COMBINATORIAL SPECIES AND LABELLED STRUCTURES

Brent Abraham Yorgey

A DISSERTATION

in

Computer and Information Sciences

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2014

Supervisor of Dissertation

Stephanie Weirich
Associate Professor of CIS

Graduate Group Chairperson

Lyle Ungar
Professor of CIS

Dissertation Committee

Steve Zdancewic (Associate Professor of CIS; Committee Chair)

Jacques Carette (Associate Professor of Computer Science, McMaster University)

Benjamin Pierce (Professor of CIS)

Val Tannen (Professor of CIS)

COMBINATORIAL SPECIES AND LABELLED STRUCTURES

ὅς ἐστιν εἰκὼν τοῦ θεοῦ τοῦ ἀοράτου,
πρωτότοκος πάσης κτίσεως,
ὅτι ἐν αὐτῷ ἐκτίσθη τὰ πάντα ἐν τοῖς οὐρανοῖς καὶ ἐπὶ τῆς γῆς, τὰ ὁρατὰ καὶ τὰ ἀόρατα, εἴτε θρόνοι εἴτε κυριότητες εἴτε ἀρχαὶ εἴτε ἐξουσίαι· τὰ πάντα δι' αὐτοῦ καὶ εἰς αὐτὸν ἔκτισται·

καὶ αὐτός ἐστιν πρὸ πάντων

καὶ τὰ πάντα ἐν αὐτῷ συνέστηκεν,

καὶ αὐτός ἐστιν ἡ κεφαλὴ τοῦ σώματος τῆς ἐκκλησίας·

ὅς ἐστιν ἀρχή,
πρωτότοκος ἐκ τῶν νεκρῶν, ἵνα γένηται ἐν πᾶσιν αὐτὸς πρωτεύων,
ὅτι ἐν αὐτῷ εὐδόκησεν πᾶν τὸ πλήρωμα κατοικῆσαι καὶ δι' αὐτοῦ ἀποκαταλλάξαι τὰ πάντα εἰς αὐτόν, εἰρηνοποιήσας διὰ τοῦ αἵματος τοῦ σταυροῦ αὐτοῦ, εἴτε τὰ ἐπὶ τῆς γῆς εἴτε τὰ ἐν τοῖς οὐρανοῖς·

ΠΡΟΣ ΚΟΛΟΣΣΑΕΙΣ 1.15–20

Acknowledgments

I first thank Stephanie Weirich, who has been a wonderful advisor, despite the fact that we have fairly different interests and different approaches to research. She has always encouraged me to pursue my passions, even to the point of allowing me to take on a dissertation topic she knew very little about. Perhaps most importantly, she has done a masterful job getting me to actually graduate (no mean feat)—by turns encouraging and challenging me, each at the appropriate moment.

Jacques Carette has been an unofficial second advisor to me. Despite having plenty of "official" advisees also demanding his time and attention, he has generously taken the time to collaborate, give feedback and advice, and even twice to host me for a week of focused, face-to-face collaboration. This dissertation literally would not exist were it not for his academic and personal generosity, for which I will always be grateful.

My family has been, and continues to be, a constant source of joy and encouragement. My wife and bestest friend Joyia, more than anyone else, is the one who encouraged me through the darkest points and convinced me to keep going. She has also sacrificed much in order to give me the time and space necessary to finish. My son Noah, too, has sacrificed—in ways he doesn't even understand—while his daddy wrote a "very long story about computers and numbers". But I could always count on him to cheer me up with tickle fights.

The other members of the Penn programming languages group—especially (though by no means limited to) Chris Casinghino, Richard Eisenberg, Nate Foster, Michael Greenberg, Peter-Michael Osera, Benjamin Pierce, Vilhelm Sjöberg, Daniel Wagner, and Steve Zdancewic—deserve a great deal of thanks for all their support over the years, through moral support and encouragement, critical feedback on papers and talks, enlightening discussions, and simply friendship. PL Club has been a wonderfully collegial community in which to learn and work.

While developing the ideas in this dissertation I have benefited over the years, both directly and indirectly, from conversations with many people in the Haskell community and the wider FP and PL communities, particularly Faris Abou-Saleh, Reid Barton, Gershom Bazerman, Conal Elliott, Jeremy Gibbons, Andy Gill, Jason Gross, Ralf Hinze, Neel Krishnaswami, Dan Licata, Peter Lumsdaine, Simon Peyton Jones, Ross Street, Andrea Vezzosi, and Nick Wu, along with many others. I am also grateful to Heinrich Apfelmus, Toby Bartels, Shachaf Ben-Kiki, Gabor Greif, David Harrison, Jay McCarthy, Colin McQuillan, David Roberts, Jon Sterling, and Ryan

Yates, all of whom read early drafts of this dissertation and sent me typo reports as well as more substantial suggestions, greatly improving the final product. Thanks also to the anonymous MSFP and MFPS reviewers, whose feedback on submissions based on this material led to many substantial improvements to the technical content.

The `diagrams` community—particularly Daniel Bergey, Chris Chalmers, Allen Gardner, Niklas Haas, Claude Heiland-Allen, Chris Mears, Jeff Rosenbluth, Carter Schonwald, Michael Sloan, Luite Stegeman, and Ryan Yates—has been a great source of joy to me during the long process of completing my PhD. Not only have they provided encouragement, camaraderie, and welcome distraction, but this dissertation itself is richer for their contributions to `diagrams`—many of the diagrams throughout this document make nontrivial use of features contributed by other members of the community. It has also been a particular joy to see the project continue humming along even during my virtual absence while writing.

It is staggering to consider the wealth of relationship accumulated during six years at City Church Philadelphia. I particularly thank Tuck and Stacy Bartholomew, Darren Bell, Dave and Katie Brindley, Zac and Joanna Brooks, Sara Cayless, Mike and Sonja Chen, Tim and Ruth Creber, Chris and Bonnie Currie, Ben Doane and Melissa McCarten, Megan and Ryan Dougherty, John Dyck, Brooke Fugate, Kevin Funderburk, Will and Margaret Kendall, Dick Landis, Colin and Lauren Marlowe, Drew and Susie Matter, Nick McAvoy, Chris and Sarah Miciek, Jeremy Millington, Cat Ricketts, Ben Smith, Josh and Kory Stamper, Ben Sykora and Beth Dyson, Matt Thanabalan and Carrie Lutjens, Gene and Laura Twilley, and Jackson Warren, all of whom, at various times and in various ways, have provided encouragement and support as I made my way through graduate school. It is certain that I have forgotten others who should also be on this list, and in any case it is not even clear where the list should stop!

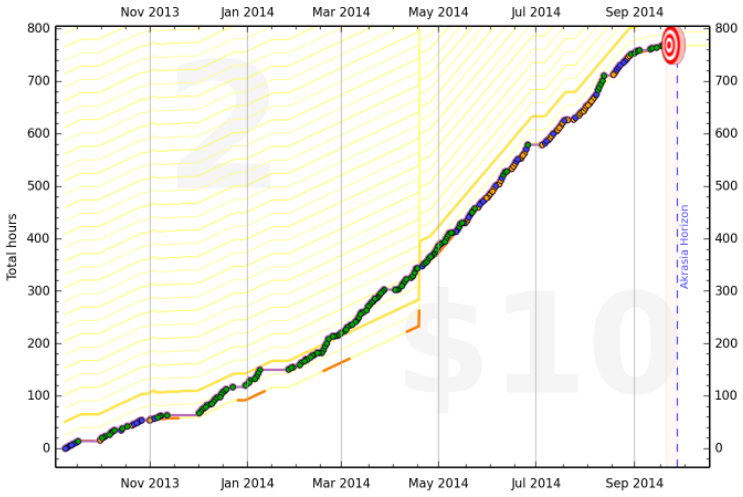
I thank the Mustard Seed Foundation for their affirmation in selecting me as a Harvey Fellow, and for their support, both financially, and in helping me think through the integration of my faith and work.

