\llparenthesis alg \rrparenthesis) \circ \mathtt{out}. \tag{4.8.11}$$

We want to apply this to `SnocList`, meaning that we need an `fmap` for `OP<T>::Fst<U>`. We will use the namespace method of organising the relationships established in the section on type functors and the beginning of the chapter (§4.3). In keeping with the Haskell language convention, let us call it `SnocF`:

```
template <typename T>
struct SnocF {

  template <typename U>
  using Of = typename MP<T>::template Left<U>;

  template <typename Fn>
  static auto fmap(Fn fn) -> Hom<Of<Dom<Fn>>, Of<Cod<Fn>>> {
    return [fn](Of<Dom<Fn>> i_or_p) -> Of<Cod<Fn>> {
      using Elem = maybe_pair_element_t<Of<Dom<Fn>>>;

      return coprod(id<I>, prod(ptr::fmap(fn), id<Elem>))(i_or_p);
    };
  }
```

We are not quite done with the struct. For convenience, we will also use the ambient namespace to simplify the definitions of

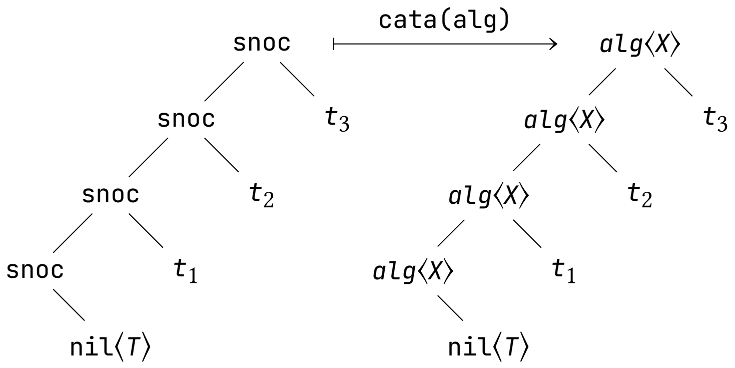
- `Alg`—the type of `SnocF` algebras. Recall that P_A^*-coalgebras have the form $\vec{x}_0 \triangledown f$ where $f : X \times A \to B$, where X is the *carrier* of the algebra.
- `cata`—the catamorphism itself.

```
  template <typename Carrier>
  using Alg = Hom<typename MP<T>::template Left<Carrier>, Carrier>;

  template <typename Carrier>
  static auto cata(Alg<Carrier> alg) -> Hom<SnocList<T>, Carrier> {
    return [alg](SnocList<T> ts) {
      return alg(fmap(cata<Carrier>(alg))(out(ts)));
      //     alg    ∘    F::fmap((|alg|))    ∘    out
    };
  }
};
```

The key things to look at are the signatures involved, and the fact that `cata`'s inner lambda is exactly the same composition of (4.8.11): "`alg` after fmaped `cata(alg)` after `out`."

Recall that list-catamorphisms can be thought of as a recursive replacement of operations. If the list is built up from expressions in `snoc` and `cata(alg)` converts snoc-lists into expressions in `alg` as depicted in the tree diagram:



(4.8.12)

A common first example of list-catamorphisms is sum totaling a list of integers. This algebra is carried by the `int` type so in the general form

$$\vec{0} \triangledown \texttt{sum\_pair},$$

`sum_pair` will add the left- and right-factors of the pair:

```
auto sum_alg = [](auto op) -> int {
  auto global_0 = [](I) -> int { return 0; };
  auto sum_pair = [](P<std::shared_ptr<int>, int> p) -> int {
    return *proj_l(p) + proj_r(p);
  };

  return fanin(global_0, sum_pair)(op);
  //     \______________________/
  //           0⃗ ▽ sum_pair
};
```

In the tree diagram, (4.8.12), the `nil` (which is just `I` disguised with a list-element type) will be replaced by the integer value `0`, and then the sum can proceed forward. The running sum is carried along the left factor of the pairs, and each iteration of the algebra adds its leaf into the running sum. We can check that it works as expected:

```
TEST_CASE("sum algebra on integer lists is as expected") {

  auto list_ints = snoc(snoc(snoc(snoc(nil<int>, 1), 2), 3), 4);
  auto sum_int_list = SnocF<int>::cata<int>(sum_alg);

  REQUIRE(sum_int_list(list_ints) == 1 + 2 + 3 + 4);
}
```

Another common first example of list-catamorphisms is based on an algebra that simply counts the elements. It is carried by `int`:

```
auto len_alg = [](auto op) -> int {
  // We don not care about the list-element type, so we deduce it:
  using ElemT = maybe_pair_element_t<decltype(op)>;

  auto global_0 = [](I) -> int { return 0; };
  auto add_one = [](P<std::shared_ptr<int>, ElemT> p) -> int {
    return *proj_l(p) + 1;
  };

  return fanin(global_0, add_one)(op);
};
```

You can see this is very similar to `sum_alg`, except that the right-factors of the pairs are not used. The left factor carries a running tally which is incremented at each branch of (4.8.12). The only added complexity is that we abstract over the element-type. This is so that the algebra does not hard-code the element type of the lists. We can see from the following unit test that the same algebra works for both `int`-lists and `A`-lists:

```
TEST_CASE("len_alg-catamorphism is as …expected") {

  SECTION("… on an integer list") {
    auto list_ints = snoc(snoc(snoc(snoc(nil<int>, 1), 2), 3), 4);
    auto len_int_list = SnocF<int>::cata<int>(len_alg);

    REQUIRE(len_int_list(list_ints) == 4);
  }

  SECTION("… on an A-list") {
    auto list_as = snoc(snoc(nil<A>, A{}), A{});
    auto len_a_list = SnocF<A>::cata<int>(len_alg);

    REQUIRE(len_a_list(list_as) == 2);
  }
}
```

*   *   *

One may notice that these snoc-lists are not an efficient way of approaching linear collections in C++. We even demonstrated that these snoc-lists as linear data representations are equivalent to `std::vector`. Indeed, the point of the exercise was not to develop production-ready code—*it was to demonstrate a model*, a proof of concept. The next chapter will explicate a model of programs for implementing control systems in C++, and this model is depicted in the language of category theory. It will use algebra and coalgebra, catamorphisms and anamorphisms, algebraic datatypes, and much more of the structure established in this chapter as features of the category C++. When developing a model of control programs we now know that

- the category ***Cpp*** has sufficient structure to support the model, and
- there are representations of that model in C++17 code.

The practice of good software engineering will preclude us from writing control systems software using code the like of which is presented in this chapter. But while developing that software, we will have a compass of mathematical intuition to guide the development and a backbone of mathematical structure to justify it.

4.9 LIMITATIONS OF THE MODEL

Machine independent thinking, as an abstraction, can be very powerful. But you also need to recognise it has a limit.
— Kevlin Henney
Six Impossible Things

In developing ***Cpp***/C++ we started by simply declaring a category where the objects are sets holding the values of any given C++ type, and the arrows are functional routines—functions pure and without side-effect. These choices were made so that we could view ***Cpp*** as something that could be embedded in the category of sets by a forgetful functor. Such a functor implies some resemblance between the two categories and by that, we hope ***Cpp*** can share some of $**et**'s nicer structure. But from the outset, we assumed at our modeling effort would preclude most ordinary C++ code.

Upon that we made our next assumption: that we could cull a subset of C++17 and define a collection of type constructors and function templates that model bicartesian closure—simultaneously giving ***Cpp*** an internal language and a "representation" of that language in C++ code. This "fast an loose reasoning" might be justified as morally correct [97], but moral correctness requires awareness of limitations.

By the creation of a BICCC of C++ programs one might be given the impression that I am implying a strong and direct relationship between the internal language of ***Cpp*** and the operational semantics of C++. I must hasten to stress that this is not the case. By analogy, the point of IEEE floating points numbers is to give numerical expressions a superficial resemblance to computations in the field of real numbers. In other words, the point is to enable the engineer to think like and idealist but compute like a pragmatist. Likewise, this is the point of working in ***Cpp***. The analogy runs deeper; surprise and disappointment await the programmer who confuses the platonic ideal with the pragmatic realization.

C++ will fail to underwrite the immaculacy of ***Cpp*** in more ways than I could possibly predict. The language standard, [158], is a massive (over

1600 pages) agglomeration of abstruse standardese. It is of both questionable usefulness and questionable feasibility to exhaustively analyse all of the ways in which the map of ***Cpp*** will fail to lay plainly over the terrain of C++. But this section provides a collection of places where it is grossly apparent that extra care must be taken because the code departs from the category.

REMARK 4.9-1. In applied mathematics, we gain a great deal of insight about the underlying physics when we study how a model departs from it. So these places where the code departs from the category, far from being a failure, are perhaps one of the most valuable aspects of the model.

There's [a gap] between category theory, [...] which is a pure subject, and programming, which is an applied subject; and pure category theory is almost never *useful in* anything... *unless you, at the very end, let yourself to break it.*
— David I. Spivak

GENERAL RECURSION In light of §2.3.1, we only allow catamorphisms/anamorphisms for recursion/corecursion in ***Cpp***/C++. In sum, Domain theory might be seen as entire field designed to deal with the misadventures one might have with general recursion. Though bicccs are logically at odds with the structure required for general recursion in many conceptions of domains [54, §3], we can take the lessons of [97] and [47] and constrain ourselves to catamorphisms (and with additional care, anamorphisms [66]). Even if ***Cpp*** allowed general recursion, function recursion in C++17 risks a stack overflow. Many compilers implement TCO (which flattens a type of recursive call into an equivalent loop) but this is not guaranteed in the standard. This means that each level of a recursive sub-call potentially consumes some portion of finite stack memory. So consider that `cata` in §4.8.5 may overflow the stack on sufficiently large lists.

ERROR BY EXCEPTIONS Exceptions are C++'s method of choice for handling errors and it is inseparably integrated into the STL. I ignore exceptions in ***Cpp*** and compilers make it easy to disable them. But in ***Cpp***/C++ I used the STL quite heavily. So by ignoring exceptions yet using libraries that rely on them for error handling, I am providing dangerous code. There are well known methods of error handling that are consonant with programming categories. Even in C++, specifically ISO/IEC 14882:2023 (C++23), the simple product type `std::expected` is outfitted with explicitly monadic operations to facilitate functional composition with non-exception error handling. So ideally, I would not have based `P` and `S` on `std::pair` and `std::variant` (respectively), instead creating my own types. But this chapter was purposed for exposition, to demonstrate CT structures in code that is simple and familiar enough to appear in the pages of a thesis.

TYPES THAT DO NOT BEHAVE AS TYPES C++ has a number of built-in types such as pointers, references and arrays that do not behave in the same way as "normal" types. Where pointers were required (in §4.8), I chose to use smart-pointers instead of raw pointers inherited from C. Even then, in EG. 4.3-4 we saw that trying to endow the `std::shared_ptr` type constructor with the structure of a functor leads to a perversion of the usual functor laws.

References are tools for performance to avoid needless copying of function arguments. They are not modelled in ***Cpp***. Outside the context of concurrency and lifetime issues, it is reasonable to have const-references in function arguments without endangering the categorical essence. An important exception is the common C++ (originally C) practice of using "output parameters". That is, function arguments which are mutated in the body of the function. This amounts to side-effects which are verboten by the axioms of ***Cpp***.

CONTROL FLOW In ***Cpp***, programs are expressions so operational control structures such as branches or loops must be handled differently and are not directly captured in ***Cpp***. Instead, branching is achieved through the use of sum types and loops are modelled by structural recursion. Notice, for example, that the implementations of `fanin`, `coprod` and other members of the cocartesian structure use `if`-statements. `Vector::fmap` internally used `std::transform` (which is written with a loop in all STL implementations I have checked). Those lower level details were obscured by the categorical abstraction surfaces. We can make efficient use of control flow structures if we wrap them tightly in functional interfaces. In the case of recursion, it is practically a requirement that we do so. Since TCO is not prescribed in the C++17 standard, recursion must be unrolled into loops to avoid risk of stack overflow.

VARIANT FOR COPRODUCT The coproduct type constructor `S` inherits from `std::variant`. As pointed out in the previous paragraph on exceptions, `S` is a proof of concept and a more sound implementation would not have used `std::variant`. With that in mind, there are a few things to be aware of:

- variants can not hold arrays ([158, pp. 12.7.3–3]), and
- variants can become `valueless_by_exception` if an exception is thrown during a type-changing assignment or emplacement.

Since raw C-style arrays have a strange relationship with raw pointers, and we have excluded raw pointers from ***Cpp***, we must take bare arrays out of the model. The use of C-style arrays is discouraged by many important sets of best-practices and guidelines due to type-safety issues and their relationship to pointers. The STL provides `std::array` which is a type-safe statically-sized array that takes their place.

UNNATURAL FUNCTION TEMPLATES In §4.4, it was pointed out that natural transformations are well represented as function templates. However, not all function templates are natural transformations. The difference is parametric vs. ad-hoc polymorphism: natural transformations require parametric polymorphism. For function templates to model natural transformations, they should work for any given template typename argument without specialisation.

TYPE-CASTING Our model relies on the C++ type system: though all of this we have let types guide the design. As I was writing all of the test cases of the chapter, I was warned about each mistake because the types did not align. For better (and sometimes worse) C++ affords programmers the tools to circumvent the type system. To model homs we used `std::function` which uses type erasure for better, and *improves* type safety. But `static_cast` and `dynamic_cast` sidestep the type system which means departing from ***Cpp***. The case of `std::function` shows that when this is done judiciously, it can be used to good effect. But it is a breach of the categorical model and should only be done locally, within a type-safe wrapper.

OBJECT LIFETIME Object lifetime is an operational reality of C++ that has no depiction in ***Cpp***. It is up to the programmer to ensure that expressions in ***Cpp***/C++ are composed of living objects sharing a scope with the expression.

STACK VS. HEAP ALLOCATION The category ***Cpp*** has no regard for C++'s underlying memory model. For that matter, it does not know about cache layers, CPU memory paging of anything else about how data is laid out. It is still the programmer's art to optimise memory.

4.10 SUMMARY & CONCLUSION

The chapter started off defining a category of C++17 programs, ***Cpp***. This category borrows structure from **Set**, representing types as the set of their values and arrows as functions. So some hand waving and squinting gives ***Cpp*** the structure of a BICCC, and since ***Cpp*** is platonic, these things on their own mean very little. The interesting part is the representation. A particular (non-unique) representation is given alongside the development of the structure which is denoted ***Cpp***/C++.

After demonstrating that ***Cpp***/C++ could satisfy the category axioms (§4.2.1), we moved on to demonstrating *type functors*, defined as endofunctors on ***Cpp***. Examples given include

- `Vector`, a functor on the `std::vector` type constructor (eg. 4.3-2),
- `Const`, the family of constant functors fixed at any given ***Cpp***-object (eg. 4.3-3), and
- `sptr`, based on the `std::shared_ptr` type constructor, which is actually not a functor but is functorial up to isomorphism (eg. 4.3-4).

In §4.4 we covered natural transformations, where, as an example, a function was defined to compute the length of a `std::vector` as a natural transformation

$$\texttt{len} : \texttt{Vector} \Rightarrow \texttt{Const}\langle\texttt{std::size\_t}\rangle,$$

from the `Vector` functor to the `Constant` functor over `std::size_t`.

With the basic trinity of category theory: (1) categories, (2) functors and (3) natural transformations; each demonstrated in ***Cpp***, our attention turned to mapping out ***Cpp*** as a biccc. Table 4.2 is a summary of the ***Cpp***/c++ names of type constructors and function templates involved in carrying that structure.

table 4.2: Summary of biccc structures and c++ types upon which they can be modelled.

biccc			
ccc			
Category	Exponential	Product	Coproduct
• `Hom` • `Dom` • `Cod` • `compose`	• `pcurry` • `puncurry` • `ev`	• `P` type constructor • `associator_fd/rv` • unit, `I` • `unitor_fw/rv` • `braid`	• `S` type constructor • `associator_co_fd/rv` • unit, `Nothing` • `unitor_co_fw/rv` • `braid_co`
		product distribution over coproduct	
		• `factorise` • `expand`	

These structures build up in abstraction with each layer depending on the previous.

A core concept in the control-systems portion of the thesis is the notion of fixpoints. In particular, initial algebras or terminal coalgebras of functors modeling linear datatypes. We used a fixpoint operator `Mu` to build fixpoints of a type constructor `MP` with a functor instance `SnocF`, modeling the functor $1 + - \times T$ (for any T). This modelled snoc-lists. Section 4.8 demonstrates that these fixpoints and their universal algebra homomorphisms (catamorphisms) can be built upon the universal arrows of ***Cpp***/C++.

The motivation of this effort is to provide a foundation for categorical reasoning about C++ programs, including control programs. In the next chapter we will develop a model of control programs based on a category theoretical depiction of Moore machines [88]. These Moore machines are structures existing in the category of sets. Because we built ***Cpp*** as a category that can trivially be embedded in $**et**, and showed that it has representations in C++ code, we can confidently employ some morally correct "fast and loose reasoning" to design control software in the next chapter.

5 CATEGORIES TO CONTROLLER CODE

Talk is cheap. Show me the code.
— Linus Torvalds

THIS chapter brings together theory and C++ code to yield two things:

1. a theoretical model of a control program as a constellation of universal arrows in the category ***Cpp***, and
2. an abstract C++ architecture that accommodates any choice of control algorithm.

It is written in a bottom-up approach: first I will describe the software design, a simple expression in Rx that embodies the I/S/O relationships inherent in feedback control application software. From there, we will use the mathematics of CH. 3 and the structure ***Cpp***/C++ of CH. 4 to work up and underpinning and justification of the design. In particular, we will see that the design lends itself to a view of control programs as Moore machines.

5.0-1 The next section introduces a simple C++ structure called `MooreMachine`, that embodies that data of the classical notion as a Goguen machine in $\mathbb{F}$**inSet**. The definition of a `MooreMachine` will be the central data structure that is used to ground all of the more abstract mathematical concepts that category theory serves. The mathematical framework does not *add* data to the `MooreMachine` concept, rather, it captures the pattern of iteration that comes organically when Moore machines are animated as instances of Arbib's input processes. In other words, because CT constructions necessarily includes arrows as well as objects, by taking a categorical approach to Moore machines we simultaneously capture both the data and mechanism of the classical Moore machine. And further, because we situate this in a category of C++ programs, we get the code as a result.

In the final section of the chapter, I give a working example: a PID-controller running with a simulated plant. This example is well placed because PID controllers (for better or worse) are ubiquitous in industrial control systems. But better yet, they admit analytical solutions to which we can compare the numerically simulated results.

The next chapter is a fuller case study involving an advanced control technique (NMPC) running on a multi-system autonomous mobile robot.

5.1 OUTLINE OF CONTROL PROGRAMS

The construction at which this chapter aims is very intricate; a detailed clockwork of precision that spans many layers of abstraction. The ultimate goal is something simple and concrete: a small piece of C++ code that, depending on some user defined types and functions, implements any control algorithm. As in CH. 4.2, we take *program* to mean an arrow of ***Cpp***—and control programs are therefore a subset of those arrows. The starting point is the concept that a controller is a dynamical system in its own right; and a control program is regarded as a Moore machine in ***Cpp*** that embodies the relationship between input and output which are structured as asynchronous lists.

Control computers take, as input, time-series data arriving from sensors and operators[1], and produces a time-series of actuator commands. These streams of values are like lists or arrays except that the values they hold are not distributed through memory but through time. We cannot index into them, or pull values at will, so these streams are *observable* but not decomposable. Instead, we can then compose a chain of operations that gives an expression for the output stream in terms of the input streams, with no reference to individual values in the stream. When values become available in any input stream, they are pushed through the computation pipeline and any dependent output values are updated with the new information. This means that the flow of data drives the computation; pushing data rather than pulling it in from a loop-structure.

I give an overview of this sort of list/stream processing DSL in §2.3.2,

[1] Operators are people or supervisory software directing the controller by providing setpoints, reference values and other non-feedback input.

and a derivation with tutorial on Rx §3.10. While there are various libraries that will work for our purposes, I use RxCpp since it is actively maintained, expertly developed and well used[2].

Consider two observable streams

```
const observable<Y> plant_state = ...;
const observable<R> reference = ...;
```

Sensor readings take value in the type `Y` and operator input take value in the type `R`. One then defines an initial state for the controller, a value from the type `C`:

```
const S s0 = ...;
```

Now define the stream of control instructions, `controls`, as

LISTING 5.1.1:

```
const auto controls =
  plant_state
    | combine_latest(reference)
    | scan(c0, state_transition_function)
    | map(r);
```

The `combine_latest`, `scan` and `map` combinators are off-the-shelf components in any Rx implementation. The *pipe operator*, (`|`), denotes that the observable stream on the left is fed to the combinator on the right[3] producing a new (transformed) observable stream.

The `combine_latest` does what you may expect: it produces values in $Y \times R$ that is always a pair of the latest values in the streams `plant_state` and `reference`. When a new value is pushed along either stream, the combinator emits the pair of latest values from both. This is illustrated in Figure 5.1 as a *marble diagram*.

The `scan` combinator is where the bulk of the calculation goes on and it is where the state of the control computation is held. Starting from the

[2] It is well used since it is the official implementation on reactivex.io.

[3] If you are familiar with the POSIX shell “pipe” operator, then you have a good intuition for its meaning here.

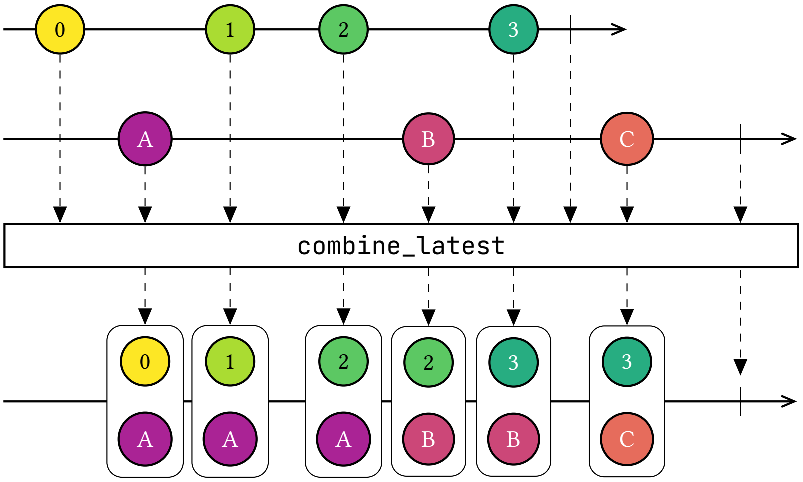


FIGURE 5.1: Marble diagram of the `combine_latest` combinator. NB: data intervals need not be constant.

initial controller state, and given a function

$$\texttt{state\_transition\_function} : \mathtt{S} \times (\mathtt{Y} \times \mathtt{R}) \to \mathtt{C},$$

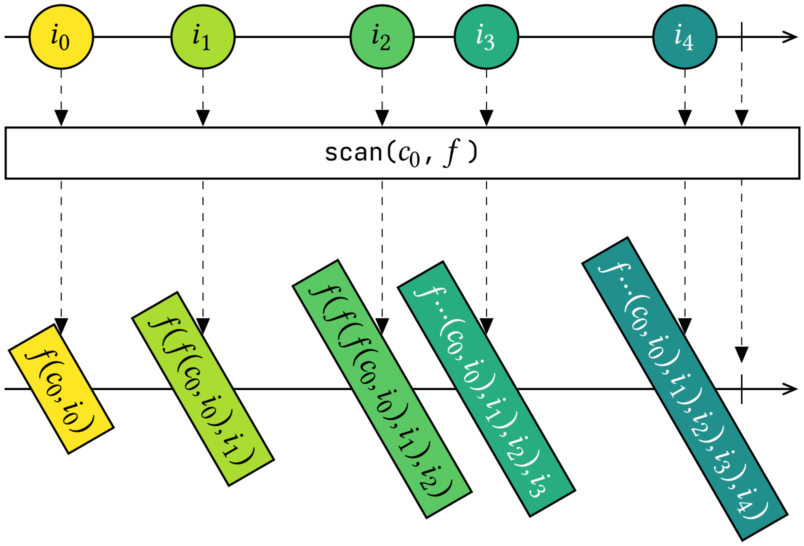
the scan combinator produces a stream of controller states updated by new input, according to `state_transition_function`. Traditional control algorithms will involve some combination of numeric integrals, finite difference schemes and anything else computed from the history of input values can be held and updated in the controller state, and propagated by `state_transition_function`.

The `map` combinator applies a function pointwise to a stream. In this case, the *readout* function,

$$\mathtt{r} : \mathtt{S} \to \mathtt{U},$$

which maps the controller's state to a value in `U` which can be exported to the actuators[4].

[4] Maybe through some sort of driver API or direct manipulation of hardware pin states.



i_0
i_1
i_2
i_3
i_4
scan(c_0, f)
$f(c_0,i_0)$
$f(f(c_0,i_0),i_1)$
$f(f(f(c_0,i_0),i_1),i_2)$
$f\ldots(c_0,i_0),i_1),i_2),i_3$
$f\ldots(c_0,i_0),i_1),i_2),i_3),i_4)$

FIGURE 5.2: Marble diagram for the `scan` combinator.

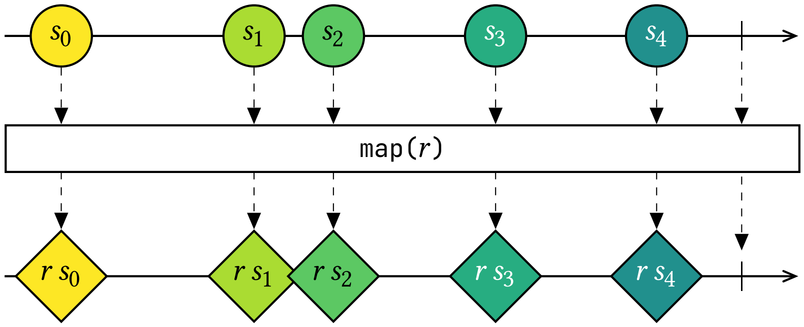
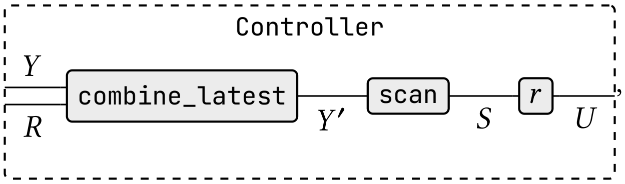
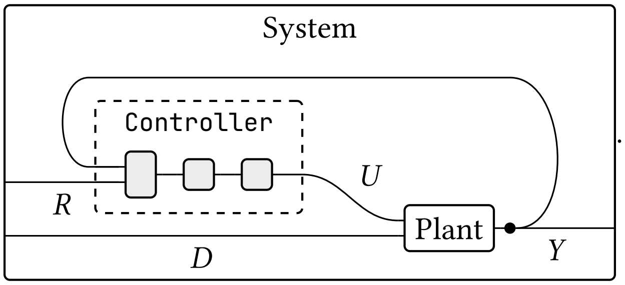


FIGURE 5.3: Marble diagram for the `map` combinator. NB: this combinator does not hold state.

This composition of `combine_latest`, `scan` and `map` can be summarised in a formal *wiring diagram*:

 (5.1.1)

which sits in the context of the larger system:



Although this system wiring diagram looks like (and is certainly comparable to) the block diagram FIG. 1.2, wiring diagrams are a pictographic expression language for composition in certain types of categories. They are cousins of *string diagrams*, and should be thought of as mathematical expressions, not thin visual metaphors. Their interpretation as such will not be a topic for now, but I direct the interested reader to [124, 182].

For intuition, one might combine the notions of marble and wiring diagrams. Instead of having time on a spatial axis, imagine a running animation where marbles travel along wires. You can see this illustrated in FIG. 5.4. The marbles enter combinators from the left and are transformed and passed back out to the right. The marbles can travel at different speeds and the combinators can introduce delay between input and output. In this model, it is the arrival of the marbles at the ports of the control program that drives the computation pipeline.

This slender code, the wiring diagrams, and the marble diagrams—are the results at which the remaining sections aims.

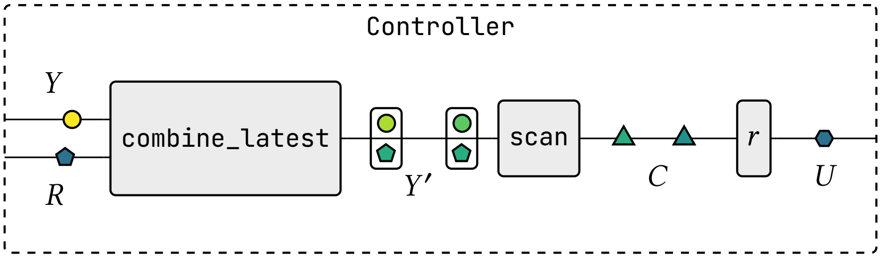


FIGURE 5.4: A wiring diagram with Rx marbles overlaid on wires. To be interpreted as a frame with animated marbles moving from left to right in time.

5.1.1 Control Programs & Moore Automata

In the point of view presented in this thesis, a control program is a composite of smaller I/S/O components each interacting through their input and output ports. We model these components as Moore machines or *automata*.

In this section, we categorify Moore machines in ***Cpp*** to capture the control program's I/S/O relationship and give it a representation in code. More specifically, we list and demonstrate a simple data structure named `MooreMachine` which will be an input for later category theoretically motivated transformations. By capturing the Moore structure and its behaviour in a universal constellation of ***Cpp***-arrows, we will have obtained two dependent structures, one algebraic and one coalgebraic, for mapping input values to output values while lawfully maintaining state.

5.2 ALGEBRA & COALGRA OF MOORE MACHINES

Moore machines can be regarded as the instance of Goguen machines erected in the category $\mathbb{S}$**et**. That is to say, they can be viewed as an algebraic construction. Section 3.9 describes how Moore machines can be regarded as coalgebra. To design the control program, we will lean heavily on the algebraic side to get us there, but the behaviour of the program will by defined by coalgebra.

Recall from DEF. 3.9-1 that a Moore machine is a 6-tuple,

$$(\, I,\, S,\, O,\, \delta,\, r,\, \vec{s}_0 \,),$$

where:

- S is a set of states and $\vec{s}_0$ is a global element evaluating as an initial state,
- I and O are sets of input and output values respectively,
- $\delta : S \times I \to S$, the transition function is an $\hat{I}$-algebra, mapping the state and input to a successor state and,
- $r : S \to O$, the readout function mapping the state to the observable output.

We can render this as a **Set**-diagram:

$$\begin{tikzcd} S \times I \arrow[r, "\delta"] & S \arrow[r, "r"] & O \\ 1 \arrow[ur, "\vec{s}_0"'] & & \end{tikzcd} . \tag{5.2.2}$$

Now consider the two functors: $- \times I$ and $I \multimap -$ denoted as $\hat{I}$ and E_I respectively. Table 5.1 shows the signatures of the co/structure maps of the algebras and coalgebras of these functors.

table 5.1: The signatures for the co/structure maps of algebras and coalgebras for the functors $\hat{I} = - \times I$ and $E_I = I \multimap -$. $\hat{I}$-coalgebra and E_I-algebras are de-emphasised because they do not contribute to the discussion.

	$\hat{I}\,S$	$E_I\,S$
algebra	$S \times I \to S$	$I \multimap S \to S$
coalgebra	$S \to S \times I$	$S \to I \multimap S$

OBSERVATION 5.2-1. Notice that the $\hat{I}$-algebra over S is exactly the signature of δ and E_I-coalgebra over S is exactly $\grave{\delta}$. This is not a coincidence since products and exponentials are adjoints [109, EG. 9.7, P. 215], so $\hat{I}$-**Alg** and E_I-**coAlg** are related in kind. That is, any $\hat{I}$-algebra can be curried into an E_I-coalgebra

NOTE. Soon we will be using C++17 to express Moore machines, but the standard (and certainly the compilers) disallow single-charater Greek names like δ. So from hereon, we use `f` in code and δ interchangeably, with f/`f` leaning toward use in code listings as well as the specific case state-transition functions of control algorithms.

We can add $\grave{\delta}$ into (5.2.2):

$$\begin{tikzcd}
S \times I \arrow[r, "\delta"] & S \arrow[r, "\grave{\delta}"] \arrow[dr, "r"'] & I \multimap S \\
1 \arrow[ur, "\vec{s}_0"'] & & O \\
\underbrace{\hphantom{S \times I \to S}}_{\text{algebra}} & & \underbrace{\hphantom{S \to I \multimap S}}_{\text{coalgebra}}
\end{tikzcd} .$$

Through a progression of code examples building on ***`Cpp`***/C++, we will start from a naïve example of a Moore machine, to and algebraic model with catamorphisms before arriving at a coalgebraic model with Rx observables.

5.3 MOORE MACHINES IN CODE

We can render the data of the classical Moore machine in a simple `struct` template:

```
template <typename I, typename S, typename O>
struct MooreMachine {
  S s0;
  std::function<S(S, I)> tmap;
  std::function<O(S)> rmap;
};
```

In the actual control systems code, we do not define `MooreMacine`: the structure is a tool for demonstration.

The following example establishes a leitmotif upon which the subsequent examples build.

EXAMPLE 5.3-1. Our Moore machine is

$$\begin{aligned}\Sigma = (S = \texttt{int},\ s_0 &= 0,\\ I &= \texttt{int},\\ O &= \texttt{int},\\ f &= (s,i) \mapsto s+i,\\ r &= \mathrm{id}_S),\end{aligned}$$

which keeps a running sum of the inputs as its state, and has full state-output. We can avail ourselves of a few (global) type aliases aimed at our running-sum problem.

```
using State = int;
using Output = int;
using Input = int;
```

This will disambiguate the three in the code, and leave `I`, `S` and `O` as template variables.

The following code is intentionally obtuse. It demonstrates *very explicitly* the pattern of induction we wish to capture. We will revisit the example as we develop category theoretical tools that encompass the behaviour, making the iteration automatic—a consequence of the essential structure of the category theoretical objects.

LISTING 5.3.1: A Catch2 `TEST_CASE` demonstrating how the constituents of a `MooreMachine` are used to turn a list of input to a list of output using a simple running-sum of integers.

```
TEST_CASE(
    "Given a MooreMachine where,\n"
    "   S = O = I = int\n"
    "  s0 = 0\n"
    "   f = (i, s) ↦ s + i\n"
    "   r = s ↦ s,\n"
    "and given an input vector `i_s` and manually computed "
    "`running_sum…`") {
  const State s0 = 0;
  const auto f = [](State s, Input i) -> State { return s + i; };
  const auto r = id<State>;
  const auto mm = MooreMachine<Input, State, Output>{s0, f, r};

  const auto i_s = std::vector<Input>{0, 1, 2, 3, 4};
  const auto running_sum = // $\{ s_0, r \circ f(s_k, i_k) \}_{k=0}^{4}$.
      std::vector<Output>{0, 0, 1, 3, 6, 10};
  //                        ↑
  //               Initial state
  AND_GIVEN(
      "a function that explicitly demonstrates the "
      "recursion of $f$ while generating a sequence of "
      "successive output values.") {

    auto manual_moore_machine =
        [&i_s, &mm]() -> std::vector<Output> {
      const auto [s0, f, r] = mm;
      return {
        r(s0),
        r(f(s0, i_s[0])),
        r(f(f(s0, i_s[0]), i_s[1])),
        r(f(f(f(s0, i_s[0]), i_s[1]), i_s[2])),
        r(f(f(f(f(s0, i_s[0]), i_s[1]), i_s[2]), i_s[3])),
        r(f(f(f(f(f(s0, i_s[0]), i_s[1]), i_s[2]), i_s[3]), i_s[4]))
      };
    };

    THEN("we should expect the running sum including the output "
         "of the initial state.") {
      REQUIRE(manual_moore_machine() == running_sum);
    }
  }
```

MooreMachines can be converted into algebras of the snoc list structure from §4.8. First, let us define a helper to reduce verbosity:

```
template <typename T, typename Carrier>
using SnocAlg = typename SnocF<T>::template Alg<Carrier>;
```

Now we can turn any Moore machine into a SnocAlg by:

```
template <typename S>
auto moore_to_snoc_algebra(MooreMachine<Input, S, Output> mm)
    -> SnocAlg<Input, S> {
  auto global_s0 = [mm](PUnit) -> S { return mm.s0; };
  auto state_trans = [mm](P<std::shared_ptr<S>, Input> p) -> int {
    auto [s, i] = p;
    return mm.tmap(*s, i);
  };

  return fanin(global_s0, state_trans);
}
```

In principle, all we have done is rearrange the data in a form that can be given life, animated by the categorical machinery of catamorphisms. This is what the last example looks like in that context. In the following example, recall the functor B from DEF. 3.6-20, mapping machines to I/O behaviours as

$$\Sigma \mapsto r \circ (\!|\vec{s}_0 \triangledown \delta|\!) : \ I^* \to O.$$

EXAMPLE 5.3-2. Extending EG. 5.3-1 (scope repeated in grey) the following test-case demonstrates that `moore_to_snoc_algebra` maps `MooreMachines` such that we can construct B `mm`.

```
TEST_CASE(
    "Given a MooreMachine where,\n"
    "   S = O = I = int\n"
    "  s0 = 0\n"
    "   f = (i, s) ↦ s + i\n"
    "   r = s ↦ s,\n"
    "and given an input vector `i_s` and manually computed "
    "`running_sum…`") {
  State s0 = 0;
  auto f = [](State s, Input i) -> State { return s + i; };
  auto r = id<State>;
  auto mm = MooreMachine<Input, State, Output>{s0, f, r};

  auto i_s = std::vector<Input>{0, 1, 2, 3, 4};
  // running_sum = { s_0, r ∘ f(s_k, i_k) }^4_{k=0}.
  auto running_sum = std::vector<Output>{0, 0, 1, 3, 6, 10};
  //                                     ↑
  //                              Initial state
  AND_GIVEN(
      "An Î•-algebra embodying δ and s_0, and "
      "corresponding I/O response function "
      "phi= B mm = r ∘ (|alg|)") {
    auto alg = moore_to_snoc_algebra(mm);
    auto phi = compose(mm.rmap, SnocF<Input>::cata<State>(alg));
    auto total =
        mm.rmap(std::accumulate(cbegin(i_s), cend(i_s), 0));

    auto snoc_is = to_snoclist(i_s);

    THEN("phi applied to an empty list should produce the "
         "initial state.") {
      REQUIRE(phi(nil<Input>) == mm.rmap(mm.s0));
    }

    THEN("phi applied to i_s should produce its sum total.") {
      REQUIRE(phi(snoc_is) == total);
    }

    // A new test will be added here in a moment.
  }
}
```

And indeed, this is a passing test included in [193].

NB: note the comment line at the end telling of an addition we will make soon.

The problem here is that we need access to the intermediate states in the calculation: a control program must produce output at every step. What we want is not B `mm` but a snoc-scan (§3.8.2). In the next section 5.4, I will show that every pointed object-algebra can be transformed (functorially) into algebras producing a scans as their catamorphisms. For now, in code, it is simply a function template:

```
template <typename I, typename S>
auto snoc_scanify(SnocAlg<I, S> alg) -> SnocAlg<I, SnocList<S>> {
  return [alg = alg](auto unit_or_p) -> SnocList<S> {
    auto global_snoc_s0 = [&alg](PUnit) {
      return snoc(nil<S>, alg(PUnit{}));
    };

    auto accum_trans =
        [&alg](P<std::shared_ptr<SnocList<S>>, Input> p)
        -> SnocList<S> {
      auto [accum, val] = p;
```

```
        const auto s0 = unsafe_head(*accum);
        return snoc(*accum, alg(P{std::make_shared<S>(s0), val}));
      };
      return fanin(global_snoc_s0, accum_trans)(unit_or_p);
    };
  }
```

This takes an algebra `alg` : $\hat{I}_\bullet S \to S$ returns an algebra of type $\hat{I}_\bullet S^* \to S^*$. Given an empty list, the "scanified" algebra produces $[\, s_0 \,]$ (the initial value wrapped in a snoc list) and given a pair $(w, i) \in S^* \times I$ taking the head of the list of states, passes it to δ along with i and then returning w with the new state appended.

Now to the test case in EG. 5.3-2 we can add (where indicated)

```
    THEN("The scanified version of that algebra should produce "
         "a list (i.e., SnocList) of the running sum.") {
      auto running_sumer = SnocF<Input>::cata<SnocList<State>>(
          snoc_scanify<Input, State>(alg));
      REQUIRE(running_sumer(snoc_is) == to_snoclist(running_sum));
    }
```

which passes.

Now we we should like to start working toward the Rx-based interface. It was shown in §4.8.4 that snoc lists are isomorphic to `std::vectors`—grounding the type-system based list into an object. In the 2020 version of the C++ standard, [171], partly adopted Eric Niebler's range-v3 library, which gives iterable collections the "pipe" syntax with the aim of making algorithms on collections composable. Notably, this includes vectors. The thesis demo code includes an example with range-v3, as an intermediate step between `std::vectors` and asynchronous collections in Rx. Without repeating the full test here, the interesting part of the code is:

```
template <typename I, typename S, typename O>
auto rang_v3_moore_machine(MooreMachine<I, S, O> mm)
    -> Hom<std::vector<I>, std::vector<S>> {
  using namespace ranges;
  return [mm](std::vector<I> i_s) {
      const auto o_s =
          i_s | views::exclusive_scan(0, mm.tmap) | views::transform(mm.rmap);
      return std::vector(std::cbegin(o_s), std::cend(o_s));
  };
}
```

which uses std::exclusive_scan and std::transform (adapted for ranges) in the expression for o_s. The vector of values *i_s* is fed into the scan, which is then mapped over by r. Contrast that with the nearly identical version for Rx:

LISTING 5.3.2:

```
template <typename I, typename S, typename O>
auto rx_moore_machine(MooreMachine<I, S, O> mm)
    -> Hom<rx::observable<I>, rx::observable<S>> {
  return [mm](rx::observable<I> i) {
    return i | rx_scanl(mm.s0, mm.tmap) | rx::map(mm.rmap);
  };
}
```

The combinator rx_scanl is a slight modification of RxCpp's built-in scan. This is to address (imperfectly) an issue that many scan algorithms disagree on whether or not the initial state is included in the output. We want the initial state in the output, and unfortunately for us, RxCpp's scan does not. Here is the modification, where we explicitly start_with the initial state:

```
template <typename I, typename S>
auto rx_scanl(S s0, Hom<Doms<S, I>, S> f)
    -> Hom<rx::observable<I>, rx::observable<S>> {
  return [s0, f](rx::observable<I> obs) {
    return obs.scan(s0, f).start_with(s0);
  };
}
```

And now, here is the full example.