A heartfelt thank you to the Williams College computer science department for the genuine care and support I have received as a visiting faculty member, and especially to Bill Lenhart for his generous gift of time in taking on most of the grungy legwork for CS 134. Beginning a new job, teaching two classes, and simultaneously completing a dissertation would be impossible even to contemplate were it not undertaken in such a supportive environment.

In Philadelphia, the Green Line Cafe, Lovers & Madmen, the Penn Graduate Student Center, and Van Pelt library—and in Williamstown, Tunnel City Coffee and the Schow science library—have all provided wonderfully conducive environments for focused writing sessions.

Last but certainly not least, a big thank you is due to Beeminder (`http://beeminder.com`), and to its cofounders, Bethany Soule and Danny Reeves. The chance that I would have successfully finished this dissertation without Beeminder is vanish-

ingly small, for the simple reason that a dissertation cannot be put off until a week before it is due. $145 for the motivation to write a dissertation is quite a steal; I owe the whole Beeminder team a round of beers!



Finally, my work has been supported by the National Science Foundation under the following grants:

- NSF 1218002, CCF-SHF Small: *Beyond Algebraic Data Types: Combinatorial Species and Mathematically-Structured Programming*
- NSF 1116620, CCF-SHF Small: *Dependently-typed Haskell*
- NSF 0910500, CCF-SHF Large: *Trellys: Community-Based Design and Implementation of a Dependently Typed Programming Language*

and by the Defense Advanced Research Projects Agency under the following grant:

- DARPA Computer Science Study Panel Phase II. *Machine-Checked Metatheory for Security-Oriented Languages.*

Statement of contribution

Parts of Chapter 3, particularly the enumeration of categorical properties needed to support various species operations, were carried out in collaboration with Jacques Carette. The rest of this dissertation is my own original work.

ABSTRACT

COMBINATORIAL SPECIES AND LABELLED STRUCTURES

Brent Abraham Yorgey

Stephanie Weirich

The theory of *combinatorial species* was developed in the 1980s as part of the mathematical subfield of enumerative combinatorics, unifying and putting on a firmer theoretical basis a collection of techniques centered around *generating functions*. The theory of *algebraic data types* was developed, around the same time, in functional programming languages such as Hope and Miranda, and is still used today in languages such as Haskell, the ML family, and Scala. Despite their disparate origins, the two theories have striking similarities. In particular, both constitute algebraic frameworks in which to construct structures of interest. Though the similarity has not gone unnoticed, a link between combinatorial species and algebraic data types has never been systematically explored. This dissertation lays the theoretical groundwork for a precise—and, hopefully, useful—bridge bewteen the two theories. One of the key contributions is to port the theory of species from a classical, untyped set theory to a constructive type theory. This porting process is nontrivial, and involves fundamental issues related to equality and finiteness; the recently developed *homotopy type theory* is put to good use formalizing these issues in a satisfactory way. In conjunction with this port, species as general functor categories are considered, systematically analyzing the categorical properties necessary to define each standard species operation. Another key contribution is to clarify the role of species as *labelled shapes*, not containing any data, and to use the theory of *analytic functors* to model labelled data

structures, which have both labelled shapes and data associated to the labels. Finally, some novel species variants are considered, which may prove to be of use in explicitly modelling the memory layout used to store labelled data structures.

Contents

List of Tables

List of Figures

This document is typeset in LaTeX using Computer Modern.

It was edited using `emacs` and stored using `git` and `github.com`.

The illustrations were produced with `diagrams` version 1.2 (`http://projects.haskell.org/diagrams`).

Chapter 0

Introduction

The theory of *algebraic data types* has had a profound impact on the practice of programming, especially in functional languages. The basic idea is that types can be built up *algebraically* from a small set of primitive types and combinators: a unit type, base types, sums (*i.e.* tagged unions), products (*i.e.* tupling), and recursion. Most languages with support for algebraic data types also add bells and whistles for convenience (such as labeled products and sums, convenient syntax for defining types as a "sum of products", and pattern matching), but the basic idea remains unchanged.

For example, in Haskell [Marlow, 2010] we can define a type of binary trees with integer values stored in the leaves as follows:

$$\begin{aligned}\textbf{data}\ \mathsf{Tree} = {} & \mathsf{Leaf}\ \mathsf{Int} \\ \mid {} & \mathsf{Branch}\ \mathsf{Tree}\ \mathsf{Tree}\end{aligned}$$

Algebraically, we can think of this as defining the type which is the least solution to the equation $T = \mathsf{Int} + T \times T$. This description says that a Tree is either an Int (tagged with Leaf) or a pair of two recursive occurrences of Trees (tagged with Branch).

This algebraic view of data types has many benefits. From a theoretical point of view, recursive algebraic data types can be interpreted as *initial algebras* (or *final coalgebras*), which gives rise to an entire theory—both semantically elegant and practical—of programming with recursive data structures via *folds* and *unfolds* [Meijer et al., 1991, Gibbons, 2002]. A fold gives a principled way to compute a "summary value" from a data structure; dually, an unfold builds up a data structure from an initial "seed value".

Folds (and unfolds) satisfy theorems which aid in transforming, optimizing, and reasoning about programs defined in terms of them. As a simple example, a map (*i.e.* applying the same function to every element of a data structure) followed by a fold can always be rewritten as a single fold. These laws, and others, allow Haskell compilers to eliminate intermediate data structures through an optimization called deforestation [Wadler, 1988, Gill et al., 1993].

An algebraic view of data types also enables *datatype-generic programming*—writing functions that operate generically over values of *any* algebraic data type by examining its algebraic structure. For example, the following function (defined using Generic Haskell-like syntax [Hinze, 2000, Clarke et al., 2002]) finds the product of all the Int values contained in a value of *any* algebraic data type.

$$
\begin{array}{lll}
\mathit{genProd}\ \{|\ \mathsf{Int} & |\}\ i & = i \\
\mathit{genProd}\ \{|\ \mathsf{Sum}\ \mathit{t1}\ \mathit{t2} & |\}\ (\mathsf{Inl}\ x) & = \mathit{genProd}\ \{|\ \mathit{t1} |\}\ x \\
\mathit{genProd}\ \{|\ \mathsf{Sum}\ \mathit{t1}\ \mathit{t2} & |\}\ (\mathsf{Inr}\ x) & = \mathit{genProd}\ \{|\ \mathit{t2} |\}\ x \\
\mathit{genProd}\ \{|\ \mathsf{Prod}\ \mathit{t1}\ \mathit{t2} & |\}\ (x, y) & = \mathit{genProd}\ \{|\ \mathit{t1} |\}\ x * \mathit{genProd}\ \{|\ \mathit{t2} |\}\ y \\
\mathit{genProd}\ \{|\ \_ & |\}\ \_ & = 1
\end{array}
$$

Datatype-generic programming is a powerful technique for reducing boilerplate, made possible by the algebraic view of data types, and supported by Haskell libraries and extensions [Jansson and Jeuring, 1997, Lämmel and Jones, 2003, Cheney and Hinze, 2002, Weirich, 2006b,a].

The theory of *combinatorial species* has been similarly successful in the area of combinatorics. First introduced by Joyal [1981], it is a unified theory of *combinatorial structures* or *shapes*. Its immediate goal was to generalize the existing theory of *generating functions*, a central tool in enumerative combinatorics (the branch of mathematics concerned with counting abstract structures). More broadly, it introduced a framework— similar to algebraic data types—in which many combinatorial objects of interest could be constructed algebraically, and in which those algebraic descriptions can be used to reason about, manipulate, and derive properties of the combinatorial structures. The theory of species has been used to give elegant new proofs of classical results (for example, Cayley's theorem giving the number of labelled trees [Joyal, 1981]), and some new results as well (for example, a combinatorial interpretation and proof of Lagrange inversion [Bergeron et al., 1998, Chap. 3]).

Not only do the theory of algebraic data types and the theory of combinatorial species have a similar algebraic flavor in general, but the specific details are tantalizingly parallel. For example, the *species* of binary parenthesizations (*i.e.* binary trees with data stored in the leaves) can be defined by the recursive species equation

$$\mathsf{P} = \mathsf{X} + \mathsf{P} \cdot \mathsf{X} \cdot \mathsf{P}$$

which closely parallels the Haskell definition given above. The theory of functional programming languages has a long history of fruitful borrowing from pure mathematics, as, for example, in the case of category theory; so the fruit seems ripe for picking in the case of combinatorial species.

There has already been some initial progress in this direction. The connection between species and computation was first explored by Flajolet, Salvy, and Zimmermann, with their work on LUO [Flajolet et al., 1989, Flajolet and Salvy, 1995], allowing the use of species in automated algorithm analysis. However, they carried out their work in a dynamically typed setting.

The first to think about species specifically in the context of strongly typed functional programming were Carette and Uszkay [2008], who explored the potential of species as a framework to extend the usual notion of algebraic data types, and described some preliminary work adding species types to Haskell. More recently, Joachim Kock has done some theoretical work generalizing species, "container types", and other notions of "extended data type" [Kock, 2012]. (Most interestingly, Kock's work points to the central relevance of homotopy type theory [Univalent Foundations Program, 2013], which also emerges as a central player in this dissertation.)

However, there has still yet to be a comprehensive treatment of the precise connections between the theory of algebraic data types and the theory of combinatorial species. Bergeron et al. [1998] give a comprehensive treatment of the theory of species, but their book is written primarily from a mathematical point of view and is only tangentially concerned with issues of computation. It is also written in a style that makes it relatively inaccessible to researchers in the programming languages community—it assumes mathematical background that many PL researchers do not have.

The investigations in this dissertation, therefore, all arise from considering the central question, **what is the connection between species and algebraic data types?** A precise connection between the two would have exciting implications. It would allow taking much of the mathematical theory developed on the basis of species—for example, enumeration, exhaustive generation, and uniform random generation of structures via Boltzmann sampling [Duchon et al., 2002, 2004, Flajolet et al., 2007, Roussel and Soria, 2009]—and applying it directly to algebraic data types. It is also possible that exploring the theory of species in an explicitly computational setting will yield additional insights into the combinatorial setting.

There is also the promise of using species not just as a tool to understand and work with algebraic data types in better ways, but directly as a foundation upon which to build (a richer notion of) algebraic data types. This is particularly interesting due to the ability of the theory of species to talk about structures which do not correspond to algebraic data types in the usual sense—particularly structures which involve *symmetry* and *sharing*.

A data structure with *symmetry* is one whose elements can be permuted in certain ways without affecting its identity. For example, permuting the elements of a bag always results in the same bag. Likewise, the elements of an ordered cycle may be cyclically permuted without affecting the cycle. By contrast, a typical binary tree structure has no symmetry: any permutation of the elements may result in a different tree. In fact, every structure of an algebraic data type has no symmetry, since every element in an algebraic structure can be uniquely identified by a *path* from the root of the structure to the element, so permuting the elements always results in an observably different value.

A data structure with *sharing* is one in which different parts of the structure may refer to the same subpart. For example, consider the type of undirected, simple graphs, consisting of a set of vertices together with a set of edges connecting pairs of

vertices. In general, such graphs may involve sharing, since multiple edges may refer to the same vertex, and vice versa.

In a language with first-class pointers, creating data structures with sharing is relatively easy, although writing correct programs that manipulate them may be another story. The same holds true for many languages without first-class pointers as well. Creating data structures with sharing in the heap is not difficult in Haskell, but it may be difficult or even impossible to express the programs that manipulate them.

For example, in the following code,

$$
\begin{array}{l}
t = \textbf{let } \mathit{t3} = \mathsf{Leaf}\ 1 \\
\quad \mathit{t2} = \mathsf{Branch}\ \mathit{t3}\ \mathit{t3} \\
\quad \mathit{t1} = \mathsf{Branch}\ \mathit{t2}\ \mathit{t2} \\
\quad \textbf{in } \mathsf{Branch}\ \mathit{t1}\ \mathit{t1}
\end{array}
$$

only one "Leaf" and three "Branch" structures will be allocated in memory. The tree *t2* will be shared in the node *t1*, which will itself be shared in the node t. Furthermore, in a lazy language such as Haskell, recursive "knot-tying" allows even cyclic structures to be created. For example,

$$\mathit{nums} = 1 : 2 : 3 : \mathit{nums}$$

actually creates a cycle of three numbers in memory.

Semantically, however, t is a tree, not a DAG, and *nums* is an infinite list, not a cycle. It is impossible to observe the sharing (without resorting to compiler-specific tricks [Gill, 2009]) in either of these examples. Even worse, applying standard functions such as *fold* and *map* destroys any sharing that might have been present and risks non-termination.

When programmers wish to work with "non-regular" data types involving symmetry or sharing, they must instead work with suitable *encodings* of them as regular data types. For example, a bag may be represented as a list, or a graph as an adjacency matrix. However, this encoding puts extra burden on the programmer, both to ensure that invariants are maintained (*e.g.* that the adjacency matrix for an undirected graph is always symmetric) and that functions respect abstract structure (*e.g.* any function on bags should give the same result when given permutations of the same elements as inputs).

The promise of using the theory of species as a foundation for data types is to be able to declare data types with symmetry and sharing, with built-in compiler support ensuring that working with such data types is "correct by construction".

The grand vision of this research program, then, is to create and exploit a bridge between the theory of species and the theory and practice of programming languages. This dissertation represents just a first step in this larger program, laying the theoretical groundwork necessary for its continued pursuit.

To even get started building a bridge between species and data types requires more work than one might naïvely expect. The fundamental problem is that the theory of

species is traditionally couched in untyped, classical set theory. To talk about data types, however, we want to work in *typed* and *constructive* foundations. Attempting to port species to a typed, constructive setting reveals many implicit assumptions that must be made explicit, as well as implicit uses of reasoning principles, such as the axiom of choice, which are incompatible with constructive foundations. The bulk of Chapter 2 defines the foundational groundwork which makes it possible to talk about species in a typed, constructive setting. In particular, the biggest issues are the difference between *equality* and *isomorphism*, and the constructive encoding of *finiteness* (which is itself related to issues of equality and isomorphism). The recently developed *homotopy type theory* [Univalent Foundations Program, 2013] turns out to be exactly what is wanted to encode everything in a parsimonious way. The development of cardinal-finite sets in HoTT (along with a related concept I term "manifestly finite sets") is novel, as is the development of HoTT analogues of the set-theoretic groupoids $\mathbf{B}$ and $\mathbf{P}$.

Chapter 3 presents the theory of species itself. Much of the chapter is not novel in a technical sense. One of the main contributions of the chapter, instead, is simply to organize and present some relevant aspects of the theory for a functional programming audience. The existing species literature is almost entirely written for either hard-core combinatorialists or hard-core category theorists, and is not very accessible to the typical FP practitioner. Any attempt to make species relevant to computer scientists must therefore first address this accessibility gap.

Chapter 3 does also make a few novel technical contributions—for example, a characterization of equipotence in terms of manifestly finite sets, and a careful discussion of finite versus infinite families of structures and the relation to species composition. Most importantly, since Chapter 3 is already attempting to present at least two different variants of species—the traditional definition based on set theory, and a novel variant based on homotopy type theory—it "bites the bullet" and considers *arbitrary* functor categories, elucidating the categorical properties required to support each species operation. Although many individual species generalizations have been considered in the past, this systematic consideration of the minimal features needed to support each operation is novel. This allows operations to be defined for whole classes of species-like things at once, and in some cases even allows for species-like things to be constructed in a modular way, by applying constructions known to preserve the required properties.

Chapter 5 goes on to explore particular species variants, evaluated through the framework of Chapter 3. Some variants have already been considered in the literature; others, such as the notion of copartial species considered in §5.2, are novel.