EXAMPLE 5.3-3. Extending EG. 5.3-1 (scope repeated in grey) the following test-case demonstrates the Rx expression above produces the output sequence of the Moore machine.

```
TEST_CASE(
    "Given a MooreMachine where,\n"
    "   S = O = I = int\n"
    "  s0 = 0\n"
    "   f = (i, s) ↦ s + i\n"
    "   r = s ↦ s,\n"
    "and given an input vector `i_s` and manually computed "
    "`running_sum…`") {
  State s0 = 0;
  auto f = [](State s, Input i) -> State { return s + i; };
  auto r = id<State>;
  auto mm = MooreMachine<Input, State, Output>{s0, f, r};

  auto i_s = std::vector<Input>{0, 1, 2, 3, 4};
  // running_sum = { s₀, r ∘ f(sₖ, iₖ) }⁴ₖ₌₀.
  auto running_sum = std::vector<Output>{0, 0, 1, 3, 6, 10};
  //                                      ↑
  //                              Initial state
  AND_GIVEN("With rx_moore_machine") {

    auto custom_moore_scan = [&i_s, &mm]() -> std::vector<Output> {
      auto oi = make_vector_observable(i_s);
      auto oo = oi | rx_moore_machine(mm);

      std::vector<Output> output;
      oo.subscribe([&output](Output v) { output.push_back(v); });

      return output;
    };

    THEN("expect a running sum without output from the initial "
         "state.") {
      REQUIRE(custom_moore_scan() == running_sum);
    }
  }
}
```

And indeed, this is a passing test included in [193].

5.3-4 In transitioning from `std::vector` to Rx observables, we have made an interesting shift. While scans on `std::vectors` were underpinned by snoc-catamorphisms, asynchronous collections do not compute catamorphically because they do not (and can not) traverse the list to its root before yielding a result. See OBS. 3.9-6 for more context. We have somehow transitioned into something behaving anamorphically. In spirit, the RxCpp scan produces a stream of intermediate results from $(\!| \grave{f} \vartriangle \mathrm{id} |\!)\, c_0$, where the readout is identity. This is why in LST. 5.3.2 we applied `r` using the `map` combinator, which is the `fmap` of the list functor.

5.4 FROM FOLDS TO SCANS

Ordinary folds can be made into scans by means of the endofunctor:

$$\begin{aligned} \texttt{scanify} : \hat{I}_{\bullet}\text{-}\mathbf{Alg} &\to \hat{I}_{\bullet}\text{-}\mathbf{Alg} \\ (A,\, \alpha) &\mapsto (A^*,\, \alpha') \\ f &\mapsto f^*. \end{aligned} \tag{5.4.3}$$

where

$$\alpha'(a) = \begin{cases} [\, \alpha() \,] & \text{if } a = () \\ w +\!\!+ [\, \pi_2 \circ \alpha(\kappa^{-1},\, i) \,] & \text{if } a = (w :\ A^*,\, i :\ I). \end{cases}$$

Because free $\hat{-}_{\bullet}$-algebras are known to exist as fixpoints in $\hat{-}_{\bullet}$-**Alg**, we can press them into service of this operation, thus the appearance of $(-)^*$ on the `output` of the machine.

It can be shown that for (5.4.3) respects the functor axioms (§B.6-1). That is, given any $(A,\, \alpha)$,

$$\texttt{scanify}(\mathrm{id}_{(A,\,\alpha)}) = \mathrm{id}_{\texttt{scanify}\,(A,\,\alpha)}$$

and given compatible algebra homomorphisms $f,\ g$,

$$\texttt{scanify}(g \circ f) = \texttt{scanify}(g) \circ \texttt{scanify}(f).$$

The algebra homomorphisms in $\hat{I}_{\bullet}$-**Alg** are just $\mathbf{Set}$-functions that preserve the structure of the algebra. In this case, showing functionality is a familiar exercise to functional programmers because `scanify`'s action on arrows is just the standard list-functor in $\mathbf{Set}$. Moreover, though it is not needed for the present work, it can be shown that one can revert α' back to α by simply taking the head of w after α'.

In LST. 5.3.2, an implementation of a Moore machine was given as an Rx expression. Confer that with the control expression, LST. 5.1.1, and the only difference is that the control expression combines, as input, a reference signal and a (measured or estimated) plant state into a unified input to the scan. And, as pointed out in **5.3-4**, we are now working in coalgebras and anamorphisms. Recall the diagram (3.9.25), repeated here for convenience:

$$\begin{tikzcd}
\hat{I}_\bullet\, I^* \arrow[r, "\hat{I}\,\llparenthesis \vec{s}_0 \triangledown \delta \rrparenthesis"] \arrow[rr, bend left, "\hat{I}\,\llparenthesis \vec{y}_\Sigma \triangledown \varphi \rrparenthesis"] \arrow[d, shift left, "\bar{+\!\!+}_\bullet"] & \hat{I}_\bullet\, S \arrow[r, "\hat{I}\,\llbracket \grave{\delta} \vartriangle r \rrbracket_M"] \arrow[d, "\vec{s} \triangledown \delta"] & \hat{I}_\bullet\, (I^* \multimap O) \arrow[d, "\vec{y}_\Sigma \triangledown \varphi"] & \\
I^* \arrow[u, shift left, "\bar{+\!\!+}_\bullet^{-1}"] \arrow[r, "\llparenthesis \vec{s}_0 \triangledown \delta \rrparenthesis"] \arrow[rr, bend right, "\llparenthesis \vec{y}_\Sigma \triangledown \varphi \rrparenthesis"'] & S \arrow[r, "\llbracket \grave{\delta} \vartriangle r \rrbracket_M"] & I^* \multimap O \arrow[r, "ev_B"] & O
\end{tikzcd} .$$

Listing 5.1.1 makes a control program into a Moore machine where we only observe output. The control algorithm is encoded in δ (or f). As mentioned in **5.3-4**, the scan produces intermediate values of a calculation that amounts to $\llbracket \grave{f} \vartriangle \mathrm{id} \rrbracket_M\, c_0$, where c_0 is the initial state of the controller. State readout is effected afterward by mapping r over the intermediates in the pipline. What remains then is to give a full componentwise coalgebraic description of the three transformations in the pipeline, illustrated in the wiring diagram (5.1.1). This portion of the thesis was published as a paper [184] which I coauthored with my supervisors.

The `combine_latest` block collects the measured plant state and reference values, which may arrive asynchronously, and puts out a value reflecting the most recent updates to each. For simplicity, let $Y' := Y \times R$. So the block's state and output are from $Y \times R$, but its input is from $Y \sqcup R$. Whichever value arrives is remembered in the state, which is passed on as output. The state dynamics can be modelled as a $M_{Y\times R}^{Y\sqcup R}$-coalgebra

$$\big(\, Y\times R,\ \psi :\ Y\times R \to (Y+R \multimap Y\times R)\times(Y\times R)\,\big).$$

where ψ is the costructure map of a Moore machine as in (3.9.26):

$$\psi(p :\ Y\times R) = \big((m :\ Y+R)\mapsto f(p,m),\ p\big)$$

with

$$f(p,m) := \begin{cases} \big(m,\ \pi_2(p)\big) & \text{if } m\in Y \\ \big(\pi_1(p),\ m\big) & \text{if } m\in R. \end{cases}$$

The `compute` block encodes the control algorithm. Its state, from set C and beginning with c_0, holds all information required for that running calculation. To model the dynamics as a Moore machine, we first need a state transition function $f :\ C\times Y' \to C$, taking the value from `combine_latest` along with the previous state of the controller to give an updated state. Then, we define the costructure map:

$$\varphi(c :\ C) = \big((y :\ Y')\mapsto f(c,y),\ c\big).$$

The computation of control values is then reduced to a scan over the input, $\llparenthesis \varphi \rrparenthesis$.

Finally, the third block, simply labelled r, reads the control output from the state of the controller: $r :\ C\to U$. It has no state, so it does not need to be modeled by anything more than it is: a function.

5.6 EXAMPLE: PID CONTROLLER

PID controllers, along with Proportional (P), Proportional-Integral (PI) and Proportional-Derivative (PD) controllers are a class of automatic reference tracking feedback control techniques widely used in industry [173]. They take, as input, the error between a desired setpoint and the actual process output, producing a control signal as the sum of three terms over the error signal: one proportional, one integral, and one derivative:

$$u(t) = k_p\, e(t) + k_i \int_0^t e(t')\,\mathrm{d}t' + k_d \frac{\mathrm{d}}{\mathrm{d}t} e(t). \tag{5.6.4}$$

where:

- $t \in \mathbb{R}_{[0,\infty)}$ is the additive monoid of continuous time,
- u is the control signal,
- e is the error signal, and
- k_p, k_i and k_d are scaling coefficients or "gain" for each term.

Each of these terms has purposes and balanced trade-offs:

The proportional term helps in reducing steady-state error, but can lead to overshoot and oscillations if the gain is too high,

The integral term reduces steady-state error, but can lead to instability if the gain is too high,

The derivative term reduces overshoot and damps oscillations, but is prone to amplify high frequency noise in the system.

The balance is struck my judicious choice of scaling coefficients, k_p, k_i and k_d.

To demonstrate the control program model, we use a discrete-time implementation of PID. Indeed, most modern applications of PID control are for computer controlled systems[5].

The simplest (perhaps crudest) techniques to discretise (5.6.4) are to approximate the derivative by 2-point forward difference and the integral by right-rectangular approximation. Consider the space of monotonically increasing sequences $(\mathbb{R},<)^{(\mathbb{N},<)}$, [6] mapping $\mathbb{N} \ni n \mapsto t_n \in \mathbb{R},$. The time-derivative of e is discretely approximated as

$$\frac{\mathrm{d}e}{\mathrm{d}t}(t_n) \approx \frac{e(t_n) - e(t_{n-1})}{t_n - t_{n-1}}, \tag{5.6.5}$$

[5] Servomotors are a quintessential example here.

[6] Where $(\mathbb{R},<)$ and $(\mathbb{N},<)$ are strict total orders with the usual "less than" ordering on real or natural numbers, respectively. Functions that preserve the ordering, $a < b$ implies $f(a) < f(b)$, are monotonically increasing.

and the integral is approximated as

$$\int_0^{t_n} e(t)\,\mathrm{d}t \approx e(t_n)\,(t_n - t_{n-1}) + \sum_{i=1}^{n-1} e(t_i)\,(t_i - t_{i-1}). \tag{5.6.6}$$

The first term is pulled out of the sum because that sum will end up in an accumulator in the final program. Substituting (5.6.5) and (5.6.6) into (5.6.4), we obtain an expression for the computation of a discrete-time PID control signal:

$$u(t_n) = k_p\,e(t_n) + k_i\left(e(t_n)\,(t_n - t_{n-1}) + \sum_{i=1}^{n-1} e(t_i)\,(t_i - t_{i-1})\right) \\ + k_d\,\frac{e(t_n) - e(t_{n-1})}{t_n - t_{n-1}}. \tag{5.6.7}$$

5.6-1 One final thing to discuss is what happens with (5.6.7) at t_0, where neither the derivative nor integral terms can produce a value. At this point we, simply define $u(t_0) \overset{\text{def}}{=} 0$. This introduces a lag suffered by the discrete PID that is not present in the continuous case. When we abstract the control algorithm into a state model, we could then adopt the nomenclature of Willems [75], that the control has 1-finite memory or a memory span of 1. This means that we must specify exactly 1 initial value to seed the calculation of the sequence u.

5.6.1 The Plant

The plant in this example is a damped harmonic oscillator with the addition of a static/DC force. The differential equation for the position of the oscillator is:

$$m\,\frac{\mathrm{d}^2 z}{\mathrm{d}t^2}(t) = -\kappa\,z(t) - \beta\,\frac{\mathrm{d}z}{\mathrm{d}t}(t) - \rho - u(t),$$

where

- mass, m,
- $z : \mathbb{R} \to \mathbb{R}$ is the position of the oscillator in time and

- $\kappa \in \mathbb{R}_{>0}$ is the Hooke's law constant, scaling the strength of the restoring force.
- $\beta \in \mathbb{R}_{\geq 0}$ is the damping coefficient, scaling the strength of the speed dependent damping.
- $u : \mathbb{R} \to \mathbb{R}$ is the controller output.

The customary approach to this problem is to reduce it to an initial value problem in the first order of a vector-valued state variable:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\kappa/m & -\beta/m \end{bmatrix}\begin{bmatrix} z(t) \\ \dot{z}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \rho + u(t) \end{bmatrix} \tag{5.6.8}$$

with initial conditions

$$\begin{bmatrix} z(0) \\ \dot{z}(0) \end{bmatrix} = \begin{bmatrix} z_0 \\ v_0 \end{bmatrix}$$

Here, the state, $\boldsymbol{x} = [\, z \;\; \dot{z} \,]^\top \in X = \mathbb{R}^{2\times 1}$, consists of position and velocity to furnish the double integration of the original problem.

Equation (5.6.8) will be solved numerically, using the 4th-order Runge-Kutta method provided in Boost.Numeric.Odeint [136], so we need not explicitly discretise the problem here.

5.6.2 The Control Program

The heart of the control program is the same expression derived in the previous section:

```
const auto s_controls =
    world_ix.get_plant_observable()
      | rx::combine_latest(position_error, world_ix.setpoint)
      | rx::scan(c0, pid_algebra(k_p, k_i, k_d))
      | rx::map([](CState c) -> double { return c.value.u; });
```

In words, we have a stream of control values, `s_controls`, that is defined in terms of a pipeline starting at the (simulated) world interface `world_ix` which provides an observable stream of plant states. In the next stage of the pipline, those plant states are combined with the setpoint using a function `position_error` to produce the error signal. This is passed to the streaming scan computation which evolves the state of the controller, starting at initial state `c0` and marching forward through application of the algebra produced by `pid_algebra` which takes as arguments the coefficients for the P, I and D terms.

Unpacking that expression further requires a breakdown of the types involved. The plant state is

```
//            ┌   ·   ┐  // time
// PState =   │ ┌ · ┐ │  // position
//            └ └ . ┘ ┘  // velocity
//            ^ ^
//            │ value
//            SignalPt
using PState = SignalPt<std::array<double, 2>>;
```

where

```
template <typename T, typename Clock = chrono::steady_clock>
struct SignalPt {
  chrono::time_point<Clock> time;
  T value;
};
```

which templates on the clock type from the `std::chrono` library. [7] The array of two double-precision floating point numbers emcompases the state vector from (5.6.8), $[\, z \;\; \dot{z} \,]^{\top} \in X = \mathbb{R}^{2\times1}$.

The setpoint is the position of the oscillator that the controller will drive toward. Its type is aliased:

```
using SetPt = double;
```

The error signal is just the difference between the oscillator position and the setpoint.

```
inline auto position_error(PState x, SetPt setp) -> ErrPt {
  return {x.time, setp - x.value[0] };
}
```

[7] In this section, we simply use the default of `steady_clock`, but we retain the felxibility of choice through templates.

where ErrPt is another alias of a signal of double:

```
using ErrPt = SignalPt<double>;
```

The state of the PID controller is

```
struct PIDState {
  double err_accum;
  double error;
  double u;
};
```

where:

- err_accum stores the accumulation for the integral term, the sum in (5.6.6),
- error stores the value of e for the next iteration, and
- u is the control output.

Since the controller performs numerical differentiation and integration, it needs to keep track of time. So the controller state will be wrapped as a SignalPt too:

```
//           ┌   ·   ┐  // time
//           │ ┌ · ┐ │  // accumulated error
// CState =  │ │ · │ │  // error
//           └ └ . ┘ ┘  // control value
//           ^ ^
//           │ value
//           SignalPt
using CState = SignalPt<PIDState>;
```

The algebra for the scan embodies the expression for the discrete-time PID signal, (5.6.7):

```
auto pid_algebra(double k_p, double k_i, double k_d)
    -> Hom<Doms<CState, ErrPt>, CState> {
  return [k_p, k_i, k_d](CState prev_c, ErrPt cur_err) -> CState {

    const auto delta_t = seconds_in(cur_err.time - prev_c.time);

    if (delta_t <= 0) // Default to P-control if Δt ≤ 0
      return  {prev_c.time,
                { prev_c.value.error
10                  , prev_c.value.err_accum
11                  , k_p * cur_err.value
12                  }};
13
14      const auto integ_err =
15          std::fma(cur_err.value, delta_t, prev_c.value.err_accum);
16
17      const auto diff_err =
18          (cur_err.value - prev_c.value.error) / delta_t;
19
20      const auto u =
21          k_p * cur_err.value + k_i * integ_err + k_d * diff_err;
22
23      return {cur_err.time, {integ_err, cur_err.value, u}};
24    };
25  }
```

In addition to the straightforward calculation of (5.6.6), it is worth pointing out that in lines 7–12 handle the matter of **5.6-1**. The timestamps on the initial states of the plant and controller are initialised with time provided by the runtime system at program startup. In the first call to a `pid_algebra`, we expect `delta` $=$ `0`. In that case, or in any other where we lose sequencing order of input timestamps, we bypass calculation of the integral and derivative terms and default to `P`-control. In this example, we can expect the timestamps to increase monotonically from then on due to the guarantees provided by `steady_clock` [158, §23.17.7.2]. If we had a completely asynchronous environment where we did not have ordering guarantees on timestamps, this behaviour would still be reasonable. If time sequence violation precludes calculation of the derivative and integral approximations, we have three options:

1. return the proportional term alone,
2. halt and wait for either temporal coherence or operator intervention, or
3. repeat the previously computed control value until coherence returns.

For this example, a simple solution suffices.

The world interface not only provides the observable stream of plant states, it also has a member function to accept a control value and step the internal simulation of the plant dynamics. So the control expression begins and ends on the interface of `world_ix` and the subscription

```
s_controls.subscribe([&world_ix](double u) {
  world_ix.controlled_step(u);
});
```

closes the feedback loop.

5.6.3 The Test Examples

The unit-step response of the PID-controller was simulated for four combinations of PID-coefficients:

(A) P-control, $(k_p, k_i, k_d) = (300, 0, 0)$;
(B) PD-control, $(k_p, k_i, k_d) = (300, 0, 10)$;
(C) PI-control, $(k_p, k_i, k_d) = (30, 70, 0)$; and
(D) PID-control, $(k_p, k_i, k_d) = (350, 300, 50)$

These are each applied to a harmonic oscillator with $m = 1$, $\kappa = 10/m$, $\beta = 20/m$, $\rho = 1$. The simulation time was 2 s with a constant sample interval of 0.001 s.

Simulation results are compared to analytical solutions for the continuous PID-controller with results illustrated in Figure 5.5. The agreement of the lines in that figure make it difficult to discern and characterise differences at the scale of the plots. Insofar as visual inspection can note a deviation of the simulation line, it is "leading" (reacting faster) than the analytical line at tight contours near the rise- and peak-times. This is predictable algorithmic error due in-part to the initial lag in the first discrete time step where the simulated PID controller initially gives nil output leading to a larger error accumulation than the continuous model. Table 5.2 provides a quantitative summary, tabulating the Root-Mean-Square Deviation (RMSD) and the percent-difference in Steady State Error (SSE) of the simulated results from the analytical results. The RMSD for all cases is in

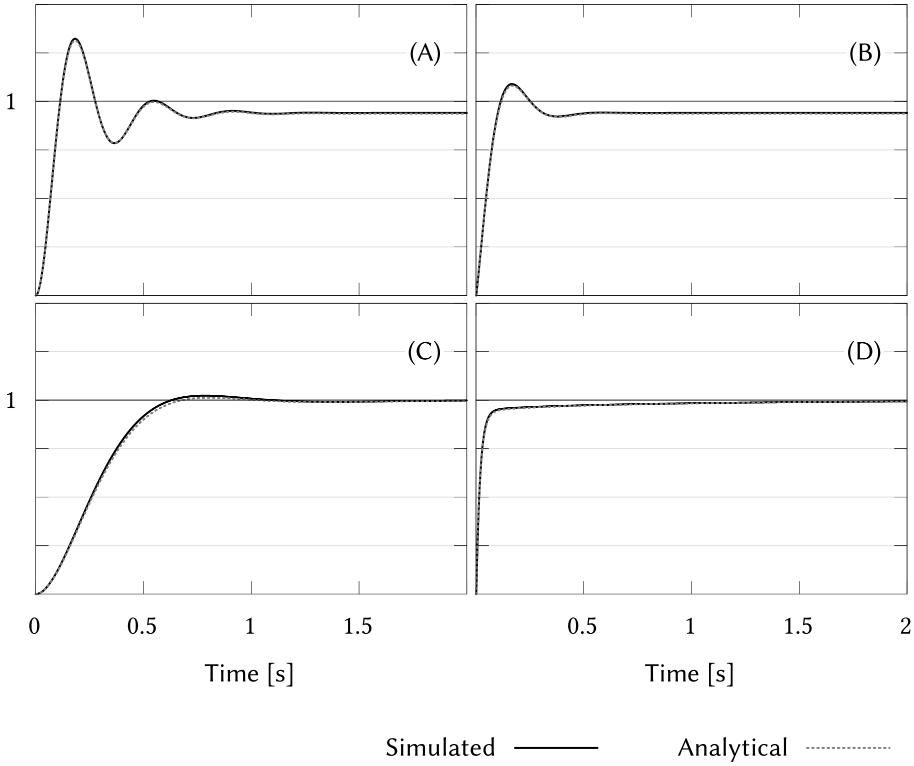


FIGURE 5.5: The unit-step response curves for each of the itemized test-cases: (A) P-, (B) PD-, (C) PI- and (D) PID-control. In each case, control is applied to the damped harmonic oscillator. The solid lines illustrate simulation results using the developed Rx control and numerically integrated oscillator; the dotted lines illustrate closed-form analytical solutions.

table 5.2: For each of the test examples, a tabulation of RMSD and percent difference of SSE between analytic solution and PID simulation result.

	(A)	(B)	(C)	(D)
RMSD	3.908	3.976	8.948	4.259
SSE diff.	0.334	0.333	0.068	0.038

the order of one part in a thousand on data that is of unit order; and the SSE of the simulation results deviates from the analytic solution by about ⅓rd of a percent or less. Note that even for the (C) and (D) cases with higher RMSD, the SSE difference is relatively low, suggesting that deviations in the acute activity of the discrete controller are transient and do not predict deviant behaviour on longer timescales.

Although the point of this exercise is not to achieve high numerical fidelity, I personally find these results satisfying. The RMSD are comfortably within tolerances we might hope for—considering the simplicity of the numerical model.

5.7 LIMITATIONS

The main limitation with the Rx representation of the model is that merging event streams is non-deterministic: when the streams contain simultaneous events, sometimes one occurrence is merged before the other, and sometimes the other way around. This is one of many motivations for eventually moving to FRP, once there is a clean and hardened implementation available.

6 BISHOP ROBOT: A CASE STUDY

SENTRYnet was a project from the Innovation for Defence Intelligence and Security (IDEaS) program by the DRDC. The focus of the project was an autonomous robot supplied by CrossWing Inc., called *Bishop* (pictured in Figure 6.1). The project was a cooperative distributed across several universities and industrial partners, and Prof. Eklund was an investigator on the project. As his graduate student, I developed the motion control system, including obstacle avoidance, for Bishop. That project was called *trusted motion control* and the system was developed under the codename *catnav* [1] used the software design detailed in CH. 5, and is the subject of this chapter.

FIGURE 6.1: CrossWing's Bishop.

Chapter 5 concluded with the example of PID-control of a simulated harmonic oscillator. The key differences between that example and this case study are:

- Integration: catnav is a component of a larger system integrated through ROS. This means that observables have to come in through ROS topics and output is published to a ROS topic.
- Algorithmic complexity: catnav uses an NMPC algorithm for computing control output.
- Plant dynamics: the robot, as a plant, is more complex. Its state representation has many more variables and is nonlinear.
- Input: the input is much more complex. It includes an entire path for tracking reference (vis-à-vis the simple set-point of the PID) but also information about obstacles and transformations between coordinate frames.

[1] The pneumonic is CATegory theoretical NAVigation

6.1 NONLINEAR MODEL PREDICTIVE CONTROL OF BISHOP

NMPC is an optimization based method for feedback control of systems with nonlinear dynamics. One of its key uses is in tracking problems like Bishop's. Given a model of the state dynamics of the plant, the controller predicts the behaviour of the plant over a finite window of future time (called the *prediction horizon*). The plant behaviour depends, of course, on the control signal over that horizon. So the controller can forecast the behaviour of the plant that would result from any given control signal (or *control plan*) over the horizon. That ability is used at the heart of an optimization step which finds the control plan that minimises some cost functional. In Bishop's case, the cost functional penalises deviation from a reference path through its environment, control effort (aggressive steering commands) and proximity to local obstacles. Some portion of that optimal control plan is executed before the calculation is repeated, with the horizon shifted ahead in time. (This is why model predictive control (MPC) is sometimes called receding-horizon control.)

In the present formulation, we will be using discrete time. The horizon is internal to the calculation of the control program, so we have the luxury of a fixed time-step interval. The horizon will be divided into a monotonically ascending sequence of $N \in \mathbb{N}$ time steps. The set $T = \{t_0, \ldots, t_{N-1}\}$, with $t_n \in \mathbb{R}$ collects these time steps. Each of the N intervals is evenly timed:

$$\Delta t = \Delta t_k = t_k - t_{k-1}$$

for $k \in \{1, \ldots, N\}$.

More formally, recall the notation $\mathcal{N} = \{0, \ldots, N-1\}$ and let $(\mathcal{N}, <)$ be a restriction of the strict total order $(\mathbb{N}, <)$. For each horizon there is a straightforward homomorphism of strict total orders, a *sequencing map*:

$$\tau : (\mathcal{N}, <) \to (T, <),$$

where $(T, <)$ inherits the ordering relation from the strict total order $(\mathbb{R}, <$ $)$. The requirement that the sequencing map is a homomorphism of strict total orders is equivalent to the requirement that T, as a sequence, is monotonically increasing. The map τ gives T the structure of a sequence and provides indexing so that $\tau(n) = t_n$ for $n \in N$.

DEFINITION 6.1-1. The pair (T, τ) as specified above, constitutes a *time horizon.*

N is obtained from the cardinality of T or the domain of τ.

In C++ and other languages, this sequencing map is roughly equivalent to the indexing operator, `operator[]` [158, §26.2.3, Table 88.], or the bounds checked `at` member function [158, §26.2.3, ¶15], in supporting STL containers. But those are set-maps and do not restrict the codomain (time) to be monotonically increasing. Given a supporting container of type $F\langle U, \ldots\rangle$ the subscripting operator is in the hom-set: `operator[]` : $F\langle U, \ldots\rangle \times$ `std::size_t` $\rightarrow U$ and does not restrict the domain to the bounds of the container, so the programmer can pass subscripts not in N causing undefined behaviour[2]. The `at` member function is more careful: `at` : $F\langle U, \ldots\rangle \times N \rightarrow U$.

NOTATION 6.1-2. The field of real numbers that underlies T, we have addition and subtraction. It shall be understood that,

- $t_n + 0\,\mathrm{s} = t_n$,
- $t_n + m\Delta t = t_{n+m}$,
- $t_n - m\Delta t = t_{n-m}$, and
- $t_m - t_n = (n - m)\Delta t$,

with appropriate care taken since T is not closed under these operations. That is, T cannot carry the structure of an additive group because addition and subtraction in the underlying real field can produce values not in T.

While the state $\boldsymbol{x}$ has value for each $t \in T$, the value $\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$ comes from a sensor measurement (possible *via* estimation filter) and is provided as input to the controller. The prediction step propagates from thereon by induction over $T \setminus t_0$.

[2] Causing a segmentation fault in the best case, or unreported bad behaviour in the worst.

NOTATION 6.1-3. As a compact notation, we use subscripts to indicate indices in sequences spanning the horizon. For example, a sequence $p : T \to Z$ (arbitrary Z) has values $\{ p_n = p \circ \tau(n) \mid n \in N \}$, and $q : N \to Z$ has values $\{ q_n = q(n) \mid n \in N \}$. In C++17 where we are using STL containers to organise data over the horizon, p_n will be roughly equivalent to `p[n]`, modulo the ever present precarities of approximating real numbers with IEEE floating point.

Over a given horizon, (T, τ), the NMPC controller solves the discrete finite horizon open loop optimization,

$$\breve{u} = \arg\min_{u} \; J(\ldots, \boldsymbol{x}, u), \tag{6.1.1}$$

where,

- $\breve{u} : T \setminus \{t_{N-1}\} \to U$ is the control plan that minimises J,
- $\boldsymbol{x} : T \to X$ is a free sequence of plant states,
- $u : T \setminus \{t_{N-1}\} \to U$ is a free sequence of controls,
- $J : \cdots \times (T \to X) \times (T \to U) \to \mathbb{R}$ is a real-valued cost function depending on $\boldsymbol{x}$, $\boldsymbol{u}$ and may depend on other ancillary values, and
- $\cdots$ the ellipses leave room for those ancillary values.

The ancillary values can always be curried into J so that it may be regarded as

$$J : (T \to X) \times (T \to U) \to \mathbb{R}.$$

The optimization will additionally be constrained so that the state-trajectory obeys a model of the plant:

$$\boldsymbol{f} : X \times U \to X,$$

which propagates state through the Initial Value Problem (IVP):

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{f} \circ (\boldsymbol{x} \vartriangle \boldsymbol{u}) \circ \tau(n) = \boldsymbol{f}(\boldsymbol{x}_n, \boldsymbol{u}_n), \\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0. \end{cases} \qquad \text{for } n \in N \setminus \{ N - 1 \}. \tag{6.1.2}$$

So given an initial $\boldsymbol{x}_0$, the state trajectory is determined by $\boldsymbol{u}$.

NOTATION 6.1-4. In the context of horizon $(T,\ \tau)$, given an initial state $\boldsymbol{x}_0$ and a control sequence $\boldsymbol{u}$, let $\boldsymbol{x}^{\boldsymbol{u}}$ denote the sequence resultant from the solution of (6.1.2).

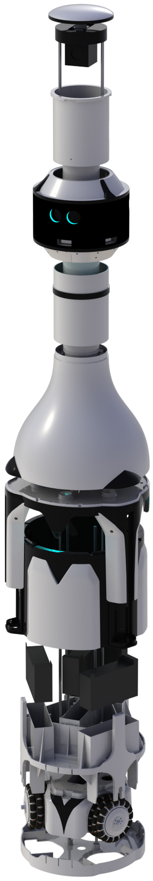
6.1.1 Vehicle Dynamics

In catnav, Bishop's state was modelled as an aggregate of position, speed, steering angle; and steering rate as the control variable. The steering angle was identified with the direction of travel (precluding the possibility of slipping). Nmpc is computationally expensive and wheel turn rates are controlled at a lower hardware layer by PID controllers; so for the sake of any given NMPC horizon, the speed is taken to be constant at value v. Speed changes could be scripted separately for smoother maneuvering, high speed cruising, and ramping up (or down) to (or from) rest. So catnav controls motion by modulating the direction of travel (by varying the steering rate) and by starting or stopping travel.

The x, y-position of the robot takes values in the Euclidean space $\mathbb{R}^2$ so that the components of the derivative of position, $\dot{x}$ and $\dot{y}$, also take values in $\mathbb{R}^2$. (That is, the tangent space at any state $p \in \mathbb{R}^2$ is $\mathsf{T}_p\mathbb{R}^2 = \mathbb{R}^2$.) So the combined rectangular position and velocity takes values in $\mathbb{R}^4$. Appending the steering angle, $\theta \in (-\pi,\ \pi]$, gives Bishop's state space: $X = \mathbb{R}^4 \times (-\pi,\ \pi]$. The steering angle is closed under modular arithmetic, so it can be given the structure of a circle, isomorphic to S^1. The tangent space over any $p \in S^1$ is $\mathbb{R}$, so the controls take value in $U = \mathbb{R}$. A state space vector $\boldsymbol{x} \in X$ is organised as a column:

$$\boldsymbol{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} & \theta \end{bmatrix}^\top$$

which express the rectangular x and y coordinates with corresponding velocity components $\dot{x}$ and $\dot{y}$, along with steering angle θ (measured the from the x-axis). The origin and orientation of the axes is determined at Bishop's startup, where it is considered to be positioned at the origin and pointing toward $+x$. The controls are steering rate: $u = \dot{\theta}$.



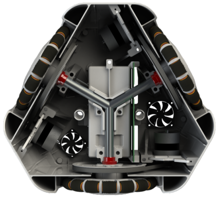


FIGURE 6.2: (top) An exploded view if Bishop's internals. (bottom) a top-down view of Bishop's triangular, 3-omniwheel base.

We discretise over the time horizon $(T,\ \tau)$ and when forecasting the state trajectory we integrate the equations of motion

$$\begin{cases} \boldsymbol{x}_{n+1} = \boldsymbol{f} \circ (\boldsymbol{x} \vartriangle u) \circ \tau\,(n) \\ \boldsymbol{x}_0 = \begin{bmatrix} x_0 & \dot{x}_0 & y_0 & \dot{y}_0 & \theta_0 \end{bmatrix}^\top, \end{cases} \qquad \text{for } \ n \in N \setminus \{\, N-1 \,\}, \tag{6.1.3}$$

with a first-order Euler scheme. Ordinary kinematics gives the transition function:

$$\boldsymbol{x}_{n+1} = \boldsymbol{f}(\boldsymbol{x}_n, u_n) = \begin{bmatrix} x_n + \dot{x}_n\,\Delta t \\ v\ \cos\theta_{n+1} \\ y_n + \dot{y}_n\,\Delta t \\ v\ \sin\theta_{n+1} \\ \theta_n + u_n\,\Delta t \end{bmatrix} \tag{6.1.4}$$

Notice that the cartesian velocity components depend on θ_{n+1}, so the angle variable must be computed prior to $\dot{x}$ and $\dot{y}$.

Also note that this is a nominal model under which perturbations are not explicitly accounted for. We assume motion without slipping.

6.1.2 Open-loop optimisation

In the continuous setting the NMPC performance index[3], the mathematical device to be optimised, is a functional of the state-space trajectory, control-space trajectory and ancillary values. In the discrete setting the functional becomes a function of the sampled discretised/sampled state and control trajectories. In our case, given the context of the horizon (T, τ), we choose a performance *cost* function,

$$J(\boldsymbol{x}^{\mathrm{ref}},\ \boldsymbol{x}^{\mathrm{tgt}},\ \Phi,\ \boldsymbol{x},\ u) = \phi(\boldsymbol{x}^{\mathrm{tgt}},\ \boldsymbol{x}_{N-1}) + \sum_{n=0}^{N-2} L(\boldsymbol{x}_n^{\mathrm{ref}},\ \Phi,\ \boldsymbol{x}_n,\ u_n), \tag{6.1.5}$$

where,

- $u:\ T \to U$ is the sequence of controls,

[3] The cost functional to be minimised, reward functional to be maximised. The two are equivalent and the choice between them is a matter of personal taste.

- $\boldsymbol{x} : T \to X$ is a sequence of plant states,
- $\boldsymbol{x}^{\mathrm{ref}} : T \to X^{\mathrm{ref}}$ where X^{ref} is the subspace of X covering the Euclidean map: $\mathbb{R} \times \{0\} \times \mathbb{R} \times \{0\} \times \{0\}$; and $\boldsymbol{x}^{\mathrm{ref}}$ the tracking reference path sampled at intervals of $v\,\Delta t$ to align with the spatial sampling of $\boldsymbol{x}^{u}$,
- $\boldsymbol{x}^{\mathrm{tgt}} \in X^{\mathrm{ref}}$ is the ultimate target, the terminus of the path of which $\boldsymbol{x}^{\mathrm{ref}}$ is a segment,
- $L : X^{\mathrm{ref}} \times X \times U \to \mathbb{R}$ is called the *running cost*,
- $\phi : X^{\mathrm{ref}} \times X \to \mathbb{R}$ is called the terminal cost, and
- $\Phi : \mathbb{R}^2 \to \mathbb{R}$ is a scalar potential field of soft constraints—regions of the cartesian map to be avoided.

Catnav uses quadratic cost terms,

$$\phi(\boldsymbol{x}^{\mathrm{tgt}}, \boldsymbol{x}_{N-1}) = (\boldsymbol{x}_{N-1} - \boldsymbol{x}^{\mathrm{tgt}})^{\top} \boldsymbol{Q}_0 (\boldsymbol{x}_{N-1} - \boldsymbol{x}^{\mathrm{tgt}})$$

and

$$L(\boldsymbol{x}_n^{\mathrm{ref}}, \Phi, \boldsymbol{x}_n, u_n) = R u_n^2 + \tilde{\boldsymbol{x}}_n^{\top} \boldsymbol{Q} \tilde{\boldsymbol{x}}_n + \Phi(x_n, y_n)$$

where,

- $\tilde{\boldsymbol{x}}_k$ is the tracking error, $\tilde{\boldsymbol{x}}_k = \boldsymbol{x}_k - \boldsymbol{x}_k^{\mathrm{ref}}$.
- $R \in \mathbb{R}$ is a positive scalar weighting coefficient, and $\boldsymbol{Q}_0, \boldsymbol{Q} \in \mathbb{R}^{5\times 5}$ are diagonal matrices of positive weighting coefficients to shift the balance of priority in the optimisation step.

Quadratic cost functions provide a convenience in their simple geometry so we do not need to check Hessians for positive-definity. The intention is to use a simple gradient decent algorithm to solve the optimisation. Of course, Φ is not quadratic, so attention must be paid to complexities it might introduce.

6.1-5 To simplify the forthcoming derivation, assume that for each horizon, the values $\boldsymbol{x}^{\mathrm{ref}}$, Φ and $\boldsymbol{x}^{\mathrm{tgt}}$ have been curried into J so that it may be regarded as a function $J(\boldsymbol{x}, u)$. Likewise for $\phi(\boldsymbol{x}_{N-1})$ and $L(\boldsymbol{x}_n, u_n)$, which also have their ancillary dependencies curried in.

In the optimisation, the terminal cost $\phi(\boldsymbol{x}_{N-1})$ biases the curvature of $\boldsymbol{x}^u$ toward $\boldsymbol{x}^{\text{tgt}}$. This is important because in the context of a finite horizon (limiting foresight) and unknown Φ, the radial symmetry of the quadratic terms can easily lead to limit cycles or otherwise sub-optimal progress toward a global target. Computational expense grows quickly with N, so including ϕ in J is a frugal way to inject some global perspective without increasing N.

The weighting scalar R deserves a brief note because Bishop's drive design (Figure 6.2) is capable of instantaneous direction changes: it is anhalonomic. The kinematics model couples the direction of travel to the orientation of the robot and forces steering instead of instantaneous orientation changes. The magnitude of R adds turn radius into the cost balance. The value of R can even be a function of velocity so that narrower turns can be executed at lower speeds.

Following similar derivations in [89, 98, 119] which seem to have originated in [35, §2.2], we solve the minimization problem (6.1.1) by discretely solving the Euler-Lagrange differential equations to obtain a variational expansion obtain gradient elements. These may all be regarded as a discritisation of the optimal control problem based on Pontryagin's maximum principle [94, §III.5]. These are used in a gradient descent scheme to solve the open-loop cost minimisation. We choose boundary conditions with t_{N-1} fixed and $\boldsymbol{x}_{N-1}$ free [94, §5.1, CASE I.]. So the optimisation problem at hand (considering **6.1-5**) given a horizon $(T,\ \tau)$, is

$$\breve{u} = \arg\min_{u}\ J(\boldsymbol{x},\, u) \quad \text{s.t.} \quad \boldsymbol{x} = \boldsymbol{x}^u.$$

The constraint $\boldsymbol{x} = \boldsymbol{x}^u$ is equivalent to $\boldsymbol{x}_{n+1} - \boldsymbol{f}(\boldsymbol{x}_n, u_n) = 0$. To enforce them the cost measure (6.1.5) is augmented with a set of Lagrange multi-

Trivia: because he spoke Russian, after the soviet war, Lotfi Zadeh was one of the first American engineers to become familiar and to popularise *Pontryagin's maximum principle.* See [76] for the fact, and see [3] if *you* read Russian.

pliers $\boldsymbol{\lambda} : N \setminus \{ 0 \} \to \mathbb{R}^5$ with one dimension for each in X:

$$\begin{aligned} J^{\text{aug}}(\boldsymbol{x}, u) &= \phi(\boldsymbol{x}_{N-1}) \\ &\qquad + \sum_{n=0}^{N-2} \Big(L(\boldsymbol{x}_n^{\text{ref}}, \boldsymbol{x}_n, u_n) + \boldsymbol{\lambda}_{n+1}^{\top} (\boldsymbol{f}(\boldsymbol{x}_n, u_n) - \boldsymbol{x}_{n+1}) \Big). \end{aligned}$$

There is no value of $\boldsymbol{\lambda}$ associated with initial state $\boldsymbol{x}_0$ since that is given at the outset as part of the IVP. The initial state is the current or last known state of the robot and is provided as input to the algorithm. As it is fixed at all parts of the open-loop calculation it makes no sense to constrain it.