Finally, Chapter 6 considers extending species to *labelled data structures*, which intuitively consist of a labelled shape, or species structure, paired with a mapping from labels to data elements. The notion of *analytic functors*, as introduced by Joyal [1986], turns out to be exactly the right framework in which to consider labelled data structures. Analytic functors can be most generally defined in terms of *Kan*

extensions, and so the chapter opens with a presentation of Kan extensions, once again aimed at functional programmers. Analytic functors are considered in the context of copartial species, which, can serve as a foundation for further work codifying data structures backed by memory storage (in applications where the memory layout really matters, *e.g.* linear algebra libraries), and also for partial species, which may help model situations where data need not be associated to every label.

Chapter 1

Preliminaries

The main content of this dissertation builds upon a great deal of mathematical formalism, particularly from set theory, category theory, and type theory. This chapter provides a brief overview of the necessary technical background, giving definitions, important intuitions, and references for further reading. Readers who merely need to fill in a few gaps may find such brief treatments sufficient; it is hoped that readers with less background will find it a useful framework and source of intuition for furthering their own learning.

1.1 Metavariable conventions and notation

A great many variables and named entities appear in this dissertation. To aid the reader's comprehension, the following metavariable conventions are (mostly) adhered to:

- Metavariables f, g, h range over functions.
- Greek metavariables (especially α, β, σ, τ, ϕ, ψ) often range over bijections.
- Blackboard bold metavariables (*e.g.* $\mathbb{C}$, $\mathbb{D}$, $\mathbb{E}$) range over categories, as do fraktur variables such as $\mathfrak{L}$ and $\mathfrak{S}$.
- Names of specific categories use boldface (*e.g.* **Set**, **Cat**, **Spe**, **B**, **P**).
- Names of types or categories defined within homotopy type theory often use a calligraphic font (*e.g.* $\mathcal{U}$, $\mathcal{B}$, $\mathcal{P}$, $\mathcal{S}$).
- Metavariables A, B, C, range over arbitrary sets or types.
- Metavariables K, L range over *finite* sets or types.
- Metavariables F, G, H range over functors (and in particular over species).

- Names of specific species use a sans-serif font (*e.g.* X, E, L, C, B, R).

This dissertation also features a menagerie of notations to indicate "sameness". To the outsider this must seem quite bewildering: how complicated can "sameness" be? Why would one ever need anything other than plain old equality (=)? On the other hand, to computer scientists and philosophers this should come as no surprise; equality turns out to be an incredibly subtle concept. Each of these symbols and their corresponding concepts will later be discussed in more depth; however, as an aid to the reader, we give a brief enumeration of them here, which can referred back to in case of confusion or forgetfulness.

- Equality (=) is the only notation which is overloaded. In the context of set theory, two sets A and B are *equal*, denoted $A = B$, when they have the same elements. In the context of homotopy type theory (§1.3), $=$ denotes *propositional* equality; $A = B$ denotes the type of *paths* between the types A and B.

- In the context of set theory, the symbol $:=$ is used to introduce a *definitional* equality; that is, $x := y$ is a definition of x rather than a proposition asserting the equality of two things.

- $A \xrightarrow{\sim} B$ denotes the set (or type) of *bijections* between sets (or types) A and B. That is, if $f : A \xrightarrow{\sim} B$ then f is a function from A to B which possesses both a left and right inverse (denoted f^{-1}). Note that in set theory, sets which are in bijection are typically not equal.

- In homotopy type theory, $\equiv$ denotes *judgmental* equality, not to be confused with propositional equality (=). A fuller discussion of judgmental versus propositional equality can be found in §1.3.2.

- The symbol $:\equiv$ also denotes a definitional equality, but in homotopy type theory rather than set theory. The symbol emphasizes the fact that a definition introduces a judgmental rather than a propositional equality.

- Again in homotopy type theory, $A \simeq B$ denotes the *equivalence* of two types A and B. Intuitively, equivalence can be thought of as a "very well-behaved" bijection, *i.e.* a bijection with some extra coherence conditions.

- $A \leftrightarrow B$ denotes *logical equivalence* of A and B, that is, that each logically implies the other; more familiarly, it can also be read as "if and only if". Via the logical interpretation of types as propositions, this is also to say that there exist functions $A \to B$ and $B \to A$. Logical equivalence is thus a weaker notion than bijection or equivalence, since there is no requirement that the functions be inverse.

$=$	(propositional) equality
$:=$	definitional equality (set theory)
$\xrightarrow{\sim}$	bijection
$\equiv$	judgmental equality
$:\equiv$	definition equality (HoTT)
$\simeq$	equivalence
$\leftrightarrow$	logical equivalence
$\cong$	isomorphism
$\overset{\#}{=}$	species equipotence
$\approx$	relabelling equivalence
$\sim$	generic equivalence relation

Table 1.1: "Sameness" relations

- An *isomorphism* is an invertible arrow in a category (§1.4), and is denoted by $A \cong B$. The precise meaning of $\cong$ thus depends on the category under consideration. For example, in **Set**, the category of sets, isomorphisms are precisely bijections; in the category of pointed sets, isomorphisms are those bijections which preserve the distinguished element, and so on. Generally speaking, isomorphisms can be thought of as "structure-preserving correspondences".

- $F \overset{\#}{=} G$ denotes the *equipotence* of two species, discussed in §3.3.

- $f_1 \approx f_2$ denotes equivalence up to relabelling of species shapes, discussed in §3.3.2.

- Finally, $x \sim y$ is often used in general to denote an equivalence relation (whichever one happens to be under consideration at the moment).

These notations are summarized in Table 1.1.

1.2 Set theory

A grasp of basic set theory (the element-of ($\in$) and subset ($\subseteq$) relations, intersections ($\cap$), unions ($\cup$), and so on) is assumed. However, no background is assumed in *axiomatic* set theory, or in particular its role as a foundation for mathematics. Issues relating to axiomatic set theory are spelled out in detail as necessary (for example, the axiom of choice, in §2.1).

The set of *natural numbers* is denoted $\mathbb{N} = \{0, 1, 2, \dots\}$. The *size* or *cardinality* of a finite set X, a natural number, is denoted $\#X$ (rather than the more traditional $|X|$, since that notation is used for another purpose; see §1.3). Given a natural number $n \in \mathbb{N}$, the canonical size-n prefix of the natural numbers is denoted $[n] = \{0, \dots, n-1\}$.

Given a function $f : A \to B$, an element $b \in B$, and subsets $X \subseteq A$ and $Y \subseteq B$,

- $f(X) = \{f(a) \mid a \in X\}$ denotes the image of X under f;
- $f^{-1}(b) = \{a \in A \mid f(a) = b\}$ denotes the preimage or *fiber* of b;
- $f^{-1}(Y) = \bigcup_{b \in Y} f^{-1}(b) = \{a \in A \mid f(a) \in Y\}$ likewise denotes the preimage of an entire set.

1.3 Homotopy type theory

Homotopy Type Theory (HoTT) is a relatively new variant of Martin-Löf type theory [Martin-Löf, 1975, Martin-Löf and Sambin, 1984] arising out of Vladimir Voevodsky's Univalent Foundations program [Voevodsky]. There is certainly not space to give a full description here; in any case, given the existence of the excellent HoTT Book [Univalent Foundations Program, 2013], such a description would be superfluous. Instead, it will suffice to give a brief description of the relevant parts of the theory, and explain the particular benefits of carrying out this work in the context of HoTT. Some particular results from the HoTT book are also reproduced as necessary, especially in §2.2. It is thus hoped that readers with no prior knowledge of HoTT will still be able to follow everything in this dissertation, at least at a high level, though a thorough understanding will probably require reference to the HoTT book.

Homotopy type theory, I will argue, is the *right* framework in which to carry out the work in this dissertation. Intuitively, this is because the theory of species is based centrally around groupoids and isomorphism—and these are topics central to homotopy type theory as well. In a sense, HoTT is what results when one begins with Martin-Löf type theory (MLTT) and then takes the principle of equivalence (§1.3.4) very seriously, generalizing equality to isomorphism in a coherent way.