Next, introduce the control Hamiltonian sequence $H : X \times U \to \mathbb{R}$ as

$$H_n(\boldsymbol{x}_n, u_n) = L(\boldsymbol{x}_n, u_n) + \boldsymbol{\lambda}_{k+1}^{\top} \boldsymbol{f}(\boldsymbol{x}_n, u_n). \tag{6.1.6}$$

Rewriting J^{aug} in terms of H and extracting the $n = 0$ term from the sum gives

$$J^{\text{aug}} = \phi(\boldsymbol{x}_{N-1}) + \boldsymbol{\lambda}_{N-1}^{\top} + H_0 + \sum_{k=1}^{N-2} \big(H_n - \boldsymbol{\lambda}_n^{\top} \boldsymbol{x}_n \big).$$

In order to obtain the gradient of J^{aug}, we take its differential

$$\begin{aligned} \mathrm{d}J^{\text{aug}} = \left(\frac{\partial \phi(\boldsymbol{x}_{N-1})}{\partial \boldsymbol{x}_{N-1}} - \boldsymbol{\lambda}_{N-1}^{\top} \boldsymbol{x}_{N-1} \right) \mathrm{d}\boldsymbol{x}_{N-1} + \frac{\partial H_0}{\partial \boldsymbol{x}_0} \, \mathrm{d}\boldsymbol{x}_0 + \frac{\partial H_0}{\partial u_0} \, \mathrm{d}u_0 \\ + \sum_{n=1}^{N-2} \left(\left(\frac{\partial H_n}{\partial \boldsymbol{x}_n} - \boldsymbol{\lambda}_n^{\top} \right) \mathrm{d}\boldsymbol{x}_n + \frac{\partial H_n}{\partial u_n} \, \mathrm{d}u_n \right), \end{aligned} \tag{6.1.7}$$

and concern ourselves with its variation with respect to u_n. Since $\boldsymbol{x}$ is ultimately determined by u and u is the target of optimisation, we can choose the Lagrange multipliers to eliminate terms of $\mathrm{d}\boldsymbol{x}_n$,

$$\boldsymbol{\lambda}_{N-1}^{\top} = \frac{\partial \phi}{\partial \boldsymbol{x}_{N-1}} \quad \text{and} \quad \boldsymbol{\lambda}_n^{\top} = \frac{\partial H_n}{\partial \boldsymbol{x}_n} + \frac{\partial H_n}{\partial \tilde{\boldsymbol{x}}_n} \frac{\partial \tilde{\boldsymbol{x}}_n}{\partial \boldsymbol{x}_n}$$

or more explicitly

$$\boldsymbol{\lambda}_{N-1}^{\top} = \tilde{\boldsymbol{x}}_{N-1}^{\top} \, Q_0 \, \frac{\partial \tilde{\boldsymbol{x}}_{N-1}}{\partial \boldsymbol{x}_{N-1}}, \tag{6.1.8}$$

and

$$\boldsymbol{\lambda}_n^\top = \tilde{\boldsymbol{x}}_n^\top Q \frac{\mathrm{d}\tilde{\boldsymbol{x}}_n}{\mathrm{d}\boldsymbol{x}_n} + \boldsymbol{\lambda}_{n+1}^\top \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}_n} + \frac{\partial \Phi(x_n, y_n)}{\partial \boldsymbol{x}_n}, \tag{6.1.9}$$

where

$$\frac{\partial \Phi(x_n, y_n)}{\partial \boldsymbol{x}_n} = \begin{bmatrix} \frac{\partial \Phi(x_n, y_n)}{\partial x_n} \\ 0 \\ \frac{\partial \Phi(x_n, y_n)}{\partial y_n} \\ 0 \\ 0 \end{bmatrix}. \tag{6.1.10}$$

The constant factors from all of the derivatives in $\mathrm{d}J^{\mathrm{aug}}$ and $\boldsymbol{\lambda}$ are absorbed into the weighting parameters. The choice of Lagrange multipliers consequently reduces (6.1.7) to

$$\mathrm{d}J^{\mathrm{aug}} = \sum_{k=1}^{N-1} \frac{\partial H_n}{\partial u_n} \mathrm{d}u_n + \boldsymbol{\lambda}_0^\top \mathrm{d}\boldsymbol{x}_0.$$

Of course, $\mathrm{d}\boldsymbol{x}_0 = 0$ since it is fixed. This leaves

$$\mathrm{d}J^{\mathrm{aug}} = \frac{\partial H_n}{\partial u_n} \mathrm{d}u_n,$$

thus $\partial H_n / \partial u_n$ is the gradient of J^{aug} with respect to the sequence u. By expanding and differentiating (6.1.6) we obtain an explicit formula for the gradient elements

$$\frac{\partial H_n}{\partial u_n} = R u_n + \boldsymbol{\lambda}_{k+1}^\top \frac{\partial \boldsymbol{f}}{\partial u_n}. \tag{6.1.11}$$

6.1-6 GRADIENT DESCENT A basic gradient descent algorithm that includes steps for forecasting and computation of the Lagrange multiplier sequence,

1. $\boldsymbol{x}_0$ and some initial guess for $\breve{u}$ is created,
2. $\boldsymbol{x}$ is forecast from $\boldsymbol{x}_0$ and u by iteration of (6.1.3),
3. $\boldsymbol{\lambda}_{N-1}$ is computed from (6.1.8),
4. $\boldsymbol{\lambda}_k$ is computed from (6.1.9),
5. gradient elements $\partial H_n / \partial u_n$ are computed from (6.1.11),

6. the guess for $\breve{u}$ is updated by elementwise subtracting $\partial H_n/\partial u_n$ (multiplied by a step-factor) from u_n, and
7. the above steps are repeated against the updated estimate of $\breve{u}$ until the norm of the gradient elements (with the sequence treated as a vector) is sufficiently small, indicating proximity to a minimum of J.

This gradient descent, compared to more advanced methods, sacrifices elegance and performance for simplicity. We had always planned to replace it with more sophisticated technology, but it performed well enough to carry the research to the end of the grant.

6.1.3 The NMPC algorithm

Catnav's NMPC algebra is templated on a static value for N. In the simplest case, when the NMPC algebra receives an input of $\boldsymbol{x}_0$, $\boldsymbol{x}^{\text{ref}}$, $\boldsymbol{x}^{\text{tgt}}$ and Φ, the open-loop optimisation step is begun using the algorithm outlined in **6.1-6**, yielding an estimate of $\breve{u}$. The first element of that estimate is executed and the process is repeated when the next input package arrives.

Section 6.3 will provide finer details, but this is the basic NMPC algorithm.

6.2 OPERATING ENVIRONMENT

SENTRYnet teams had a standardised running environment facilitated by Docker. The image was based on Ubuntu 18.04 and ROS melodic (ROS-1). This, even at the time, was slightly out of date but was assumed to be more reliable and familiar to the research groups. The image included the following packages:

- `python-rosdep`,
- `ros-melodic-desktop-full`,
- `ros-melodic-dingo-control`,
- `ros-melodic-dingo-description`,

- ros-melodic-effort-controllers,
- ros-melodic-global-planner,
- ros-melodic-move-base,
- ros-melodic-ridgeback-gazebo-plugins,
- ros-melodic-slam-toolbox,
- ros-melodic-teb-local-planner, and
- ros-melodic-tf2-sensor-ms.

OpenCV is also brought in as a dependency.

Catnav additionally requires the packages

- ros-melodic-rxros, and
- ros-melodic-rxros-tf

which builds on RxCpp to provide tooling, such as observables/observers for ROS topics and the tf system.

6.2.1 Integration with move_base

The standard ROS move_base package includes a "local_planner" to accomplish the navigation task. It takes the path provided by the global_planner while dealing with fine-grained details of execution and the local_costmap Catnav replaces that component: it takes the occupancy grid (local_costmap) and global path and interacts directly with the mobile base. The dataflow among components is illustrated in Figure 6.3

Catnav uses the local_costmap, which is simply an occupancy grid centred on the robot, to obtain Φ, (see §6.4). It uses the path from the plan from the global_planner to compute $\boldsymbol{x}^{\mathrm{ref}}$. Additionally, catnav obtains coordinate transformations broadcast on the tf/tf2 system.

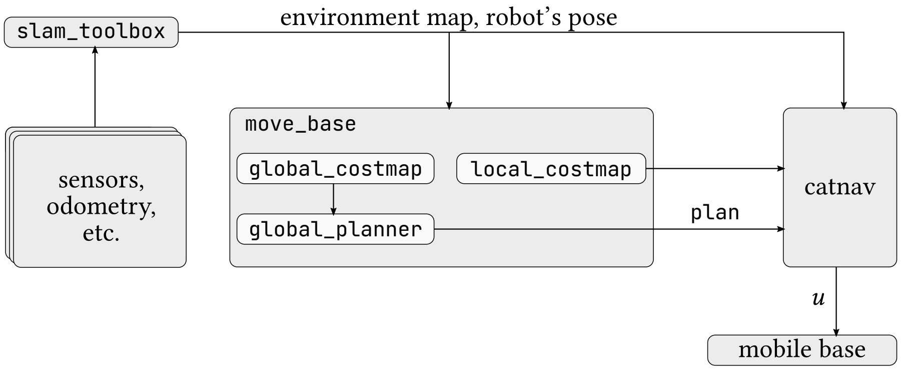


FIGURE 6.3: Component/information-flow diagram for selected components of Bishop's ROS-based navigation stack—including catnav.

6.3 IMPLEMENTATION

6.3.1 The Plant & Controller State Types

The plant state type is dubbed `WorldState` in the code, for largely historical reasons [4]. It contains a time-stamp, the pose of the robot, the global target, the desired cruising speed, the path from the global planner and a function modeling $\partial\Phi(x, y)/\partial \boldsymbol{x} : \mathbb{R}^2 \to \mathbb{R}^2$ from (6.1.10):

```
template <typename Clock = std::chrono::steady_clock>
struct WorldState {
  typename Clock::time_point time;
  Pose pose;
  std::optional<Target> target;
  double cruise_speed;
  std::vector<XY> global_plan;
  std::function<std::optional<XY>(XY)> Dphi
                          = [](XY) { return std::nullopt; };
};
```

The `XY` type (LST. C.3.1.) is a simple type modeling a 2-dimensional (2D) Euclidean vector as a pair of `doubles`. It has basic equality and linear-algebraic operations and can be converted to a `std::tuple` for structured binding.

[4] Confer the term *world interface* from the PID example from the end of CH. 5.

6.3-1 A `Pose` is the position and orientation of the robot

```
struct Pose {
  XY position = {0, 0};
  double yaw = 0.;
};
```

where `yaw` is measured counter-clockwise (CCW) from the x-axis. A `Target` is a pose with some extra tooling for proximity detection

```
struct Target {
  BallTolerable<XY> position;
  BallTolerable<double> yaw;
};
```

Because we must determine if a target has been reached by comparing the current pose of the robot to the target, we must be very particular about what we mean by "reached". The errors in measurement, floating-point calculation and the mechanics of the robot all contribute to the brute fact that we can never expect a strict equality of the pose of the robot with an arbitrary target. The `BallTolerable` type expects difference and absolute-value operations so that we can see if one value is within a ball swept by a radius

```
template <typename T>
struct BallTolerable {
  T value;
  double tolerance;
  bool is_tolerable(T& x) { return abs(x - value) <= tolerance; }
};
```

So a `Target` is a `Pose` where position and yaw have a radius tolerance for the inequality comparison `is_tolerable`. Note that a `BallTolerable` is a slight variation[5] of what Kálmán calls a "tolerance space" in [20, Pt. I, §6.4] (the definition is thereby attributed to Zeeman). In the case of an `XY`, the function `abs` returns the square-norm (ℓ^2-norm) of the vector.

[5] Zeeman's description (*via.* Kálmán) is of a pair (X, ξ) of a set X and reflexive symmetric relation ξ on X where $(x, y) \in \xi$ just in case $x \in X$ and $y \in X$ are tolerably close, by some measure.

Note that Dphi is a std::function and not Hom as in previous chapters. The demo code libraries of the previous chapters was not included in catnav.

The state of the NMPC controller is large and mostly comprised of values for the horizon and open-loop optimisation and templates on the horizon size, N.

```
template <std::size_t N, typename Clock = std::chrono::steady_clock>
struct NMPCState {
  std::chrono::time_point<Clock> time;
  std::chrono::duration<double> dt =
                            std::chrono::duration<double>(1. / 3);
  // State Vector:
  std::array<double, N> x = {{0}};
  std::array<double, N> y = {{0}};
  std::array<double, N> th = {{0}};
  std::array<double, N> Dx = {{0}};
  std::array<double, N> Dy = {{0}};
  std::array<double, N - 1> Dth = {{0}};
  std::array<double, N - 1> v = {{0}};
  // Tracking reference and resulting error $(x, y) - (x^{\mathrm{ref}}, y^{\mathrm{ref}})$:
  std::array<double, N - 1> xref = {{0}};
  std::array<double, N - 1> yref = {{0}};
  std::array<double, N> ex = {{0}};
  std::array<double, N> ey = {{0}};
  // Obstacle potential gradient for all but starting point of
  //   the trajectory:
  std::array<double, N - 1> Dphi_x = {{0}};
  std::array<double, N - 1> Dphi_y = {{0}};
  // Lagrange Multipliers:
  std::array<double, N - 1> px = {{0}};
  std::array<double, N - 1> py = {{0}};
  std::array<double, N - 1> pDx = {{0}};
  std::array<double, N - 1> pDy = {{0}};
  std::array<double, N - 1> pth = {{0}};
  // Optimisation gradients:
  std::array<double, N - 1> grad = {{0}};
  double curGradNorm = 0;
  double prvGradNorm = 0;
  // Coefficients
  double R = 0,  // Control effort penalty
      Q = 0,     // Tracking error penalty
      Q0 = 0;    // Terminal error penalty
  // Collection of obstacles used to compute (DPhiX, DPhiY).
  InfoFlag infoFlag = InfoFlag::OK;
};
```

Instead of storing the horizon (T, τ), we simply have the horizon size N, starting time, `time`, and the (constant) sampling interval Δt, `dt`, and (T, τ) can be computed from them. The state vector

$$\begin{bmatrix} x & \dot{x} & y & \dot{y} & \theta \end{bmatrix}^\top$$

for the horizon are stored as individual arrays of `doubles`: `x`, `Dx`, `y`, `Dy` and `th`.

The position error $\tilde{\boldsymbol{x}}$ is stored as `ex` and `ey` and will be calculated from the tracking references `xref` and `yref` storing data for $\boldsymbol{x}^{\text{ref}}$.

The values `Dphi_x/y` are populated with the gradient of Φ at (x_n, y_n).

The Lagrange multipliers correspond to each dimension of the state vector. In the derivation, they were denoted by $\boldsymbol{\lambda}$. C++ does not allow names to start with Greek letters, so I chose `p`, as we physicists tend to denote them in Lagrangian mechanics due to their relationship with momentum.

The gradient elements $\partial H_n / \partial u_n$ are stored in `grad`. We also store two values (`current` and `previous`) of the square-norm of the array `grad`, treated as an $N - 1$-dimensional vector.

The `InfoFlag` is a value from the enumeration

```
enum class InfoFlag {
  OK,
  TargetReached,
  NoTarget,
  MissingWorldData,
  STOP,
  Null
};
```

which can be used to log, make choices or handle errors during the state transition calculations in the NMPC-algebra to be described in the next subsection.

6.3.2 The Nmpc Algebra

The NMPC algebra is, superficially, fairly simple

```
template <std::size_t N, typename Clock>
constexpr auto nmpc_algebra(
      NMPCState<N, Clock> c,
          const WorldState<Clock>& w) -> NMPCState<N, Clock> {
  if (!w.target) {
    c.infoFlag = InfoFlag::TargetReached;
    return dtl::with_init_from_world(c, w);
  }
  if (w.time - c.time ⩽ std::chrono::seconds{0}) {
    c.infoFlag = InfoFlag::STOP;
    return c;
  }
  if (l2norm(w.target.value().position.value - w.pose.position)
      < w.target.value().position.tolerance) {
    c.infoFlag = InfoFlag::TargetReached;
    return dtl::with_init_from_world(c, w);
  }

  c.infoFlag = InfoFlag::OK;
  return dtl::sd_optimise(
      dtl::plan_reference(dtl::with_init_from_world(c, w), w), w);
}
```

The bulk is logic regarding the `infoFlag` and `WorldState` integrity; and then the return line is a composite expression. With the namespaces removed

```
sd_optimise(plan_reference(with_init_from_world(c, w), w), w);
```

Each of these functions is type

$$\texttt{NMPCState}\langle\texttt{N},\texttt{Clock}\rangle \times \texttt{WorldState}\langle\texttt{Clock}\rangle \to \texttt{NMPCState}\langle\texttt{N},\texttt{Clock}\rangle,$$

so they compose along the first argument. Each function uses data from the world, `w`, and the contents of the controller state `c`, to make an updated controller state.

The routine `with_init_from_world` is *impure*. It the world state to set $\boldsymbol{x}_0$ and v:

```
template <std::size_t N, typename Clock>
constexpr NMPCState<N, Clock> with_init_from_world(
    NMPCState<N, Clock> c, const WorldState<Clock>& w) noexcept {
  c.time = Clock::now();
  c.x[0] = w.pose.position.x;
  c.y[0] = w.pose.position.y;
  c.th[0] = w.pose.yaw;
  for (auto& each : c.v) each = w.cruise_speed;
  c.Dx[0] = c.v[0] * std::cos(c.th[0]);
  c.Dy[0] = c.v[0] * std::sin(c.th[0]);

  return c;
}
```

It is impure because it calls out to `Clock::now`. It can be made pure if the clock were to come in as a separate source, but this adds needless complexity.

The function `plan_reference`, as its name suggests, populates `xref` and `yref` in `c`. Recall that the `global_plan` is included in the `WorldState` as a `std::vector⟨XY⟩`. It is a discretely sampled set of map coordinates. It is not, in general, sampled appropriately for `xref` and `yref` which need to be spaced $v\,\Delta t$ appart. This function resamples the `global_path` by linear interpolation populating `xref`/`yref`. While it is a simple calculation, it is understandably verbose, and so not presented here.

The `sd_optimise` function implements the algorithm outlined in **6.1-6**

```
template <std::size_t N, typename Clock>
constexpr NMPCState<N, Clock> sd_optimise(
    NMPCState<N, Clock> c, const WorldState<Clock>& w) noexcept {
  const auto step = [&w](auto c) noexcept {
    return descend(lagrange_gradient(forecast(c, w)));
  };

  constexpr auto gradNorm_outside = [](double epsilon) {
    return [epsilon](auto& c) noexcept { return c.curGradNorm >= epsilon; };
  };

  const double tol = ((c.R + c.Q) * (N - 1) + c.Q0) / N / c.dt.count() / 100;

  return iterate_while(step, gradNorm_outside(tol), c);
}
```

The `tolerance` calculation is somewhat arbitrary and is based on the idea that there is a certain amount of sub-optimality per horizon step. The `iterate_while` function does `step(c)` while `gradNorm_outside(tol)`. The interesting part is the function `step`, which returns

```
  descend(lagrange_gradient(forecast(c, w)))
```

The `forecast` function computes $\boldsymbol{x}^u$ and populates `x`/`yref` and `Dphi_x`/`y`:

```
template <std::size_t N, typename Clock>
constexpr NMPCState<N, Clock> forecast(NMPCState<N, Clock> c,
                                      const WorldState<Clock>& w) noexcept {
  for (std::size_t k = 1; k < N; ++k) {
    c.th[k] = fma(c.Dth[k - 1], c.dt.count(), c.th[k - 1]);
    c.x[k] = fma(c.Dx[k - 1], c.dt.count(), c.x[k - 1]);
    c.Dx[k] = c.v[k - 1] * std::cos(c.th[k]);
    c.y[k] = fma(c.Dy[k - 1], c.dt.count(), c.y[k - 1]);
    c.Dy[k] = c.v[k - 1] * std::sin(c.th[k]);

    c.ex[k - 1] = c.x[k] - c.xref[k - 1];
    c.ey[k - 1] = c.y[k] - c.yref[k - 1];

    if (auto Dphi = w.Dphi({c.x[k], c.y[k]})) {
      std::tie(c.Dphi_x[k - 1], c.Dphi_y[k - 1]) = Dphi.value();
    } else {
      c.infoFlag = InfoFlag::MissingWorldData;
      c.Dphi_x[k - 1] = 0.;
      c.Dphi_y[k - 1] = 0.;
    }
  }

  if (w.target) {
    c.ex[N - 1] = c.x[N - 1] - w.target->position.value.x;
    c.ey[N - 1] = c.y[N - 1] - w.target->position.value.y;
  } else {
    c.ex[N - 1] = 0;
    c.ey[N - 1] = 0;
  }

  return c;
}
```

The bulk of the calculation is iteration of (6.1.4). You can see an error flag set if, for whatever reason, `w.Dphi` returns a `std::nullopt`. Before the return statement, you can see that if there is a `w.target`, then $\tilde{\boldsymbol{x}}_{N-1}$ is set relative to the global target for the calculation of the terminal cost $\phi(\boldsymbol{x}_{N-1})$.

The `lagrange_gradient` function populates the Lagrange multipliers by (6.1.8) and (6.1.9) and `c.grad` by (6.1.11).

```
constexpr NMPCState<N, Clock> lagrange_gradient(
    NMPCState<N, Clock> c) noexcept {
  static_assert(N >= 3, "Gradient calculation requires N >= 3.");

  // Starting with (6.1.8):
  c.prvGradNorm = c.curGradNorm;
  c.px[N - 2] = c.Q0 * c.ex[N - 2];
  c.py[N - 2] = c.Q0 * c.ey[N - 2];
  c.pDx[N - 2] = 0;
  c.pDy[N - 2] = 0;
  c.pth[N - 2] = 0;
  c.grad[N - 2] = c.pth[N - 2] * c.dt.count() + c.Dth[N - 2] * c.R;
  c.curGradNorm = c.grad[N - 2] * c.grad[N - 2];

  for (unsigned int k = N - 3; k != UINT_MAX; k--) {
    // Now use (6.1.9):
    c.px[k] = c.px[k + 1] + c.Q * c.ex[k] - c.Dphi_x[k];
    c.pDx[k] = c.px[k + 1] * c.dt.count();
    c.py[k] = c.py[k + 1] + c.Q * c.ey[k] - c.Dphi_y[k];
    c.pDy[k] = c.py[k + 1] * c.dt.count();
    c.pth[k] =
      c.pth[k + 1] + c.pDy[k + 1] * c.v[k] * std::cos(c.th[k])
        - c.pDx[k + 1] * c.v[k] * std::sin(c.th[k]);

    // Compute the gradient elements ∂H/∂u_k by (6.1.11):
    c.grad[k] = c.R * c.Dth[k] + c.pth[k + 1] * c.dt.count();
    c.curGradNorm += c.grad[k] * c.grad[k];
  }
  c.curGradNorm = std::sqrt(c.curGradNorm);

  return c;
}
```

Finally, the `descend` function subtracts a fraction of `c.grad` from `c.Dth` elementwise

```
template <std::size_t N, typename Clock>
constexpr NMPCState<N, Clock> descend(NMPCState<N, Clock> c) noexcept {
  constexpr double sdStepFactor = 0.01;
  for (std::size_t i = 0; i < N - 1; ++i) {
    c.Dth[i] -= sdStepFactor * c.grad[i];
  }

  return c;
}
```

Once the `sd_optimise` loop breaks and the `nmpc_algebra` returns, the controller state is left with `c.Dth` optimal to within tolerance, and the rest of the horizon values correspondingly set.

In the next section, the readout after the scan only needs to extract `Dth[0]` from the controller state and broadcast it over a ROS topic to be received and executed by the mobile base.

6.3.3 The `main` Function of the Ros Node

The `main` function of the program begins as any ROS node

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "catnav");
```

and it will end with `ros::spin`.

6.3-2 THE CONTROL EXPRESSION The controller includes the same RxCpp pipeline expression derived in the last chapter

```
const auto controls =
  global_plan
  | combine_latest(
      combine_reference_and_feedback,
      odometry,
      obstacle_Dphi)
  | scan(c0, nmpc_algebra<N, Clock>);
```

When we close the loop by subscription to `controls`, you can see the first use of RxROS, to publish to a ROS topic

```
const auto closed_loop =
  controls
    | map(extract_vel_cmd_from_cstate)
    | rxros::operators::
        publish_to_topic<geometry_msgs::Twist>("/cmd_vel");
```

The string argument to `publish_to_topic` is the topic identifier.

The observables `global_plan`, `odometry` and `obstacle_Dphi` come in from ROS topics. In ROS, each topic is strongly typed and has a Unix path-like identifier.

The `global_plan` is defined as:

```
const auto global_plan = rxros::observable::from_topic<nav_msgs::Path>(
    "/move_base/GlobalPlanner/plan");
```

The string argument to `from_topic` is the topic identifier. The …`/plan` topic's type is `Path` from ROS's `nav_msgs`[6]. The global plan is used straightforwardly, but the odometry and obstacle gradient have additional processing.

The expression for `obstacle_Dphi` builds on an observable topic published by `move_base`

```
const auto obstacle_Dphi =
    rxros::observable::from_topic<nav_msgs::OccupancyGrid>(
        "/move_base/local_costmap/costmap")  // in `map` frame
    | map(
        [](nav_msgs::OccupancyGrid& m) {
          return GradientOfGrid(m);
        }
      );
```

The topic type of …`/costmap` is `nav_msgs::OccupancyGrid`[7]. Over the topic's observable, we map the object constructor for the catnav type `GradientOfGrid`. The details of this type will be expounded in §6.4. So `obstacle_Dphi`'s value type is `GradientOfGrid`.

The `odometry` observable is intricate because it involves timing

```
const auto odometry =
    rxros::observable::from_topic<nav_msgs::Odometry>(
        "/odom")  // in `odom` frame
    | debounce(10ms)
    | combine_latest(
        [](auto o, auto tr) {
```

[6] See `http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Path.html`

[7] See `https://docs.ros.org/en/noetic/api/nav_msgs/html/msg/OccupancyGrid.html`

```
     const Pose p = {
       {
         o.pose.pose.position.x,
         o.pose.pose.position.y
       }
       , tf2::getYaw(o.pose.pose.orientation)
     };

     return transform_pose(tr, p);
    },
    odom_to_map_transform
   );
```

The `/odom` topic's messages (of type `nav_msgs::Odometry`[8]) are debounced to a 10 ms interval, meaning that data arriving through the topic are ignored except every 10 ms, at which point a new value is allowed into the observable stream. This emulates a 100 Hz update frequency by rejecting intermediate values. This is done because the odometry updates rapidly compared to the other observables and we do not want to overdrive the control calculation. Next, the poses undergo a coordinate transform facilitated by the `combine_latest` combinator, which allows us to combine the odometry stream with a stream of value type `tf::StampedTransform`. The source of the coordinate transform is ROS's `tf` package, which is also exposed by RxROS

```
const auto odom_to_map_transform =
  rxros::observable::from_transform("odom", "map").retry();
```

The `transform_pose` function in part of catnav and is a straightforward C++ spelling of the linear algebra involved. Its signature is

$$\texttt{transform\_pose} : \texttt{tf::StampedTransform} \times \texttt{Pose} \to \texttt{Pose},$$

so the value type of the stream is catnav's `Pose` type (from **6.3-1**). The Rx standard `retry` combinator[9] overrides the `on_error` contingency of Rx observables forcing a re-subscription instead. This is necessary because, as

[8] See `https://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html`

[9] See `https://reactivex.io/documentation/operators/retry.html` for details.

the ROS system is coming online, catnav is often started before this transform becomes available.

* * *

Now knowing the structure of the input observables, we can understand how `combine_reference_and_feedback` takes the global tracking reference path `p`, odometry estimated pose `o`, and the `GradientOfGrid` modeling the gradient of the obstacle potential field, and packs these into a `WorldState`, `w`

```
auto combine_reference_and_feedback(
    nav_msgs::Path p,
    Pose o,
    std::function<std::optional<XY>(XY)> f) -> PState {
  PState w;
  w.time = std::chrono::steady_clock::now();

  w.pose = o;

  if (p.poses.size() > 0) {
    w.target = {
      {
        {
          p.poses.back().pose.position.x,
          p.poses.back().pose.position.y
        }
        , 0.25
      },
      {
        tf2::getYaw(p.poses.back().pose.orientation),
        5. * M_PI / 180.
      }
    };

    w.global_plan.reserve(p.poses.size());
    for (auto& plan_pose : p.poses) {
      w.global_plan.push_back(
          {plan_pose.pose.position.x, plan_pose.pose.position.y});
    }

    w.cruise_speed = 0.25;  // m/s
  } else {
    w.target = {};
    w.global_plan = {};
  }

  w.Dphi = f;

  return w;
}
```

The world state is time-stamped and the pose from the odometry reading `o` sets `w.pose`. If a non-empty tracking reference path is provided by the global planner then data is organised into `w` as follows:

- The terminus of `p` is used to set `w.target`, and then all of `p`'s map locations are copied into `w.global_plan`.
- The speed of the nominal (cruising) speed of the robot was planned to come as input from another SENTRYnet component project, but that never came to be. So `w.cruise_speed` is set to a safe but arbitrary value.

Otherwise, if no global plan is available, then the target and global plan in `w` are empty/nullopt. The potential gradient `w.Dphi` is straightforwardly assigned the `GradientOfGrid` object coming in from `obstacle_Dphi`.

Finally, the scan in the control expression must be seeded by an initial controller state. This is where the coefficients and horizon are set

```
constexpr CState c0 = []() {
  CState c;
  c.infoFlag = InfoFlag::NoTarget;
  for (auto& each : c.v) each = 0.25;
  c.dt = 1.s / 5;
  for (auto& each : c.Dth) each = 0.5;
  c.Q0 = 0;
  c.Q = 8;
  c.R = 0.25;

  return c;
}();
```

The main function ends with the usual `ros::spin`:

```
  ros::spin();

  return EXIT_SUCCESS;
}
```

Some details were kept out of the description of the main function, but nearly all of the complexity comes from conversions dealing with various message types from ROS topics. The core of catnav is in the simple control expression, 6.3-2 and in the NMPC-algebra, §6.3.2. One important mechanism remains: the gradient of the obstacle potential encompasses in the `GradientOfGrid` type, covered in the next section.

6.4 OBSTACLE AVOIDANCE

Equation (6.1.5) for the objective function J has a running cost that includes a scalar potential field $\varPhi : \mathbb{R}^2 \to \mathbb{R}$ that causes J to accumulate value based on the x and y coordinates along the state trajectory $\boldsymbol{x}$. Minimization of J requires only the gradient

$$\frac{\partial \varPhi(x_n, y_n)}{\partial \boldsymbol{x}_n} = \begin{bmatrix} \frac{\partial \varPhi(x_n, y_n)}{\partial x_n} \\ 0 \\ \frac{\partial \varPhi(x_n, y_n)}{\partial y_n} \\ 0 \\ 0 \end{bmatrix}.$$

This potential field is the mechanism by which obstacles in the map (provided by `move_base`) are avoided. That is to say that an obstacle emits a potential field that is large in proximity to the obstacle and declines with distance. Minimization of J encourages the robot to accumulate as little of this potential as possible over the course $\boldsymbol{x}$ by keeping distance from the obstacles.

In the previous section, the expression for `obstacle_Dphi` was

```
const auto obstacle_Dphi =
    rxros::observable::from_topic<nav_msgs::OccupancyGrid>(
        "/move_base/local_costmap/costmap")  // in `map` frame
    | map(
        [](nav_msgs::OccupancyGrid& m) {
          return GradientOfGrid(m);
        }
      );
```

where topic type of the costmap is `nav_msgs::OccupancyGrid`. In practice, the costmap is in fact a simple occupancy grid. The grid's resolution is included in the `nav_msgs::OccupancyGrid` data structure, allowing us to turn map coordinates into a grid index

```
auto xy_coord_to_map_idx(const nav_msgs::MapMetaData& grid, XY p)
    -> std::optional<std::pair<size_t, size_t>> {
  const auto origin = grid.origin.position;
  if (p.x < origin.x || p.y < origin.y) return std::nullopt;

  const auto cells_x = grid.width;
  const auto cells_y = grid.height;
  const auto resolution = grid.resolution;

  const auto mp = std::optional<std::pair<size_t, size_t>>{
      {static_cast<size_t>((p.x - origin.x) / resolution),
       static_cast<size_t>((p.y - origin.y) / resolution)}};

  return (mp->first < cells_x && mp->second < cells_y) ? mp : std::nullopt;
}
```

Internally, `move_base` maintains the grid so that the robot is always (roughly) centered in the grid.

The potential field is computed from the occupancy grid. The tools used to do so are routines included in OpenCV (namespace `cv` in code to follow). The occupancy grid is converted into a raster image or rather, a `cv::Mat`. That image is then blurred (with a Gaussian kernel), raised to a power (pixelwise), differentiated (via Scharr filter), normalised and then scaled. The process is illustrated in Figure 6.4. The resulting gradient is stored in the `GradientOfGrid` structure

```
struct GradientOfGrid {
  cv::Mat grad_x, grad_y;
  nav_msgs::MapMetaData info;
```

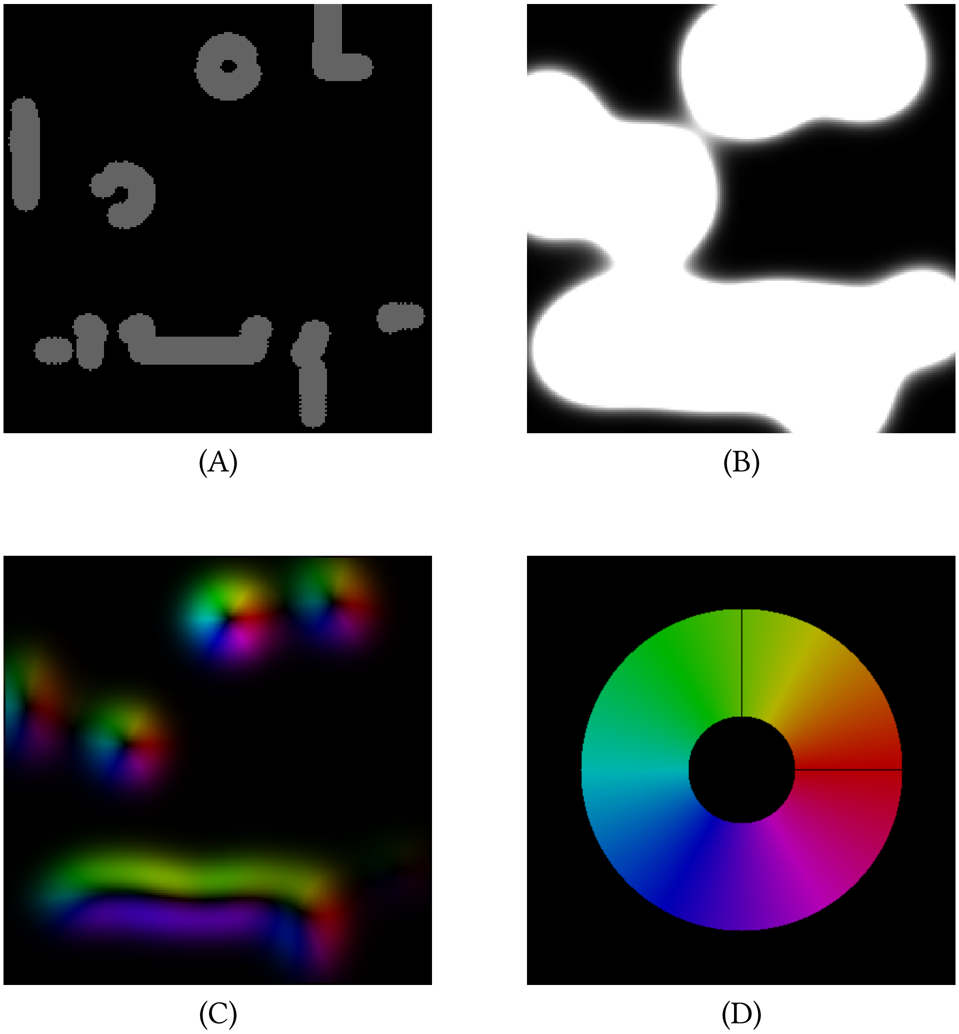


FIGURE 6.4: Illustration of the process of turning an occupancy grid into a gradient. (A) an occupancy grid, black pixels are clear and grey pixels occupied. (B) Gaussian blur and pixel-wise exponential power is applied to (A). (C) OpenCV's Scharr filter is applied to numerically differentiate (B); pixel brightness indicates magnitude of gradient and hue indicates angle according to (D). (D) Hue colour key for angle of (C) image; $+x$-axis is right (toward red) and $+y$-axis is up (toward yellow-green) and are indicated with thin black lines; angle is measured ccw from $+x$.

```
  GradientOfGrid(nav_msgs::OccupancyGrid);
  std::optional<XY> operator()(XY) const;
};
```

and here is the constructor

```
GradientOfGrid::GradientOfGrid(nav_msgs::OccupancyGrid grid)
  : info(grid.info) {

  cv::Mat costmap( // convert grid to a Mat signed 8-bit ints.
    grid.info.height,
    grid.info.width,
    CV_8S,
    grid.data.data()
  );

  costmap.convertTo(costmap, CV_64F); // 8-bit ints to 64-bit floats.

  cv::Mat costmap_smooth(grid.info.height, grid.info.width, CV_64F);

  const auto kernel_size = cv::Size(91, 91);
  // Using rule of thumb that the kernel diameter should be
  //   approximately 6σ:
  const double sigma =
    static_cast<double>(kernel_size.width - 1) / 6.;

  cv::GaussianBlur(
    costmap,
    costmap_smooth,
    kernel_size,
    sigma,
    sigma,
    cv::BORDER_ISOLATED
  );
  cv::pow(costmap_smooth, 4, costmap_smooth);
  cv::Scharr(costmap_smooth, grad_x, CV_64F, 1, 0); // x-axis
  cv::Scharr(costmap_smooth, grad_y, CV_64F, 0, 1); // y-axis

  auto image_norm = max_norm(grad_x, grad_y);

  if (image_norm != 0) {
    const auto scalef = -2 / image_norm;
    grad_x = grad_x * scalef;
    grad_y = grad_y * scalef;
  }
}
```

The `max_norm` function returns the highest absolute value in the given `cv::Mats`. Due to a lack of a thorough refactoring process, there are a few so-called *magic numbers*:

- 91 × 91, the size of the Gaussian kernel, which was tuned based on the size of the occupancy grid, 200 × 200 pixels in Bishop's case.
- 4, the power to which each pixel was raised. This was tuned to how quickly the potential should fade with distance from the occupied grid. This was determined manually, and is otherwise unjustified. We studied the diffraction properties of the robot through doorways: if the robot had difficulty entering doorways then the drop-off was too gradual.
- 2, the scale factor of the normalised image, which was likewise manually tuned in proportion to the coefficients R, $\boldsymbol{Q}_0$ and $\boldsymbol{Q}$ of the optimisation problem. Good sense would have this number in the optimisation calculation instead of in the constructor of `GradientOfGrid`, which would have happened in a final refactoring pass. In that case, it could be set when `c0` is created, and could change dynamically.

Once the constructor computes and sets `grad_x/y`, it is `operator()` which gives access to the gradient in terms of map coordinates:

```
std::optional<XY> GradientOfGrid::operator()(XY p) const {
  if (const auto mp = xy_coord_to_map_idx(this->info, p)) {
    const auto cv_pt = cv::Point(mp->first, mp->second);
    return {{grad_x.at<double>(cv_pt), grad_y.at<double>(cv_pt)}};
  } else
    return std::nullopt;
}
```

A null value is returned just in case `p` references a coordinate exceeds the bounds of the occupancy grid, otherwise the gradient values are retrieved from `grad_x/y` and returned as an `XY`.

6.5 DISCUSSION & CONCLUSION

This chapter on the catnav project is described as a case study for the category theoretically motivated control program design described in the previous chapter. Having described how the design was used to implement an NMPC controller for Bishop's motion control and obstacle avoidance, let us conclude with a discussion of the benefits and consequences of the design from a software engineering standpoint.

MODULARITY The design enabled a particularly clean separation of concerns with loose coupling and good functional cohesion. The NMPC-algebra and related code had no dependence on any part of ROS or Rx. The NMPC algebra was included in the catnav ROS node as header-only. The `nmpc_algebra` header only includes standard library headers, `XY.hpp`, and `geometry.hpp` which has tools for interpolating along the global plan to get the tracking reference sampled for the horizon. And both of those headers only include standard library headers.

The design would have been simpler if `NMPCState` and `WorldState` internally used ROS's types: those in `geometry_msgs` and `nav_msgs` namespaces. However, this would have reduced memory economy, possibly impacting iteration time due to additional heap allocation and ultimately cache thrashing. It also would have needlessly increased coupling. CrossWing, the company that designed Bishop's hardware, ultimately wants a software infrastructure independent of ROS, so adding those dependencies to the NMPC code would have required them to later modify the code. As it is, the ROS dependencies are limited to RxROS and the types required to retrieve data from topics and the `tf` transforms. Writing ROS out of the control system only requires that RxROS is replaced with plain RxCpp, and that whatever messaging system they choose to replace topics is adapted as an RxCpp observable.

The design of the NMPC algebra is very function-oriented with no real data-hiding. In theory, there are disadvantages to this: the lack of a ridged interface to the `NMPCState` means that anyone extending functionality may fail to consider contracts and invariants that would otherwise be encoded in the interface. This is balanced by the fact that everything is exposed to for extensive unit testing.

TESTABILITY In the NMPC related components of catnav, the data structures are pure structures: they have no member functions or private members. Recall that the most core and complex structures are the `NMPCState` and `WorldState`. Their interface may be regarded as the collection of free functions in which they appear as factors of the domain and codomain. This arrangement is ideal for testing because data hiding can often make it difficult to isolate components for tests. Moreover, state changes can be well tracked during integration tests. However, this design choices comes at the cost of strict invariants.

In the ROS node where the control expression lives, the majority of code is merely free functions to extract or convert data from the ROS topics. So that is similarly straightforward to test.

The RxCpp and RxROS libraries is independently tested, so there are no additional tests for their components. That said, it is quite straightforward to test that expressions in the Rx DSL mean what they are expected to mean. Since observable collections can be converted to/from lists, testing an Rx expression is a matter of creating a mock input-sequence and validating the output.

EXTENSIBILITY In my and others experiences working on catnav, the design has proven to be very extensible. Because of the good margins separating domains of the code and the expression oriented nature of the Rx-oriented design of the controller interface, extension requires little or no accidental complexity.

Future control and software engineers attempting to extend the code will benefit from two key design choices:

- To the controls engineers, the NMPC algebra and subordinate components are designed with loose coupling and simple functional interfaces. The abstraction boundaries in the NMPC code are straightforwardly motivated by (or depicted in) the underlying mathematical relationships. So, insofar as a controls engineer is able to grapple with the mathematics of the algorithm they should be able to extend the code.
- To the software engineers, the Rx is *very* well documented in many web-sites and books, and is a fairly common skill in the general labour pool of programmers. The Rx expressions are to programming what pipes are to plumbing, so adding input is a matter of subscribing to a source, extending the `WorldState` and making the necessary accommodations in the control algebra.

There is one particular aspect of the design that is possibly detrimental to extension: purely public data in the `NMPCState`. Normally invariants are enforced by a public interface to private data. In the case of `NMPCState`, we might expect the values of the prediction horizon $\boldsymbol{x}$ to be always coherent with the values of the control horizon u. With public data, a programmer can change `Dth` and neglect to call `forecast`, meaning that the arrays of the prediction horizon are no longer consistent with the control horizon. In the future, it may be beneficial to redesign the free-function interface to be finer-grained so that invariants are at least better communicated, if not strictly enforced.

MAINTAINABILITY Because the external dependency footprint is narrow, version drift should not be a problem in the foreseeable future—except for one *very* notable exception. At the beginning of the SENTRYnet project, the investigators adopted a ROS-1 platform which was already superseded by ROS-2 at that time. However, because the ROS components are well isolated from the rest of the system, even a migration to ROS-2 would require no modification of the core components related to control calculation.

Regarding the Rx components, Rx is well established as an abstraction for linear, asynchronous collections and has a stable and consistent implementation across many programming languages. Even though RxCpp may mature, there should be few, if any, breaking changes to the interface. RxROS, which started as an academic project, was adopted by the *ROSin project*,[10] a European Union-funded project aimed at developing and maintaining open-source ROS tooling for industrial application. Hopefully, this means it will be well maintained in the future. Since it is a straightforward wrapping of RxCpp over ROS topics, it is not a large or complicated codebase and CrossWing is sufficiently resourced to fork and maintain it internally if necessary.

PERFORMANCE The development began with a deliberate effort not to pessimise or prematurely optimise for performance. Instead, once the catnav node was properly integrated with components from the other SENTRYnet projects, there was a planned process of performance and memory benchmarking that would drive a cycle of refactoring for performance. This effort would have been well supported by the suite of unit tests. Unfortunately, the SENTRYnet project ended before the teams could collaborate on integration.

During the lifetime of the project, performance was never an issue. In fact, the catnav node ran fast enough that we had to debounce the odometry stream to limit the throughput of the controller.

LOGGING *&* ERROR HANDLING How well does the design handle error conditions, exceptions, and edge cases? Is there proper error reporting, logging, and graceful recovery mechanisms?

[10] `http://rosin-project.eu/`

The catnav node does not use exceptions for error handling. This is a point of weakness, since it relies on STL components which do in fact use exceptions for error feedback. Eventually, CrossWing will want catnav to work in a real-time environment. Since C++'s exceptions do not yet have a system to make stack-unwinding a deterministic-time process, all current standards for C++ in an industrial real-time context preclude their use for error handling.

In functional languages, functions that can fail often return a sum type with terms for a target value or an object encoding errors. The `std::optional` template is the simplest example of that mechanism. The reader may have noticed that some of the members of the controller and world states are optional. It was planned that some of the optional returns would be extended with more descriptive error indicators. This is especially true in the horizon optimisation routines. As mentioned earlier, the optimiser was supposed to be rewritten to wrap the functionality of a scientific/industrial grade library component. The resulting interface would return an optimised controller state or an error type encoding whatever feedback is available from the chosen library. A monadic interface such as the one for the proposed `std::expected` [129, 157] greatly simplifies the process of composing functions that return these sorts of error sums.

The debug compilation target for catnav includes verbose logging output. This is done using the Rx `tap` combinator. Both the `NMPC` and `WorldStates` have routines to "stringafy" them into YAML.

6.5.1 Future Development

Unfortunately, the project ended with the control system unfinished. The unfinished portions largely has to do with integration with other SENTRYnet components, with new kinds of input and output values for those interactions. That stage was to be in place before the final phases of development such as performance tuning and fault hardening/error handling. In particular:

- Refactoring was needed to eliminate magic numbers, select better names and eliminate some regrettable design choices.
- Performance based optimization would have eliminated needless copying (particularly in function arguments where reference semantics would be indicated). Finalisation of the input/output structures of the controller would was also to facilitate heap/stack usage analysis and to optimise for CPU cache behaviour on Bishop's specific hardware.
- Open-loop speed control was not properly implemented because speed scripting was supposed to come from an external planner downstream from a specialised computer vision system and an artificial intelligence (AI)-based system for human-robot interaction.
- We intended to adopt a more a more efficient and robust optimisation algorithm, ideally from an off-the-shelf optimisation library (such as from the GNU (GNU) scientific library). Along with that update, a monadic error reporting and handling system was to be implemented. This also relied on a finalisation of the plant model which would was to include summary state information from other components.

6.5.2 Conclusion

Though catnav was not fully completed vis-à-vis the SENTRYnet project goals, it faithfully implemented the control program design of the previous chapter. It functioned well as a basic motion/tracking controller, especially as compared to our prior designs using more conventional software designs, [135, 151]. Still more importantly, the SENTRYnet project required a great deal of adaptation and extension of the design in order to integrate with the ROS-based platform; and the design proved to be amply flexible and extensible. I conjecture that this is due to the abstraction boundaries arising from the mathematical design of the architecture. In particular, there is a separation of concerns that reduces internal coupling of the code (compared to other designs I and my coworkers have considered) and facilitates a straightforward mental model of the program logic. During extension, parts of the programming process feel more like plumbing than programming because it is a matter of matching and joining input to output in a functional pipeline. Emphasis in the code is shifted toward expressions of meaning rather than instruction of operation. Of course, at the lowest levels of the program, the code looks like more typical C/C++—but this is where reasoning is local and the scope of complexity is limited to what a human programmer can reasonably behold.

DISCUSSION & CONCLUSION

7

This dissertation set forth to study control systems software at its foundation. This foundation rests not on silicon chips or copper wire, but on the science of structure, order, and relation. It is true of all engineering that the fundamentals of practice take root in mathematical models of dynamical interplay among information, matter and energy. Yet somehow, this is less true of software engineering than of its elder siblings. And to suggest so is not pejorative of software engineering! It is indicative of a horizon to be explored; an undiscovered country to which coming generations of researchers will stake claim. I repeat an epigram of an earlier chapter, a quote from Anthony Ralston and Mary Shaw [43]:

> Mathematical reasoning does play an essential role in all areas of computer science which have developed or are developing from an art to a science. Where such reasoning plays little or no role in an area of computer science, that portion of our discipline is still in its infancy and needs the support of mathematical thinking if it is to mature.

I feel this idea resonates in the parts of software engineering that depend on computer science: the architecture, design and implementation of software programs.

Control systems software, the target of the thesis' aims, is an interesting contact surface. Controls engineering, in contrast to software, is one of the most mathematically rich of the engineering disciples. When controls engineers pass specifications to software engineers who in turn pass it to code, there is surely a disruption at the boundary where the mathematics ends and the software begins. Despite this boundary, the tendency is to simply identify the software with the math. As others have pointed out, [92], the software should be treated as a dynamical system in its own right.

I have posited that the solution is to formulate an algebraic theory that treats dynamical system at a level of abstraction that subsumes the dynamics of all components of the controlled system, *including its software*. This way, a specification can be carried from the hands of the controls engineer all the way to the code by way of the same unifying mathematical language. This places the onus equally on the controls and software engineers to renew or extend their practice.

7.1 CONTRIBUTION

In §1.2, I itemised the main goals as

- a deeper understanding of control systems software that organises the fine detail of these programs in a way that eases specification and implementation,
- a mathematically motivated design of a control program written in C++,
- a model of a control program that bears interpretation in the same categorical setting as ordinary dynamical systems.
- demonstration of the above by means of simulation and by means of implementation on a real electromechanical device designed for a non-academic purpose,

Toward that ends, I have presented an algebraic (and coalgebraic) model of a control program borne out in C++17 code. That model is a thin assemblage, but it is built upon a deep and profound calculus of relationships found in the literature of theoretical computer science and abstract algebra. How thin is this assemblage exactly? We can measure–5-simple lines of code:

LISTING 7.1.1: The control expression