We begin our brief tour of HoTT with its syntax.

1.3.1 Terms and types

Some familiarity with dependent type theory on the part of the reader is assumed; we simply note quickly the standard features of HoTT, including:

- an empty type $\bot$, with no inhabitants;
- a unit type $\top$, with inhabitant $\star$;
- sum types $A + B$, with constructors $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$, as well as a case construct for doing case analysis;
- dependent pairs $(x : A) \times B(x)$, with constructor $\langle -, - \rangle$, and projection functions $\pi_1 : (x : A) \times B(x) \to A$ and $\pi_2 : (p : (x : A) \times B(x)) \to B(\pi_1\ p)$;
- dependent functions $(x : A) \to B(x)$; and

- a hierarchy of type universes $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2 \ldots$.

Following standard practice, universe level subscripts will usually be omitted, with $\mathcal{U}$ being understood to represent whatever universe level is appropriate in the context.

HoTT also allows inductive definitions. For example, $\mathbb{N} : \mathcal{U}_0$ denotes the inductively-defined type of natural numbers, with constructors $\mathsf{O} : \mathbb{N}$ and $\mathsf{S} : \mathbb{N} \to \mathbb{N}$; we will use Arabic notation like 3 as a shorthand for $\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O}))$. We also have $\mathsf{Fin}\ : \mathbb{N} \to \mathcal{U}_0$, which denotes the usual indexed type of finite sets, with constructors $\mathsf{FO} : \mathsf{Fin}\ (\mathsf{S}\ n)$ and $\mathsf{FS} : \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{S}\ n)$. For example, one can check that $\mathsf{Fin}\ 3$ has the three inhabitants FO, $\mathsf{FS}\ \mathsf{FO}$, and $\mathsf{FS}\ (\mathsf{FS}\ \mathsf{FO})$, and that in general $\mathsf{Fin}\ n$ is the type-theoretic counterpart to $[n] = \{0, 1, \ldots, n-1\}$.

Although Agda notation [Norell, 2007] is mostly used in this dissertation for dependent pairs and functions, the traditional notations $\sum_{x:A} B(x)$ and $\prod_{x:A} B(x)$ (instead of $(x : A) \times B(x)$ and $(x : A) \to B(x)$, respectively) are sometimes used for emphasis. As usual, the abbreviations $A \times B$ and $A \to B$ denote non-dependent (*i.e.* when x does not appear free in B) pair and function types, respectively.

1.3.2 Equality

HoTT distinguishes between two different types of equality:

- *Judgmental* equality, denoted $x \equiv y$, is defined via a collection of judgments stating when things are equal to one another, and encompasses things like basic rules of computation. For example, the application of a lambda term to an argument is judgmentally equal to its β-reduction. Judgmental equality is reflexive, symmetric, and transitive as one would expect. Note, however, that judgmental equality is not reflected as a proposition in the logical interpretation of types, so it is not possible to reason about or to prove judgmental equalities internally to HoTT.

- *Propositional* equality. Given $x, y : A$, we write $x =_A y$ for the proposition that x and y are equal (at the type A). The A subscript may also be omitted, $x = y$, when it is clear from the context. Unlike judgmental equality, where $x \equiv y$ is a *judgment*, the propositional equality $x = y$ is a *type* (or a *proposition*) whose inhabitants are evidence or *proofs* of the equality of x and y. Thus propositional equalities can be constructed and reasoned about *within* HoTT. Inhabitants of $x = y$ are often called *paths* from x to y; the intuition, taken from homotopy theory, is to think of paths between points in a topological space. The most important aspect of this intuition is that a path from a point x to a point y does not witness the fact that x and y are literally the *same* point, but rather specifies a *process* for getting from one to the other. The analogue of this intuition in type theory is the fact that a path of type $x = y$ can have *nontrivial computational content* specifying how to convert between x and y. There is a special value $\mathsf{refl}_x : x = x$ which witnesses the reflexivity of propositional

equality, and corresponds to a "trivial path with no computational content"; but, as the discussion above indicates, there can be other inhabitants of path types besides refl.

Note that it is possible (and often useful!) to have nontrivial higher-order paths, *i.e.* paths between paths, paths between paths between paths, and so on.

1.3.3 Path induction

To make use of a path $p : x = y$, one may use the induction principle for paths, or *path induction*. Path induction applies when trying to prove a statement of the form

$$\forall x, y.\ (p : x = y) \to P(x, y, p). \tag{1.3.1}$$

For the precise details of path induction, see the HoTT book [Univalent Foundations Program, 2013]. For this work, however, a simple intuition suffices: to prove 1.3.1 it suffices to assume that p is refl and that x and y are literally the same, *i.e.* it suffices to prove $\forall x.\ P(\mathsf{refl}, x, x)$.

It is important to note that this does *not* imply all paths are equal to refl! It simply expresses that all paths must suitably "act like" refl, inasmuch as proving a statement holds for refl is enough to guarantee that it will hold for all paths, no matter how they are derived or what their computational content.

Path induction has some immediate consequences. First, it guarantees that functions are always functorial with respect to propositional equality. That is, if $e : x = y$ is a path between x and y, and f is a function of an appropriate type, then it is possible to construct a proof that $f(x) = f(y)$ (or a suitable analogue in the case that f has a dependent type). Indeed, this is not hard to prove via path induction: it suffices to show that one can construct a proof of $f(x) = f(x)$ in the case that e is refl, which is easily done using refl again. Given $e : x = y$ we also have $P(x) \to P(y)$ for any type family P, called the *transport* of $P(x)$ along e and denoted $\mathsf{transport}^P(e)$, or simply e_* when P is clear from context. For example, if $e : A = B$ then $\mathsf{transport}^{X \mapsto X \times (X \to C)}(e) : A \times (A \to C) \to B \times (B \to C)$. Transport also follows easily from path induction: it suffices to note that $\mathsf{id} : P(x) \to P(x)$ in the case when e is refl.

1.3.4 Equivalence and univalence

Another notion of "sameness" definable in HoTT is *equivalence*. An equivalence between A and B, written $A \simeq B$, is a "coherent bijection", that is, a pair of inverse functions $f : A \to B$ and $g : B \to A$, along with an extra condition ensuring coherence of higher path structure. The precise details are unimportant for the purposes of this dissertation, and can be found in the HoTT book [2013, Chapter 4]. The important point is that equivalence and bijection are logically equivalent—that is, each implies the other. In particular, to prove an equivalence it suffices to exhibit a bijection.

The identity equivalence is denoted by id, and the composition of $h : B \simeq C$ and $k : A \simeq B$ by $h \circ k : A \simeq C$. As a notational shortcut, equivalences of type $A \simeq B$ can be used as functions $A \to B$ where it does not cause confusion.

HoTT's main novel feature is the *univalence axiom*, which states that equivalence is equivalent to propositional equality, that is, $(A \simeq B) \simeq (A = B)$. One direction, $(A = B) \to (A \simeq B)$, follows easily by path induction. The interesting direction, which must be taken as an axiom, is $\mathsf{ua} : (A \simeq B) \to (A = B)$. This formally encodes the *principle of equivalence* [nLab, 2014e], namely, that sensible properties of mathematical objects must be invariant under equivalence. Univalence, in conjunction with transport, implies that equivalent values are completely interchangeable.

Propositional equality thus takes on a meaning richer than the usual conception of equality. In particular, $A = B$ does not mean that A and B are *identical*, but that they can be used interchangeably—and moreover, interchanging them may require some work, computationally speaking. Thus an equality $e : A = B$ can have non-trivial computational content, particularly if it is the result of applying ua to some equivalence.

As of yet, univalence has no direct computational interpretation[1], so using it to give a computational interpretation of species may seem suspect. However, ua satisfies the β law $\mathsf{transport}^{X \mapsto X}(\mathsf{ua}(f)) = f$, so univalence introduces no computational problems as long as applications of ua are only ultimately used via $\mathsf{transport}$. In particular, sticking to this restricted usage of ua still allows a convenient shorthand: packaging up an equivalence into a path and then transporting along that path results in "automatically" inserting the equivalence and its inverse in all the necessary places throughout the term. For example, let $P(X) :\equiv X \times (X \to C)$ as in the example from the end of §1.3.3, and suppose $e : A \simeq B$, so $\mathsf{ua}\ e : A = B$. Then $\mathsf{transport}^P(\mathsf{ua}(e)) : P(A) \to P(B)$, and in particular $\mathsf{transport}^P(\mathsf{ua}(e))\langle a, g\rangle = \langle e(a), g \circ e^{-1}\rangle$, which can be derived mechanically by induction on the shape of P.

1.3.5 Propositions, sets, and n-types

As noted previously, it is possible to have arbitrary higher-order path structure: paths between paths, paths between paths between paths, and so on. This offers great flexibility but also introduces many complications. It is therefore useful to have a vocabulary for explicitly talking about types with limited higher-order structure.

Definition 1.3.1. A *mere proposition*, or (-1)-*type*, is a type for which any two inhabitants are propositionally equal.

The word "mere" is often used for emphasis ("*mere* proposition") but is also sometimes dropped ("proposition"). Intuitively, the only interesting thing that can be said about a mere proposition is whether it is inhabited or not. Although it may

[1] Though as of this writing there seems to be some good progress on this front via the theory of *cubical sets* [Bezem et al., 2014].

have many *syntactically* different inhabitants, they are all equal and hence internally indistinguishable. Such types are called "propositions" since they model the way one usually thinks of propositions in, say, first-order logic. There is no value in distinguishing the different possible proofs of a proposition; what counts is only whether or not the proposition is provable at all.

Definition 1.3.2. A type A is a *set*, or 0-*type*, if there is (up to propositional equality) at most one path between any two elements; that is, more formally, for any $x, y : A$ and $p, q : x = y$, there is a path $p = q$. Put another way, for any $x, y : A$, the type $x = y$ is a proposition.

Standard inductive types such as $\mathbb{N}$, $\mathsf{Fin}\ n$, and so on, are sets, although proving this takes a bit of work. Generally, one shows via induction that paths between elements of the type are equivalent to an indexed type given by $\bot$ when the elements are different and $\top$ when they are the same; $\bot$ and $\top$ are mere propositions and hence so is the type of paths. See the HoTT book for proofs in the particular case of $\mathbb{N}$, which can be adapted to other inductive types as well [Univalent Foundations Program, 2013, §2.13, Example 3.1.4, §7.2].

As noted above, propositions and sets are also called, respectively, (-1)-types and 0-types. As these names suggest, there is an infinite hierarchy of n-types (beginning at $n = -2$ for historical reasons) which have no interesting higher-order path structure above level n. As an example of a 1-type, consider the type of all sets,

$$\mathsf{Set} :\equiv (A : \mathcal{U}) \times \mathsf{isSet}(A),$$

where $\mathsf{isSet}(A) :\equiv (x, y : A) \to (p, q : x = y) \to (p = q)$ is the proposition that A is a set. Given two elements $A, B : \mathsf{Set}$ it is not the case that all paths $A = B$ are equal; such paths correspond to bijections between A and B, and there may be many such bijections.

Note that $\mathsf{isSet}(A)$ itself is always a mere proposition for any type A (see Lemma 3.3.5 in the HoTT book).

1.3.6 Higher inductive types

Another novel feature of HoTT (albeit one that is not yet fully understood) is the presence of *higher inductive types* (HITs). Standard inductive data types are specified by a collection of *data constructors* which freely generate all values of the type. For example, the values of $\mathbb{N}$ are precisely those constructed by any (finite) combination of the constructors O and S. HITs add the possibility of constructors which build not *values*, but *paths* between values (or paths between paths, or...). They also come with an induction principle requiring uses of the values to respect all the equalities built by the higher constructors.

This gives a natural way to build *quotient types*. For example, consider the HIT $T : \mathcal{U}$ with data constructors $\mathsf{TO} : T$ and $\mathsf{TS} : T \to T$, as well as a higher path

constructor $\mathsf{P2} : (t : T) \to t = \mathsf{TS}\ (\mathsf{TS}\ t)$. This corresponds to quotienting $\mathbb{N}$ by the reflexive, transitive closure of the relation $n = n + 2$. In this case, we can see (and could even prove formally) that T is equivalent to the type $\mathbf{2}$ with two inhabitants. However, if we really have in mind the quotient $\mathbb{N}/(n = n + 2)$, instead of the type $\mathbf{2}$, it may be more convenient to work with it directly using the HIT T. For example, in order to define functions $T \to A$, one specifies a function $f : \mathbb{N} \to A$ and then proves separately that f is compatible with the equality $n = n + 2$. In any case, there are also many HITs which are not equivalent to some standard inductive type, so the presence of HITs really does represent a large jump in expressive power. For a good example of a nontrivial "real-world" application of HITs, see Angiuli et al. [2014].

1.3.7 Truncation

The last important concept from HoTT to touch upon is *propositional truncation*, which is also an example of a nontrivial higher inductive type. If A is a type, then $\|A\|$ is also a type, with a data constructor $|-| : A \to \|A\|$ that allows injecting values of A into $\|A\|$. However, in addition to being able to construct *values* of $\|A\|$, there is also a way to construct *paths* between them: in particular, for any two values $x, y : \|A\|$, there is a path $x =_{\|A\|} y$. Thus, $\|A\|$ is a copy of A but with all values considered equal. This is called the *propositional truncation* of A since it evidently turns A into a proposition, which can intuitively be thought of as the proposition "A is inhabited".

If we have an inhabitant of $\|A\|$, we know some $a : A$ must have been used to construct it. However, the recursion principle for $\|A\|$ places some restrictions on how we are allowed to use the underlying $a : A$. In particular, to construct a function $\|A\| \to P$, one must give a function $f : A \to P$, along with a proof that f respects the equalities introduced by the higher constructor of $\|A\|$. Hence *all* the outputs of f must be equal—that is, P must be a mere proposition. That is, a function $A \to P$ can be used to construct a function $\|A\| \to P$ if and only if P is a mere proposition. Intuitively, this means that one is allowed to look at the value of type A hidden inside a value of $\|A\|$, as long as one "promises not to reveal the secret". Keeping this promise means producing an inhabitant of a *proposition* P, because it cannot "leak" any information about the precise inhabitant $a : A$. Up to propositional equality, there is at most one inhabitant of P, and hence no opportunity to convey information.

What about defining a function $\|A\| \to B$, when B is *not* a mere proposition? In this case, the recursion principle for propositional truncation cannot be applied directly. However, there is a useful trick which makes it possible: if one can *uniquely characterize* a particular value of B—that is, create a predicate $Q : B \to \mathcal{U}$ such that $(b : B) \times Q(b)$ is a mere proposition—one can then define a function $\|A\| \to (b : B) \times Q(b)$ from a function $A \to (b : B) \times Q(b)$, and finally project out the B to obtain a function $\|A\| \to B$. Since the function guarantees to always construct the same, uniquely characterized value of B for any input type A, it cannot reveal any information about the particular value of A it is given. This trick is detailed

in the HoTT book [2013, §3.9]; Exercise 3.19 affords some good intuition for this phenomenon.

As in the HoTT book (see Chapter 3), the adjective "mere" will be used more generally to refer to truncated things. In particular, an important example is the distinction between the type

$$(a : A) \times B(a),$$

pronounced "there *constructively* exists an inhabitant of A such that B", and its truncation

$$\|(a : A) \times B(a)\|,$$

pronounced "there *merely* exists an inhabitant of A such that B". The latter more closely corresponds to the notion of existence in classical logic: classically, given a proof of an existence statement, it may not be possible to extract an actual witness. Given an inhabitant of $\|(a : A) \times B(a)\|$, we know only that some $(a : A)$ satisfying B exists, without getting to know its identity.