```
const auto controls =
  plant_state
    | combine_latest(reference)
    | scan(c0, state_transition_function)
    | map(r);
```

This short code snippet is deceptively simple because it has a precise meaning in an algebraic language internal to a category of C++ programs, introduced in this thesis. Nothing is left open to interpretation, and every exquisite detail is accounted for. This expression was suggested by the math and not the other way around. The mathematical structure does not provide the implementation, but gives us a set of abstraction boundaries for architecture and design. So while it can be implemented in a variety of ways, the engineer has a clear, coherent mental model, test surfaces amenable to intense scrutiny, and tests which are themselves suggested by the axioms of the structure. This gives the application architecture with

- loosely coupled internals,
- flexibility for extension, and
- a constabulary test suite.

Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains.
— Steve Jobs

Even though an engineer may come to this code on their own, through intuition, the mathematical formulation gives scientific justification to what would otherwise be art.

In abstract, LST. 7.1.1 represents a sequential state machine in the category ***Cpp***. The description of this category and its bicartesian closure is also one of the main contributions of the thesis. ***Cpp*** is a mathematical object and does not prescribe implementation. It is not a mathematical model of C++, but rather a model of data, structure and computation that can be implemented in C++ in non-unique ways. A particular implementation is given which demonstrates the ability of C++ to, in some capacity, be mathematically faithful. Also, the axioms of the category are encoded in C++ as unit tests.

Two examples were given of software control implemented using the model:

1. *in silicio:* stabilisation of a simulated spring-mass-damper with PID, and

2. *in vivo:* motion tracking of a mobile robot with NMPC.

7.2 DISCUSSION

The practice in computer science of structuring programs so that they can be identified with certain types of mathematical structures is known as FP. This is not always an explicitly category theoretical endeavour, but that relationship has been established at least since the 1970s. A 1975 paper by Goguen [34] cites some relevant literature and even notes that control theory and computer science both share a common concern with realisation theory. In purely mathematical terms, the rapprochement of control and automata theory began in the 1960s [5, 10]. The idea of controlling robots with functional (reactive) programming techniques was explored in the early 2000s [93], in the language Haskell, which is quintessentially functional. But it remained a challenge to implement these ideas in a systems language like C++ that is most commonly used for control applications.

This thesis addresses the issue of using these theoretical tools in C++, with immediate application to any language supporting the implementation of a DSL for asynchronous lists, such as Rx.

7.2.1 Limitations

It is a reality of applied mathematics that one must always be concerned with the limitations of their model, aware of obvious failure and vigilant for the unexpected ones. Limitations of ***Cpp*** are discussed at the end of CH. 4, with the caveat that there may certainly be ways in which ***Cpp*** and C++ diverge that I have not yet found or imagined, and many I would find surprising. I just can not resist bringing out the old cliche: *the map is not the terrain.* But I very much think of ***Cpp*** as my map when I write C++ programs. I plan my route through the category, and make detours when necessary, due to either inconsistencies of the map, realities of code performance, or most importantly: where it is simply too unwieldy to write category theory in C++ spelling. That last point is an important engineering trade-off because warping a language to fit a model will make the program code less clear to the detriment of maintainers and collaborators. In both of the two implementation examples there were points where I detoured from the functional model into imperative or OO programming style for the sake of ease. But even then, having boundaries prescribed by the model made those departures safer: I had contractual obligations that limited the scope of the departures.

The expression LST. 7.1.1, is fairly unlimited in terms of the behaviour of control programs by which can be implemented. It is merely a factorisation of the program into an I/S/O system, which can describe any state-mediated input/output relationship that the computer running it is capable of producing. It is limited in terms of the fact that programmers should conform to the FP model near these abstraction boundaries. For example, the state transition and readout functions, `f` and `r`, should be pure functions, without side effects. If the programmer does otherwise, they should do it with deliberation and care. The implementation of the DSL for the push-based list interface may introduce performance issues and those should be addressed at the level of the library. Moreover, on MCUs where programs are organised into a `setup` and `loop` functions instead of a `main` function, then the expression no longer makes sense at an architectural level. But interpretation of control programs as a sequential state machine in ***Cpp*** can still be applied with some additional effort. (This is perhaps a future direction for research.)

7.3 OUTLOOK & FUTURE DIRECTIONS

The broader picture is of a categorical unification of systems both of the discrete variety of the present case and continuous models associated with physical processes. This is currently an active area of research, CF. [87, 148, 153, 176]. In particular, the efforts of the Spivak and colleagues [172, 178] are particularly intriguing because of their broad view that treats interconnection, dynamics and interaction simultaneously, which can all be couched in the stunningly well appointed category of polynomial functors [190] and lenses. In fact, while writing this thesis I became aware that if I attempted to set my thesis in the context of their work, I would be chasing a moving target. While their work did nothing to invalidate my own, they accomplished quite a lot that located the present model in a context as a special case of something much greater. One of the natural follow-on directions is to reformulate the mathematics in therms of $\mathbb{P}$**oly**, and in terms of the sheaf theoretic view of event based systems described in [176].

In CH. 2, I briefly described FRP and mentioned that I planned to work with Blackheath on an implementation of FRP in C++23 which would then allow me to harmonise with Hudak group's research in Haskell [93]. This has the appeal of explicitly adding continuous time, which is fundamental to FRP, to the present model.

All told, there is already a vast and rich literature of categorical systems theory that is still rapidly developing. There is so much to be understood about how this theory can be passed to praxis and the benefits of doing so. My personal research interests, reflected in the present thesis, is merely to bridge that gap between theory and praxis, and understanding the benefits and trade-offs.

MATHEMATICAL NOTATION

A

We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful. In fact, mathematics is, to a large extent, invention of better notations.
— Richard P. Feynman

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.
— A.N. Whitehead

PERHAPS because the thesis overlaps several disciplines across engineering, mathematics and science, I feel a particular responsibility to be unusually careful and clear about notation.

As a starting reference, the mathematics in this document generally adheres to the provisions set forth in the ISO/IEC standard for mathematical signs and symbols [106], ISO/IEC 80000-2:2009. Beyond the scope of that standard, or where where we depart from it, notation is introduced as it naturally falls in first-use, either in definitions or demarcated in numbered passages:

NOTATION A.0-1 (Notation block). This is the body of a *notation block*. Such passages shall be referenced as NTN. A.0-1.

This appendix has two purposes and to those ends, is organised into two sections. The first describes general mathematical notation that does not fall naturally into the exposition of the thesis. This includes

- some features of ISO/IEC 80000-2:2009,
- conventions specific to subfields of engineering, science or mathematics and
- specific uses of things, like delimiters, that are often ambiguous from author to author.

The remaining section is a tabular reference guide to collect and index important notation from across these pages and ISO/IEC 80000-2:2009—mostly as a summary aid to those not following in linear progression.

A.1 NOTATION *NOT* OTHERWISE DOCUMENTED

The ISO/IEC 80000-2:2009 document is an international standard and has some provisions that may land strangely on the North American eye. So to begin, let us highlight some features of the standard.

Quantities which are not variable across time or context (such as immutable constants of mathematics) are set in upright type. For examples,

- the imaginary unit: $\mathrm{i}^2 = -1$,
- Euler's number: $\mathrm{e} = \exp(1)$,

- the ratio of a circle's diameter to its circumference (in flat spaces) is $\mathrm{\pi}$
- differential/integral operators, as in

$$\frac{\mathrm{d}f}{\mathrm{d}x} \quad \text{or} \quad \partial_x f \quad \text{or} \quad \mathrm{D}_x f \quad \text{or} \quad \int f(x)\,\mathrm{d}x,$$

and so on. Variables, parameters, contextual constants, running numbers and alike are set in italic type. So in the expression $\sum_i a_i = 2 + 4\mathrm{i}$, the italic i is a counter while the upright i is the imaginary unit.[1] Likewise for functions, we write $f(x)$ with an italic f for some contextual definition of f, but $\sin(x)$ is in upright type. (In TEX documents, it is particularly easy to forget the backslash in `\sin(x)` giving "*sin*(x)" which recognisable as a common error. So the only unusual thing here is seeing this convention of uprights extended to constants.)

* * *

Compliant with ISO/IEC 80000-2:2009 Item № 2-7.19, but beyond its scope, we disambiguate the use of delimiters.

NOTATION A.1-1. Parentheses, (...), will be used as traditional delimiters in mathematical expressions, indicating or disambiguating order of operations. They will also be used to indicate tupeling as in $(a, b, c) \in A \times B \times C$, when contents are conjoined with commas.

NOTATION A.1-2. Braces, {...}, identify sets. For example, $\{\, a_1, a_2, \dots, a_n \,\}$.

NOTATION A.1-3. The previous set may also be expressed with ranged brace notation: $\{\, a_k \,\}_{k\,=\,1}^{n}$.

NOTATION A.1-4. Brackets, [...], will be used to enclose composite structures such as lists, vectors and matrices. Commas will distinguish lists from row vectors:

$$\text{a list } [1, 2, 3] \;\; \textit{vs.} \text{ a row vector } [1\ 2\ 3].$$

NOTATION A.1-5 (Set membership *&* judgement)**.** An expression of membership $a \in A$ is a logical proposition and has a truth value. We can say that something holds if or when or for $a \in A$, but it is not a judgement. To declare that a is in A we write $a :\; A$.

[1] Of course, one should not foster confusion by needlessly overloading letters in a single expression.

In function declarations as $f : A \to B$, the colon notation for type judgement is quite conventional. But in light of colon meaning type judgement, the expression $A \to B$ can be thought of as the set of functions between A and B (and making clear that the arrow $\to$ is itself an operator).

NOTATION A.1-6 (Function application). Given a function $f : A \to B$ and a value $a : A$, then the juxtaposed fa indicates function application. This is in agreement with standard practices in functional programming literature, but deviates from ISO/IEC 80000-2:2009.

Parentheses, as in $f(a)$, may be used for clarity, especially where the parameter is an expression and order of operations may be unclear. Since parentheses are also used in conjunction with the comma operator for tupeling (see NTN. A.1-1) application of a multivariate function, say $f : A\times B \to C$, is more familiar: $f(a, b)$.

In a sufficiently abstract setting, functions themselves are seen as mere values of a set. It is important then to be able to depict these values without naming them and stressing the economy of our namespace.

NOTATION A.1-7 (Lambda functions). We make use of the *maps-to* arrow ($\mapsto$) to express elements of function sets without giving them names. Taking for example f as defined above, we could simply write $(a \mapsto fa) \in A \to B$, with some redundancy since we have already named f. But the target on the right side of the maps-to arrow can be, for example, an expression in some sort of algebra.

For example of the above NTN. A.1-7, take $\left(x \mapsto x^2\right) \in \mathbb{R} \to \mathbb{R}$, the square function on real numbers. We could then write $\left(x \mapsto x^2\right)(2.5) = 6.25$.

NOTATION A.1-8 (Function definition). Stacking the notions of type judgement and lambda-notation, we obtain a thorough notation for defining named functions:

$$\begin{aligned} f : A &\to B \\ a &\mapsto fa. \end{aligned}$$

where presumably, fa would be some sensible expression rather than a circular redundancy. The notation may also be adapted inline using a semicolon: $f : A \to B; a \mapsto fa$.

In general, sets will be denoted with italic roman capitals $A, B, C, \ldots$.

NOTATION A.1-9 (Assumed set elements). If a passage of text refers to a set denoted with a roman capital, then lowercase variables of that letter, possibly with super/sub-scripts or other decorations will be understood to denote arbitrary members of that set. For examples, $i \in I$, $x' \in X$, $s_0 \in S$ and so on.

NOTATION A.1-10 (Infix vs. prefix morphisms). Given a binary infix morphism, for example $\otimes$, if it benefits clarity at any point to use it in prefix form, it will be placed in parentheses: $(\otimes)(A, B) \equiv A \otimes B$. In reverse, I borrow notation from Haskell in decorating a prefix morphism f with back-ticks, turning it into an infix morphism: $f(a, b) \equiv a \grave{}f\grave{}\ b$.

A.2 table *of* notation

This section curates, tabulates and indexes key pieces of notation from across the text. This includes those defined in notation blocks and those in ISO/IEC 80000-2:2009. It is not exhaustive, but should include everything that may be unusual, is unusually nuanced, or is ambiguous. For examples: function application without parentheses, $f x$, is unusual; the notation $f : A \to B$ is standard, but the fact that it may also be used for category arrows is unusually nuanced; the notation $\mathbb{N}$ for the set of natural numbers is common but different authors disagree on whether or not $\mathbb{N}$ includes zero, so it is ambiguous.

table A.1: Notation with... Ref № is a reference to a passage in this text, or a table line item in ISO/IEC 80000-2:2009.

Ref №	Expression	Meaning	Remarks/Examples
ISO 2-6.1	$\mathbb{N}$	the set of natural numbers $\mathbb{N} = \{ 0, 1, 2, 3, \dots \}$	Restrictions can be specified with subscript: $\mathbb{N}_{>0} = \{ 1, 2, 3, \dots \}$
ISO 2-6.2	$\mathbb{Z}$	the set of integers $\mathbb{Z} = \{ \dots, -2, -1, 0, 1, 2, \dots \}$	Restrictions can be specified with subscript: $\mathbb{Z}_{\geq 0} = \mathbb{N}$, for example
ISO 2-6.4	$\mathbb{R}$	the set of all real numbers	Restrictions can be specified with subscript: $\mathbb{R}_{\geq 0} = \{ x \in R \mid x \geq 0 \}$
NTN. 3.2-1	$\mathsf{n}, \mathsf{1}, \mathsf{2}, \dots$	the n-th enumeration set	Sans-serif lowercase variables or sans-serif uppercase numerals. Given $n \in \mathbb{N}$, $\mathsf{n} = \{ 0, \dots, n - 1 \}$. E.g. $\mathsf{3} = \{ 0, 1, 2 \}$ Corner case: in this document, most often $\mathsf{1} = \{ () \}$ instead of $\{ 0 \}$. See B.1-12.
DEF. B.1-2 DEF. B.1-3 DEF. B.8-6	$\mathbf{1}, \mathbf{2}, \mathbf{3}$	the —th finite category	Bold sans-serif uppercase digits. Categories with a finite number of objects (specified by the digit) and arrows. Only **1**–**3** are used, and no general/inductive definition is given.

Ref №	Expression	Meaning	Remarks/Examples
NTN. B.9-1	$\dot{\mathbf{1}}, \dot{\mathbf{2}}, \dot{\mathbf{3}}, \ldots, \dot{\boldsymbol{n}}$	the n-th discrete finite category	Bold, sans-serif uppercase digit with over-dot, or bold lowercase sans-serif letter. For some $n \in \mathbb{N}$, the category with enumerated objects $1, 2, \ldots, n$, and where the only arrows are identities: $\dot{\boldsymbol{n}} \coloneqq \boxed{1 \quad 2 \quad \cdots \quad n}$.
NTN. A.1-3	$\{ a_k \}_{k=m}^{n}$ $\{ a_k \}_{k=\{ m, \ldots, n \}}$	the set of values a_k where k ranges from m to n	E.g. $\{ a_n \}_{n=3} = \{ a_n \}_{n=0}^{2} = \{ a_0, a_1, a_2 \}$
NTN. B.6-6	$(a_i)_{i \in n}$	an n-indexed family of A-values.	For a set A and values $a_i \in A$, and enumeration set n. Note: not to be confused with an indexed set, or ranged brace. The parentheses invoke tupeling, which has a formal equivalence with functions $n \to A$.
NTN. B.1-12	$()$	• the empty tuple • the unique member of a singleton	With enumeration set $1 = \{ 0 \}$, it is useful to denote the lone element as $()$. This way, functions with domain 1 can be given empty argument as in $f()$ instead of $f0$.
NTN. B.1-4	$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$	arbitrary categories	Italic sans-serif Latin capitals.
NTN. B.1-5	$A, B, C, \ldots$	category-objects	Sans serif, bold-italic Latin capitals.
NTN. B.1-4	f, g, h	• category-arrows • morphisms	Italic Latin miniscules.
ISO 2-11.3	$f : A \to B$	f takes A to B	The morphism or category-arrow f has domain A and codomain B

Ref №	Expression	Meaning	Remarks/Examples
DEF. B.1-1	$A \xrightarrow{f} B$	f takes A to B $f : A \to B$	To be read as a small diagram meaning the morphism or category-arrow f has domain A and codomain B. Not mere redundancy with ISO 2-11.3, this notation makes commutative diagrams a first-class citizen with respect to inline mathematics.
DEF. B.2-1	$A \hookrightarrow B$	• An injection from A to B. (Injective function.) • Monomorphism in a category from A to B.	An injection is a mono in $\$\mathbf{et}$. More generally, authors in CT use $\rightarrowtail$ instead of the hook-arrow. Since the present work focuses on concrete categories, the hook-arrow is appropriately suggestive of inclusion, and a necessary precursor for the sub-set relationship: $A \subset B$,
DEF. B.2-2	$A \twoheadrightarrow B$	• A surjection from A to B. (Surjective function.) • Epimorphism in a category from A to B.	A surjection is an epi in $\$\mathbf{et}$.
NTN. B.6-2	F, G, H	arbitrary functors	Sans serif, Latin capitals.
NTN. A.1-5	$a : A$	asserting a is of A	Like $a \in A$, but is a judgement, not a propositional function. *E.g.* we can say *if* a ∈ A then ..., but we can assert that we have $a : A$.
ISO 2-5.16	$A \times B$	the Cartesian product of sets A and B the categorical product of objects A and B	The elements belonging to A_1 and A_2, with information about which they came from. $A \times B \cong \{ (a, b) \mid a \in A \text{ and } b \in B \}$
—	$A_1 \sqcup A_2$	the disjoint union of sets A_1 and A_2 AKA the discriminated union of sets A_1 and A_2	The elements belonging to A_1 and A_2, with information about which they came from. $A_1 \sqcup A_2 \cong \{(i, x) \mid x \in A_i \text{ and } i \in \{1, 2\}\}$

Ref №	Expression	Meaning	Remarks/Examples
ISO 2.5-13	$A \setminus B$	A minus B	The elements belonging to A but not to B. $A \setminus B = \{\, a \mid a \in A \text{ and } a \notin B \,\}$
NTN. A.1-6	$f(x)$ fx	f at x	Function application is basic: the main reason for including it here is to highlight that the parentheses are optional.
NTN. A.1-7	$(x \mapsto fx)$	the lambda function mapping any x in the domain to fx of the codomain	A notation for making anonymous functions. Typically, the expression on the right-hand side of the maps-to arrow would not simply be a named function. E.g. $(x \mapsto x^2)(3) = 9$
NTN. A.1-8	$f : A \to B$ $\quad a \mapsto fa$ $f : A \to B; a \mapsto fa$	the morphism f takes A to B by mapping $a \in A$ to $fa \in B$	Combines ISO 2-11.3 with Notation NTN. A.1-7 to give a comprehensive way of defining morphisms.
NTN. B.9-7	$f \vartriangle g$	*f and g*	If $f : C \to A$ and $g : C \to B$ then $f \vartriangle g : C \to A \otimes B$ where $\otimes$ is the product operator in the ambient category. E.g. in $\mathbf{\$et}$ take $\sin \vartriangle \cos : \mathbb{R}_{[0,\,2\pi)} \to \mathbb{R} \times \mathbb{R}$ which maps the 2π-interval to the unit circle.

Ref №	Expression	Meaning	Remarks/Examples
NTN. B.10-3	$f \triangledown g$	f *or* g	If $f : A \to X$ and $g : B \to X$ then $f \triangledown g : A \oplus B \to X$ where $\oplus$ is the coproduct operator in the ambient category. E.g. in 𝕊**et** where $\oplus = \sqcup$, take $X \coloneqq \{$ “red”, “black” $\}$, $f \coloneqq \{$♥, ♦, ♠, ♣$\} \to X$ and $g \coloneqq \{$■, ■, ♦$\} \to X$, mapping the symbols to their colour in the obvious way. Then, $f \vartriangle g : \{$♥, ♦, ♠, ♣$\} \sqcup \{$■, ■, ♦$\} \to \{$ “red”, “black” $\}$, in the obvious way, but accepting items from either domain. But note that “♦” is distinct in the domain: there are two red triangles distinguish as being from the left or right of the $\sqcup$ symbol.
ISO 2-11.7	$g \circ f$ gf	g after f	$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend right, "g \circ f"'] & B \arrow[r, "g"] & C \end{tikzcd}$ Composition of functions, maps or category-arrows. In the case of functions, $(g \circ f)(x) = gf = g(f(x))$. NB: gf is not part of the ISO standard.
NTN. B.1-7	$f \mathbin{;} g$	f then g or g after f	Like the ring operator, but with arguments reversed: $f \mathbin{;} g = g \circ f$ $\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend right, "f ; g"'] & B \arrow[r, "g"] & C \end{tikzcd}$
§B.4	$\vec{a}$	the global element a	For a in any set A, $\vec{a} : 1 \to A; () \mapsto a$. E.g. given $\pi \in \mathbb{R}$, $\vec{\pi}() = \pi$
NTN. B.11-3	$\|A\|$	underlying set of A	For monoids, groups and other algebraic structures over sets, the operator $\|\cdot\|$ references their underlying set. E.g. for a monoid, $\|(M, \circ, e)\| = M$

Ref №	Expression	Meaning	Remarks/Examples
§B.16.1	U	A forgetful functor: of monoids, of groups, of programs, *etc.*	For monoids, groups and other algebraic structures over sets, the functor U maps to their underlying sets and their homomorphisms to underlying functions in \$**et**. E.g. for a monoid, $U(M, \circ, e) = M$, identical to the universal algebraic concept of underlying set: $\|(M, \circ, e)\| = M$.
NTN. B.6-3	$F : \boldsymbol{C} \to \boldsymbol{D}$ $\begin{cases} X \mapsto FX \\ f \mapsto Ff \end{cases}$	functor F takes $\boldsymbol{C}$ to $\boldsymbol{D}$ by mapping objects X to FX and arrows f to Ff	Like NTN. A.1-8, but with a brace and two mappings to reflect the twofold nature of functors: having maps for both objects and arrows.
NTN. B.7-2	$\alpha, \beta, \gamma, \ldots$	arbitrary natural transformations	Greek miniscules.
DEF. B.7-1	$\alpha : F \Rightarrow G$ $F \overset{\alpha}{\Rightarrow} G$	the natural transformation α maps the functor F to the functor G	The double arrow ⇒ distinguishes these from other morphisms. Note that in a category of functors, ⇒ becomes →.
DEF. B.7-1	α_X	the component of the natural transformation α at object X	Given functors $F, G : \boldsymbol{C} \to \boldsymbol{D}$, if $\alpha : F \Rightarrow G$ and $X \in \boldsymbol{C}$ then α_A is the component of α at the object X.
‹B.7-6›	$\vartheta * \eta$	the horizontal composite of ϑ and η	Given functors $F, G : \boldsymbol{C} \to \boldsymbol{D}$ and $F', G' : \boldsymbol{D} \to \boldsymbol{E}$, and natural transformations $\eta : F \Rightarrow G$ and $\vartheta : F' \Rightarrow G'$, the horizontal composite has two equivalent (componentwise) formulas, one of which is: $(\vartheta * \eta)_X = G'\eta_X \circ \vartheta_{FX}$
DEF. B.2-3	$A \cong B$	A is isomorphic to B	To objects are isomorphic if there exists a pair of arrows f, g such that $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$

Ref №	Expression	Meaning	Remarks/Examples
NTN. B.15-6	$(\grave{\ })$ or $\grave{f}$	"*f curried*" The exponential transpose of the arrow f.	In a symmetric monoidal category, $(\boldsymbol{C}, \otimes)$, with internal homs $(\multimap)$ there is a bijection $\boldsymbol{C}(A \otimes B, C) \xrightarrow{\sim} \boldsymbol{C}(A, B \multimap C)$. Given an arrow $f : A \otimes B \to C$ then $\grave{f} : A \to B \multimap C$. The operation is called currying, and the bijective inverse is called uncurrying.
NTN. B.16-2	$L \dashv R$ $\begin{tikzcd} \boldsymbol{C} \arrow[r, bend right, "R"'] \arrow[r, phantom, "\perp"] & \boldsymbol{D} \arrow[l, bend right, "L"'] \end{tikzcd}$	For functors L, R, L is the *left adjoint of* R and R is the *right adjoint of* L	NB: In all cases, the post of the turnstile symbol, $\dashv$, points toward the left adjoint.
NTN. 3.3-4	μF	The least-fixpoint (initial algebra) of the functor F.	An endofunctor F induces a category F-**Alg** and μF denotes its initial object. In the context of the category on which F maps, μF instead refers only to the underlying object $\lvert\mu F\rvert$.
NTN. 3.3-8	νF	The greatest-fixpoint (terminal coalgebra) of the functor F.	An endofunctor F induces a category F-**coAlg** and νF denotes its terminal object. In the context of the category on which F maps, νF instead refers only to the underlying object $\lvert\nu F\rvert$.
—	$w \mathbin{+\!\!+} w'$	Concatenation of tuples, lists or strings.	
DEF. 3.5-1	$P_A\, B$	The Cartesian product $B \times A$.	A notational convenience for composing more complex functors and discussion of fixpoints. EG. The greatest fixpoint νP_A is carried by the set of all infinite sequences, A^{ω} and the least fixpoint, $\mu(1 \;+\; P_A)$, is carried by the set of all finite sequences (or words), A^*.

Ref №	Expression	Meaning	Remarks/Examples
def. 3.5-1	$\hat{A}$, $\hat{B}$, …	Object functors: P_A, P_B, …	A notation that gives purchase on the notion of “objects as functors” in categories with Cartesian products.
def. 3.6-6	$\hat{A}_\bullet$, $\hat{B}_\bullet$, …	Pointed object functors: $1 + P_A$, $1 + P_B$, …	Object functor summed with a terminal object, which gives “pointed” algebras. That is, algebras with a global element or *base-point*.
ntn. 3.9-3	E_A	Covariant internal hom-functor: $E_A\,X = A \multimap X$.	A notational convenience for composing more complex functors and discussion of fixpoints.
—	S^1	The 1-dimensional circle manifold, aka the 1-sphere.	See [95, p. 7].

B CATEGORY THEORY

> A mathematical object is determined by its relationships to other objects. Practically speaking, this suggests that an often fruitful way to discover properties of an object is *not* to investigate the object itself, but rather to study the collection of maps to or from the object.
>
> — Tai-Danae Bradley
>
> https://www.math3ma.com/blog/the-most-obvious-secret-in-mathematics

“Abstract-nonsense”—is an idiom many people hear upon introduction to Category Theory (CT). Practitioners use the term affectionately, with tongue-in-cheek. The idiom is a recognition of CT as a general theory of structure and abstraction without context. Categorical constructions inherit meaning from their domain of application. While this makes the theory feel somewhat nebulous at first reading, a sufficient study of CT opens a door in the mind that remains permanently ajar. Light from this door illuminates patterns of organisation and abstraction anywhere you see structured relationships. This might leave you with the impression that *category theory is meant to be applied*, and I endorse that view.

This appendix provides an introduction to basic CT with emphasis on topics that aid the presentation of the present thesis. For general introductions, I recommend the texts of Awodey [109], Leinster [130] and Spivak [133]. For the reader with computer scientific inclination, Barr and Wells' [82] is an excellent introduction, as is Pierce [64] and Buurlage [162]. The standard reference in category theory is the venerable text of Mac Lane [81]. Most definitions in this appendix are based on these sources, with some variation in structure, focus, detail and approach.

B.1 DEFINITION *of* A CATEGORY

The following definition of a category is formidable, but let us front-load it with this intuitive summary: a category is a collection of objects as nodes with a composable network of directional arrows connecting them, forming a directed graph. The arrows must be closed under composition: any two compatible arrows must be equivalent to a third arrow with equal meaning. The composition must further be associative and unital. This means that each object must have a special arrow that loops back on itself—its identity arrow that composes with other arrows in such a way that the composition has no net effect, the same way multiplying any number by 1 has no net effect.

DEFINITION B.1-1 (Category). A ***category***, $\boldsymbol{C}$, consists of the following data:

- a collection of ***objects***: $A, B, C, \dots$ collectively denoted $\mathrm{ob}\,\boldsymbol{C}$,
- a collection of ***arrows*** among the objects: $f, g, h, \dots$, collectively denoted $\hom \boldsymbol{C}$
- to each arrow f is associated two objects called its

  domain, denoted $\mathrm{dom}\, f$, and

  codomain, denoted $\mathrm{cod}\, f$,

  which is summarised as $f : A \to B$, or equivalently $A \xrightarrow{f} B$, both symbolising $\mathrm{dom}\, f = A$ and $\mathrm{cod}\, f = B$,
- a binary product on arrows ($\circ$) denoting composition.

These data are obliged to the following axioms:

C-1: (Closure) Given arrows f and g where $\mathrm{cod}\, f = \mathrm{dom}\, g$, there is a composite arrow $g \circ f$, making the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[rr, bend right, "g\circ f"'] & B \arrow[r, "g"] & C\,.
\end{tikzcd}$$

C-2: (Associativity) Given compatible arrows f, g, h, the composition is associative: $h \circ (g \circ f) = (h \circ g) \circ f$, reflected in the commutativity of the diagram

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h\circ g\circ f"'] \arrow[dr, "g\circ f"] & B \arrow[d, "g"] \arrow[dl, "h\circ g"'] \\
D & C\,. \arrow[l, "h"]
\end{tikzcd}$$

C-3: (Identity) For each $\boldsymbol{C}$-object, $A \in \mathrm{ob}\,\boldsymbol{C}$, there is an *identity arrow*, denoted $\mathrm{id}_A : A \to A$, acting as the left- and right-unit for the composition operator. So for any arrow, $f : A \to B$, the following diagram commutes:

$$\begin{tikzcd}
A \arrow[loop left, "\mathrm{id}_A"] \arrow[r, "f"] & B \arrow[loop right, "\mathrm{id}_B"]\,.
\end{tikzcd}$$

This means that $f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f$.

Let us see some trivial examples. In set theory, we might start with the empty set and singleton sets as the simplest examples. The analogues for categories are the *empty category*, **0**, and *singleton category*, **1**.

DEFINITION B.1-2. The empty category, **0**, has no objects and therefore no arrows. The category can be depicted as

$$\mathbf{0} := \boxed{} .$$

DEFINITION B.1-3. The ***singleton category***, **1**, contains a single object and a single arrow. Each object in any category has an identity arrow. Since **1** only has the single arrow, the lone arrow must be the identity arrow. The category **1** can be illustrated graphically as:

$$\mathbf{1} := \boxed{\begin{tikzcd} () \arrow[loop right, "\mathrm{id}_{()}"] \end{tikzcd}} ,$$

with the lone object represented as an empty tuple as we do with singleton sets. Since identity arrows are axiomatically required, they are customarily omitted from diagrams:

$$\mathbf{1} := \boxed{()} .$$

This is also a trivial example of a ***discrete category***, in which all arrows are identity arrows.

NOTATION B.1-4. Arbitrary categories will be labelled with a sans-serif bold-italic uppercase Latin letter: $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ and so on. Established categories will be denoted in boldface with initial-case, such as in **$et**, the category of sets and functions, which will be defined later.

NOTATION B.1-5 (Category-objects)**.** Given a category $\boldsymbol{C}$, any object $A \in \operatorname{ob} \boldsymbol{C}$ is called a $\boldsymbol{C}$-object. We may simply write $A \in \boldsymbol{C}$.

NOTATION B.1-6 (Category-arrows)**.** The collection of arrows between $\boldsymbol{C}$-objects A and B is denoted $\boldsymbol{C}(A, B)$. Any arrow in $\boldsymbol{C}$ is called a ***C*-*arrow***. When we want to refer to arrows without reference to their category, we can write $\hom(A, B)$.

The associative binary operation on arrows, denoted with the ring operator ($\circ$) in DEF. B.1-1, is sometimes at odds with clarity. This is because the arrow notation,

$$A \xrightarrow{f} B \xrightarrow{g} C,$$

reads contralaterally to the argument order of the ring operator: $g \circ f$.

NOTATION B.1-7 (Reversed composition operator, $\mathbin{⨟}$)**.** Given two compatible arrows f, g, their composite $g \circ f$ may be written as $f \mathbin{⨟} g$ when it benefits clarity.

The following is the associativity diagram from C-2, rephrased with the reversed operator:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "f⨟g⨟h"'] \arrow[dr, "f⨟g"] & B \arrow[d, "g"] \arrow[dl, "g⨟h"'] \\
D & C \arrow[l, "h"] .
\end{tikzcd}$$

I suspect the reader may find that diagram generates a little less cognitive friction when following the arrows around.

B.1-8 ORDINALS AND LINEAR ORDERS AS CATEGORIES Linear orders are posets with the additional property that for any two elements a, b if $a \neq b$ then either $a \leq b$ or $b \leq a$. For any $n \in \mathbb{N}$, the n-th ***linear order*** consists of the pair $(n, \leq)$ of the n-th enumeration set with the usual linear ordering on integers. Illustrated as a category,

$$\boxed{0 \xrightarrow{\leq} 1 \xrightarrow{\leq} 2 \xrightarrow{\leq} 3 \xrightarrow{\leq} \cdots \xrightarrow{\leq} n-1}\ .$$

Any finite linear order is of the preceding form, and is therefore isomorphic[1] to $(n, \leq)$ for some n.

The collection of objects in a category may either be a set or a proper class, and so too for the collection of arrows. If those collections are both proper classes, the category is called a *large category.* If both are sets, then it is a ***small category***. If a category has a proper class of objects, but for

[1] DEF. B.2-3 defines isomorphism, or read as *bijective* for now.

any objects A, B, hom(A, B) is a set, then the category is ***locally small***. Unless there is specific emphasis on size, the arrows in any hom is called a ***hom-set***.

B.1.1 The (or a) Category of Sets & Functions

Perhaps the most venerable category in all of CT, the category of sets and functions is the setting for most of the present thesis.

DEFINITION B.1-9 (The category of sets and functions)**.** Denoted $\mathbb{S}$**et**, the category has
objects: the class of all sets
arrows: all functions from each set into another.
The identity on a given set X is the *identity function*:

$$\begin{aligned} \mathrm{id}_X :\ & X \to X \\ & x \mapsto x. \end{aligned}$$

Functions have set-theoretical domains and codomains which agree perfectly with the notion of domain and codomain of categorical arrows. Furthermore, function composition provides the composition operation on arrows.

B.1-10 $\mathbb{S}$**et** is a substrate for numerous others. By adding structure to the sets and restricting the morphisms to those that preserve the structure, we get categories of groups, posets, rings and so on. A map exists between these categories and $\mathbb{S}$**et** that *forgets* the additional structure, allowing them to reveal their inner character as a play of shadow in $\mathbb{S}$**et**. In a sense, $\mathbb{S}$**et** can be seen as a sort of minimal structure from which algebraic objects may be cultivated. Intuitively, this has something to do with the spirit of set theory as the foundational system for mathematics. By studying sets in the context of CT, we obtain deeper insights that carry across boundaries of mathematical theories—the relationships deep enough to transcend the internal details to which CT is so apathetic.

This does have some limitations. Without being able to inspect the labels of the elements, any two sets with the same cardinality may as well be the same. For this reason, many constructions in set will only be unique up to bijection. Later, we will define *products* in $\mathbb{S}$**et** which is in part the Cartesian product of sets. From a category theoretical perspective, the product $A \times B$ of sets A and B can be any set that has cardinality $(\mathrm{card}\, A)(\mathrm{card}\, B)$—CT cannot tell us the difference. Certainly, the traditional $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$ minimizes cognitive overhead, and nothing stops us from imagining it in place, while being aware that is not a unique representation of the data.

* * *

Notice that DEF. B.1-1 and all of the subsequent notations completely avoid any tooling that might give a view into the objects. In a category, objects are like points in a space: they have no internal life of their own. Instead have interrelationships to other points. Like Ernest Rutherford revealing the structure of the atom by indirect observation of scattered rays, category theorists do not need direct means to measure the inner structure of the objects that make up their worlds. Instead of peering inward as with a microscope, they gaze outward as with a telescope to the constellation of objects and the arrows connecting them. This is why categories take their meaning from the context of application, and why they are a basis to reveal patterns across those contexts.

B.1-11 It is perhaps not too early to give a taste of this telescopic view. So called *global elements* [2] will play an important role in the thesis: specifically, global elements in $\mathbf{\$et}$. In §B.3 the notion of *terminal objects* will be properly introduced. These are objects in a category with a unique arrow from every other object in the category terminating there. In $\mathbf{\$et}$, any singleton (or single-element) set has that property.

B.1-12 (Singleton with empty tuple). It is standard in CT to name terminal objects 1. In $\mathbf{\$et}$, singletons are terminal. For notational simplicity, we often name the single element (), so $1 = \{()\}$. This collides with NTN. 3.2-1 for enumeration sets which says $1 = \{0\}$. We tolerate it as a corner case because the empty tuple is superior in application of CT to software.

Let us take $1 = \{()\} \in \mathbf{\$et}$ as our singleton. While any set A has only a single function in $\mathbf{\$et}(A, 1)$, turning the spotlight around, $\mathbf{\$et}(1, A)$ is bijective with A itself. That is, each choice of function in $\mathbf{\$et}(1, A)$ is equivalent to a choice of and element of A. This fact is so well used in the thesis that is convenient to have a special notation to make a mechanism of it:

[2] See §B.4, p. 272.

NOTATION B.1-13. (Global Set-Elements) Given a set A, for each $a \in A$ the function $\vec{a}: 1 \to A$ (with the arrow over the element label) is the function such that $\vec{a}() = a$. So $\vec{\pi} : 1 \to \mathbb{R}$ gives $\vec{\pi}() = 3.14159\ldots$.

The next three sections will formalise these concepts, allowing us to properly generalise.

B.2 EPIMORPHISM, MONOMORPHISM & ISOMORPHISM

Stage the following definitions in an arbitrary category, $\boldsymbol{C}$. Also consider arbitrary $\boldsymbol{C}$-objects A, B, C and $\boldsymbol{C}$-arrows g, h, i, j related as follows.

DEFINITION B.2-1 (Monomorphism)**.** An arrow $f : A \to B$ is a ***monomorphism*** (denoted $f : A \hookrightarrow B$) if, for all other C and all g, h as in

$$\begin{tikzcd} C \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & A \arrow[r, "f"] & B \end{tikzcd} , \tag{B.2.1}$$

it holds that $f \circ g = f \circ h$ implies $g = h$.

In $\mathbf{Set}$, in order for g and h to be truly arbitrary functions and to ensure that $f \circ g = f \circ h$ implies $g = h$, f must be *injective.* Of course, injections do not map more than one element of the domain to any given element of the codomain. If there exists an $b \in B$ such that the preimage through f has more than one element then we could find witnesses g and a h that would differ only in that preimage, giving $fg = fh$ while $g \neq h$.

DEFINITION B.2-2 (Epimorphism)**.** An arrow $f : A \to B$ is a ***epimorphism*** (denoted $f : A \twoheadrightarrow B$) if, for all other C and all g, h as in

$$\begin{tikzcd} A \arrow[r, "f"] & B \arrow[r, shift left, "g"] \arrow[r, shift right, "h"'] & D \end{tikzcd} ,$$

it holds that $gf = hf$ implies $g = h$.

This time in $\mathbf{Set}$, in order for g and h to be truly arbitrary functions and to ensure that $gf = hf$ implies $g = h$, f must be *surjective.* Of course, the image of a surjection covers its entire codomain. If it did not, then we could find witnesses g, h that differ in their mapping of elements outside of the image of f, but give $gf = hf$.

Epimorphisms and monomorphisms are also called “epis” and “monos”. An arrow may be described as “epic” or “monic”.

Isomorphisms, sometimes simply called "isos" are a stricter but simpler idea.

DEFINITION B.2-3. An arrow f is an isomorphism, denoted $f : A \xrightarrow{\sim} B$, if there exists another arrow $g : B \to A$ such that

$$g \circ f = \mathrm{id}_A \text{ and } f \circ g = \mathrm{id}_B.$$

When an isomorphism exists between two objects, they are said to be ***isomorphic***, denoted symbolically as $A \cong B$.

In $\mathbb{S}$**et**, isomorphisms correspond to bijections, where the g is simply denoted f^{-1}. Bijections are functions which are simultaneously injective and surjective.

In general, all isos are epic and monic. The converse is not generally true: epi-monos are not necessarily isos—but in $\mathbb{S}$**et** they are.

B.3 INITIAL/TERMINAL OBJECTS

DEFINITION B.3-1 (Initial & terminal objects)**.** Consider an arbitrary category $\boldsymbol{C}$. A $\boldsymbol{C}$-object 0 is ***initial*** if, for all $A \in \boldsymbol{C}$, there exists a unique arrow $0 \xrightarrow{0_A} A$. Dually, an object 1 is ***terminal*** if, for all $A \in \boldsymbol{C}$, there exists a unique arrow $A \xrightarrow{!_A} 1$.[3] An object that is both initial and terminal is called a ***zero object***.

When a category has these objects, the objects are unique up to isomorphism.

As we saw in ‹B.1-11›, singleton sets are terminal in $\mathbb{S}$**et**. There are infinitely many singletons, but they are all isomorphic. The empty set, $0 = \{\ \}$ is initial in $\mathbb{S}$**et**, and it happens to be unique. (There is no content in 0 to vary.)

There are even initial and terminal categories.

DEFINITION B.3-2. The category $\mathbb{C}$**at** has
objects: all *small* categories,
arrows: functors among the categories.

Analogous to $\mathbb{S}$**et**, the initial object is the empty category, **0**, and any **1** is terminal. Though we have yet to discuss what a map $\mathbf{1} \to \boldsymbol{C}$ might look like.

[3] Morphisms to terminal objects or from initial objects are often called !, possibly emphasising their uniqueness as in the notation ∃! meaning "there exists a unique…".

B.4 GLOBAL ELEMENTS

In $\mathbf{Set}$, there are is an isomorphism for each $\mathbf{Set}$-object A:

$$A \cong \mathbf{Set}(1, A)$$

owing to the fact that each function in the hom-set picks an element from A. These what we called global elements of A, back in ‹B.1-11›. For each $a \in A$, its global element is denoted with a small arrow: $\vec{a}$ (NTN. B.1-13).

Two functions, $f, g : A \to B$ are equal if, for each $\vec{a} \in \mathbf{Set}(1, A)$, $f \circ \vec{a} = g \circ \vec{a}$. In light of this, reconsider monos as injections. In $\mathbf{Set}$, instead of considering the entire category as we did in (B.2.1), it is sufficient to inspect an arbitrary arrow $f : A \to B$ with respect to the global elements:

$$\begin{tikzcd} 1 \arrow[r, shift left, "\vec{a}_1"] \arrow[r, shift right, "\vec{a}_2"'] & A \arrow[r, "f"] & B . \end{tikzcd}$$

If for all $\vec{a}_1, \vec{a}_2 \in \mathbf{Set}(1, A)$, $f\,\vec{a}_1 = f\,\vec{a}_2$ if an only if $\vec{a}_1 = \vec{a}_2$, then f is clearly injective. It reads almost exactly as the typical set-theoretical definition.

This may all seem like a way to smuggle set-theory into CT, but this is only because we are focusing in the category of sets. But it should not be surprising that the structure of the objects will be manifest in the arrows of the category.

B.5 DUALITY AND OPPOSITE CATEGORIES

Proofs in CT have a twofold economy for effort. Any category theoretical construct that is based on statements about objects, arrows and the tools and axioms provided in DEF. B.1-1 has a dual construct. These duals arise by formal reversal of arrows. We have already seen some examples. Monos are the duals of epis, and terminal objects are dual to initial objects.

DEFINITION B.5-1. Every category $\boldsymbol{C}$ has an opposite category $\boldsymbol{C}^{\mathrm{op}}$ where they share the same objects and arrows but all of $\boldsymbol{C}^{\mathrm{op}}$'s arrows are formally reversed: $\boldsymbol{C}(A, B) = \boldsymbol{C}^{\mathrm{op}}(B, A)$.

The dual of everything true in $\boldsymbol{C}$ is true in $\boldsymbol{C}^{\mathrm{op}}$. Each arrow f in $\boldsymbol{C}$ has a counterpart f^{op} in $\boldsymbol{C}^{\mathrm{op}}$ where $\operatorname{dom} f = \operatorname{cod} f^{\mathrm{op}}$ and $\operatorname{cod} f = \operatorname{dom} f^{\mathrm{op}}$. Moreover,

- if f is monic then f^{op} is epic,

- if f is an isomorphism so it is in f^{op},
- terminal objects in $\boldsymbol{C}$ are initial objects in $\boldsymbol{C}^{\mathrm{op}}$ and vise-versa.

The dual of a construct is often named with the prefix "co-".

The *duality principle* (see [81, Ch. II]) implies that any proof is likewise a proof of the dual concept. If you have developed a theorem, you need not provide proof of the dual theorem.

The structure of the theory presented herein is based on four key dualities,

- algebras and coalgebras,
- catamorphism and anamorphism,
- control and estimation, and
- observable and iterable structures.

Catamorphism and anamorphism form the core of the control and estimation software models. But to find and formalise these structures, we must map in and among categories themselves. As functions relate sets in and among themselves, *functors* relate categories.

B.6 FUNCTORS

Functors are structure preserving maps between categories. Since the data of a category consists of two collections, a functor embodies a map for each.

DEFINITION B.6-1. Given categories $\boldsymbol{C}$ and $\boldsymbol{D}$, a functor F, mapping the first to the second is denoted

$$\mathsf{F} : \boldsymbol{C} \to \boldsymbol{D},$$

and consists of

- an *object map* assigning to each $\boldsymbol{C}$-object a $\boldsymbol{D}$-object,
- an *arrow map* assigning to each $\boldsymbol{C}$-arrow a $\boldsymbol{D}$-arrow,

preserving categorical structure by the following axioms:

F-1: (Functors preserve composition) For any composite $\boldsymbol{C}$-arrow, $g \circ f$,

$$\mathsf{F}(g \circ f) = \mathsf{F}g \circ \mathsf{F}f,$$

F-2: (Functors preserve identity) For any $\boldsymbol{C}$-object X,

$$\mathsf{F}\mathrm{id}_X = \mathrm{id}_{\mathsf{F}X}.$$

NOTATION B.6-2. Arbitrary functors will be denoted by uppercase italic sans-serif Latin letters: $\mathsf{F}, \mathsf{G}, \mathsf{H}$, and so on.

Functors come in covariant and contravariant types. The difference is that a contravariant functor reverses the arrows in the image of the functor. This will be covered in more detail later. For now, we focus on covariant functors.

The following diagram depicts the structure preserving nature of covariant functors:

$$\begin{tikzcd}
\boldsymbol{D} & FA \arrow[r, "Ff"] \arrow[loop above, "\mathrm{id}_{FA}"] \arrow[rr, bend left=50, "F(gf)"] & FB \arrow[r, "Fg"] \arrow[loop above, "\mathrm{id}_{FB}"] & FC \arrow[loop above, "\mathrm{id}_{FC}"] \\
\boldsymbol{C} \arrow[u, "F"] & A \arrow[r, "f"'] \arrow[loop below, "\mathrm{id}_A"] \arrow[rr, bend right=50, "gf"'] & B \arrow[r, "g"'] \arrow[loop below, "\mathrm{id}_B"] & C . \arrow[loop below, "\mathrm{id}_C"]
\end{tikzcd}$$

The bottom half is an amalgam of the diagrams of category axioms C-1 and C-2, and the top half is the image of the diagram through the functor. The top and bottom are not connected because they live in two different categories.[4] It is the small diagram on the left showing $\boldsymbol{C} \xrightarrow{F} \boldsymbol{D}$ that connects the top to bottom.

To define a functor is to define two functional maps. That inspires the following notation.

NOTATION B.6-3. We extend the traditional notation for defining functions $f : A \to B; a \mapsto fa$ to functors by adding a brace encompassing the two maps:

$$F : \boldsymbol{C} \to \boldsymbol{D}$$
$$\begin{cases} X \mapsto FX \\ f \mapsto Ff \end{cases}$$

B.6.1 Diagrams in a Category

B.6-4 CHOOSING AN OBJECT Passage ‹B.1-11› describes how maps from a singleton set to a given set are in one-to-one correspondence with a

[4] In circles of functional programmers, the functions f and g are commonly said to be "lifted" into the functor as Ff and Fg respectively.

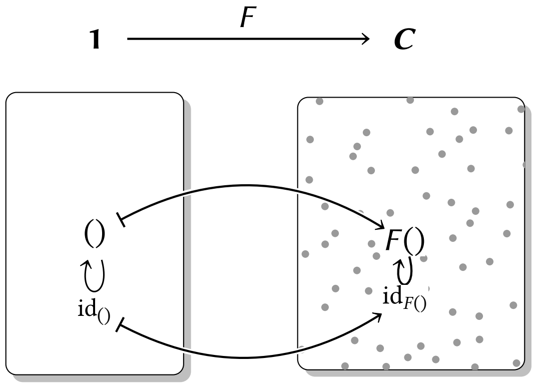


FIGURE B.1: Depicted are Categories **1** (left) and $\boldsymbol{C}$ (right, arrows omitted for clarity, objects are gray dots). Functor F maps () to $F\,() \in \boldsymbol{C}$ and their identities accordingly as per AX. F-2.

choice of elements from that given set. That can be extended for functors as well. Given a category $\boldsymbol{C}$, consider a functor $F : \mathbf{1} \rightarrow \boldsymbol{C}$. The image of **1** in $\boldsymbol{C}$ will consist of a $\boldsymbol{C}$-object and its identity (see FIG. B.1 in the margin). Therefore, a choice of F is equivalent to a choice of a $\boldsymbol{C}$-object: the set of functors $\mathbf{1} \rightarrow \boldsymbol{C}$ is bijective to $\operatorname{ob} \boldsymbol{C}$.

Categories offer much more structure than sets, so perhaps it is not surprising that this idea of choosing an object can be generalised to locate much more complex structures. It turns out that this is a key idea that opens the door to much of CT.

DEFINITION B.6-5 (Diagram)**.** Let $\boldsymbol{I}$ and $\boldsymbol{C}$ be categories. An $\boldsymbol{I}$-shaped ***diagram*** in $\boldsymbol{C}$ is a (covariant) functor $D : \boldsymbol{I} \rightarrow \boldsymbol{C}$. The category $\boldsymbol{I}$ is called the ***index category***.

Notice that if $\boldsymbol{I} = \mathbf{1}$, we recover the notion of ‹B.6-4›. In fact, diagrams are generalisation of indexed families. For example, given a set A and an enumeration set n, an n-indexed family of A-values, $(\, a_i \,)_{i \in n}$, is equivalent to a function of type $n \rightarrow A$.

NOTATION B.6-6 (Indexed family)**.** An n-indexed family of A values is written $(\, a_i \,)_{i \in n}$.

The index category does much more because it also picks out arrows.

In the next section, we study transformations between functors: *natural transformations.* Together with diagrams, they lead to a formal framework for identifying unique or optimal manifestations of a particular structure within a category—a *universal construction.*

B.7 NATURAL TRANSFORMATIONS

DEFINITION B.7-1. Given categories $\boldsymbol{C}$ and $\boldsymbol{D}$ and two functors, $F, G : \boldsymbol{C} \to \boldsymbol{D}$, a ***natural transformation***, $\eta : F \Rightarrow G$ is a family of $\boldsymbol{D}$-arrows,

$$\left(FA \xrightarrow{\eta_A} GA\right)_{A \in \boldsymbol{C}},$$

indexed by the $\boldsymbol{C}$-objects such that for every pair A, B of $\boldsymbol{C}$-objects, and each (A, B)-arrow f, it holds that

$$\eta_B \circ Ff = Gf \circ \eta_A. \tag{B.7.2}$$

The arrows in the family are individually called the ***components of the natural transformation.*** If each component of a natural transformation is an isomorphism, then the natural transformation is called a ***natural isomorphism***.[5]

NOTATION B.7-2. Arbitrary natural transformations will be lowercase italic Greek letters: $\alpha, \beta, \gamma, \ldots$ and so on.

Natural transformations are summarised in the following:

$$\begin{tikzcd}
\boldsymbol{D} & GA \arrow[r, "Gf"] & GB \\
 & FA \arrow[u, "\eta_A"] \arrow[r, "Ff"] & FB \arrow[u, "\eta_B"'] \\
\boldsymbol{C} \arrow[uu, bend left, "F"] \arrow[uu, bend right, "G"'] \arrow[uu, phantom, "\overset{\eta}{\Longrightarrow}"] & A \arrow[r, "f"] & B
\end{tikzcd} \tag{B.7.3}$$

Note how η is represented in the diagram as a double ($\Rightarrow$) between the two functor arrows. The bottom row of the diagram exists in $\boldsymbol{C}$, and the upper rows are in $\boldsymbol{D}$. The functors F and G lift the diagram $A \xrightarrow{f} B$ forming

[5] Isomorphism is defined in §B.2. If the reader is unfamiliar with the notion, it will suffice to think about invertible mappings such as bijections on sets.

the rows of the rectangle in $\boldsymbol{D}$. The components η_A and η_B of the natural transformation connect the rows. The constraint (B.7.2) is called the ***naturality condition*** and it is encoded in the diagram above as the commutativity condition of the square diagram—where there are two paths from FA to GB each with two edges, and both paths must be equivalent. That diagram is called a ***naturality square***.

B.7-3 ιd, THE IDENTITY NATURAL TRANSFORMATION The analogue of the identity function, the identity natural transformation on a functor $F : \boldsymbol{C} \to \boldsymbol{D}$, denoted with a Greek "ι" instead of an "i" (just to be cute): $\iota\mathrm{d}_F$. As a family of functions, it is simply $(\ \mathrm{id}_{FX}\ )_{X \in \boldsymbol{C}}$

Natural transformations are maps, so we should expect them to compose. They give rise to interesting composite structures that interact with functors. These come in two varieties: vertical and horizontal composition. A special case of horizontal composition called *whiskering* is also important.

B.7-4 ***Vertical composition*** is the most straightforward. Given categories $\boldsymbol{C}, \boldsymbol{D}$ and functors $F, G, H : \boldsymbol{C} \to \boldsymbol{D}$, two natural transformations $\eta : F \Rightarrow G$ and $\vartheta : G \Rightarrow H$ may be composed so that

$$\begin{tikzcd} \boldsymbol{C} \arrow[r, bend left=60, "F"] \arrow[r, "G" description] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, shift left=3, "\Downarrow\eta"] \arrow[r, phantom, shift right=3, "\Downarrow\vartheta"] & \boldsymbol{D} \end{tikzcd} \qquad \text{becomes} \qquad \begin{tikzcd} \boldsymbol{C} \arrow[r, bend left=60, "F"] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, "\Downarrow\vartheta\eta"] & \boldsymbol{D} \end{tikzcd}.$$

This composition is performed componentwise, using the underlying composition of arrows in $\boldsymbol{D}$:

$$(\vartheta\eta)_X = \vartheta_X \circ \eta_X \quad \text{for all} \quad X \in \mathrm{ob}\,\boldsymbol{C}.$$

The naturality square from (B.7.3), is extended by the composite as

$$\begin{tikzcd} HA \arrow[r, "Hf"] & HB \\ GA \arrow[u, "\vartheta_A"] \arrow[r, "Gf"] & GB \arrow[u, "\vartheta_B"'] \\ FA \arrow[u, "\eta_A"] \arrow[r, "Ff"] & FB \end{tikzcd},$$

where all paths commute. For example, $\eta_A \fatsemi \vartheta_A \fatsemi Hf = \eta_A \fatsemi Gf \fatsemi \vartheta_B$, which is a slightly more involved equation than we have seen in diagrams before now.

This suggests a category!

DEFINITION B.7-5. A ***functor category*** $\mathbf{Fun}(\boldsymbol{C}, \boldsymbol{D})$ consists of

Objects: the collection of all functors from $\boldsymbol{C}$ to $\boldsymbol{D}$, and

Arrows: natural transformations among the functors.

The identity arrows are the identity natural transformations ιd, and arrows are closed under vertical composition.

In these functor categories, because the arrows are natural transformations, they are written with a single arrow, →, instead of the double ⇒.

B.7-6 ***Horizontal composition*** is slightly more intricate, involving a third category $\boldsymbol{E}$ with functors $F', G' : \boldsymbol{D} \to \boldsymbol{E}$. Natural transformations are now $\eta : F \Rightarrow G$ and $\vartheta : F' \Rightarrow G'$. and their horizontal composite is denoted $\eta * \vartheta$. With horizontal composition,

$$\begin{tikzcd}
\boldsymbol{C} \arrow[r, bend left=50, "F", ""{name=U1, below}] \arrow[r, bend right=50, "G"', ""{name=D1}] & \boldsymbol{D} \arrow[r, bend left=50, "F'", ""{name=U2, below}] \arrow[r, bend right=50, "G'"', ""{name=D2}] & \boldsymbol{E} & \text{becomes} & \boldsymbol{C} \arrow[r, bend left=50, "F'F", ""{name=U3, below}] \arrow[r, bend right=50, "G'G"', ""{name=D3}] & \boldsymbol{E} \, .
\arrow[Rightarrow, from=U1, to=D1, "\eta"]
\arrow[Rightarrow, from=U2, to=D2, "\vartheta"]
\arrow[Rightarrow, from=U3, to=D3, "\vartheta * \eta"]
\end{tikzcd} \tag{B.7.4}$$

Each object $X \in \operatorname{ob} \boldsymbol{C}$ gives rise to the following naturality square in $\boldsymbol{D}$, the diagonal of which is the component of $\vartheta * \eta$ at X:

$$\begin{tikzcd}
F'F\,X \arrow[r, "F'\eta_X"] \arrow[d, "\vartheta_{FX}"'] \arrow[dr, "(\vartheta * \eta)_X"] & F'G\,X \arrow[d, "\vartheta_{GX}"] \\
G'F\,X \arrow[r, "G'\eta_X"'] & G'G\,X \, .
\end{tikzcd} \tag{B.7.5}$$

This yields two formulas for the composite, one for each non-diagonal path from $F'F\,X$ to $G'G\,X$:

$$(\vartheta * \eta)_X = \vartheta_{FX} \fatsemi G'\eta_X \quad \text{or} \quad G'\eta_X \circ \vartheta_{FX}, \tag{B.7.6i}$$

and

$$= F'\eta_X \mathbin{;} \vartheta_{GX} \quad \text{or} \quad \vartheta_{GX} \circ F'\eta_X, \tag{B.7.6ii}$$

which are equivalent as a consequence of naturality.

B.7-7 LEFT- AND RIGHT-WISKERING Special cases of horizontal composition, were the left or right side is ıd, gives rise to an operation called left and right ***whiskering***.

Starting with (B.7.4) and letting $G = F$ and $\eta = \imath\mathrm{d}_F$ gives

$$\begin{tikzcd} \boldsymbol{C} \arrow[r, "F"] & \boldsymbol{D} \arrow[r, bend left=50, "F'"{name=U}] \arrow[r, bend right=50, "G'"'{name=L}] \arrow[Rightarrow, from=U, to=L, "\vartheta"] & \boldsymbol{E} & \text{becoming} & \boldsymbol{C} \arrow[r, bend left=50, "F'F"{name=U2}] \arrow[r, bend right=50, "G'F"'{name=L2}] \arrow[Rightarrow, from=U2, to=L2, "\vartheta F"] & \boldsymbol{E}\,. \end{tikzcd}$$

The natural transformation ϑF is called the ***left whiskering*** of ϑ and F, appearing, perhaps oddly, as the composite of a functor and a natural transformation. The components $(\vartheta F)_X$ for $X \in \operatorname{ob} \boldsymbol{C}$ can be derived from either (B.7.6ɪ) or (B.7.6ɪɪ) by substituting $G = F$ and $\eta = \imath\mathrm{d}_F$:

$$\begin{aligned} & (\vartheta F)_X \coloneqq (\vartheta * \imath\mathrm{d}_F)_X \\ = \quad & \{\text{substituting (B.7.6ɪ) and } \eta_X = \mathrm{id}_X\} \\ & G'\mathrm{id}_X \circ \vartheta_{FX} \\ = \quad & \{G'\mathrm{id}_X = \mathrm{id}_{G'X} \text{ by axiom F-2 and can be factored out by C-3.}\} \\ & \vartheta_{FX} \end{aligned}$$

So

$$(\vartheta F)_X = \vartheta_{FX} : F'F\,X \to G'F\,X.$$

Starting again from (B.7.4) and this time letting $G' = F'$ and $\vartheta = \imath\mathrm{d}_{F'}$ gives

$$\begin{tikzcd} \boldsymbol{C} \arrow[r, bend left=50, "F"{name=U}] \arrow[r, bend right=50, "G"'{name=L}] \arrow[Rightarrow, from=U, to=L, "\eta"] & \boldsymbol{D} \arrow[r, "F'"] & \boldsymbol{E} & \text{becoming} & \boldsymbol{C} \arrow[r, bend left=50, "F'F"{name=U2}] \arrow[r, bend right=50, "F'G"'{name=L2}] \arrow[Rightarrow, from=U2, to=L2, "F'\eta"] & \boldsymbol{E}\,. \end{tikzcd}$$

The natural transformation $F'\eta$ is called the ***right whiskering*** of F' and η and as with the left case, the components can be derived from either (B.7.6ɪ) or (B.7.6ɪɪ) by substitution and simplification giving

$$(F'\eta)_X = F'\eta_X : F'F\,X \to F'G\,X.$$

Interplay between vertical and horizontal composition is governed by the ***interchange law***. Given three categories and functors with natural transformations related as

$$\begin{tikzcd}
\boldsymbol{C} \arrow[r, bend left=60, "F"] \arrow[r, ""] \arrow[r, bend right=60, "G"'] \arrow[r, phantom, shift left=3.5ex, "\Downarrow \eta"] \arrow[r, phantom, shift right=3.5ex, "\Downarrow \vartheta"] & \boldsymbol{D} \arrow[r, bend left=60, "F'"] \arrow[r, ""] \arrow[r, bend right=60, "G'"'] \arrow[r, phantom, shift left=3.5ex, "\Downarrow \eta'"] \arrow[r, phantom, shift right=3.5ex, "\Downarrow \vartheta'"] & \boldsymbol{E} ,
\end{tikzcd}$$

the following equivalence holds:

$$(\vartheta' * \vartheta)(\eta' * \eta) = (\vartheta'\eta') * (\vartheta\eta) : \; F'F \to G'G.$$

We can now rewrite (B.7.5) cast in the functor category $\mathbf{Fun}(\boldsymbol{C}, \boldsymbol{E})$ where arrows are natural transformations, including the whiskers:

$$\begin{tikzcd}
F'F \arrow[r, "F'\eta"] \arrow[d, "\vartheta F"'] \arrow[dr, "\vartheta * \eta"] & F'G \arrow[d, "\vartheta G"] \\
G'F \arrow[r, "G'\eta"'] & G'G .
\end{tikzcd}$$

This cleaner style is free of componentwise tedium.

B.8 LIMITS & COLIMITS

Co/limits of diagrams reveal single objects within a category that individually capture some aspect of a relationship or substructure among other objects—a sort of single-object embodiment of a structure.

Let $\boldsymbol{I}$ be the discrete category with $\operatorname{ob}\boldsymbol{I} = \{ 1, 2 \}$ and consider the diagram

$$D_{A,B} : \boldsymbol{I} \to \mathbf{Set}$$
$$\begin{cases} 1 \mapsto A; \; 2 \mapsto B \\ \mathrm{id}_1 \mapsto \mathrm{id}_A; \; \mathrm{id}_2 \mapsto \mathrm{id}_B. \end{cases}$$

Because the index category contains only two objects (and their identity arrows) this diagram "chooses" two fixed sets, A and B, in $\mathbf{Set}$. Now, entertain what may seem a strange question: *could a single object in* $\mathbf{Set}$ *embody* *the structure of the image?* In this simple case, that would entail a single set C that somehow embodies the data of both A and B. To make the relationship complete, we would need morphisms relating C with A and B. This can go one of two ways.

1. Morphisms from C to each of A and B—functions to "unpack" C elements, giving an A and a B:

   $$\begin{tikzcd} A & C \arrow[l, "p_1"'] \arrow[r, "p_2"] & B . \end{tikzcd}$$

   A diagram of this particular shape is called a ***span*** and C is called the centre of the span.
2. Morphisms from each of A and B to C—functions embedding the elements of A and B into C, giving elements of C a quality of either A or B:

   $$\begin{tikzcd} A \arrow[r, "i_1"] & C & B \arrow[l, "i_2"'] . \end{tikzcd}$$

   A diagram of this particular shape is called a ***cospan*** and C is the centre of the cospan.

Compare the span (item 1) with the binary Cartesian product of sets, along with canonical projections π_1 and π_2:

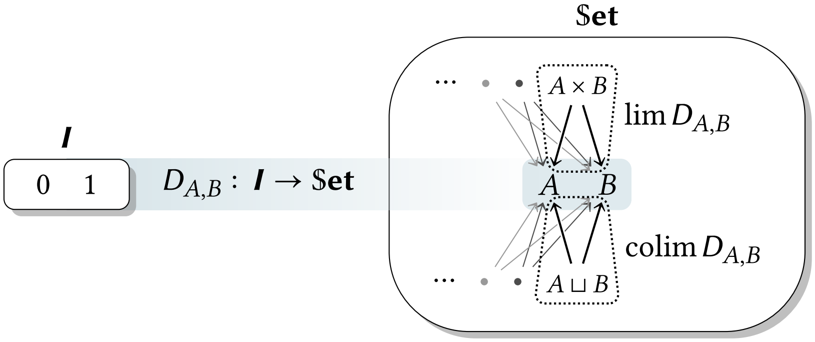
$$\begin{tikzcd} A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B . \end{tikzcd}$$

There is no coincidence here. The elements of the Cartesian product set are an optimal simultaneous embodiment of the elements of "*A and B*," for some definition of "optimal." Limits of diagrams formalise this notion of optimality, and the Cartesian product is the categorical *product* in $\mathbb{S}$**et**. Dually, the colimit of $D_{A,B}$ gives the "optimal" cospan (item 2) embodying "*A or B*," for some definition of "optimal." This leads to the dual of the Cartesian product—the disjoint union of sets:

$$\begin{tikzcd} A \arrow[r, "\iota_1"] & A \sqcup B & B \arrow[l, "\iota_2"'] . \end{tikzcd}$$

The disjoint union, in relation to the Cartesian product, is called the *coproduct* in $\mathbb{S}$**et**.

Of course, there are infinitely many spans and cospans in $\mathbb{S}$**et**, with many choices for their central objects and arrows. The co/limit constructions privilege the "optimal" ones in a way that will become clear once those concepts are properly defined.



𝕊**et**
I
0 1
$D_{A,B} : I \to \mathbb{S}\mathbf{et}$
$A \times B$
$\lim D_{A,B}$
A B
$\operatorname{colim} D_{A,B}$
$A \sqcup B$

FIGURE B.2: Objects A and B are chosen in $\mathbb{S}\mathbf{et}$ by $D_{A,B}$. Spans over and cospans under $A, B \in \mathbb{S}\mathbf{et}$ (grey dots and arrows) are numerous. Cartesian product and disjoint union highlighted as the limit and colimit respectively.

Before introducing rigour, it is useful to step back and recapitulate the steps taking us from the two-object index category to the product and coproduct in $\mathbb{S}\mathbf{et}$. Illustrated in FIG. B.2, the two object category casts an image in $\mathbb{S}\mathbf{et}$ through the diagram $D_{A,B}$, fixing the objects A and B as the target of the construction. Any of the gray dots $\bullet \in \mathbb{S}\mathbf{et}$ from FIG. B.2 (along with their co/spanning arrows) may be candidates for the role of single-object embodiment. The span centred on the Cartesian product with canonical projections is the limit, $\lim D_{A,B}$, and the cospan centred on the disjoint union with canonical injections is the colimit, $\operatorname{colim} D_{A,B}$. The co/limit construction allows us to pick the true categorical products and coproducts out of the infinity of spans and cospans in the category.

This intuitive characterisation generalises to diagrams of various shapes like this:

- limits tend to collect aspects of their diagram into a structures with sequential order.
- colimits tend to collect aspects of their diagram into a unification that requires selection on the basis of membership.

REMARK B.8-1. Computer scientists will readily recognise the contrast of the previous itemization as the distinction of

- “sequence *vs.* selection”, or
- “parallel *vs.* sequential” execution.