1.3.8 Why HoTT?

In the context of this dissertation, homotopy type theory is much more than just a convenient choice of concrete type theory to work in. It is, in fact, quite central to this work. It is therefore appropriate to conclude with a summary of specific ways that this work benefits from its use. Many of these points are explored in more detail later in the dissertation.

- HoTT gives a convenient framework for making formal the idea of "transport": using an isomorphism $\sigma : L_1 \xrightarrow{\sim} L_2$ to functorially convert objects built from L_1 to ones built from L_2. This is a fundamental operation in HoTT, and is also central to the definition of species (§3.2). In fact, when constructing species with HoTT as a foundation, transport simply comes "for free"—in contrast to using set theory as a foundation, where transport must be tediously defined (and proved correct) for each new species. In other words, within HoTT it is simply impossible to write down an invalid species; any function giving the action of a species on objects extends automatically to a functor. In a material set theory, on the other hand, it is quite possible to define non-functorial functions on objects.

- The *univalence axiom* (§1.3.4) and *higher inductive types* (§1.3.6) make for a rich notion of propositional equality, over which the "user" has a relatively high degree of control. For example, using higher inductive types, it is easy to define various quotient types which would be tedious to define and work with in Martin-Löf type theory. One particular manifestation of this general idea is *coends* (§1.4.3) which can be directly defined as a higher inductive type (§2.2.2).

- Homotopy type theory allows doing category theory without using the axiom of

choice (§2.1, §2.2), which is essential in a constructive or computational setting. It also makes many constructions simpler; for example, a coend over a functor with a groupoid as its domain is just a plain Σ-type, with no need for higher inductive types at all.

- *Propositional truncation* (§1.3.7) is an important tool for properly modelling concepts from classical mathematics in a constructive setting. In particular we use it to model the concept of *finiteness* (§2.4).

Although not the main goal, I hope that this work can serve as a good example of the "practical" application of HoTT and its benefits for programming. Much of the work on HoTT has so far been aimed at mathematicians rather than computer scientists—appropriately so, since mathematicians tend to be more skeptical of type theory in general and constructive foundations in particular. However, new foundations for constructive mathematics must almost by necessity have profound implications for the foundations of programming as well [Martin-Löf, 1982].

1.4 Category theory

This dissertation makes extensive use of category theory, which is the natural language in which to explore species and related concepts. A full understanding of the contents of this dissertation requires an intermediate-level background in category theory, but a grasp of the basics should suffice to understand the overall ideas. A quick introduction to necessary concepts beyond the bare fundamentals is presented here, with useful intuition and references for further reading. It is hoped that this section can serve as a useful guide for "bootstrapping" one's knowledge of category theory, for those readers who are so inclined.

This section presents traditional category theory as founded on set theory, to make it easy for readers to correlate it with existing literature. However, as explained in §2.2 and as evidenced throughout this work, HoTT makes a much better foundation for category theory than set theory does! §2.2 highlights the most significant differences and advantages of HoTT-based category theory; most of the other definitions and inutition carry over essentially unchanged.

1.4.1 Category theory fundamentals

At a bare minimum, an understanding of the foundational trinity of category theory (categories, functors, and natural transformations) is assumed, along with some standard universal constructions such as terminal and initial objects, products, and coproducts. For an introduction to these concepts (and a much more leisurely introduction to those sketched below), the reader should consult one of the many excellent references on the subject [Lawvere and Schanuel, 2009, Awodey, 2006, Pierce, 1991, Barr and Wells, 1990, Mac Lane, 1998].

The notations $f\,;g = g \circ f$ are both used to denote composition of morphisms.

Standard categories We begin by listing some standard categories which will appear throughout this work.

- $\mathbf{1} = \bullet$, the trivial category with a single object and only the required identity morphism.
- $\mathbf{2} = \bullet \to \bullet$, the category with two objects and one nontrivial morphism between them (as well as the required identity morphisms).
- $|\mathbf{2}| = \bullet \quad \bullet$, the discrete category with two objects and only identity morphisms. A *discrete* category is a category with only identity morphisms; $|\mathbb{C}|$ denotes the discrete category with the objects of $\mathbb{C}$. Also, note that any set can be treated as a discrete category.
- **Set**, the category with sets as objects and (total) functions as morphisms.
- **FinSet**, like **Set** but with only finite sets as objects.
- **Cat**, the category of all small categories (a category is *small* if its objects and morphisms both form *sets*, as opposed to proper classes; considering the category of *all* categories gets us in trouble with Russell). Note that **1** is the terminal object in **Cat**.
- **Grp**, the category of groups and group homomorphisms.
- $\mathbf{Vec}_K$, the category of vector spaces over the field K, together with linear maps.
- **Hask**, the category whose objects are Haskell types and morphisms are (total) functions definable in Haskell.[2]

Bifunctors The concept of *bifunctors* can be formalized as a two-argument analogue of functors; bifunctors thus map from *two* categories to a single category. One can define a bifunctor $B : \mathbb{C}, \mathbb{D} \to \mathbb{E}$ as a function sending pairs of objects to a single object, pairs of morphisms to a single morphism, and so on, but this turns out to be equivalent to a regular (one-argument) functor $B : \mathbb{C} \times \mathbb{D} \to \mathbb{E}$, where $\mathbb{C} \times \mathbb{D}$ denotes the *product category* of $\mathbb{C}$ with $\mathbb{D}$. Product categories are given by the usual universal product construction in **Cat**; objects in $\mathbb{C} \times \mathbb{D}$ are pairs of objects from $\mathbb{C}$ and $\mathbb{D}$, and likewise morphisms in $\mathbb{C} \times \mathbb{D}$ are pairs of morphisms from $\mathbb{C}$ and $\mathbb{D}$. One place that bifunctors come up often is in the context of monoidal categories; see §1.4.2.

[2]Technically, this is a polite fiction, and requires pretending that $\perp$ does not exist. Fortunately, laziness does not play an important role in this work.

Natural transformations To denote a natural transformation η between functors $F, G : \mathbb{C} \to \mathbb{D}$, we use the notation $\eta : F \xrightarrow{\bullet} G$, or sometimes just $\eta : F \longrightarrow G$ when it is clear that F and G are functors. The notation $\eta : \forall A.\ FA \to GA$ will also be used, which meshes well with the intuition of Haskell programmers: naturality corresponds to *parametricity*, a property enjoyed by polymorphic types in Haskell [Reynolds, 1983, Wadler, 1989]. This notation is also more convenient when the functors F and G do not already have names but can be readily specified by expressions, especially when those expressions involve A more than once or in awkward positions[3]—for example, $\forall A.\ A \otimes A \to \mathbb{C}(B, HA)$. This notation can be rigorously justified using *ends*; see §1.4.3.

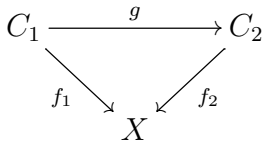
Hom sets In a similar vein, the set of morphisms between objects A and B in a category $\mathbb{C}$ is usually denoted $\mathbb{C}(A, B)$ or $\mathrm{Hom}_{\mathbb{C}}(A, B)$, but I will also occasionally use the notation $A \Rightarrow B$, or $A \Rightarrow_{\mathbb{C}} B$ when the category $\mathbb{C}$ should be explicitly indicated.

$- \Rightarrow_{\mathbb{C}} - : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Set}$ is a bifunctor, contravariant in its first argument and covariant in the second argument; its action on morphisms is given by

$$(f \Rightarrow_{\mathbb{C}} g)\ h = f\,;h\,;g.$$

We will often have occasion to make use of the fact that $X \Rightarrow -$ preserves limits (for example, $(X \Rightarrow Y \times Z) \cong (X \Rightarrow Y) \times (X \Rightarrow Z)$), and, dually, $- \Rightarrow X$ turns colimits into limits (for example, in a category with coproducts and products, $(Y + Z \Rightarrow X) \cong (Y \Rightarrow X) \times (Z \Rightarrow X)$).