Both are correct. They are distinguished only technically, by the duality of memory and time.

REMARK B.8-2. (Co)limits of variously shaped diagrams give interesting structures. For examples, Initial and terminal objects are the limit and colimit (respectively) of the single-object discrete diagram, and pullbacks and pushouts are generated as limits of the diagram of a span and cospan respectively.

REMARK B.8-3. The foregone discussion has focussed on diagrams in $\mathbf{Set}$, but concepts like tupeling and unions make no sense in many categories, where there are no underlying elements in the objects. The product in a poset category is the *greatest lower bound* operator [109, §2.5] and dually the coproduct is the *lest upper bound* [109, §3.2].

* * *

There are several equivalent ways to define (co)limits. I choose a mainstream approach in this section drawing inspiration from of [130], [81] and [109].[6] We have already assembled some of the concepts we need: diagrams, index categories, natural transformations and initial/terminal objects. Limits and colimits will be defined in terms of these things, plus one additional notion: (co)cones.

DEFINITION B.8-4 (Cone)**.** Given a category $\boldsymbol{C}$, a small category $\boldsymbol{I}$ called the index category and a diagram $D : \boldsymbol{I} \to \boldsymbol{C}$, a ***cone on*** $\boldsymbol{D}$ is a pair $\left(A, \Delta_A \overset{\lambda}{\Longrightarrow} D\right)$ of

- A, a $\boldsymbol{C}$-object called the ***cone-point***,
- λ, a family of morphisms from the cone-point to the objects in the image of D thought of as a natural transformation from the constant functor Δ_A to the diagram.

The naturality condition on λ enforces commutativity of the diagrams

$$\begin{tikzcd}
 & A \arrow[dl, "\lambda_I"'] \arrow[dr, "\lambda_J"] & \\
DI \arrow[rr, "Df"'] & & DJ \text{ ,}
\end{tikzcd} \tag{B.8.7}$$

for all $I, J \in \boldsymbol{I}$ and $f \in \boldsymbol{I}(I, J)$. The diagrams (B.8.7) are called the *faces* of the cone.

[6] Spivak takes a more structured approach in [133] that I personally favour, however it requires some tools that would otherwise go unused in the thesis.

At the beginning of the section, the Cartesian product was used to introduce the notion of a limit. In that example, illustrated in FIG. B.2, spans over the sets A and B are cones.

B.8-5 CONE FACES In the progression of finite categories, we have already seen $\mathbf{0}$ and $\mathbf{1}$ (DEF. B.1-2 and DEF. B.1-3).

B.8-6 . The category $\mathbf{3}$ is depicted as

$$\mathbf{3} := \boxed{\begin{tikzcd} X \arrow[rr, "f_{XY}"] & & Y \arrow[dl, "f_{YZ}"] \\ & Z \arrow[ul, "f_{ZX}"] & \end{tikzcd}}\,,$$

where the objects and arrows have been labelled for use in this example.

Let $D : \mathbf{3} \to \boldsymbol{C}$ be a diagram of $\mathbf{3}$ in an arbitrary category $\boldsymbol{C}$. A cone on D, (A, λ), looks like this:

$$\begin{tikzcd} & A \arrow[dl, "\lambda_X"'] \arrow[dr, "\lambda_Y"] \arrow[dd, "\lambda_Z"] & \\ DX \arrow[rr, "f_{XY}"] & & DY \arrow[dl, "f_{YZ}"] \\ & DZ \arrow[ul, "f_{ZX}"] & \end{tikzcd} \quad (B.8.8)$$

where the image of D has been faded grey for clarity and the constituents of the pair (A, λ) are drawn black.[7] Each triangular face of the cone must commute.

Since a cone is a constellation of objects and arrows in the codomain of a diagram, the arrows of the codomain can map between cones if they preserve the cone structure.

DEFINITION B.8-7 (Cone morphism)**.** Given two cones (A, α), (B, β) over a diagram $D : \boldsymbol{I} \to \boldsymbol{C}$, a morphism $f : (A, \alpha) \to (B, \beta)$ between the cones is a $\boldsymbol{C}$-arrow $f : A \to B$ such that the following diagram commutes for each $J \in \boldsymbol{I}$:

$$\begin{tikzcd} A \arrow[rr, "f"] \arrow[dr, "\alpha_J"'] & & B . \arrow[dl, "\beta_J"] \\ & DJ & \end{tikzcd} \quad (B.8.9)$$

[7] Roughly speaking, if one imagines a sort of *volume of rotation* swept by the arrows λ as the triangular base is rotated, the motivation behind the name *cones* is apparent.

Having a collection of cones over a diagram and morphisms among them suggests a category.

DEFINITION B.8-8 (Cone category)**.** A diagram D induces a category $\mathbf{Cone}_D$ with

objects: cones over D
arrows: cone morphisms.

A limit then will stand out as a special object the category of cones.

DEFINITION B.8-9. A ***limit*** of a diagram D, denoted $\lim D$, is a terminal object in $\mathbf{Cone}_D$. This is a so-called ***universal cone***.

Perhaps a good way to visualise this is by taking a cross-section of $\mathbf{Cone}_D$ in $\boldsymbol{C}$ along a single face as in (B.8.7). Connecting the cones, we will also have the commuting triangles from (B.8.9). For a diagram $D : \boldsymbol{I} \to \boldsymbol{C}$ and for each $I, J \in \boldsymbol{I}$, the corresponding faces of the cones in $\mathbf{Cone}_D$ (with cone points • and terminal object $\lim D$) form the following series of nested commuting triangles with cone morphisms drawn red:

$$\begin{tikzcd}
 & \bullet \arrow[ddddl, bend right] \arrow[ddddr, bend left] \arrow[ddd, bend right, red] & \\
 & \vdots & \\
 & \bullet \arrow[ddl, bend right] \arrow[ddr, bend left] \arrow[d, red] & \\
 & \lim D \arrow[dl, "\lambda_I"'] \arrow[dr, "\lambda_J"] & \\
DI \arrow[rr, "Df"'] & & DJ\,.
\end{tikzcd} \tag{B.8.10}$$

Intuitively, $\lim D$ can be seen as the purest, simplest, or “closest” cone. Any other cone (A, α) has a morphism $f : A \to \lim D$ such that for all $I, J \in \boldsymbol{I}$, it holds that $\alpha_I = \lambda_I \circ f$. In this sense, all other cones are said to “factor” through $\lim D$. Compare that to the informal illustration for the opening example of the section, Figure B.2.

B.8-10 Aside from the product, we have seen another example of a limit: terminal objects (§B.3). The index category for the construction of a terminal object is $\mathbf{0}$, the empty category[8]:

$$D : \mathbf{0} \to \boldsymbol{C}$$

[8] For empty category, see DEF. B.1-2, p. 266

Clearly, there can be no data in D since its domain is empty: D is only a functor in the trivial sense. The cones over the empty diagram are similarly rendered vacant: since there is no image of $\mathbf{0}$ through D then the cone is merely a cone-point: no edges and thus no faces. Put plainly, $\operatorname{ob}\mathbf{Cone}_D = \operatorname{ob}\boldsymbol{C}$, and all of the requisite cone-face relationships are satisfied vacuously. The limit "cone" is the $\boldsymbol{C}$-object to which every other cone-point has a unique morphism. Therefore, the limit over the empty diagram is precisely a terminal object in $\boldsymbol{C}$.[9]

* * *

As cones are to limits, cocones are to colimits. Cocones are cones with the arrows from the cone-point reversed. (That is, a cocone is a cone in $\boldsymbol{C}^{\text{op}}$.)

DEFINITION B.8-11. Consider an arbitrary category $\boldsymbol{C}$, an index category $\boldsymbol{I}$ and a diagram $D : \boldsymbol{I} \to \boldsymbol{C}$. A ***cocone under*** D is a pair $\big(Z, D \overset{\gamma}{\Longrightarrow} \Delta_Z\big)$ of

- Z, a $\boldsymbol{C}$-object called the ***cocone-point***,
- γ, a family of morphisms from the objects in the image of D to the cocone-point that can be thought of as a natural transformation from the diagram to the constant functor Δ_Z.

B.8-12 The cocone analogue of (B.8.8) from B.8-5 (p. 284) is

$$\begin{tikzcd}
 & DZ \arrow[dl, "f_{ZX}"'] \arrow[dd, "\gamma_Z"] & \\
DX \arrow[rr, "f_{XY}"'] \arrow[dr, "\gamma_X"'] & & DY \arrow[ul, "f_{YZ}"'] \arrow[dl, "\gamma_Y"] \\
 & A &
\end{tikzcd}$$

B.8-13 COCONES IN BINARY DISJOINT UNION At the beginning of the section, the disjoint union was used to introduce the notion of a colimit. In that example, illustrated in Figure B.2, cospans under the sets A and B are cocones.

[9] It is a common abuse of notation to identify a limit (which is a cone) with the limit cone-point—but in this unique situation it is correct because cones over the empty diagram *are* just cone-points.

Given two cocones (A,α), (B,β) under a diagram $D : \boldsymbol{I} \to \boldsymbol{C}$, a morphism $f : (A,\alpha) \to (B,\beta)$ between the cones is a $\boldsymbol{C}$-arrow $f : A \to B$ such that the following diagram commutes for each $J \in \boldsymbol{I}$:

$$\begin{tikzcd}
A \arrow[rr, "f"] & & B . \\
& DJ \arrow[ul, "\alpha_J"] \arrow[ur, "\beta_J"'] &
\end{tikzcd}$$

As with the category of cones, a diagram D dually presents a category $\mathbf{Cocone}_D$ of cones under D.

DEFINITION B.8-14. A ***colimit*** of a diagram D, denoted $\operatorname{colim} D$, is an initial object in $\mathbf{Cocone}_D$. This is a so-called ***universal cocone*** that *factors through* all other cocones.

As in (B.8.10), we can look at a cross section of the cocones in $\boldsymbol{C}$:

$$\begin{tikzcd}
DI \arrow[rr, "Df"] \arrow[dr, "\gamma_I"] \arrow[ddr, bend right] \arrow[ddddr, bend right=60] & & DJ . \arrow[dl, "\gamma_J"'] \arrow[ddl, bend left] \arrow[ddddl, bend left=60] \\
& \operatorname{colim} D \arrow[d, red] \arrow[ddd, red, bend right] & \\
& \bullet & \\
& \vdots & \\
& \bullet &
\end{tikzcd} \tag{B.8.11}$$

Compare this to the informal illustration for the opening example of the section, FIG. B.2.

B.8-15 In ‹B.8-10› we see that terminal objects are definable through limits. Since terminal objects are dual to initial objects, it should come as little surprise that initial objects are definable as colimits. They are colimits of the diagram of the empty category:

$$D : \mathbf{0} \to \boldsymbol{C}.$$

The cocones under the empty diagram contain only the cocone-point, so all the requisite cocone-face relationships are vacuously satisfied. The colimit is precisely a $\boldsymbol{C}$-object from which every other cocone-point has a unique morphism. In other words, the colimit of the empty diagram is precisely an initial object.

B.9 PRODUCTS

This section opened with a discussion of products and coproducts in 𝕊**et** to lend intuition. With limits defined we circle around to formally define categorical products in general.

Abstract products in a category $\boldsymbol{C}$ are limits of diagrams of discrete finite categories:

$$\begin{aligned} \dot{\mathbf{2}} &:= \boxed{1 \quad 2}\,, \\ \dot{\mathbf{3}} &:= \boxed{1 \quad 2 \quad 3}\,, \\ \dot{\mathbf{4}} &:= \boxed{1 \quad 2 \quad 3 \quad 4}\,, \\ &\vdots\ . \end{aligned}$$

An n-fold product is determined by the limit of a diagram with the shape of a discrete category with n objects.

NOTATION B.9-1. For $n \in \mathbb{N}$, the n-th ***discrete finite category*** is the category with enumerated objects $1, 2, \ldots, n$, and where the only arrows are identities:

$$\dot{\boldsymbol{n}} := \boxed{1 \quad 2 \quad \cdots \quad n}\,.$$

Let us first reconstruct[10] the simplest such object, the 2-fold (binary) product, before moving to the n-fold case.

B.9-2 Let D be the diagram $D : \dot{\mathbf{2}} \to \boldsymbol{C};\ 1 \mapsto A;\ 2 \mapsto B$. A cone over D with cone-point $C \in \boldsymbol{C}$ is a span,

$$A \xleftarrow{f_A} C \xrightarrow{f_B} B\,.$$

A ***binary product*** in $\boldsymbol{C}$ is a span $(A \otimes B, \{\pi_1, \pi_2\}) = \lim D$. That is, if for all other spans $\big(C, \{\, f_A, f_B \,\}\big)$, there exists a unique morphism $u : C \to A \otimes B$ such that the following diagram commutes:

$$\begin{tikzcd} & C \arrow[dl, "f_A"'] \arrow[d, dashed, "\exists!\ u"'] \arrow[dr, "f_B"] & \\ A & A \otimes B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B\,. \end{tikzcd} \tag{B.9.12}$$

[10] This was done informally and in the specific context of 𝕊**et** in the beginning of this section, ca. p. 280.

Equation (B.9.12) is a specialisation of (B.8.10) to the case of the binary product. Since the commutativity condition of (B.9.12) requires that $\pi_1 u = f_A$ and $\pi_2 u = f_B$, the only information we need to construct u is the pair f_A and f_B. In this sense, an arrow into a product object is reducible to a pair of arrows. In many texts on general CT, u is denoted $\langle f_A, f_B \rangle$, but a more specialised notation will be introduced later. (See the "fan-out operator", NTN. B.9-7.)

Higher order products, multifunctors of the form $A \otimes B \otimes C \otimes \cdots$, are constructed the same way, but on diagrams with more discrete objects. An n-ary or n-fold product is constructed from a diagram of $\dot{\boldsymbol{n}} \to \boldsymbol{C}$.

DEFINITION B.9-3 (n-fold product)**.** Given an arbitrary category $\boldsymbol{C}$ and some $n \in \mathbb{N}_{\geq 2}$, consider a diagram $D : \dot{\boldsymbol{n}} \to \boldsymbol{C}$ that sends $i \in \dot{\boldsymbol{n}}$ to $X_i \in \boldsymbol{C}$. Cones on the diagram are n-spans:

$$C \xrightarrow{f_{X_i}} X_i \qquad \text{for all } i \in \dot{\boldsymbol{n}}.$$

The ***n*-fold product** is universal cone:

$$\lim D = \left(\bigotimes_{j \in \dot{\boldsymbol{n}}} X_j \xrightarrow{\pi_i} X_i \right)_{i \in \dot{\boldsymbol{n}}},$$

This means that for all other cones there exists a unique arrow u making the following diagrams commute:

$$\begin{tikzcd}
C \arrow[d, dashed, "\exists!\, u"'] \arrow[dr, "f_{X_j}"] & \\
\bigotimes_{i \in \dot{\boldsymbol{n}}} X_i \arrow[r, "\pi_j"'] & X_j
\end{tikzcd} \qquad \text{for all } j \in \dot{\boldsymbol{n}}.$$

PROPOSITION B.9-4 ([109, Prop. 2.7, p. 40])**.** Products in a category, when they exist, are unique up to isomorphism.

NOTATION B.9-5. Consider a category $(\boldsymbol{C}, \otimes, 1_{\boldsymbol{C}})$ with countable products and terminal object. In the special cases of iterated products on the same $\boldsymbol{C}$-object, that is, limits of the constant diagrams $\dot{\boldsymbol{n}} \ni n \mapsto X$ for fixed $X \in \boldsymbol{C}$, we enjoy the compact notation:

$$X^{\otimes n} := \bigotimes_{n} X,$$

with $X^{\otimes 0} \cong 1_{\boldsymbol{C}}$, $X^{\otimes 1} \cong X$ and inductively $X^{\otimes n} \cong X^{\otimes(n-1)} \otimes X$.

Intuition from the case $(\boldsymbol{C}, \otimes, 1_{\boldsymbol{C}}) = (\mathbf{Set}, \times, 1)$ is that $X^{\times n}$ the set of all n-tuples of X, which will be made explicit in §B.9.1.

PROPOSITION B.9-6 ([33, Prop. 2.5])**.** In a category with countable products there exists a natural isomorphism:

$$\varpi_X^{m,n} : X^{\otimes m} \otimes X^{\otimes n} \xrightarrow{\sim} X^{\otimes(m+n)}$$

for $m, n \in \mathbb{N}$, called ***consolidation***.

The Greek word for “consolidate” is “παγιώνω”, according to Google. Since π is already used for product projection, I use the variant form of pi, ϖ, for the consolidation of products.

B.9.1 Products in Set reprised

In the beginning of the section we discussed products in $\mathbf{Set}$. From an intuitive but informal approach, we arrived at the Cartesian product. We are now in a position to approach it with the formal definition of a categorical product and see once again that the categorical product is the Cartesian.

Consider a $\mathbf{Set}$-span centered on the singleton 1:

$$\begin{tikzcd} A & 1 \arrow[l, "\vec{a}"'] \arrow[r, "\vec{b}"] & B \end{tikzcd},$$

where A and B are any sets. Global elements reveal the elements of a set, manifesting them as arrows: $\{\, \vec{n} \mid n \in \text{◌} \,\} \cong \hom(1, \text{◌})$.[11] We will use this span of global elements to probe the elements of the categorical product by substituting it into (B.9.12):

$$\begin{tikzcd} & 1 \arrow[dl, "\vec{a}"'] \arrow[d, dashed, "\exists!\, u"'] \arrow[dr, "\vec{b}"] & \\ A & A \otimes B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B \end{tikzcd}.$$

The singleton element () contains no information in and of itself so the fact that

$$\pi_1 \circ u() = \vec{a}() = a \quad \text{and} \quad \pi_2 \circ u() = \vec{b}() = b$$

implies that $\pi_1 \circ u$ and $\pi_2 \circ u$ hold any structural information required to locate and retrieve a and b from $A \otimes B$. Specifically, u locates an element in

[11] See §B.4.

$A \otimes B$ that can be “unpacked” by the projections to give an A-element and a B-element—making clear that the elements of $A \otimes B$ contain both. Since u is unique, there can be only one such element in $A \otimes B$, constraining the product to a minimal representation. An ordered pair is the minimal structured expression of a and b implying that $u() = (a, b)$. Next, like some sort of categorical solid of revolution, we can vary $\vec{a}$ over all A and likewise for B, mapping out the full contents of $A \otimes B$ and revealing that the product is indeed the Cartesian product:

$$A \otimes B = A \times B = \left\{ (a, b) \mid a \in A; b \in B \right\}.$$

This makes the projections

$$\begin{aligned} \pi_1 : \ A \times B &\to A \\ (a, b) &\mapsto a, \end{aligned} \qquad \text{and} \qquad \begin{aligned} \pi_2 : \ A \times B &\to B \\ (a, b) &\mapsto b. \end{aligned}$$

The previous exercise demonstrates that the unique factorization $u : 1 \to A \times B$ is determined by the choice of $\vec{a}$ and $\vec{b}$. Likewise, going back to the general case of spans $A \xleftarrow{f_A} C \xrightarrow{f_B} B$, illustrated in (B.9.12), the unique factorization $u : C \to A \times B$ is determined by the choice of f_A and f_B. Given an element $c \in C$, u must produce a pair by applying f_A and f_B separately and packaging the results into a pair. This operation has enough general utility to warrant special notation, $u = f_A \vartriangle f_B$.

NOTATION B.9-7. The operator $\vartriangle$, sometimes called ***fan-out***, is defined in $\mathbb{S}$**et** as

$$\begin{aligned} f_A \vartriangle f_B : \ C &\to A \times B \\ c &\mapsto (f_A\, c, f_B\, c). \end{aligned}$$

As a mnemonic aid, think of $\vartriangle$ as as geometric amalgam of the logical “and” operator, $\wedge$, and function composition, $\circ$, since we combine the left *and* right functions giving a two-fold map.

For example, take $\sin \vartriangle \cos : \mathbb{R}_{[0, 2\pi)} \to \mathbb{R}_{[-1, 1]} \times \mathbb{R}_{[-1, 1]}$ which maps the 2π-interval to the unit circle. So $\sin \vartriangle \cos (\pi/3) = (\sqrt{3}/2,\ 1/2)$.

Though the foregone discussion has focussed on the binary product, it all generalises straightforwardly to n-ary finite Cartesian products. In fact, it will soon be made clear that, in the category of sets,

$$A \times B \times C \cong A \times (B \times C) \cong (A \times B) \times C,$$

so higher order products can be constructed from iterated binary products. Furthermore, the notation $X^{\times n}$ extrapolated from NTN. B.9-5 simplifies nicely to X^{n}, for $n \in \mathbb{N}$ and $n = \{ 0, \dots, n \}$. This suggests (quite correctly) that $\mathrm{Hom}(n, X) \cong X^{\times n}$. This allows us to think of an array of X-values indexed by n, with $X^0 \cong 1$, $X^1 \cong X$ (the global elements) and inductively $X^n \cong X^{\{0, \dots, n-1\}} \times X$. Pick any $m, n \in \mathbb{N}$, the natural isomorphism $X^m \times X^n \cong X^{m \sqcup n}$ is like concatinating an m-tuple and n-tuple (forward) or splitting an $(m+n)$-tuple into m- and n-tuples (reverse).

REMARK B.9-8. In $\mathbb{S}\mathbf{et}$ one can construct a product for any finite collection of sets. We could say that $\mathbb{S}\mathbf{et}$ admits all limits of discrete shape, but more naturally we say that it *admits all finite products.* This leads to the notion of multivariate functions as arrows from product sets: $\mathbb{S}\mathbf{et}(A \times B \times \cdots, Z)$.[12]

B.10 COPRODUCTS

Coproducts are dual to products, so the following discussion strongly parallels §B.9. Abstract coproducts in a category $\boldsymbol{C}$, also called the ***categorical sum***, are colimits of diagrams of discrete finite categories:

$$\begin{aligned} \dot{\mathbf{2}} &:= \boxed{1 \quad 2}\,, \\ \dot{\mathbf{3}} &:= \boxed{1 \quad 2 \quad 3}\,, \\ \dot{\mathbf{4}} &:= \boxed{1 \quad 2 \quad 3 \quad 4}\,, \\ &\vdots \;\; . \end{aligned}$$

An n-fold coproduct is determined by the colimit of a diagram with the shape of a discrete category with n objects. We shall construct the simplest such object, the 2-fold (binary) coproduct before moving to the n-fold case.

B.10-1 Let D be the diagram $D : \dot{\mathbf{2}} \to \boldsymbol{C}$; $1 \mapsto A$; $2 \mapsto B$, selecting the summands. A cocone over D with cocone-point $C \in \boldsymbol{C}$ is a cospan,

$$A \xrightarrow{f_A} C \xleftarrow{f_B} B\,.$$

A *binary coproduct* in $\boldsymbol{C}$ is a cospan $(A \oplus B, \{ \iota_1, \iota_2 \}) = \operatorname{colim} D$. That is, if for all other cospans $\big(C, \{ f_A, f_B \}\big)$, there exists a unique morphism

[12] This is how categories and *operads* can be related.

$u\colon A \oplus B \to C$ such that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "\iota_1"] \arrow[dr, "f_A"'] & A \oplus B \arrow[d, dashed, "\exists!\, u"'] & B . \arrow[l, "\iota_2"'] \arrow[dl, "f_B"] \\
& C &
\end{tikzcd} \tag{B.10.13}$$

The previous diagram is a specialisation of (B.8.11) to the case of the binary coproduct. A coproduct is precisely a product in $\boldsymbol{C}^{\mathrm{op}}$ so if they exist, they too are unique up to isomorphism. The functions ι are called *injections* in contrast to *projections*, but they need not be injective (or epi).[13] The commutativity condition of (B.10.13) requires that $u\,\iota_1 = f_A$ and $u\,\iota_2 = f_B$. In this sense, an arrow from a coproduct object is reducible to a pair of arrows. In mainstream texts on general CT, the arrow u is denoted $[f_A, f_B]$, but a more specialised notation will be introduced later. (See the "fan-in operator", NTN. B.9-7.)

Higher order coproducts, $A \oplus B \oplus C \oplus \cdots$, are constructed on diagrams with more discrete objects.

DEFINITION B.10-2. Given an arbitrary category $\boldsymbol{C}$ and some $n \in \mathbb{N}_{\geq 2}$, consider a diagram $D\colon \dot{\boldsymbol{n}} \to \boldsymbol{C}$ that sends $i \in \dot{\boldsymbol{n}}$ to $X_i \in \boldsymbol{C}$. Cocones on the diagram are n-cospans:

$$C \xleftarrow{f_{X_i}} X_i \qquad \text{for all } i \in \dot{\boldsymbol{n}}.$$

The n***-fold coproduct*** is universal cocone:

$$\operatorname{colim} D = \left(\bigoplus_{j \in \dot{\boldsymbol{n}}} X_j \xleftarrow{\iota_i} X_i \right)_{i \in \dot{\boldsymbol{n}}} ,$$

This means that for all other cones there exists a unique arrow u making the following diagrams commute:

$$\begin{tikzcd}
\bigoplus_{j \in \dot{\boldsymbol{n}}} X_j \arrow[d, dashed, "\exists!\, u"'] & X_i \arrow[l, "\iota_i"'] \arrow[dl, "f_{X_i}"] \\
C &
\end{tikzcd} \qquad \text{for all } i \in I.$$

[13] Awodey briefly calls them "coprojections" in [109], which has a satisfying clarity, but is not conventional terminology.

Products in a category, when they exist, are unique up to isomorphism.

B.10.1 Coproducts in **Set** reprised

Products in **Set** (from the last section, §B.9.1) entail the organisation and unpacking of tuples—sequences of elements of sets. In this sense, there is a deep relationship between **Set**-products and sequencing. Dually to that sense, there is a deep relationship between **Set**-coproducts and selection. Instead of tuples, the machinery around coproducts will entail case-wise structures.

In the beginning of the section we discussed coproducts in **Set**. From an intuitive but informal approach, we arrived at the Cartesian product. We are now in a position to approach it from the general notion of a categorical product and land once again on the disjoint union.

As with we did with binary **Set**-products[14], we will try to use a simple probe to tease meaning out of these diagrams. Consider the case where $A = B = 1$, then given $c_1 \in C$ and $c_2 \in C$ the corresponding cospan is

$$1 \xrightarrow{\vec{c_1}} C \xleftarrow{\vec{c_2}} 1\,.$$

Substituting it into (B.10.13) we get the commutative diagram

$$\begin{tikzcd}
1 \arrow[r, "\iota_1"] \arrow[dr, "\vec{c_1}"'] & 1 \oplus 1 \arrow[d, dashed, "\exists!\, u"'] & B\,. \arrow[l, "\iota_2"'] \arrow[dl, "\vec{c_2}"] \\
& 1 &
\end{tikzcd}$$

which encodes

$$u \circ \iota_1 = \vec{c_1} \quad \text{and} \quad u \circ \iota_2 = \vec{c_2}.$$

The injections are canonical, and locate the element of each summand into the coproduct object $1 \oplus 1$. It is the job of u to map the images of the injections to their targets in C as determined by $\vec{c_1}, \vec{c_2}$. But the summands, the left- and right-hand singletons, are identical: how does u discern when to use c_1 or c_2? The object $1 \oplus 1$ must not only embody the elements of the summands, but must also encode their origin in either the left or right summand.

$$1 \oplus 1 \cong \{\,(a,1),(b,2)\,\},$$

[14] in §B.9.1, p. 290

which is a disjoint union of the singletons, $1 \sqcup 1$, as expected. The *tags*[15] "1" and "2" mirror the injection labels. The injections conspire in the scheme by appending the appropriate tag:

$$\begin{aligned} \iota_1 :\ & 1 \to 1 \sqcup 1 \\ & () \mapsto ((), 1) \end{aligned} \qquad \text{and} \qquad \begin{aligned} \iota_2 :\ & 1 \to 1 \sqcup 1 \\ & () \mapsto ((), 2). \end{aligned}$$

Allowing the summands as A and B to vary among all sets, the situation generalises back to

$$\begin{tikzcd} A \arrow[r, "\iota_1"] \arrow[dr, "f_A"'] & A \sqcup B \arrow[d, dashed, "\exists!\, u"'] & B \arrow[l, "\iota_2"'] \arrow[dl, "f_B"] \\ & C & \end{tikzcd},$$

with injections

$$\begin{aligned} \iota_1 :\ & A \to A \sqcup B \\ & a \mapsto (a, 1) \end{aligned} \qquad \text{and} \qquad \begin{aligned} \iota_2 :\ & B \to A \sqcup B \\ & b \mapsto (b, 2). \end{aligned}$$

The unique $u :\ A \sqcup B \to C$ encompasses as a special composite of f_A and f_B and gets its own notation. Dual to the *fan-out* operator, $\vartriangle$, we have

NOTATION B.10-3. The operator $\triangledown$, sometimes called ***fan-in***, is defined in $\mathbf{Set}$ as

$$\begin{aligned} f_A \triangledown f_B :\ & A \sqcup B \to C \\ & (x, i) \mapsto \begin{cases} f_A\, x & \text{if} \quad i = 1 \\ f_B\, x & \text{if} \quad i = 2 \end{cases} \end{aligned}$$

As a mnemonic aid, think of $\triangledown$ as visually similar to logical "or" operator, $\vee$, since it applies the left *or* right function depending on the tag of the argument.

Though the foregone discussion has focussed on the binary coproduct, it all generalises straightforwardly to n-ary finite coproducts. In fact, it will soon be made clear that, in the category of sets,

$$A \sqcup B \sqcup C \cong A \sqcup (B \sqcup C) \cong (A \sqcup B) \sqcup C,$$

so higher order coproducts can be constructed from binary coproducts.

[15] The tags themselves are not important. We could use text strings "A" and "B" or "left" and "right" or anything else that uniquely identifies the source of each element.

REMARK B.10-4. We can construct a coproduct for any finite collection of sets. We could say that **\$et** admits all colimits of discrete shape, but more naturally we say that it admits all finite coproducts.

B.11 MONOIDAL STRUCTURES IN CATEGORIES

Monoids are sometimes called a group without inverses or a semigroup with identity. Since monoids will be a thread throughout the thesis, we should define it formally:

DEFINITION B.11-1. An (algebraic) ***monoid*** is a triple $(M, \textcircled{M}, e)$ consisting of

- a set M,
- a binary operation, $\textcircled{M} : M \times M \rightarrow M$, and
- a value $e \in M$

which obey three axioms:

M-1: (Closure) for any $a, b \in M$, $(a \textcircled{M} b) \in M$.
M-2: (Associativity) for any $a, b, c \in M$, $(a \textcircled{M} b) \textcircled{M} c = a \textcircled{M} (b \textcircled{M} c)$
M-3: (Identity) The element $e \in M$, called the ***identity element*** and for any $a \in M$, $e \textcircled{M} a = a \textcircled{M} e = a$

DEFINITION B.11-2. Given monoids $(A, \textcircled{A}, e_A)$ and $(B, \textcircled{B}, e_B)$, a homomorphism of monoids is a function $f : A \rightarrow B$ such that:

- $h(a \textcircled{A} a') = h(a) \textcircled{B} h(a')$
- $h(e_A) = e_B$,

thus respecting the structure of the monoids.

NOTATION B.11-3 (Underlying set of a monoid)**.** Given a monoid M, the set that underlies it is denoted $|M|$. At the risk of self-referencing definitions, we may introduce arbitrary monoids as $M = (|M|, \textcircled{M}, e)$.

DEFINITION B.11-4. There is a category of algebraic monoids, **Mon**(**\$et**) with

objects: all algebraic monoids,
arrows: all monoid homomorphisms, DEF. B.11-2, as arrows in **\$et** that respect the structure of the monoids.

The forgetful functor $U_{\mathbf{Mon}} : \mathbf{Mon}(\mathbf{\$et}) \rightarrow \mathbf{\$et}$ maps each monoid M to its underlying set $|M|$ and each monoid homomorphism h to its underlying function $|h|$.

Monoids provide a very fundamental model of composition and it is perhaps unsurprising that monoidal structures are readily seen within categories which are all about composition. Examples of monoids are prolific: some familiar ones are $(\mathbb{N}, +, 0)$, $(\mathbb{R}, \times, 1)$. In this section, we discuss and classify monoidal structures made from the objects and arrows of a category.

A binary operation on the constituents of a category will take the form of binary functors or *bifunctors.* These can be thought of as functors of two arguments, or a single-variate functor on products in $\mathbb{C}\mathbf{at}$. A bifunctor $\circ : \boldsymbol{C} \times \boldsymbol{D} \to \boldsymbol{E}$ (written in infix notation) satisfies

$$\begin{cases} \mathrm{id}_A \circ \mathrm{id}_B = \mathrm{id}_{A \circ B} \\ (f \circ g)(f' \circ g') = (ff') \circ (gg') \end{cases}$$

when the composites are well formed.

The product and coproduct operators are bifunctors:

$$\otimes, \oplus : \boldsymbol{C} \times \boldsymbol{C} \to \boldsymbol{C}.$$

on a category $\boldsymbol{C}$. In order to build monoidal structure from these bifunctors, we need to find units and demonstrate associativity and identity. In a category, we might relax the monoid axioms as associativity and identity *up to natural isomorphism.* So, instead of demanding that $(A \otimes B) \otimes C = A \otimes (B \otimes C)$, we need only demonstrate a natural isomorphism associating both sides of the equation.

DEFINITION B.11-5. A ***monoidal structure*** in a category $\boldsymbol{C}$ is a tuple $(\circ, I, \alpha, \ell, \varrho)$ consisting of

- $\circ : \boldsymbol{C} \times \boldsymbol{C} \to \boldsymbol{C}$, a bifunctor called the ***monoidal product*** (or sometimes ***tensor product***) in the category,
- I, a $\boldsymbol{C}$-object called the ***monoidal unit*** (or ***tensor unit***),
- α, a natural isomorphism called ***the associator*** with components of the form

$$\alpha_{A,B,C} : A \circ (B \circ C) \xrightarrow{\sim} (A \circ B) \circ C \quad \text{for all } A, B, C \in \mathrm{ob}\,\boldsymbol{C},$$

- ℓ, ϱ, natural isomorphisms called (respectively) the left- and right-unitors with components of the forms

$$\ell_A : I \circ A \xrightarrow{\sim} A$$
$$\varrho_A : A \circ I \xrightarrow{\sim} A,$$

  for all $A \in \mathrm{ob}\,\boldsymbol{C}$.

For all $\boldsymbol{C}$-objects A, B, C, D, the monoidal structure observes the commutativity of the following two diagrams:

1. The associator diagram depicting the equivalence of the various groupings of the product $A \circ B \circ C \circ D$:

$$\begin{tikzcd}
(A \circ B) \circ (C \circ D) \arrow[dr, bend left, "\alpha_{(A \circ B), C, D}"] & \\
A \circ (B \circ (C \circ D)) \arrow[u, "\alpha_{A, B, (C \circ D)}"] \arrow[d, "\mathrm{id}_A \circ \alpha_{B,C,D}"'] & ((A \circ B) \circ C) \circ D \\
A \circ ((B \circ C) \circ D) \arrow[r, "\alpha_{A, (B \circ C), D}"'] & (A \circ (B \circ C)) \circ D\,. \arrow[u, "\alpha_{A,B,C} \circ \mathrm{id}_D"']
\end{tikzcd} \tag{B.11.14}$$

2. The unitor diagram depicting the behaviour of the left- and right-unitors on the product $A \circ I \circ B$ grouped on the left and right:

$$\begin{tikzcd}
A \circ (I \circ B) \arrow[r, "\alpha_{A,I,B}"] \arrow[d, "\mathrm{id}_A \circ \ell_B"'] & (A \circ I) \circ B\,. \arrow[dl, bend left, "\varrho_A \circ \mathrm{id}_B"] \\
A \circ B &
\end{tikzcd} \tag{B.11.15}$$

Having relaxed the associativity and unitary conditions to isomorphisms, a ***strict monoidal structure*** is one in which α, ℓ and ϱ are identities.

REMARK B.11-6. The subscripts on the natural transformations of (B.11.14) and (B.11.15) may be helpful but are unnecessary. The index of a natural transformation is uniquely determined by its signature. It is often helpful to be explicit—especially in a diagram purposed for revealing the intricacies of the natural transformation. Other times, it is desirable to suppress the visual noise they present.

Because the associator and unitors are isomorphisms, they have inverses:

$$\begin{aligned}
\alpha^{-1} :&\; (A \circ B) \circ C \xrightarrow{\sim} A \circ (B \circ C) \\
\ell^{-1} :&\; \qquad\qquad A \xrightarrow{\sim} I \circ A \\
\varrho^{-1} :&\; \qquad\qquad A \xrightarrow{\sim} A \circ I
\end{aligned}$$

DEFINITION B.11-7. A ***braided monoidal structure*** is a tuple $(\circ, I, \alpha, \ell, \varrho, \gamma)$, a monoidal structure with the addition of a natural isomorphism, γ, with components

$$\gamma_{A,B} : A \circ B \xrightarrow{\sim} B \circ A$$

called the ***braiding***. Braiding constitutes a weak commutative relationship in the monoid. The braiding is compatible with the associator (in in turn, the unitors) by commutativity of the diagrams:

$$\begin{tikzcd}
(A \circ B) \circ C \arrow[r, "\gamma_{(A\circ B),C}"] \arrow[d, swap, "\alpha^{-1}"] & C \circ (A \circ B) \arrow[d, "\alpha"] & A \circ (B \circ C) \arrow[r, "\gamma_{A,(B\circ C)}"] \arrow[d, swap, "\alpha"] & (B \circ C) \circ A \arrow[d, "\alpha^{-1}"] \\
A \circ (B \circ C) \arrow[d, swap, "\mathrm{id}_A \circ \gamma_{B,C}"] & (C \circ A) \circ B \arrow[d, "\gamma_{C,A} \circ \mathrm{id}_B"] & (A \circ B) \circ C \arrow[d, swap, "\gamma \circ \mathrm{id}_C"] & B \circ (C \circ A) \arrow[d, "\mathrm{id}_B \circ \gamma"] \\
A \circ (C \circ B) \arrow[r, "\alpha"] & (A \circ C) \circ B\,, & (B \circ A) \circ C \arrow[r, "\alpha^{-1}"] & B \circ (A \circ C)\,.
\end{tikzcd} \tag{B.11.16}$$

DEFINITION B.11-8. If the braiding of a braided monoidal structure is self-inverse,

$$\gamma \circ \gamma = \mathrm{id},$$

then the structure is called a ***symmetric monoidal structure***.

When the category associated with these structures appear alongside them in a tuple, they are named to reflect the monoidal structure.

DEFINITION B.11-9. A ***monoidal category*** is a category $\boldsymbol{C}$ together with a monoidal structure on $\boldsymbol{C}$ that can be summarised by the tuple $(\boldsymbol{C}, \circ, I)$ where it is understood that the monoidal product and unit have the structure of DEF. B.11-5. The terms ***strict monoidal category***, ***braided monoidal category*** and ***symmetric monoidal category*** are defined likewise, with the additional structure implicit.

REMARK B.11-10. Any category with finite categorical products is monoidal [81, §VII.1].

DEFINITION B.11-11 (Co/cartesian monoidal category)**.** The monoidal category with monoidal structure given by the category theoretic product, and the monoidal unit given by a terminal object is called a ***Cartesian***

monoidal category. A monoidal category with monoidal structure given by the categorical *coproduct* an unit provided by an initial object is called a ***Cocartesian monoidal category.***

A truly erudite treatment of monoidal categories and braided/symmetric monoidal categories is found in Mac Lane's [81, §VII.1] and [81, Ch. XI] respectively. Mac Lane does not take the approach of distinguishing a monoidal category from its monoidal structure, but the foregone description is otherwise deliberately compatible. In [168, §4.4.3], Fong and Spivak put forth less formal but very succinct and intuitive definitions and provide a wealth of insights relating to the application of monoidal categories and the graphical calculus they import [114].

B.12 **Set** as a cartesian monoidal category

The category of sets together with the Cartesian product and any singleton set, $(\mathbf{Set}, \times, 1)$, constitutes a Cartesian monoidal category. In this section, the details of the associator, left- and right-unitors and the braiding are given. The main ideas are this:

- The set $A \times B \times C$ is isomorphic to $(A \times B) \times C$ and $A \times (B \times C)$ since the tuples (a, b, c), $((a, b), c)$ and $(a, (b, c))$ are mutually interchangeable by merely rearranging parentheses. This gives the associator.
- A set A is isomorphic to $A \times 1$ and $1 \times A$ because the singleton introduces no opportunity for choice. Any such pair $(a, ())$ can be projected back to just a restored by re-appending the singleton. This gives the unitors.
- The set $A \times B$ is isomorphic to $B \times A$ since any pair (a, b) can be rearranged as (b, a) and back. This gives the braiding.

First, consider the associator for the Cartesian product. If we use the strictly binary form of the product, we can construct the type $(A \times B) \times C$ by first taking the product $A \times B$ and then applying the binary product again with C on the right. Given elements $a \in A$, $b \in B$ and $c \in C$, we can construct an element of $(A \times B) \times C$ as $((a, b), c)$. From that, it is a trivial exercise in rearranging parentheses to form $(a, (b, c)) : A \times (B \times C)$. This is the job of the associator:

$$\begin{aligned}
\alpha_{A,B,C} :\ A \times (B \times C) &\xrightarrow{\sim} (A \times B) \times C \\
(a, (b, c)) &\mapsto ((a, b), c) \\
(a, (b, c)) &\mapsfrom ((a, b), c)
\end{aligned}$$

As the above notation makes apparent, the inverse follows trivially from the forward operations since reordering parentheses is not a destructive operation: no information is lost, so no external relationships determine the inverse. It is short work to show that this definition satisfies (B.11.14).

Now to the unitors. The singleton, with its lone element, adds no variety to a product: $1 \times A$ is just $\{ ((), a) \mid a \in A \}$ which is in 1-to-1 correspondence to A. The same is true, of course, if the singleton appears on the right of the product. Therefore, the left- and right-unitors are merely product projections in one direction, with the obvious inverse. For the left-case,

$$\begin{aligned} \ell_A :\ & 1 \times A \to A \\ ((), a) & \overset{\pi_2}{\mapsto} a \\ ((), a) & \mapsfrom a \end{aligned}$$

The inverse operation relies on the fact that $()$ is the only possibility for the left element. With only a single element, 1 can only provide embellishment to a product, not variety. And so, $((), a)$, a and $(a, ())$ are equivalent representations of the same information, with only adornment to distinguish them. Likewise,

$$\begin{aligned} \varrho_A :\ & A \times 1 \to A \\ (a, ()) & \overset{\pi_1}{\mapsto} a \\ (a, ()) & \mapsfrom a. \end{aligned}$$

It is short work, once again, to verify that these definitions for α, ℓ and ϱ satisfy (B.11.15).

The Cartesian monoidal structure on **Set** is symmetric. The braiding merely swaps the order of pairs. For all sets A, B,

$$\begin{aligned} \gamma_{A,B} :\ & A \times B \xrightarrow{\sim} B \times A \\ (a, b) & \mapsto (b, a) \\ (a, b) & \mapsfrom (b, a) \end{aligned}$$

Moreover, doubly applying this braiding to a pair $(a, b) \in A \times B$ is ineffectful:

$$\gamma\gamma\,(a, b) = (a, b),$$

making the braiding symmetric.

B.13 $\mathbf{Set}$ AS A COCARTESIAN MONOIDAL CATEGORY

Now, consider the Cocartesian monoidal category $(\mathbf{Set}, \sqcup, 0)$. In this section, the details of the associator, left- and right-unitors and the braiding are given. The main ideas are this:

- The sets $A \sqcup B \sqcup C$, $A \sqcup (B \sqcup C)$ and $(A \sqcup B) \sqcup C$ are either equal or isomorphic depending on choice of tag structure. The tag structure exists only to distinguish the original membership of the union elements and is not unique. If these sets differ, then they can be interchanged by manipulating the tags. This gives the associator.
- A set A is isomorphic to $A \sqcup 0$ and $0 \sqcup A$ since the empty set contributes nothing to the union. An element of $A \sqcup 0$ or $0 \sqcup A$ can only be an element of A, and can therefore be placed in 1-to-1 correspondence with A. This gives the left- and right-unitors.
- The sets $A \sqcup B$ and $B \sqcup A$ are equal or isomorphic since they differ (at most) by tags.

Braving a descent into pedantry, consider a construction of these ternary disjoint unions by repeated application of the binary disjoint union as defined in §B.10.1:

$$\begin{aligned}
&A \sqcup (B \sqcup C) = \{\, (a,1) \mid a \in A \,\} \cup \{\, (x,2) \mid x \in B \sqcup C \,\} \\
&\qquad\qquad \text{where} \\
&\qquad\qquad\quad B \sqcup C = \{\, (b,1) \mid b \in B \,\} \cup \{\, (c,2) \mid c \in C \,\}
\end{aligned}$$

and

$$\begin{aligned}
&(A \sqcup B) \sqcup C = \{\, (x,1) \mid x \in A \sqcup B \,\} \cup \{\, (c,2) \mid c \in C \,\} \\
&\qquad\qquad \text{where} \\
&\qquad\qquad\quad A \sqcup B = \{\, (a,1) \mid a \in A \,\} \cup \{\, (b,2) \mid b \in B \,\}.
\end{aligned}$$

This awkwardly yields elements of varying product structure, like $(a,2)$ and $((b,1),2)$, but this is merely a notational inconvenience—a consequence of naïvely using the binary product to construct the ternary products. The associator fomented by this accounting system is merely a "re-tagger:"

$$\begin{aligned}
\alpha_{A,B,C} :\ & A \sqcup (B \sqcup C) \xrightarrow{\sim} (A \sqcup B) \sqcup C \\
& \begin{cases} \quad (x,1) \mapsto ((x,1),1) & (\text{will be } \ x \in A) \\ ((x,1),2) \mapsto ((x,2),1) & (\text{will be } \ x \in B) \\ ((x,2),2) \mapsto (x,2) & (\text{will be } \ x \in C) \end{cases}
\end{aligned}$$

Because the labels are unique, the associator is really a bijection between the sets of tags, and this is therefore an isomorphism; the directions of the maplet arrows above may be simply reversed to obtain α^{-1}.

The empty set provides a unit for disjoint union because it contributes no elements:

$$A \sqcup 0 = \{\, (a, 1) \mid a \in A \,\} \cong 0 \sqcup A \cong A$$

and

$$0 \sqcup A = \{\, (a, 2) \mid a \in A \,\} \cong A \sqcup 0 \cong A.$$

The canonical injections, ι_1, ι_2, for such coproducts are merely decorators, furnishing the elements of A with a tag. The left- and right-unitors discard the tag (by projection) inversely to the injection:

$$\begin{array}{rl} \ell_A : & 0 \sqcup A \xrightarrow{\sim} A \\ & (x, 2) \overset{\pi_1}{\mapsto} x \\ & (x, 2) \overset{\iota_2}{\mapsfrom} x \end{array} \qquad \text{and} \qquad \begin{array}{rl} \varrho_A : & A \sqcup 0 \xrightarrow{\sim} A \\ & (x, 1) \overset{\pi_1}{\mapsto} x \\ & (x, 1) \overset{\iota_1}{\mapsfrom} x. \end{array}$$

From all this, we can also find braiding:

$$\begin{array}{l} \gamma : \ A \sqcup B \xrightarrow{\sim} B \sqcup A \\ \begin{cases} (x, 1) \mapsto (x, 2) \\ (x, 2) \mapsto (x, 1) \end{cases} \end{array}$$

that is self-inverse: $\gamma\gamma = \mathrm{id}$ (by inspection of the above). Therefore, the Cocartesian category, $(\mathbf{\$et}, \sqcup, 0)$, is a symmetric monoidal category.

REMARK B.13-1. Instead of using integers as tags to implement disjoint union, one could use some sort of string label uniquely determined by the sets in the coproduct. In that case, relabelling would not have been necessary.

B.14 MONADS

> All told, a monad X is just a monoid in the category of endofunctors of X, with product $\times$ replaced by composition of endofunctors and unit set by the identity endofunctor.
>
> — SAUNDERS MACLANE
>
> *[81, p. 138]*

Monads encapsulate algebraic theories and algebras are models of those theories. Culturally, monads has obtained a legendary status among some programmers because of its usefulness in modelling computations with side effects [59, 63].The sub-section's opening epigraph is often wrongly attributed to Wadler possibly because of the following quote from Iry's *A Brief, Incomplete, and Mostly Wrong History of Programming Languages* [105]:

> 1990 - A committee formed by Simon Peyton-Jones, Paul Hudak, Philip Wadler, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language. Haskell gets some resistance due to the complexity of using monads to control side effects. Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem?"

Humour aside, this is actually an effective scholarly description, even if it is unhelpful as an answer to the question, "what is a monad, anyway?" So the approach in this section will be to clarify Mac Lane's portrayal, building from the tools we have already established.

Recall that an endofunctor is a functor from a category into itself. A natural transformation $\vartheta : F \Rightarrow G$ between endofunctors $F, G : \boldsymbol{C} \to \boldsymbol{C}$ is an assignment to each $\boldsymbol{C}$-object an appropriately structure preserving $\boldsymbol{C}$-arrow, collectively called the components of ϑ. Structure preserving means that for all $f \in \operatorname{Hom}\boldsymbol{C}$ it holds that

$$\vartheta_{\operatorname{cod} f} \circ Ff = Gf \circ \vartheta_{\operatorname{dom} f}.$$

That is the naturality condition given in (B.7.2) and the commutativity condition of the square diagram in (B.7.3).

Endofunctors on a category constitute a special case of a functor category, $\mathbf{End}_{\boldsymbol{C}} = \mathbf{Fun}(\boldsymbol{C}, \boldsymbol{C})$. Because signature alignment is guaranteed by definition, endofunctors can be iterated ad libitum. Each endofunctor T entails $T^2, T^3, \ldots \in \mathbf{End}_{\boldsymbol{C}}$, enabling us to build natural transformations among them. This gives us a set, and a mechanism for defining operations on the set. For a natural transformation $\mu : T^2 \Rightarrow T$ the domain T^2 should be regarded as a binary product as in $M \bigcirc M$ from DEF. 3.5-6 where the binary operation is functor composition.

DEFINITION B.14-1. A ***monad***, T, in a category $\boldsymbol{C}$ is a monoid object of $\mathbf{End}_{\boldsymbol{C}}$

$$\mathsf{id}_{\boldsymbol{C}} \xrightarrow{\eta} T \xleftarrow{\mu} T^2 \, , ,$$

specified as a triple (T, η, μ) where,

- η is a natural transformation called the unit.
- μ is a natural transformation called the multiplication,

such that the identity triangles and the associativity square (ordered respectively) commute:

$$\begin{tikzcd}
T \arrow[r, "T\eta"] \arrow[dr, swap, "\mathrm{id}_T"] & T^2 \arrow[d, "\mu"] & T \arrow[l, swap, "\eta T"] \arrow[dl, "\mathrm{id}_T"] \\
& T &
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
T^3 \arrow[r, "\mu T"] \arrow[d, swap, "T\mu"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, swap, "\mu"] & T
\end{tikzcd}$$

DEFINITION B.14-2. A monad ($T : \boldsymbol{C} \to \boldsymbol{C}$, η, μ) induces a category $\boldsymbol{C}^T$ with

objects: Algebras, (A, $T\,X \overset{\alpha}{\longrightarrow} A$) for all $A \in \boldsymbol{C}$; and
arrows: Algebra homomorphisms h observing

$$\begin{tikzcd}
T\,A \arrow[r, "T\,h"] \arrow[d, swap, "A"] & T\,B \arrow[d, "\beta"] \\
A \arrow[r, "h"] & B
\end{tikzcd}$$

This is often called the *Eilenberg-Moore* category of the monad.

B.15 CLOSED CATEGORIES

It is a great philosophical curiosity (and very useful fact) that the collection of arrows between two sets, $\mathbf{Set}(A, B)$, is itself a set and is therefore a citizen among the objects of $\mathbf{Set}$. In this strange way, the category of sets *knows* itself, embodying a reflection of its outer structure among its internal constituents. This internal knowledge is organised by the hom-functor.

The hom operator was introduced in the definition of a category (DEF. B.1-1) as the notational device to reference the collection of arrows between two objects. To promote it to a functor we need to restrict the discussion to small categories and define the action of hom on arrows.

DEFINITION B.15-1 (Hom functor)**.** Given a locally small category $\boldsymbol{C}$, arbitrary objects $A, B, X, Y \in \boldsymbol{C}$ and arrows

- $f : X \to A$
- $g : B \to Y$

the ***hom-functor*** is

$$\begin{aligned} \mathrm{hom}:\ \boldsymbol{C}^{\mathrm{op}} \times \boldsymbol{C} &\to \mathbb{S}\mathbf{et} \\ &\begin{cases} (A, B) \mapsto \mathrm{hom}(A, B) \\ (f, g) \mapsto (h \mapsto g \circ h \circ f) \end{cases} \end{aligned}$$

such that the following diagram commutes:

$$\begin{tikzcd}
\mathrm{hom}(A, B) \arrow[r, "\mathrm{hom}(f{,}B)"] \arrow[d, "\mathrm{hom}(A{,}g)"'] & \mathrm{hom}(X, B) \arrow[d, "\mathrm{hom}(X{,}g)"] \\
\mathrm{hom}(A, Y) \arrow[r, "\mathrm{hom}(f{,}Y)"] & \mathrm{hom}(X, Y)\,.
\end{tikzcd}$$

In the covariant argument, $\mathrm{hom}(A, g)$, g is lifted to $h \mapsto g \circ h$ for all $h \in \mathrm{hom}(A, B)$. Likewise in the contravariant argument, in $\mathrm{hom}(f, B)$ the function f is lifted to $h \mapsto h \circ f$ for $h \in \mathrm{hom}(A, B)$. Then the diagonal in the diagram, mapping $\mathrm{hom}(A, B)$ to $\mathrm{hom}(X, Y)$, is the composite $h \mapsto g \circ h \circ f$.

The special property of $\mathbb{S}\mathbf{et}$ having internal knowledge of itself is make explicit by substituting $\boldsymbol{C} = \mathbb{S}\mathbf{et}$ in the previous definition. But some monoidal categories can mirror this self-knowledge in more abstract ways by admitting their own *internal hom-functor*.

DEFINITION B.15-2 (Internal hom-functor, currying & evaluation)**.** A given monoidal category $(\boldsymbol{C}, \otimes, I)$ has *internal homs* if it admits an ***internal hom-functor***, $(\multimap):\ \boldsymbol{C}^{\mathrm{op}} \times \boldsymbol{C} \to \boldsymbol{C}$, that sends pairs $(A, B) \in \boldsymbol{C} \times \boldsymbol{C}$ to an "object of morphisms" or ***hom-object*** from A to B such that there is a natural isomorphism

$$\lambda:\ \mathrm{hom}(A \otimes B, C) \xrightarrow{\sim} \mathrm{hom}(A, B \multimap C),$$

called ***currying***, along with a special arrow *ev* called ***evaluation*** both reflected in the commutativity of the diagram

$$\begin{tikzcd}
B \multimap C & & (B \multimap C) \otimes B \arrow[r, "ev"] & C\,. \\
A \arrow[u, "\lambda f"] & & A \otimes B\,. \arrow[u, dashed, "(\lambda f) \otimes \mathrm{id}_B"] \arrow[ur, "f"'] &
\end{tikzcd}$$

On arrows, the internal hom-functor acts through composition as the set-valued one:

$$\begin{tikzcd}
A \multimap B \arrow[r, "f \multimap B"] \arrow[d, "A \multimap g"'] & X \multimap B \arrow[d, "X \multimap g"] \\
A \multimap Y \arrow[r, "f \multimap Y"] & X \multimap Y ,
\end{tikzcd}$$

for $A, B, X, Y \in \boldsymbol{C}$ and $f : X \to A$, $g : B \to Y$.

This is a formidable definition on its own, so here is some intuition. Function application is a very concrete matter of daily activity while working with sets and functions. To think of it in the abstract, so that it may be encoded in the arrows of an arbitrary category, we think about it in functional terms. In the definition, let us substitute $(\boldsymbol{C}, \otimes, I) = (\$\mathbf{et}, \times, 1)$ so we can speak in familiar terms. Further, let us suppose we have $g \in B \multimap C$ and some $b \in B$. Application looks like $(g, b) \overset{ev}{\longmapsto} (gb \in C)$. Like any other universal construction [16], we consider a morphism $A \otimes B \xrightarrow{f} C$ purporting to mirror the behaviour of $(B \multimap C) \otimes B \xrightarrow{ev} C$, and state that there must be a unique arrow $(\lambda f) \otimes \mathrm{id}_B$ so that it factors through ev. Of course, using $(\$\mathbf{et}, \times, 1)$ gave us two conceptual conveniences: (1) we were able to think in terms of elements and functions and (2) we were able to conceptualise the monoidal product as Cartesian. But the definition relies on neither of those ideas.

REMARK B.15-3. In fact, it is a special case where the monoidal operation in DEF. B.15-2 is the categorical product. Just in that case, hom-objects are called ***exponential objects***. They are traditionally denoted as $B^A := A \multimap B$, but we will prefer the infix symbol.

B.15-4 A WORD ON CURRYING The natural transformation λ is no mere scaffolding to support DEF. B.15-2, but is extremely useful in its own right. Given a function $f : A \times B \to C$, $\lambda f : A \to (B \multimap C)$ allows us to partially apply variables in multivariate functions. The curried version of f allows us to write $\lambda f(a)(b) = f(a, b)$. This *currying* operation is a cornerstone of functional programming, and in some languages (notably Haskell [111], named after Haskell Curry) functions are curried by default. This will be important in the thesis, allowing us to write functions on infinite products

[16] Unlike other universal constructions we have seen that use co/limits, hom objects are examples of a universal that are not expressible as co/limits.

that model an internal notion of state and produce intermediate values in an infinite (stateful) calculation.

Recall that λ is a natural isomorphism. In the category of sets, a curried function $f' : A \to B \multimap C$, may be uncurried quite simply with the tools at hand:

$$\lambda^{-1} f' = ev \circ (f' \otimes \mathrm{id}_B) : A \otimes B \to C.$$

Furthermore, $\lambda^{-1}\lambda f = f$ and $\lambda \lambda^{-1} f' = f'$. This isomorphism is actually a manifestation of the exponential relationship,

$$A \otimes B \multimap C \cong A \multimap B \multimap C,$$

which is perhaps more familiar in the exponential notation:

$$C^{A \otimes B} \cong \left(C^B\right)^A.$$

If A, B and C were real numbers, the above isomorphism would be a familiar identity from school-math.

NOTATION B.15-5. The arrow and ringmap operators are right-associative, so in the later case,

$$A \multimap (B \multimap (C \multimap D))) = A \multimap B \multimap C \multimap D.$$

Also, and without much ado, we might also curry functions of arbitrary arity:

$$\begin{aligned} A \otimes B \multimap C &\cong A \multimap B \multimap C \\ A \otimes B \otimes C \multimap D &\cong A \multimap B \multimap C \multimap D \\ A \otimes B \otimes C \otimes \multimap E &\cong A \multimap B \multimap C \multimap D \multimap E \\ &\vdots \end{aligned}$$

NOTATION B.15-6. As a shorthand for currying, given $f : A \otimes B \to C$ let the decoration $(\tilde{\ })$ denote the curried form $\tilde{f} : A \to B \multimap C$ satisfying $f(a, b) = \tilde{f}\, a\, b$.

And now, a bit more nomenclature of categories with hom-functors.

DEFINITION B.15-7. A ***closed category*** is a braided or symmetric monoidal category that admits internal homs with currying.

DEFINITION B.15-8. A ***Cartesian Closed Category*** (CCC) is a monoidal closed category where the monoidal operation is the categorical product.

REMARK B.15-9. Awodey in [109, §6.6] writes "CCC ∼ λ-calculus". In fact, Church's simply typed λ-calculus is the *internal language* of CCC, a relationship upon which functional programming leans heavily.

DEFINITION B.15-10. A CCC that is extended with finite coproducts is called a ***bicartesian closed category*** (biCCC).

PROPOSITION B.15-11 ([33, Prop. 2.1])**.** If a CCC has binary coproducts ⊕ then there exist a natural isomorphism

$$d_{A,B,C} : \; A \otimes (B \oplus C) \xrightarrow{\sim} (A \otimes B) \oplus (A \otimes C),$$

called the ***distributor***, defined over objects A, B, C spanning $\boldsymbol{C}$. By extension to countable coproducts, we have natural isomorphisms:

$$\begin{aligned} A \otimes \bigoplus_n B_n &\xrightarrow{\sim} \bigoplus_n (A \otimes B_n) \\ \Big(\bigoplus_n A_n\Big) \otimes B &\xrightarrow{\sim} \bigoplus_n (A_n \otimes B) \\ \Big(\bigoplus_m A_m\Big) \otimes \Big(\bigoplus_n B_n\Big) &\xrightarrow{\sim} \bigoplus_{m,n} (A_m \otimes B_n) \end{aligned}$$

for $m, n \in \mathbb{N}$. These are also denoted with d, relying on the domain to indicate which particular distributive law is being called to service.

PROPOSITION B.15-12 ([128])**.** In a biCCC, products always distributes over coproducts.

B.16 ADJUCTIONS

> Indeed, I will make the admittedly provocative claim that adjointness is a concept of fundamental logical and mathematical importance that is not captured elsewhere in mathematics.
>
> — STEVE AWODEY
> *[109, p. 207]*

In a chapter of brisk summaries and definitions, this section is a stands out. It will present adjunctions visually to lend intuition, but this section will do little justice to the very deep topic. For a more comprehensive view of adjunctions, see sources such as [109, Ch. 9], [133, §7.1], [81, Ch. IV] and [64, §2.4].

Adjunctions provide a comprehensive means of universal construction. Products (§B.9), coproducts (§B.10) and exponential objects (§B.15) are common textbook examples of adjunctions. Exponential objects are particularly interesting examples because they cannot be defined by limit/colimit constructions.

The term adjunction is a shorthand for "an adjoint pair of functors". Consider the diagram in $\mathbb{C}\mathbf{at}$:

$$\begin{tikzcd} \boldsymbol{C} \arrow[r, bend right, "R"'] & \boldsymbol{D} \arrow[l, bend right, "L"'] \end{tikzcd}.$$

If it turned out that

$$RL = \mathrm{id}_{\boldsymbol{D}} \quad \text{and} \quad LR = \mathrm{id}_{\boldsymbol{C}} \tag{B.16.17}$$

then we would conclude that $\boldsymbol{C} \cong \boldsymbol{D}$. Isomorphism of categories is a very stringent constraint! To relax that constraint, change equality in (B.16.17) to isomorphism:

$$RL \cong \mathrm{id}_{\boldsymbol{D}} \quad \text{and} \quad LR \cong \mathrm{id}_{\boldsymbol{C}}$$

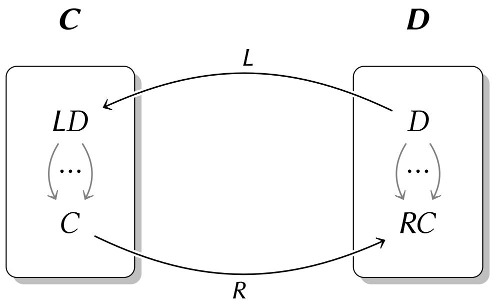
This is called an *equivalence of categories* $\boldsymbol{C}$ and $\boldsymbol{D}$. Adjunction is a further relaxation of equivalence.

DEFINITION B.16-1. (Preliminary definition) An ***adjunction*** between categories $\boldsymbol{C}$ and $\boldsymbol{D}$ is a pair of functors $R : \boldsymbol{C} \to \boldsymbol{D}$ and $L : \boldsymbol{D} \to \boldsymbol{C}$ such that for all $C \in \boldsymbol{C}$ and $D \in \boldsymbol{D}$ there is an isomorphism of hom-sets:

$$\boldsymbol{C}(LD, C) \cong \boldsymbol{D}(D, RC) \tag{B.16.18}$$

natural in C and D.

The data of the former definition can be illustrated in respect to their categories:



$$\boldsymbol{C}(LD, C) \xrightarrow{\qquad\sim\qquad} \boldsymbol{D}(D, RC)\ ,$$

where the arrow $\boldsymbol{C}(LD, C) \xrightarrow{\sim} \boldsymbol{D}(D, RC)$, at the bottom indicates a bijection between the hom-sets shown with gray arrows. Since this bijection exists for any choice of C and D, then there is a family of bijections for any choice of C and D: a natural isomorphism.

NOTATION B.16-2. The adjoint relationship is not symmetrical. We say that L is the *left adjoint of* R and R is the *right adjoint of* L written $L \dashv R$ or in full display:

$$\begin{tikzcd} \boldsymbol{C} \arrow[r, bend right, "R"'] \arrow[r, phantom, "\perp"] & \boldsymbol{D} \arrow[l, bend right, "L"'] \end{tikzcd}$$

(NB: In all cases, the post of the turnstile symbol, $\dashv$, points toward the left adjoint.)

A less intuitive (perhaps) but often useful way to define adjunction is in terms of two natural transformations, the so-called *unit*, η, and *counit*, ε, of the adjunction. Recall that

$$RL \cong \mathrm{id}_{\boldsymbol{D}} \tag{B.16.19}$$

and

$$LR \cong \mathrm{id}_{\boldsymbol{C}}. \tag{B.16.20}$$

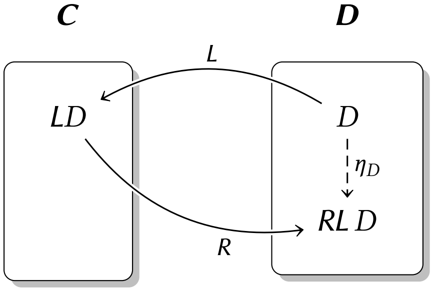
The unit comes from (B.16.19) as a natural transformation

$$\eta : \ \mathrm{id}_{\boldsymbol{D}} \Rightarrow RL,$$

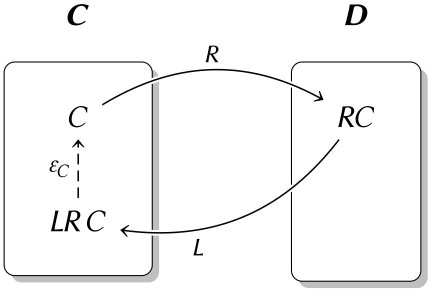
which is one half of that natural isomorphism. Likewise the counit is one direction of (B.16.20):

$$\varepsilon : LR \Rightarrow \mathrm{id}_{\boldsymbol{C}}.$$

If we let $C = LD$ in (B.16.18), we get $\boldsymbol{C}(LD, LD) \cong \boldsymbol{D}(D, RL\,D)$. The hom-set $\boldsymbol{C}(LD, LD)$ has at least one arrow: id_{LD}, and since it is in bijection with $\boldsymbol{D}(D, RL\,D)$ we can expect at least one arrow there too. That arrow is η_D, the component of the unit at D. Graphically, this looks like:



Similarly for the counit, let $D = RC$ in (B.16.18) giving $\boldsymbol{C}(LR\,C, C) \cong \boldsymbol{D}(RC, RC)$. We know that $\mathrm{id}_{RC} \in \boldsymbol{D}(RC, RC)$ and we map that through the natural isomorphism of hom-sets to obtain the component of ε at C. Graphically,



Because the unit and counit are parts of a larger natural isomorphism, (B.16.18), they must be constrained to preserve forward and backward mapping. That leads to the triangle identities in the following definition.

DEFINITION B.16-3. An ***adjunction*** between categories $\boldsymbol{C}$ and $\boldsymbol{D}$ is a quintuple $(L, R, \eta, \varepsilon)$ consisting of pair of functors

$$L : \boldsymbol{D} \to \boldsymbol{C} \quad \text{and} \quad R : \boldsymbol{C} \to \boldsymbol{D}$$

and natural transformations

$$\eta : \mathrm{id}_{\boldsymbol{D}} \Rightarrow LR \quad \text{and} \quad \varepsilon : RL \Rightarrow \mathrm{id}_{\boldsymbol{C}},$$

called unit and counit respectively, both satisfying the triangle identities:

$$\begin{tikzcd}
L \arrow[r, "L\eta"] \arrow[dr, "\mathrm{id}_F"'] & LRL \arrow[d, "\varepsilon F"] \\
& L
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
R \arrow[r, "\eta R"] \arrow[dr, "\mathrm{id}_R"'] & RLR \arrow[d, "R\varepsilon"] \\
& R
\end{tikzcd}
\qquad \text{(B.16.21)}$$

which live in $\mathbf{End}_{\boldsymbol{C}}$ and $\mathbf{End}_{\boldsymbol{D}}$ respectively.

Of course, DEF. B.16-1 and DEF. B.16-3 are equivalent. To connect the two, we can show how to obtain the isomorphism of hom-sets from the unit and counit.

Let $h : LD \to C$ be an arrow in $\boldsymbol{C}(LD, C)$. By means of the adjunction $L \dashv R$ we know that $\boldsymbol{C}(LD, C)$ is isomorphic to $\boldsymbol{D}(D, RC)$. Call the partner arrow, the image of h on the other side of the isomorphism, $h' : D \to RC$. We have the unit and counit components to work with:

$$\eta_D : D \to LR\,D \quad \text{and} \quad \varepsilon_C : RL\,C \to C.$$

Putting these into diagrams we have

$$\begin{tikzcd}
& D \arrow[dl, "\eta_D"'] \arrow[d, "h'"] \\
RL\,D \arrow[r, "Rh"'] & RC
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
LD \arrow[d, "h"'] \arrow[dr, "Lh'"] & \\
C & LR\,C \arrow[l, "\varepsilon_C"]
\end{tikzcd}$$

the commutativity of which gives us the equations

$$h' = Rh \circ \eta_D$$

and

$$h = \varepsilon_C \circ Lh'.$$

This works for all h and h' in their respective hom-sets, and simple substitution shows that these are isomorphisms under the constraints of the triangle equalities of DEF. B.16-3.

B.16.1 Free construction & free-forgetful adjunction

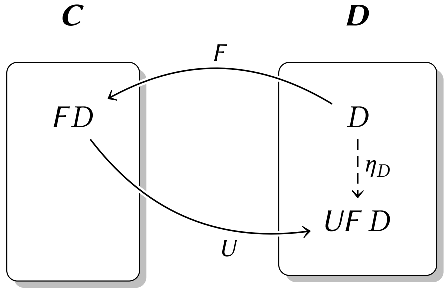
Many structures in mathematics consist of a set with adorning structure. Monoids, groups, rings, magmas, vector spaces, topological spaces, manifolds, and many others are all erected atop an "underlying" set. These are often termed "sets with additional structure". Categorical analysis places these objects into categories as objects connected along their respective homomorphisms. These homomorphisms are functions between underlying sets that preserve the additional structure. Let $\boldsymbol{C}$ be such a category. Then a ***forgetful functor***, commonly denoted U, is a faithful functor $\boldsymbol{C} \to \mathbf{\$et}$ that maps $\boldsymbol{C}$-objects to their underlying sets, and $\boldsymbol{C}$-arrows to their underlying functions. When it exists, the left adjoint $F \dashv U$ is called the "free functor", mapping sets to free objects in $\boldsymbol{C}$. So if $\boldsymbol{C} = \mathbf{Mon}(\mathbf{\$et})$, then F maps sets to free monoids over those sets.

Right-adjoints, dually, are called ***cofree***:

$$\text{free} \dashv \text{forgetful} \dashv \text{cofree}.$$

but cofreeness is not covered here.

The unit, η, of a free-forgetful adjunction, depicted:



is regarded as an "insertion of generators". That is to say that the free $\boldsymbol{C}$-object over a set D can be regarded as containing all expression trees in the semantics of $\boldsymbol{C}$'s constituents (monoids, groups, etc) with leaves on D. This set of expression trees is once again a set, and η identifies the singleton (single-leaf-only) trees, one for each D.

Conversely, the counit ε of the adjunction acts in $\boldsymbol{C}$:

$$\begin{tikzcd}
\boldsymbol{C} & \boldsymbol{D} \\
C \arrow[r, bend left, "U"] & UC \arrow[dl, bend left, "F"] \\
FU\,C \arrow[u, dashed, "\varepsilon_C"] &
\end{tikzcd}.$$

The set at the heart of every $\boldsymbol{C}$-object, C, maps to its free counterpart $FU\,C$. This free object may be regarded as embodying all free expressions of the algebraic structures captured in $\boldsymbol{C}$, taking constants in UC. The counit ε can be seen as an evaluator of all of those free expressions, giving them a value in C.

There is a more “local” notion, a “universal mapping property” (UMP), that captures the concept of a free object. The whole situation is depicted in the following diagram:

$$\begin{tikzcd}
C \arrow[rrr, bend left, "U"] & & UF\,A \arrow[r, "Uf'"] & UC \\
FA \arrow[u, dashed, "\exists!\ f'"] \arrow[urr, "U"] & & A \arrow[u, "\eta_A"] \arrow[ll, "F"] \arrow[ur, bend right, "f"'] &
\end{tikzcd} \tag{B.16.22}$$

$$\begin{tikzcd}
\boldsymbol{C} \arrow[r, shift left, "U"] \arrow[r, phantom, "\top"] & \boldsymbol{D} \arrow[l, shift left, "F"]
\end{tikzcd}$$

Given an arbitrary $N \in \boldsymbol{C}$ and arbitrary function $f : A \to UN$, the diagram inscribes the existence of a unique $\boldsymbol{C}$-arrow $f' : FA \to N$ that underwrites the commutativity of the triangle in $\boldsymbol{D}$. That triangle gives the *universal condition*:

$$f = Uf' \circ \eta_A.$$

This is actually the UMP for the unit. Likewise, there is a UMP for the counit as well [109, p. 214]. It reads, for any $C,\ D \in \boldsymbol{C},\ \boldsymbol{D}$ and $g : F\,C \to D$, there exists a unique $g' : C \to U\,D$ such that

$$g = \varepsilon_D \circ F\,g'.$$

This can all be generalised by replacing $\mathbf{Set}$ by an arbitrary category, which is covered in ch. 3.

SELECTED CODE LISTINGS

C

C.1 From Chapter 3

listing C.1.1: Creating a box-filter from RxCpp components. Running this test also prints output with timing data.

```
TEST_CASE("RxCpp box filter test.") {
  // clang-format off
  auto avg_buffer =
      [](const std::vector<double> &buffer) -> double {
    auto sum = std::accumulate(buffer.begin(), buffer.end(), 0.);
    return sum / buffer.size();
  };

  const auto start_time = std::chrono::steady_clock::now();
  const auto now_string = [&start_time]() -> std::string {
      auto now = std::chrono::steady_clock::now();
      auto duration = std::chrono::duration_cast<std::chrono::microseconds>
        (now - start_time).count();
      std::ostringstream oss;
      oss << "[" << std::setw(4)
                 << std::setfill(' ')
          << duration << " µs]";
      return oss.str();
  };

  auto source =
      rx::observable<>::range(1, 6)
        | rx::map([](auto x) -> double { return (double) x; });

  constexpr auto bufw = 3;

  auto output =
    source
      | rx::tap([&](double x){
          printf("%s src: %.2f\n", now_string().c_str(), x); })
      | rx::buffer(bufw, 1)
      | rx::take_while([](auto& v){return v.size() == bufw;})
      | rx::tap([&](const std::vector<double>& v){
          printf("%s buf: ", now_string().c_str());
          for (const auto& each : v) printf("%.2f ", each);
          printf("\n");
      })
      | rx::map(avg_buffer);

  auto output_record = std::vector<double>();

  output.subscribe(
      [&](double x) {
        printf("%s OnNext: %.2f\n", now_string().c_str(), x);
        output_record.push_back(x);
      },
      [&](){ printf("%s OnComplete\n", now_string().c_str()); });

  REQUIRE( output_record == std::vector<double>{2., 3., 4., 5.} );
  // clang-format on
}
```

C.2 FROM CHAPTER 4

LISTING C.2.1: The `hom` and `dom` notation are implemented as a thin layer of template metaprogram around `std::function`. Though it is simple, credit here goes to Igor Tandetnik on the StackExchange post `https://stackoverflow.com/q/75668291/1827360`.

```
template <typename... Ts>
struct dom {};

template <typename Dom, typename Cod>
struct Hom : public std::function<Cod(Dom)> {
  using std::function<Cod(Dom)>::function;
};

template <typename Cod, typename... Ts>
struct Hom<dom<Ts...>, Cod>
    : public std::function<Cod(Ts...)> {
  using std::function<Cod(Ts...)>::function;
};

template <typename T>
constexpr decltype(auto) id(T &&x) {
  return std::forward<T>(x);
}
```

* * *

LISTING C.2.2: Fundamental types, traits and operators representing ***Cpp*** in C++.

```
template <typename... Ts>
struct Doms {};

template <typename Dom, typename Cod>
struct Hom : public std::function<Cod(Dom)> {
  using std::function<Cod(Dom)>::function;
};

template <typename Cod, typename... Ts>
struct Hom<Doms<Ts...>, Cod> : public std::function<Cod(Ts...)> {
  using std::function<Cod(Ts...)>::function;
};

namespace impl {
  template <typename T>
  struct function_traits;

  template <typename T>
  struct function_traits
      : public function_traits<decltype(&T::operator())> {};

  template <typename Ret, typename... Args>
  struct function_traits<Ret (*)(Args...)> {
    using return_type = Ret;
    using arg_types = std::tuple<Args...>;
  };

  template <typename Ret, typename T, typename... Args>
  struct function_traits<Ret (T::*)(Args...)> {
    using return_type = Ret;
    using arg_types = std::tuple<Args...>;
    using struct_type = T;
  };

  template <typename Ret, typename T, typename... Args>
  struct function_traits<Ret (T::*)(Args...) const> {
    using return_type = Ret;
    using arg_types = std::tuple<Args...>;
    using struct_type = T;
  };

  template <typename Ret, typename T, typename... Args>
  struct function_traits<Ret (T::*)(Args...) volatile> {
    using return_type = Ret;
    using arg_types = std::tuple<Args...>;
    using struct_type = T;
  };

  template <typename Ret, typename T, typename... Args>
  struct function_traits<Ret (T::*)(Args...) const volatile> {
    using return_type = Ret;
    using arg_types = std::tuple<Args...>;
    using struct_type = T;
  };

  template <typename Ret, typename... Args>
  struct function_traits<std::function<Ret(Args...)>> {
    using return_type = Ret;
    using arg_types = std::tuple<Args...>;
  };

} // namespace impl

template <typename Fn>
using Dom = typename std::tuple_element_t<0,
    typename impl::function_traits<Fn>::arg_types>;

template <typename Fn>
using Cod = typename impl::function_traits<Fn>::return_type;
```

* * *

LISTING C.2.3: ***Cpp***/C++: more efficient and explicitly typed implementations of composition and currying.

```
  template <typename Fn>
  constexpr auto compose(Fn fn) {
    return [fn](Dom<Fn> x) -> Cod<Fn> { return fn(x); };
  }

  template <typename Fn, typename Gn>
  constexpr auto compose(Fn fn, Gn gn) {
```

```
  return [fn, gn](Dom<Gn> x) -> Cod<Fn> { return fn(gn(x)); };
}

template <typename Fn, typename Gn, typename Hn>
constexpr auto compose(Fn fn, Gn gn, Hn hn) {
  return [fn, gn, hn](Dom<Hn> x) -> Cod<Fn> {
    return fn(gn(hn(x)));
  };
}

template <typename Fn, typename Gn, typename Hn, typename In>
constexpr auto compose(Fn fn, Gn gn, Hn hn, In in) {
  return [fn, gn, hn, in](Dom<In> x) -> Cod<Fn> {
    return fn(gn(hn(in(x))));
  };
}

template <typename F>
constexpr auto curry(F f) {
  if constexpr (std::is_invocable_v<F>) {
    return std::invoke(f);
  } else if constexpr (
      std::tuple_size_v<
          typename impl::function_traits<F>::arg_types> == 1) {
    return [f](Dom<F> x) -> Cod<F> { return f(x); };
  } else if constexpr (
      std::tuple_size_v<
          typename impl::function_traits<F>::arg_types> == 2) {

    using Dm0 = std::tuple_element_t<0,
        typename impl::function_traits<F>::arg_types>;
    using Dm1 = std::tuple_element_t<1,
        typename impl::function_traits<F>::arg_types>;

    return [fn = f](Dm0 d0) -> Hom<Dm1, Cod<F>> {
      return [&fn, d0](Dm1 d1) {
        return std::invoke(fn, d0, d1);
      };
    };
  } else if constexpr (
      std::tuple_size_v<
          typename impl::function_traits<F>::arg_types> == 3) {

    using Dm0 = std::tuple_element_t<0,
        typename impl::function_traits<F>::arg_types>;
    using Dm1 = std::tuple_element_t<1,
        typename impl::function_traits<F>::arg_types>;
    using Dm2 = std::tuple_element_t<2,
        typename impl::function_traits<F>::arg_types>;

    return [fn = f](Dm0 d0) -> Hom<Dm1, Hom<Dm2, Cod<F>>> {
      return [&fn, d0](Dm1 d1) -> Hom<Dm2, Cod<F>> {
        return [&fn, &d0, d1](Dm2 d2) -> Cod<F> {
          return std::invoke(fn, d0, d1, d2);
        };
      };
    };
  } else if constexpr (
      std::tuple_size_v<
          typename impl::function_traits<F>::arg_types> == 4) {

    using Dm0 = std::tuple_element_t<0,
        typename impl::function_traits<F>::arg_types>;
    using Dm1 = std::tuple_element_t<1,
        typename impl::function_traits<F>::arg_types>;
    using Dm2 = std::tuple_element_t<2,
        typename impl::function_traits<F>::arg_types>;
    using Dm3 = std::tuple_element_t<3,
        typename impl::function_traits<F>::arg_types>;

    return [fn = f](Dm0 d0)
               -> Hom<Dm1, Hom<Dm2, Hom<Dm3, Cod<F>>>> {
      return [&fn, d0](Dm1 d1) -> Hom<Dm2, Hom<Dm3, Cod<F>>> {
        return [&fn, &d0, d1](Dm2 d2) -> Hom<Dm3, Cod<F>> {
          return [&fn, &d0, &d1, d2](Dm3 d3) -> Cod<F> {
            return std::invoke(fn, d0, d1, d2, d3);
          };
        };
      };
    };
  }
}
```

* * *

LISTING C.2.4: ***Cpp***/C++ product type and supporting structure.

```
template <typename T, typename U>
struct P : std::pair<T, U> {
  using std::pair<T, U>::pair;
```

```
};

template <typename T, typename U>
P(T, U) -> P<T, U>;

namespace std {
  template <typename T, typename U>
  struct tuple_element<0, P<T, U>> {
    using type = T;
  };

  template <typename T, typename U>
  struct tuple_element<1, P<T, U>> {
    using type = U;
  };
} // namespace std

template <typename T, typename U>
auto proj_l(P<T, U> tu) -> T {
  return std::get<0>(tu);
}

template <typename T, typename U>
auto proj_r(P<T, U> tu) -> U {
  return std::get<1>(tu);
}

template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename X = Cod<Fn>, typename Y = Cod<Gn>>
auto prod(Fn fn, Gn gn) -> Hom<P<T, U>, P<X, Y>> {
  return [fn, gn](P<T, U> tu) -> P<X, Y> {
    auto [t, u] = tu;
    return {fn(t), gn(u)};
  };
}

struct Pair {

  template <typename T, typename U>
  using Of = P<T, U>;

  template <typename Fn, typename Gn>
  static auto bimap(Fn f, Gn g)
      -> Hom<P<Dom<Fn>, Dom<Gn>>, P<Cod<Fn>, Cod<Gn>>> {
```

```
    return prod(std::forward<Fn>(f), std::forward<Gn>(g));
  }
};

template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename X = Cod<Fn>, typename Y = Cod<Gn>>
auto fanout(Fn fn, Gn gn) -> Hom<T, P<X, Y>> {
  static_assert(std::is_same_v<T, U>);
  return [fn, gn](auto t) -> P<X, Y> {
    static_assert(std::is_invocable_v<Fn, decltype(t)>);
    static_assert(std::is_invocable_v<Gn, decltype(t)>);

    return {fn(t), gn(t)};
  };
}

template <typename T, typename U, typename V>
auto associator_fd(P<T, P<U, V>> t_uv) -> P<P<T, U>, V> {
  auto [t, uv] = t_uv;
  auto [u, v] = uv;

  return {{t, u}, v};
}

template <typename T, typename U, typename V>
auto associator_rv(P<P<T, U>, V> tu_v) -> P<T, P<U, V>> {
  auto [tu, v] = tu_v;
  auto [t, u] = tu;

  return {t, {u, v}};
}

struct I { // monoidal unit for P
  bool operator==(const I) const { return true; }
};

template <typename T>
auto l_unitor_fw(P<I, T> it) -> T {
  return std::get<1>(it);
}

template <typename T>
auto l_unitor_rv(T t) -> P<I, T> {
  return {I{}, t};
```

```
}

template <typename T>
auto r_unitor_fw(P<T, I> ti) -> T {
  return std::get<0>(ti);
}

template <typename T>
auto r_unitor_rv(T t) -> P<T, I> {
  return {t, I{}};
}

template <typename T, typename U>
auto braid(P<T, U> tu) -> P<U, T> {
  auto [t, u] = tu;
  return {u, t};
}
```

* * *

LISTING C.2.5: ***Cpp***/C++ closure of the cartesian monoidal structure.

```
template <typename T, typename U>
auto ev(P<Hom<T, U>, T> fn_and_arg) {
  auto [fn, x] = fn_and_arg;
  return fn(x);
}

template <typename Fn, typename TU = Dom<Fn>,
          typename T = std::tuple_element_t<0, TU>,
          typename U = std::tuple_element_t<1, TU>,
          typename V = Cod<Fn>>
auto pcurry(Fn fn) -> Hom<T, Hom<U, V>> {
  return [fn](T t) -> Hom<U, V> {
    return [fn, t](U u) -> V { return fn({t, u}); };
  };
}

template <typename Fn,              typename T = Dom<Fn>,
          typename UtoV = Cod<Fn>, typename U = Dom<UtoV>,
                                   typename V = Cod<UtoV>>
auto puncurry(Fn fn) -> Hom<P<T, U>, V> {
  return [fn](P<T, U> p) -> V { return fn(p.first)(p.second); };
}
```

*   *   *

Listing C.2.6: ***Cpp***/c++ sum type and supporting structure.

```cpp
template <typename T, typename U>
struct S : std::variant<T, U> {
  using std::variant<T, U>::variant;

  S() = delete;
};

template <typename T, typename U>
S(T, U) -> S<T, U>;

template <std::size_t N, typename S>
struct sum_term;

template <typename T, typename U>
struct sum_term<0, S<T, U>> {
  using type = T;
};

template <typename T, typename U>
struct sum_term<1, S<T, U>> {
  using type = U;
};

template <std::size_t N, typename S>
using sum_term_t = typename sum_term<N, S>::type;

struct Never { // Monoidal unit for S
  Never() = delete;
  Never(const Never &) = delete;
  bool operator==(const Never &) const {
    throw std::domain_error(
        "`Never` instances should not exist, "
        "and someone must have done something perverse.");
  }
};

template <typename T, typename U>
auto inject_l(T t) -> S<T, U> {
  return S<T, U>(std::in_place_index<0>, t);
}

```

```cpp
template <typename T, typename U>
auto inject_r(U t) -> S<T, U> {
  return S<T, U>(std::in_place_index<1>, t);
}

template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename V = Cod<Fn>>
auto fanin(Fn fn, Gn gn) -> Hom<S<T, U>, V> {

  static_assert(std::is_same_v<V, Cod<Gn>>);

  return [fn, gn](S<T, U> t_or_u) -> V {
    static_assert(std::is_invocable_v<Fn, T>);
    static_assert(std::is_invocable_v<Gn, U>);

    if (t_or_u.index() == 0)
      return fn(std::get<0>(t_or_u));
    else
      return gn(std::get<1>(t_or_u));
  };
}

template <typename Fn,          typename Gn,
          typename T = Dom<Fn>, typename U = Dom<Gn>,
          typename X = Cod<Fn>, typename Y = Cod<Gn>>
auto coprod(Fn fn, Gn gn) -> Hom<S<T, U>, S<X, Y>> {
  using TorU = S<T, U>;
  using XorY = S<X, Y>;

  return [fn, gn](TorU t_or_u) -> XorY {
    if (t_or_u.index() == 0)
      return inject_l<X, Y>(fn(std::get<0>(t_or_u)));
    else
      return inject_r<X, Y>(gn(std::get<1>(t_or_u)));
  };
}

struct Either {
  template <typename T, typename U>
  using Of = P<T, U>;

  template <typename Fn, typename Gn>
  static auto bimap(Fn fn, Gn gn) {
    using T = Dom<Fn>;
    using U = Dom<Gn>;
    using X = Cod<Fn>;
    using Y = Cod<Gn>;
    using TorU = S<T, U>;
    using XorY = S<X, Y>;

    return [fn, gn](TorU t_or_u) -> XorY {
      if (t_or_u.index() == 0)
        return inject_l<X, Y>(fn(std::get<0>(t_or_u)));
      else
        return inject_r<X, Y>(gn(std::get<1>(t_or_u)));
    };
  };

  template <typename Fn, typename U>
  static auto lmap(Fn fn) {
    return [fn](Of<Dom<Fn>, U> tu) { return bimap(fn, id<U>); };
  };

  template <typename Gn, typename T>
  static auto rmap(Gn gn) {
    return [gn](Of<T, Dom<Gn>> tu) { return bimap(id<T>, gn); };
  };
};

template <typename T, typename U, typename V>
auto associator_co_fd(S<T, S<U, V>> t_uv) -> S<S<T, U>, V> {
  if (t_uv.index() == 0) {
    if constexpr (!std::is_same_v<T, Never>)
      return inject_l<S<T, U>, V>(std::get<0>(t_uv));
  } else {
    auto &uv = std::get<1>(t_uv);
    if (uv.index() == 0) {
      if constexpr (!std::is_same_v<U, Never>)
        return inject_l<S<T, U>, V>(std::get<0>(uv));
    } else {
      if constexpr (!std::is_same_v<V, Never>)
        return inject_r<S<T, U>, V>(std::get<1>(uv));
    }
  }
  throw std::domain_error("Recieved a variant with no value.");
}

template <typename T, typename U, typename V>
auto associator_co_rv(S<S<T, U>, V> tu_v) -> S<T, S<U, V>> {
  if (tu_v.index() == 0) {
    auto &tu = std::get<0>(tu_v);
    if (tu.index() == 0) {
      if constexpr (!std::is_same_v<T, Never>)
        return inject_l<T, S<U, V>>(std::get<0>(tu));
    } else {
      if constexpr (!std::is_same_v<U, Never>)
        return inject_r<T, S<U, V>>(std::get<1>(tu));
    }
  } else {
    if constexpr (!std::is_same_v<V, Never>)
      return inject_r<T, S<U, V>>(std::get<1>(tu_v));
  }
  throw std::domain_error("Recieved a variant with no value.");
}

template <typename T>
struct S<T, Never> : std::variant<T, Never> {
  using std::variant<T, Never>::variant;

  S() : std::variant<T, Never>{inject_l<T, Never>(T{})} {}

  S(const S &other)
      : std::variant<T, Never>(
            std::in_place_type<T>, std::get<T>(other)) {}
};

template <typename T>
struct S<Never, T> : std::variant<Never, T> {
  using std::variant<Never, T>::variant;

  S() : std::variant<Never, T>{inject_r<Never, T>(T{})} {}

  S(const S &other)
      : std::variant<Never, T>(
            std::in_place_type<T>, std::get<T>(other)) {}
};

template <typename T>
auto l_unitor_co_fw(S<Never, T> just_t) -> T {
  return std::get<1>(just_t);
}

template <typename T>
auto l_unitor_co_rv(T t) -> S<Never, T> {
  return inject_r<Never, T>(t);
}

template <typename T>
auto r_unitor_co_fw(S<T, Never> just_t) -> T {
  return std::get<0>(just_t);
}

template <typename T>
auto r_unitor_co_rv(T t) -> S<T, Never> {
  return inject_l<T, Never>(t);
}

template <typename T, typename U>
auto braid_co(S<T, U> t_or_u) -> S<U, T> {
  if (t_or_u.index() == 0)
    return inject_r<U, T>(std::get<0>(t_or_u));
  else
    return inject_l<U, T>(std::get<1>(t_or_u));
}
```

*   *   *

LISTING C.2.7: ***Cpp***/C++'s distributive structure of ***Cpp***'s product over its coproduct.

```
template <typename T, typename U, typename X>
auto factorise(S<P<T, X>, P<U, X>> tx_ux) -> P<S<T, U>, X> {
  const auto universal_factorise = fanin(
        prod(inject_l<T, U>, id<X>),
        prod(inject_r<T, U>, id<X>)
      );

  return universal_factorise(tx_ux);
}

template <typename T, typename U, typename Z>
auto expand(P<S<T, U>, Z> t_or_u_and_x) -> S<P<T, Z>, P<U, Z>> {
  // tz : T × Z → (T × Z) + (U × Z)
  const auto tz =
      pcurry(compose(
              inject_l<P<T, Z>, P<U, Z>>,
              id<P<T, Z>>
            ));
  // uz : U × Z → (T × Z) + (U × Z)
  const auto uz =
      pcurry(compose(
              inject_r<P<T, Z>, P<U, Z>>,
              id<P<U, Z>>
            ));

  // tz_uz : (T + U) → Z ⊸ (T × Z) + (U × Z)
  const auto tz_uz = fanin(tz, uz);
  // universal_expand : (T + U) × Z  ⊸  (T × Z) + (U × Z)
  const auto universal_expand = puncurry(tz_uz);

  return universal_expand(t_or_u_and_x);
}
```

* * *

LISTING C.2.8: A collection of templates for helping SnocList (§4.8.2) and establishing an isomorphism between SnocList and std::vector.

```
template <typename T>
using maybe_pair_element_t = typename sum_term_t<1, T>::second_type;

template <typename Lst>
using snoclist_element_type = typename std::remove_reference<
    decltype(std::get<1>(out(std::declval<Lst>())))>::type::
    second_type;

template <typename T, typename U>
constexpr bool has_pair(S<I, P<T, U>> const &i_or_val) {
  return i_or_val.index() == 1;
}

template <typename Lst, typename T = snoclist_element_type<Lst>>
auto operator==(Lst const &lhs, Lst const &rhs) -> bool {
  auto l = out(lhs);
  auto r = out(rhs);

  while (has_pair(l) && has_pair(r)) {
    auto [lhs_tail, lhs_val] = std::get<1>(l);
    auto [rhs_tail, rhs_val] = std::get<1>(r);
    if (!(lhs_val == rhs_val)) {
      return false;
    }
    l = *lhs_tail;
    r = *rhs_tail;
  }

  // If one of l or r still hold a pair at this point, then lhs
  // and rhs are different lengths and are not equal.
  return !has_pair(l) && !has_pair(r);
}
```

C.3 FROM CHAPTER 6

LISTING C.3.1: The XY type modelling a 2D Euclidean vector.

```
struct XY {
  double x;
  double y;

  template <typename T>
  inline operator std::tuple<T &, T &>() {
    return std::tuple<T &, T &>{x, y};
  }
};

inline constexpr bool operator==(const XY &a, const XY &b) noexcept {
  return (a.x == b.x) && (a.y == b.y);
}

inline constexpr bool operator!=(const XY &a, const XY &b) noexcept {
  return !(a == b);
}

inline constexpr XY operator+(const XY &a, const XY &b) {
  return XY{a.x + b.x, a.y + b.y};
}

inline constexpr XY operator-(const XY &a, const XY &b) {
  return XY{a.x - b.x, a.y - b.y};
}

inline constexpr XY operator*(const double c, const XY &p) {
  return XY{c * p.x, c * p.y};
}

inline constexpr XY operator*(const XY &p, const double c) { return c * p; }

inline constexpr XY operator/(const XY &p, const double c) {
  return {p.x / c, p.y / c};
}

inline constexpr double dot(const XY& a, const XY& b) {
  return a.x * b.x + a.y * b.y;
}

// TODO: #1 std::hypot isn't constexpr. If I replace with multiplication, then
// l2norm can be constexpr.
inline double l2norm(const XY &p) { return std::hypot(p.x, p.y); }

inline double abs(const XY &p) { return l2norm(p); }

inline constexpr double quadrance(const XY &p) { return p.x * p.x + p.y * p.y; }

inline XY normalise(const XY &p) {
  const double norm = l2norm(p);
  return {p.x / norm, p.y / norm};
}
```

GLOSSARY

Alias Template A type alias with template parameters. I.e., A name that refers to a family of types.

C# C$^{\#}$ is a general-purpose, high-level, multi-paradigm, statically typed programming language developed at Microsoft. It borrows from OCaml and can therefore be regarded as a descendant of ML.

CAML An acronym for Categorical Abstract Machine Language, Caml is a dialect of ML developed from the mid 1980s until CA. 2002. Notably, OCaml is a descendant in common usage today.

Catch2 The Catch2 unit testing framework for C++ [197]. See `https://github.com/catchorg/Catch2`.

Clang The native C language family front-end for LLVM

concrete category A category with a faithful functor into $\mathbb{S}\mathbf{et}$. Most notably, categories with forgetful functors into $\mathbb{S}\mathbf{et}$ are concrete.

Denumerable is a special case of a *countable* set. A set is countable iff its cardinality is less than or equal to $\aleph_0$. A set is *denumerable* iff its cardinality is exactly $\aleph_0$.

Erlang A general-purpose, concurrent, functional high-level programming language originally developed by Ericsson in the mid 1980s for concurrent, real-time telecommunications applications.

F# F$^{\#}$ is a mostly-functional but multiple paradigm, general-purpose, statically typed programming language developed at Microsoft Research.

G++ The GNU compiler collection's C++ compiler

Haskell The quintessential functional programming language named for Haskell Curry, the discoverer of λ-calculus. It was largely developed in academia but has been adopted by parts of the software industry. See the language specification, [111].

LLVM The LLVM compiler infrastructure, a portable set of compiler and toolchain technologies including C and C++ compilers and static analysis tools.

loset Linearly ordered set, AKA a total order.

member function A C++ function formally associated with a data structure (a struct or class) through the language and type system. When the class is instantiated as an object `a`, if the class has a member function `foo` then `a.foo()` is like calling `foo(&a)` because all member functions take a pointer to an instance as an implicit first argument. In the body of the definition of the member function, the implicit pointer is called `this`.

ML Meta Language (ML) is a general-purpose functional programming language, known for its use of the Hindley-Milner type system. It was developed academically and is the basis for many popular functional programming languages. (As popular as a functional language can be at this point in history.)

NumPy A Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. See `http://numpy.org/`.

O'Caml An extension to the CAML dialect of ML, with object oriented features.

poset Partially ordered set.

POSIX IEEE Portable Operating System Interface (The IEEE family of standards for a portable operating system interface.)

Scala A JVM-based language aimed specifically at combining object-oriented and functional programming. (Note: Scala can also be compiled to native machine code through an experimental LLVM front-end.)

Terminator Any particular terminal object in a category (see §B.3.) In short, a terminator 1 in a category $\boldsymbol{C}$ is a $\boldsymbol{C}$-object for which $\boldsymbol{C}(A, 1)$ has exactly one arrow for all $A \in \boldsymbol{C}$.

BIBLIOGRAPHY

[1] S. Eilenberg and S. MacLane, "General theory of natural equivalences," *Transactions of the American Mathematical Society*, vol. 58, pp. 231–294, 1945, ISSN: 0002-9947. DOI: 10.1090/S0002-9947-1945-0013131-6. JSTOR: 1990284.

[2] S. C. Kleene, "Representation of events in nerve nets and finite automata," *Automata Studies: Annals of Mathematics Studies*, no. 34, p. 3, 1956.

[3] Л. С. Понтрягин, В. Г. Болтянский, Р. В. Гамкрелидзе, and Е. Ф. Мищенко, "Математическая теория оптимальных процессов. (towards a theory of optimal processes)," *Доклады Академии Наук СССР—Proceedings of the USSR Academy of Science*, vol. 110, no. 1, pp. 7–10, 1956.

[4] F. W. Lawvere, "Functorial semantics of algebraic theories and some algebraic problems in the context of functorial semantics of algebraic theories," Ph.D. dissertation, Columbia University, 1963.

[5] M. A. Arbib, "A common framework for automata theory and control theory," *Journal of the Society for Industrial and Applied Mathematics Series A Control*, vol. 3, no. 2, pp. 206–222, 1965. DOI: 10.1137/0303017.

[6] Y. Give'on, "Toward a homological algebra of automata," University of Michigan, Logic of Computers Group, Ann Arbor, MI, Tech. Rep., Feb. 1965, U.S. Army Research Office, Durham NC, Grant № DA-ARO(D)-31-124-G588, Project № 4049-M and Office of Naval Research, DC., Contract № Nonr-1224(21).

[7] ——, "Transparent categories and categories of transition systems," University of Michigan, Logic of Computers Group, Tech. Rep., May 1965, U.S. Office of Naval Research, DC., Contract № Nonr-1224(21).

[8] G. Hotz, "Eine algebraisierung des syntheseproblems von schaltkreisen i.," *Elektronische Informationsverarbeitung und Kybernetik*, vol. 1, no. 3, pp. 185–205, 1965. [Online]. Available: https://ncatlab.org/nlab/files/HotzSchaltkreise.pdf.

[9] ——, "Eine algebraisierung des syntheseproblems von schaltkreisen i.," *Elektronische Informationsverarbeitung und Kybernetik*, vol. 1, no. 4, pp. 209–231, 1965. [Online]. Available: https://ncatlab.org/nlab/files/HotzSchaltkreise.pdf.

[10] M. A. Arbib, "Automata theory and control theory—a rapprochement," *Automatica*, vol. 3, no. 3, pp. 161–189, 1966, ISSN: 0005-1098. DOI: 10.1016/0005-1098(66)90011-2.

[11] A. W. Burks, "Language and automata, Final report," University of Michigan, Logic of Computers Group, Ann Arbor, MI, Tech. Rep., Oct. 1966, U.S. Army Research Office, Durham NC, Grant № DA-ARO(D)-31-124-G665.

[12] Y. Give'on, "On some categorical algebra aspects of automata theory: The categorical properties of transition systems," University of Michigan, Logic of Computers Group, Ann Arbor, MI, Tech. Rep., Feb. 1966, U.S. Army Research Office, Durham NC, Grant № DA-ARO(D)-31-124-G665, Project № 4049-M and Office of Naval Research, DC., Contract № Nonr-1224(21).

[13] S. Eilenberg and J. B. Wright, "Automata in general algebras," *Information and Control*, vol. 11, no. 4, pp. 452–470, 1967, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(67)90670-5.

[14] E. G. Manes, "Manes a tripie miscellany, Some aspects of the theory of algebras over a triple," Ph.D. dissertation, Wesleyan University, 1967.

[15] M. A. Arbib and Y. Give'on, "Algebra automata I, Parallel programming as a prolegomena to the categorical approach," *Information and Control*, vol. 12, no. 4, pp. 331–345, 1968, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(68)90374-4.

[16] Y. Give'on and M. A. Arbib, "Algebra automata II, The categorical framework for dynamic analysis," *Information and Control*, vol. 12, no. 4, pp. 346–370, 1968, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(68)90381-1.

[17] J. Lambek, "A fixpoint theorem for complete categories," *Mathematische Zeitschrift*, vol. 103, pp. 151–161, 2 1968, ISSN: 1432-1823. DOI: 10.1007/BF01110627.

[18] M. A. Arbib and H. P. Zeiger, "On the relevance of abstract algebra to control theory," *Automatica*, vol. 5, no. 5, pp. 589–606, 1969, ISSN: 0005-1098. DOI: 10.1016/0005-1098(69)90026-0.

[19] J. R. Hindley, "The principal type-scheme of an object in combinatory logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969, ISSN: 0002-9947.

[20] R. E. Kálmán, P. L. Falb, and M. A. Arbib, *Topics in mathematical system theory* (International series in pure and applied mathematics), W. T. Martin and E. H. Spainer, Eds. New York: McGraw-Hill, 1969.

[21] M. Barr, "Coequalizers and free triples," *Mathematische Zeitschrift*, vol. 116, no. 4, pp. 307–322, Dec. 1970, ISSN: 1432-1823. DOI: 10.1007/BF01111838.

[22] R. Davis, "Universal coalgebra and categories of transition systems," *Mathematical systems theory*, vol. 4, no. 1, pp. 91–95, Mar. 1970, ISSN: 1433-0490. DOI: 10.1007/BF01705889.

[23] M. A. Arbib and E. G. Manes, "Machines in a category, Preliminary report," in *Notices of the American Mathematical Society*, vol. 19, 1972, (A–)389.

[24] M. L. Laplaza, "Coherence for distributivity," in *Coherence in Categories*, G. M. Kelly, M. Laplaza, G. Lewis, and S. Mac Lane, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1972, pp. 29–65, ISBN: 978-3-540-37958-4.

[25] J. Adámek, "Free algebras and automata realizations in the language of categories," *Commentationes Mathematicae Universitatis Carolinae*, vol. 15, no. 4, pp. 589–602, 1974.

[26] M. A. Arbib and E. G. Manes, “Basic concepts of category theory applicable to computation and control,” in *Proceedings of the Proceedings of the First International Symposium on Category Theory Applied to Computation and Control*, Berlin, Heidelberg: Springer-Verlag, 1974, pp. 1–34, ISBN: 978-3-540-07142-6.

[27] ——, “Foundations of system theory, Decomposable systems,” *Automatica*, vol. 10, no. 3, pp. 285–302, 1974, ISSN: 0005-1098. DOI: 10.1016/0005-1098(74)90039-9.

[28] ——, “Machines in a category: An expository introduction,” *SIAM Review*, vol. 16, no. 2, pp. 163–192, 1974. DOI: 10.1137/1016026.

[29] S. Alagić, “Natural state transformations,” *Journal of Computer and System Sciences*, vol. 10, no. 2, pp. 266–307, 1975, ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80045-6.

[30] M. A. Arbib and E. G. Manes, “A categorist’s view of automata and systems,” in *Category Theory Applied to Computation and Control*, E. G. Manes, Ed., vol. 25, Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 51–64, ISBN: 978-3-540-37426-8. DOI: 10.1007/3-540-07142-3_61.

[31] ——, “Adjoint machines, state-behavior machines, and duality,” *Journal of Pure and Applied Algebra*, vol. 6, no. 3, pp. 313–344, 1975, ISSN: 0022-4049. DOI: 10.1016/0022-4049(75)90028-6.

[32] ——, *Arrows, Structures, and Functors, The Categorical Imperative.* New York: Academic Press, 1975.

[33] J. A. Goguen, “Discrete-time machines in closed monoidal categories. i,” *Journal of Computer and System Sciences*, vol. 10, no. 1, pp. 1–43, 1975, ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80012-2.

[34] J. A. Goguen, “Semantics of computation,” in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1975, pp. 151–163. DOI: 10.1007/3-540-07142-3_75.

[35] A. E. B. Jr. and Y.-C. Ho, *Applied Optimal Control, Optimization, Estimation, and Control.* Taylor & Francis, 1975, ISBN: 978-0-89116-228-5.

[36] V. Trnková, J. Adámek, V. Koubek, and J. Reiterman, “Free algebras, input processes and free monads,” *Commentationes Mathematicae Universitatis Carolinae*, vol. 16, no. 2, pp. 339–351, 1975.

[37] J. Adámek, “Realization theory for automata in categories,” *Journal of Pure and Applied Algebra*, vol. 9, no. 2, pp. 281–296, 1977, ISSN: 0022-4049. DOI: 10.1016/0022-4049(77)90071-8.

[38] J. Adámek and V. Koubek, “Remarks on fixed points of functors,” in *Fundamentals of Computation Theory*, M. Karpiński, Ed., Berlin, Heidelberg: Springer, 1977, pp. 199–205, ISBN: 978-3-540-37084-0.

[39] ——, “Least fixed point of a functor,” *Journal of Computer and System Sciences*, vol. 19, no. 2, pp. 163–178, 1979, ISSN: 0022-0000. DOI: 10.1016/0022-0000(79)90026-6.

[40] M. A. Arbib and E. G. Manes, "Intertwined recursion, tree transformations and linear systems," *Information and Control*, vol. 40, no. 2, pp. 144–180, 1979, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(79)90370-X.

[41] J. Adámek, H. Ehrig, and V. Trnková, "On an equivalence of system-theoretical and categorical concepts.," *Kybernetika*, vol. 16, no. 5, pp. 389–410, 1980.

[42] M. A. Arbib and E. G. Manes, "Machines in a category," *Journal of Pure and Applied Algebra*, vol. 19, pp. 9–20, 1980, ISSN: 0022-4049. DOI: 10.1016/0022-4049(80)90090-0.

[43] A. Ralston and M. Shaw, "Curriculum '78—is computer science really that unmathematical?" *Commun. ACM*, vol. 23, no. 2, pp. 67–70, Feb. 1980, ISSN: 0001-0782. DOI: 10.1145/358818.358820.

[44] J. Adámek, "Observability and nerode equivalence in concrete categories," in *Fundamentals of Computation Theory*, F. Gécseg, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 1–15, ISBN: 978-3-540-38765-7.

[45] J. Adámek and V. Trnková, "Varietors and machines in a category," *Algebra Universalis*, vol. 13, no. 1, pp. 89–132, 1981, ISSN: 1420-8911. DOI: 10.1007/BF02483826.

[46] S. A. Greibach, "Formal languages: Origins and directions," *IEEE Annals of the History of Computing*, vol. 3, no. 1, pp. 14–41, 1981. DOI: 10.1109/mahc.1981.10006.

[47] M. A. Arbib and E. G. Manes, "Parametrized data types do not need highly constrained parameters," *Information and Control*, vol. 52, no. 2, pp. 139–158, 1982, ISSN: 0019-9958. DOI: 10.1016/S0019-9958(82)80026-0.

[48] D. S. Scott, "Domains for denotational semantics," in *Automata, Languages and Programming*, M. Nielsen and E. M. Schmidt, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 577–610, ISBN: 978-3-540-39308-5.

[49] M. B. Smyth and G. D. Plotkin, "The category-theoretic solution of recursive domain equations," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 761–783, 1982. DOI: 10.1137/0211062.

[50] E. W. Dijkstra *et al.*, "Invariance and non-determinacy," *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 312, no. 1522, pp. 491–499, 1984. DOI: 10.1098/rsta.1984.0072.

[51] C. Gunter, "Profinite solutions for recursive domain equations," Ph.D. dissertation, Carnegie-Mellon University, 1985.

[52] M. A. Arbib and E. G. Manes, *Algebraic Approaches to Program Semantics*, 1st ed. New York, NY: Springer, 1986, ISBN: 978-1-4612-4962-7. DOI: 10.1007/978-1-4612-4962-7.

[53] R. C. Davis, "Toposes, monoid actions, and universal coalgebra," *Proceedings of the American Mathematical Society*, vol. 98, no. 4, pp. 547–552, 1986. DOI: 10.1090/s0002-9939-1986-0861747-9.

[54] P. Dybjer, "Category theory and programming language semantics, An overview," in *Category Theory and Computer Programming: Tutorial and Workshop, Guildford, U.K. September 16–20, 1985 Proceedings*, D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 163–181, ISBN: 978-3-540-47213-1. DOI: `10.1007/3-540-17162-2_121`.

[55] J. Adámek and V. Trnková, *Automata and Algebras in Categories* (Mathematics and its Applications), 1st ed. Springer Dordrecht, 1989, ISBN: 978-0-7923-0010-6.

[56] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Feb. 1989. DOI: `10.1093/comjnl/32.2.98`.

[57] G. R. Malcolm, "Algebraic data types and program transformation," Ph.D. dissertation, Rijksuniversiteit Groningen, 1990.

[58] ——, "Data structures and program transformation," *Science of Computer Programming*, vol. 14, no. 2, pp. 255–279, 1990, ISSN: 0167-6423. DOI: `10.1016/0167-6423(90)90023-7`.

[59] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, ser. LFP '90, Nice, France: ACM, 1990, pp. 61–78, ISBN: 0-89791-368-X. DOI: `10.1145/91556.91592`.

[60] M. R. Fellows, "Computer science and mathematics in the elementary schools," 1991.

[61] *Journal of Functional Programming* 1991–, Published by Cambridge University Press, ISSN: 1469-7653.

[62] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Proc. 5th ACM Conference on Functional Programming Languages and Computer Architecture*, J. Hughes, Ed., vol. 523, Springer-Verlag, Aug. 1991, pp. 124–144.

[63] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991, ISSN: 0890-5401. DOI: `10.1016/0890-5401(91)90052-4`.

[64] B. C. Pierce, *Basic Category Theory for Computer Scientists.* Cambridge, Massachusetts: The MIT Press, 1991, ISBN: 0-262-66071-7.

[65] P. Wadler, "The essence of functional programming," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, New York, USA: ACM Press, 1992, pp. 1–14, ISBN: 0897914538. DOI: `10.1145/143165.143169`.

[66] M. Barr, "Terminal coalgebras in well-founded set theory," *Theoretical Computer Science*, vol. 114, no. 2, pp. 299–315, 1993, ISSN: 0304-3975. DOI: `10.1016/0304-3975(93)90076-6`.

[67] D. Čubrić, "Interpolation property for bicartesian closed categories," *Archive for Mathematical Logic*, vol. 33, no. 4, pp. 291–319, 1994, ISSN: 1432-0665. DOI: `10.1007/BF01270628`.

[68] C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi, "Tbag: A high level framework for interactive, animated 3d graphics applications," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '94, New York, NY, USA: Association for Computing Machinery, 1994, pp. 421–434, ISBN: 978-0-89791-667-7. DOI: `10.1145/192161.192276`.

[69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994, ISBN: 0201633612.

[70] J. Adámek and V. Koubek, "On the greatest fixed point of a set functor," *Theoretical Computer Science*, vol. 150, no. 1, pp. 57–75, 1995, ISSN: 0304-3975. DOI: `10.1016/0304-3975(95)00011-K`.

[71] J. J. M. M. Rutten, "A calculus of transition systems (towards universal coalgebra)," Amsterdam, The Netherlands, Tech. Rep., 1995.

[72] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd. Cambridge, MA, USA: MIT Press, 1996, ISBN: 0-262-01153-0.

[73] B. Jacobs, "Objects and classes, co-algebraically," in *Object Orientation with Parallelism and Persistence*, B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, Eds. Boston, MA: Springer US, 1996, pp. 83–103, ISBN: 978-1-4613-1437-0. DOI: `10.1007/978-1-4613-1437-0_5`.

[74] G. R. Malcolm, "Behavioural equivalence, bisimulation, and minimal realisation," in *Recent Trends in Data Type Specification*, M. Haveraaen, O. Owe, and O.-J. Dahl, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 359–378, ISBN: 978-3-540-70642-7.

[75] J. C. Willems, "The behavioural approach to systems and control," *European Journal of Control*, vol. 2, no. 4, pp. 250–259, 1996, ISSN: 0947-3580. DOI: `10.1016/S0947-3580(96)70050-X`.

[76] L. A. Zadeh, "The evolution of systems analysis and control: A personal perspective," *IEEE Control Systems Magazine*, vol. 16, no. 3, pp. 95–98, 1996. DOI: `10.1109/37.506401`.

[77] R. Bird and O. de Moor, *Algebra of Programming* (International Series in Computer Science), C. Hoare, Ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1997, ISBN: 0-13-507245-X.

[78] C. Elliott and P. R. Hudak, "Functional reactive anmation," *ACM SIGPLAN Notices*, vol. 32, no. 8, pp. 263–273, 1997.

[79] L. S. Moss and N. Danner, "On the foundations of corecursion," *Logic Journal of the IGPL*, vol. 5, no. 2, pp. 231–257, Mar. 1997, ISSN: 1367-0751. DOI: `10.1093/jigpal/5.2.231`.

[80] B. Jacobs, "Coalgebraic reasoning about classes in object-oriented languages," *Electronic Notes in Theoretical Computer Science*, vol. 11, pp. 231–242, 1998, CMCS '98, First Workshop on Coalgebraic Methods in Computer Science, ISSN: 1571-0661. DOI: `10.1016/S1571-0661(04)00061-1`.

[81] S. Mac Lane, *Categories for the Working Mathematician*, 2nd ed. Springer, 1998, ISBN: 978-0-387-98403-2.

[82] M. Barr and C. Wells, *Category Theory for Computing Science.* Centre de Recherches Mathématiques, 1999.

[83] L. S. Moss, "Coalgebraic logic," *Annals of Pure and Applied Logic*, vol. 96, no. 1, pp. 277–317, 1999, ISSN: 0168-0072. DOI: 10.1016/S0168-0072(98)00042-6.

[84] P. Taylor, *Practical Foundations of Mathematics* (Cambridge Studies in Advanced Mathematics). Cambridge University Press, 1999, ISBN: 0-521-63107-6.

[85] J. Goguen and G. R. Malcolm, "A hidden agenda," *Theoretical Computer Science*, vol. 245, no. 1, pp. 55–101, 2000, ISSN: 0304-3975. DOI: 10.1016/S0304-3975(99)00275-3.

[86] J. Hughes, "Generalising monads to arrows," *Science of Computer Programming*, vol. 37, no. 1, pp. 67–111, 2000, ISSN: 0167-6423. DOI: 10.1016/S0167-6423(99)00023-4.

[87] B. Jacobs, "Object-oriented hybrid systems of coalgebras plus monoid actions," *Theoretical Computer Science*, vol. 239, no. 1, pp. 41–95, 2000, ISSN: 0304-3975. DOI: 10.1016/S0304-3975(99)00213-3.

[88] J. J. M. M. Rutten, "Universal coalgebra: A theory of systems," *Theoretical Computer Science*, vol. 249, no. 1, pp. 3–80, 2000, Modern Algebra, ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00056-6.

[89] G. J. Sutton and R. R. Bitmead, "Performance and computational implementation of nonlinear model predictive control on a submarine," in *Nonlinear Model Predictive Control*, ser. Progress in Systems and Control Theory, F. Allgöwer, A. Zheng, and C. I. Byrnes, Eds., vol. 26, Birkhäuser Basel, 2000, pp. 461–472, ISBN: 978-3-0348-8407-5. DOI: 10.1007/978-3-0348-8407-5_27.

[90] Z. Wan and P. Hudak, "Functional reactive programming from first principles," *ACM SIGPLAN Notices*, 2000.

[91] Z. Wan, "Functional reactive programming for real-time reactive systems," Ph.D. dissertation, Yale University, 2002, ISBN: 978-0-493-87916-1.

[92] H. Gill and J. Bay, "Introduction to the SEC vision," in *Software-Enabled Control: Information Technology for Dynamical Systems*, T. Samad and G. Balas, Eds., Wiley-IEEE Press, 2003, ch. 1, pp. 3–8, ISBN: 978-0-471-23436-4.

[93] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Summer School on Advanced Functional Programming 2002, Oxford University*, ser. Lecture Notes in Computer Science, vol. 2638, Springer-Verlag, 2003, pp. 159–187.

[94] D. E. Kirk, *Optimal Control Theory: An Introduction.* Dover Publications, 2004, ISBN: 0486434842.

[95] M. Spivak, *A comprehensive introduction to differential geometry.* 3rd ed. Houston, Texas, USA: Publish or Perish Inc., 2005, vol. I, ISBN: 0-914098-70-5.

[96] *Unified modeling language specification*, 19501, ISO/IEC, 2005.

[97] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons, "Fast and loose reasoning is morally correct," *SIGPLAN Not.*, vol. 41, no. 1, pp. 206–217, Jan. 2006, ISSN: 0362-1340. DOI: 10.1145/1111320.1111056.

[98] F. Fahimi, "Non-linear model predictive formation control for groups of autonomous surface vessels," *International Journal of Control*, vol. 80, no. 8, pp. 1248–1259, Aug. 2007, ISSN: 0020-7179. DOI: 10.1080/00207170701280911.

[99] M. Hyland and J. Power, "The category theoretic understanding of universal algebra, Lawvere theories and monads," *Electronic Notes in Theoretical Computer Science*, vol. 172, pp. 437–458, 2007, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.02.019.

[100] H. Liu and P. Hudak, "Plugging a space leak with an arrow," *Electronic Notes in Theoretical Computer Science*, vol. 193, pp. 29–45, 2007, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.10.006.

[101] J. C. Willems, "The behavioral approach to open and interconnected systems," *IEEE Control Systems Magazine*, vol. 27, no. 6, pp. 46–99, Dec. 2007, ISSN: 1066-033X. DOI: 10.1109/MCS.2007.906923.

[102] J. R. Hindley and J. P. Seldin, *Lambda-calculus and combinators, an introduction*, 2nd ed. Cambridge University Press, 2008, ISBN: 978-0-521-89885-0.

[103] C. M. Elliott, "Push-pull functional reactive programming," in *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, ser. Haskell '09, Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 25–36, ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596643.

[104] J. Gibbons and B. C. d. S. Oliveira, "The essence of the iterator pattern," *Journal of Functional Programming*, vol. 19, no. 3-4, pp. 377–402, 2009. DOI: 10.1017/S0956796809007291.

[105] J. Iry. "A brief, incomplete, and mostly wrong history of programming languages." (2009), [Online]. Available: http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html (visited on 07/08/2023).

[106] *Quantities and units–Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*, 80000-2:2009, ISO/IEC TC12, 2009.

[107] P. Zarchan and H. Musoff, *Fundamentals of Kalman Filtering, A Practical Approach* (Progress in Astronautics and Aeronautics), 3rd ed., F. K. Lu, Ed. American Institute of Aeronautics & Astronautics, 2009, vol. 232, ISBN: 978-1-60086-718-7. DOI: 10.2514/4.867200.

[108] J. Adámek, S. Milius, and L. S. Moss, "Initial algebras and terminal coalgebras, A survey," Unplished draft, used as course text at the European Summer School in Logic, Language and Information 2010, 2010.

[109] S. Awodey, *Category Theory*, 2nd ed. Oxford University Press, 2010, ISBN: 978-0-19-923718-0.

[110] R. Hinze, "Reasoning about codata," in *Central European Functional Programming School: Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, Z. Horváth, R. Plasmeijer, and V. Zsók, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 42–93, ISBN: 978-3-642-17685-2. DOI: 10.1007/978-3-642-17685-2_3.

[111] S. Marlow, Ed., *Haskell 2010 language report*, haskell.org, 2010. [Online]. Available: https://wiki.haskell.org/Language_and_library_specification.

[112] E. Meijer, "Subject/observer is dual to iterator," in *FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation*, 2010.

[113] *Rx design guidelines*, version 1.0, Microsoft Corp., 2010. [Online]. Available: http://go.microsoft.com/fwlink/?LinkID=205219.

[114] P. Selinger, "A survey of graphical languages for monoidal categories," in *New Structures for Physics* (Lecture Notes in Physics), B. Coecke, Ed., Lecture Notes in Physics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 813, pp. 289–355, ISBN: 978-3-642-12821-9. DOI: 10.1007/978-3-642-12821-9_4.

[115] J. C. Baez and M. Stay, "Physics, topology, logic and computation: A rosetta stone," in *New Structures for Physics*, B. Coecke, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–172, ISBN: 978-3-642-12821-9. DOI: 10.1007/978-3-642-12821-9_2.

[116] *Information technology–programming languages–C++*, 3rd ed., 14882:2011, ISO/IEC JTC1/SC22/WG21, 2011.

[117] H. J. M. Meijer, J. W. Dyer, and D. A. Manolescu, "Push-based application program interface based on duals of a pull-based application program interface," Appl. № 12/633,160, Pub. №. US 2011/0138403 A1, 2011, US Cl. 719/328.

[118] R. Bringhurst, *The Elements of Typographic Style*. 2012, ISBN: 978-0-88179-110-5.

[119] J. M. Eklund, J. Sprinkle, and S. S. Sastry, "Switched and symmetric pursuit-/evasion games using online model predictive control with application to autonomous aircraft," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 3, pp. 604–620, May 2012, ISSN: 1063-6536. DOI: 10.1109/TCST.2011.2136435.

[120] E. Meijer, "Your mouse is a database," *Queue*, vol. 10, no. 3, 20:20–20:33, Mar. 2012, ISSN: 1542-7730. DOI: 10.1145/2168796.2169076. [Online]. Available: http://doi.acm.org/10.1145/2168796.2169076.

[121] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys*, vol. 45, no. 52, 4 Aug. 2013, ISSN: 0360-0300. DOI: 10.1145/2501654.2501666.

[122] N. Gambino and J. Kock, "Polynomial functors and polynomial monads," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 154, no. 1, pp. 153–192, 2013. DOI: 10.1017/S0305004112000394.

[123] E. Neibler. "F-algebras and C++." (2013), [Online]. Available: `http://ericniebler.com/2013/07/16/f-algebras-and-c/`.

[124] D. I. Spivak, "The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits," May 2013. eprint: `arXiv:1305.0297[cs.DB]`.

[125] T. A. V. Teatro and J. M. Eklund, "Nonlinear model predictive control for omnidirectional robot motion planning and tracking," English, in *Canadian Conference on Electrical and Computer Engineering*, Ieee, May 2013, pp. 1–4, ISBN: 978-1-4799-0033-6. DOI: `10.1109/ccece.2013.6567713`.

[126] D. A. Turner, "Some history of functional programming languages," in *Trends in Functional Programming*, H.-W. Loidl and R. Peña, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–20, ISBN: 978-3-642-40447-4.

[127] D. Ahman and T. Uustalu, "Coalgebraic update lenses," *Electronic Notes in Theoretical Computer Science*, vol. 308, pp. 25–48, 2014, Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, ISSN: 1571-0661. DOI: `10.1016/j.entcs.2014.10.003`.

[128] M. Benini, "Cartesian closed categories are distributive," Jun. 2014. arXiv: `1406.0961 [math.CT]`.

[129] V. J. B. Escribá and P. Talbot, *Proposal to add a utility class to represent expected monad*, N4015, ISO/IEC JTC1/SC22/WG21/LWG, 2014. [Online]. Available: `https://wg21.link/n4015`.

[130] T. Leinster, *Basic Category Theory*. Cambridge University Press, 2014, ISBN: 978-1-107-04424-1. DOI: `10.1017/CBO9781107360068`.

[131] E. Meijer. "What does it mean to be Reactive?" ▶, Talk presented at React Conference. (2014), [Online]. Available: `https://youtu.be/sTSQlYX5DU0`.

[132] S. Pinker, *The sense of style, The thinking person's guide to writing in the 21st century*. Penguin Books, 2014, ISBN: 978-0-670-02585-5.

[133] D. I. Spivak, *Category theory for the sciences*. Cambridge, Massachusetts: The MIT Press, 2014, ISBN: 978-0-262-02813-4.

[134] H. Sutter, *Unified call syntax*, N4165, ISO/IEC JTC1/SC22/WG21/EWG, 2014. [Online]. Available: `https://wg21.link/n4165`.

[135] T. A. V. Teatro, J. M. Eklund, and R. Milman, "Nonlinear model predictive control for omnidirectional robot motion planning and tracking with avoidance of moving obstacles," *Canadian Journal of Electrical and Computer Engineering*, vol. 37, no. 3, pp. 151–156, 2014, ISSN: 0840-8688. DOI: `10.1109/cjece.2014.2328973`.

[136] K. Ahnert and M. Mulansky. "Boost.Numeric.Odeint." Boost C++ Libraries. (2015), [Online]. Available: `https://www.boost.org/libs/numeric/odeint/`.

[137] Z. Hu, J. Hughes, and M. Wang, "How functional programming mattered," *National Science Review*, vol. 2, no. 3, pp. 349–370, Sep. 2015. DOI: `10.1093/nsr/nwv042`.

[138] T. A. V. Teatro, P. McNelles, and J. M. Eklund, "A formulation of rod based nonlinear model predictive control of nuclear reaction with temperature effects and xenon poisoning," in *Proceedings of the 23rd International Conference on Nuclear Engineering*, Chiba, Japan: Jsme, 2015.

[139] D. Vagner, D. I. Spivak, and E. Lerman, "Algebras of open dynamical systems on the operad of wiring diagrams," *Theory and Applications of Categories*, vol. 30, no. 51, pp. 1793–1822, 2015.

[140] J. Adámek, V. Koubek, and T. Palm, "Fixed points of set functors: How many iterations are needed?" *Applied Categorical Structures*, vol. 24, no. 5, pp. 649–661, Oct. 2016, ISSN: 1572-9095. DOI: `10.1007/s10485-016-9451-1`.

[141] D. Ahman and T. Uustalu, "Directed containers as categories," *Electronic Proceedings in Theoretical Computer Science*, vol. 207, pp. 89–98, Apr. 2016. DOI: `10.4204/eptcs.207.5`.

[142] B. Beckman. "Kalman folding." version 1. viXra: `1606.0328`. (2016), [Online]. Available: `http://vixra.org/abs/1606.0328`.

[143] ——, "Kalman folding 1.5: Running statistics." version 1. viXra: `1609.0044`. (2016), [Online]. Available: `http://vixra.org/abs/1609.0044`.

[144] ——, "Kalman folding 2: Tracking and system dynamics." version 2. viXra: `1606.0348`. (2016), [Online]. Available: `http://vixra.org/abs/1606.0348`.

[145] ——, "Kalman folding 3: Derivations." version 1. viXra: `1607.0059`. (2016), [Online]. Available: `http://vixra.org/abs/1607.0059`.

[146] ——, "Kalman folding 4: Streams and observables." version 1. viXra: `1607.0141`. (2016), [Online]. Available: `http://vixra.org/abs/1607.0141`.

[147] S. Blackheath and A. Jones, *Functional Reactive Programming*. Manning Publications, 2016, ISBN: 978-1-63343-010-5.

[148] B. Fong, P. Sobociński, and P. Rapisarda, "A categorical approach to open and interconnected dynamical systems," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '16, New York, NY, USA: Association for Computing Machinery, 2016, pp. 495–504, ISBN: 9781450343916. DOI: `10.1145/2933575.2934556`.

[149] J. Kock, "Notes on polynomial functors," 2016. [Online]. Available: `http://mat.uab.cat/~kock/cat/polynomial.html`.

[150] K. Sawada and T. Watanabe, "Emfrp: A functional reactive programming language for small-scale embedded systems," in *Companion Proceedings of the 15th International Conference on Modularity*, ser. MODULARITY Companion 2016, Málaga, Spain: Association for Computing Machinery, 2016, pp. 36–44, ISBN: 978-1-4503-4033-5. DOI: `10.1145/2892664.2892670`.

[151] T. A. V. Teatro and J. M. Eklund, "An object oriented framework for a generic model predictive controller," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, IEEE, May 2016, pp. 1–4, ISBN: 978-1-4673-8721-7. DOI: `10.1109/ccece.2016.7726652`.

[152] B. Beckman. "Kalman folding for the brave and true," ▶ YOW! 2017 conference talk. (2017), [Online]. Available: https://youtu.be/peJFVe-4qz4.

[153] M. Behrisch, S. Kerkhoff, R. Pöschel, F. M. Schneider, and S. Siegmund, "Dynamical systems in categories," *Applied Categorical Structures*, vol. 25, no. 1, pp. 29–57, Feb. 2017, ISSN: 1572-9095. DOI: 10.1007/s10485-015-9409-8.

[154] E. Bertoluzzo, "The essence of reactive programming, A theoretical approach," Master's thesis, Delft University of Technology, 2017.

[155] S. Brand, *Monadic operations for std::optional*, P0798r0, ISO/IEC JTC1/SC22/WG21/[SG14,LEWG], 2017. [Online]. Available: https://wg21.link/p0798r0.

[156] L. Dionne. "Boost.Hana." Boost C++ Libraries. (2017), [Online]. Available: http://boostorg.github.io/hana/.

[157] N. Douglas. "Introduction to the Proposed std::expected," ▶, Talk presented at Meeting C++ 2017. (2017), [Online]. Available: https://youtu.be/JfMBLx7qE0I.

[158] *Information technology–programming languages–C++*, 5th ed., ISO/IEC 14882:2017, ISO/IEC JTC1/SC22/WG21, 2017.

[159] B. Jacobs, *Introduction to coalgebra, towards mathematics of states and observation* (Cambridge tracts in theoretical computer science 59). Cambridge, United Kingdom: Cambridge University Press, 2017, ISBN: 978-1-107-17789-5.

[160] J. Adámek, S. Milius, and L. S. Moss, "Fixed points of functors," *Journal of Logical and Algebraic Methods in Programming*, vol. 95, pp. 41–81, 2018, ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2017.11.003.

[161] M. A. Arbib, "From cybernetics to brain theory, and more, a memoir," *Cognitive Systems Research*, vol. 50, pp. 83–145, 2018, ISSN: 1389-0417. DOI: 10.1016/j.cogsys.2018.04.001.

[162] J.-W. Buurlage, *Categories and Haskell*, commit hash: 02f119a. 2018, A manuscript based on lecture notes for a seminar at Centrum Wiskunde & Informatica, Netherlands. [Online]. Available: https://github.com/jwbuurlage/category-theory-programmers.

[163] I. Čukić, *Functional Programming in C++*. Manning Publications, 2018, ISBN: 978-1-61729-381-8.

[164] P. Fultz II. "Boost.HOF." Boost C++ Libraries. (2018), [Online]. Available: http://boostorg.github.io/hof/.

[165] P. Pai and P. Abraham, *C++ Reactive Programming*. Packt Publishing Limited, 2018, ISBN: 978-1-78862-977-5.

[166] T. A. V. Teatro, *tfunc–FP utilities in C++17*, , 2018. [Online]. Available: https://github.com/timtro/tfunc.

[167] *Working draft–standard for programming–language C++*, N4762, ISO/IEC JTC1/SC22/WG21, 2018. [Online]. Available: https://wg21.link/standard.

[168] B. Fong and D. I. Spivak, *An Invitation to Applied Category Theory: Seven Sketches in Compositionality.* Cambridge University Press, 2019, ISBN: 978-1-108-71182-1.

[169] B. Milewski, *Category Theory for Programmers*, I. Tabachnik, Ed. Blurb, 2019, ISBN: 978-0-464-24387-8. [Online]. Available: https://github.com/hmemcpy/milewski-ctfp-pdf.

[170] J. J. M. M. Rutten, *The Method of Coalgebra, exercises in coinduction.* Amsterdam, The Netherlands: Centre for Mathematics and Computer Science, 2019, ISBN: 978-90-6196-568-8.

[171] *Information technology–programming languages–C++*, 6th ed., ISO/IEC 14882:2020, ISO/IEC JTC1/SC22/WG21, 2020.

[172] D. I. Spivak, "Poly: An abundant categorical setting for mode-dependent dynamics," 2020. DOI: 10.48550/ARXIV.2005.01894. arXiv: 2005.01894 [math.CT].

[173] R. P. Borase, D. K. Maghade, S. Y. Sondkar, and S. N. Pawar, "A review of PID control, tuning methods and applications," *International Journal of Dynamics and Control*, vol. 9, no. 2, pp. 818–827, Jul. 2021. DOI: 10.1007/s40435-020-00665-4.

[174] T. Clingman, B. Fong, and D. I. Spivak, *Regular calculi I: Graphical regular logic*, 2021. arXiv: 2109.14123 [math.CT].

[175] D. I. Spivak, "Learners' languages," 2021. arXiv: 2103.01189 [math.CT].

[176] G. Zardini, D. I. Spivak, A. Censi, and E. Frazzoli, "A compositional sheaf-theoretic framework for event-based systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 333, pp. 139–153, Feb. 2021. DOI: 10.4204/eptcs.333.10.

[177] E. Di Lavore, G. de Felice, and M. Román, "Monoidal streams for dataflow programming," in *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '22, Haifa, Israel: Association for Computing Machinery, 2022, ISBN: 978-1-4503-9351-5. DOI: 10.1145/3531130.3533365.

[178] D. J. Myers, *Categorical systems theory*, commit hash: 8e662c5. 2022–. [Online]. Available: https://github.com/DavidJaz/DynamicalSystemsBook, in preparation.

[179] ——, "Double categories of open dynamical systems (extended abstract)," D. I. Spivak and J. Vicary, Eds., 2022. DOI: 10.4204/EPTCS.333.11.

[180] N. Niu and D. I. Spivak, *Collectives: Compositional protocols for contributions and returns*, 2022. arXiv: 2112.11518 [math.CT].

[181] B. T. Shapiro and D. I. Spivak, *Duoidal structures for compositional dependence*, 2022. arXiv: 2210.01962 [math.CT].

[182] D. I. Spivak and N. Niu, *Polynomial Functors, A General Theory of Interaction.* Topos Institute, 2022, Draft of July 5, 2023.

[183] T. A. V. Teatro. “Pid-unfolding,” . (2022), [Online]. Available: https://github.com/timtro/tfunc.

[184] T. A. V. Teatro, J. M. Eklund, and R. Milman, “Toward a coalgebraic model of control programs,” in *2022 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2022, pp. 328–335. DOI: 10.1109/ccece49351.2022.9918479.

[185] K. Brown and D. I. Spivak, *Dynamic tracing: A graphical language for rewriting protocols*, 2023. arXiv: 2304.14950 [cs.LO].

[186] O. Lynch, B. T. Shapiro, and D. I. Spivak, *All concepts are* $\mathbb{C}\mathbf{at}^{\#}$, 2023. arXiv: 2305.02571 [math.CT].

[187] B. T. Shapiro and D. I. Spivak, “Dynamic operads, dynamic categories: From deep learning to prediction markets,” *Electronic Proceedings in Theoretical Computer Science*, vol. 380, pp. 183–202, Aug. 2023. DOI: 10.4204/eptcs.380.11.

[188] ——, *Structures on categories of polynomials*, 2023. arXiv: 2305.00167 [math.CT].

[189] T. S. C. Smithe, “Mathematical foundations for a compositional account of the bayesian brain,” Ph.D. dissertation, Oxford University, 2023. arXiv: 2212.12538 [q-bio.NC].

[190] D. I. Spivak, “A reference for categorical structures on **Poly**,” 2023. arXiv: 2202.00534 [math.CT].

[191] ——, *Functorial aggregation*, 2023. arXiv: 2111.10968 [math.CT].

[192] ——, “Polynomial functors and shannon entropy,” *Electronic Proceedings in Theoretical Computer Science*, vol. 380, pp. 331–343, Aug. 2023. DOI: 10.4204/eptcs.380.19.

[193] T. A. V. Teatro, *ct-control-program-cpp demo*, , 2023. [Online]. Available: https://github.com/timtro/ct-control-program-cpp.

[194] ——, *Thesis demo code*, , 2023. [Online]. Available: https://github.com/timtro/phd-thesis-demos.

[195] S. Blackheath, *SodiumFRP C++*, . [Online]. Available: https://github.com/SodiumFRP/sodium-cxx.

[196] P. Fultz II, *Fit: C++ function utility library*, . [Online]. Available: https://github.com/pfultz2/Fit.

[197] P. Nash *et al.*, *Catch2, A modern, C++-native, header-only, test framework for unit-tests, TDD and BDD*, . [Online]. Available: https://github.com/catchorg/Catch2.

[198] K. Shoop, *RxCpp*, , version 2. [Online]. Available: https://github.com/ReactiveX/RxCpp.

[199] T. A. V. Teatro, *Kalman-folding—categorical structure in the Kalman filter*, . [Online]. Available: https://github.com/timtro/kalman-folding.

[200] “The topos institute colloquium.” (), [Online]. Available: `https://topos.site/topos-colloquium/`.

[201] “The topos institute website.” (), [Online]. Available: `https://topos.institute/`.

INDEX