Slice categories Given a category $\mathbb{C}$ and an object $X \in \mathbb{C}$, the *slice category* $\mathbb{C}/X$ has as its objects diagrams $C \xrightarrow{f} X$, that is, pairs (C, f) where $C \in \mathbb{C}$ and f is a morphism from C to X. Morphisms $(C_1, f_1) \to (C_2, f_2)$ in $\mathbb{C}/X$ are morphisms $g : C_1 \to C_2$ of $\mathbb{C}$ which make the relevant diagram commute:



A good intuition is to think of the morphism $f : C \to X$ as giving a "weighting" or "labelling" to C. The slice category $\mathbb{C}/X$ thus represents objects of $\mathbb{C}$ weighted or labelled by X, with weight/label-preserving maps as morphisms. For example, objects of $\mathbf{Set}/\mathbb{R}$ are sets where each element has been assigned a real-number weight; morphisms in $\mathbf{Set}/\mathbb{R}$ are weight-preserving functions.

[3]As Haskell programmers are well aware, writing everything in point-free style does not necessarily improve readability!

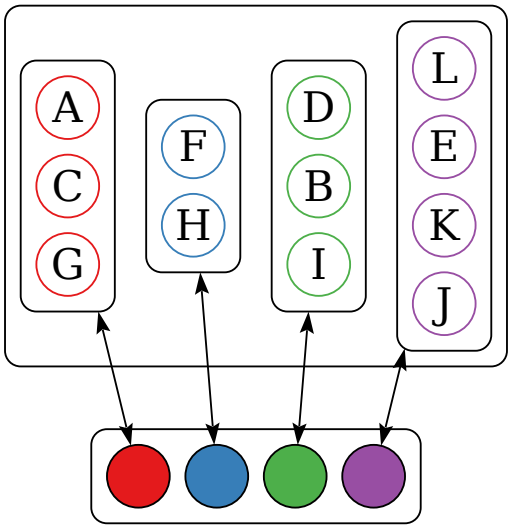


Figure 1.1: An element of $\mathbf{Set}^X$ or $\mathbf{Set}/X$

Functor categories Given two categories $\mathbb{C}$ and $\mathbb{D}$, the collection of functors from $\mathbb{C}$ to $\mathbb{D}$ forms a category (a *functor category*), with natural transformations as morphisms. This category is denoted by both of the notations $\mathbb{C} \Rightarrow \mathbb{D}$ and $\mathbb{D}^{\mathbb{C}}$, as convenient.[4] The notation $\mathbb{D}^{\mathbb{C}}$ is often helpful since intuition for exponents carries over to functor categories; for example, $\mathbb{C}^{\mathbb{D}+\mathbb{E}} \simeq \mathbb{C}^{\mathbb{D}} \times \mathbb{C}^{\mathbb{E}}$, $(\mathbb{C} \times \mathbb{D})^{\mathbb{E}} \simeq \mathbb{C}^{\mathbb{E}} \times \mathbb{D}^{\mathbb{E}}$, and so on. (In fact, this is not specific to functor categories; for example, the second isomorphism holds in any Cartesian closed category.)

Given a set X, the functor category $\mathbf{Set}^X$ (considering X as a discrete category) is equivalent to the slice category $\mathbf{Set}/X$. In particular, a functor of type $X \to \mathbf{Set}$ is an X-indexed collection of sets, whereas an object of $\mathbf{Set}/X$ is a set S with a function $f : S \to X$ labelling each element of S by some $x \in X$. Taking the preimage or *fiber* $f^{-1}(x)$ of each $x \in X$ recovers an X-indexed collection of sets; conversely, given an X-indexed collection of sets we may take their disjoint union and construct a function assigning each element of the disjoint union its corresponding element of X. Figure 1.1 illustrates the situation for $X = \{\text{red}, \text{blue}, \text{green}, \text{purple}\}$. Following the arrows from bottom to top, the diagram represents a functor $X \to \mathbf{Set}$, with each element of X mapped to a set. Following the arrows from top to bottom, the diagram represents an object in $\mathbf{Set}/X$ consisting of a set of 12 elements and an assignment of a color to each.

Equivalence of categories When are two categories "the same"? In traditional category theory, founded on set theory, there are quite a few different definitions of

[4] Traditionally the notation $[\mathbb{C}, \mathbb{D}]$ is also used, but $\mathbb{C} \Rightarrow \mathbb{D}$ is consistent with the general notation for *exponentials* explained in §1.4.2; the functor category $\mathbb{C} \Rightarrow \mathbb{D}$ is an exponential object in the Cartesian closed category of all small categories, **Cat**.

sameness for categories. There are many different notions of "equality" (isomorphism, equivalence, ...), and they often do not correspond to the underlying equality on sets, so one must carefully pick and choose which notions of equality to use in which situations (and some choices might be better than others!). Many concepts come with "strict" and "weak" variants, depending on which version of equality is being used. Maintaining the principle of equivalence in this setting requires hard work and vigilence.

A naïve first attempt is as follows:

Definition 1.4.1. Two categories $\mathbb{C}$ and $\mathbb{D}$ are *isomorphic* if there are inverse functors $\mathbb{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathbb{D}$, such that $GF = 1_{\mathbb{C}}$ and $FG = 1_{\mathbb{D}}$.

This definition has the right idea in general, but it is subtly flawed. It talks about *equality* of functors (GF and FG must be *equal to* the identity). However, two functors H and J can be isomorphic without being equal. In particular, two functors are *naturally isomorphic* if there is a pair of natural transformations $\phi : H \xrightarrow{\bullet} J$ and $\psi : J \xrightarrow{\bullet} H$ such that $\phi \circ \psi$ and $\psi \circ \phi$ are both equal to the identity natural transformation. For example, consider the functors given by the Haskell types

```
data Rose a = Node a [Rose a]
data Fork a = Leaf a | Fork (Fork a) (Fork a)
```

These are obviously not *equal*, but they are isomorphic (though not obviously so!), in the sense that there are natural transformations, *i.e.* polymorphic functions, *rose2fork*:: $\forall a.\mathsf{Rose}\ a \to \mathsf{Fork}\ a$ and *fork2rose* :: $\forall a.\mathsf{Fork}\ a \to \mathsf{Rose}\ a$, such that *rose2fork* $\circ$ *fork2rose* $= id$ and *fork2rose* $\circ$ *rose2fork* $= id$ [Yorgey, 2010, Hinze and James, 2010].

Definition 1.4.1 therefore violates the *principle of equivalence*—to be discussed in more detail in §2.1—which states that properties of mathematical structures should be invariant under isomorphism. Here, then, is a better definition:

Definition 1.4.2. Categories $\mathbb{C}$ and $\mathbb{D}$ are *equivalent* if there are functors $\mathbb{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathbb{D}$ which are inverse up to natural isomorphism, that is, there are natural isomorphisms $1_{\mathbb{C}} \cong GF$ and $FG \cong 1_{\mathbb{D}}$.

That is, the compositions of the functors F and G do not *literally* have to be the identity functor, but only (naturally) *isomorphic* to it.[5] This does turn out to be a well-behaved notion of sameness for categories [nLab, 2014c].

[5]The astute reader may note that the stated definition of natural isomorphism of functors mentions *equality* of natural isomorphism—do we also need to replace this with some sort of isomorphism to avoid violating the principle of equivalence? Is it turtles all the way down (up)? This is a subtle point, but it turns out that there's nothing wrong with equality of natural transformations. For the usual notion of category, there is no higher structure after natural transformations, *i.e.* no nontrivial morphisms (and hence no nontrivial isomorphisms) between natural transformations.

There is much more to say about equivalence of categories; §2.1 picks up the thread with a much fuller discussion of the relationships among equivalence of categories, equality, the axiom of choice, and classical versus constructive logic.

Adjunctions The topic of *adjunctions* is much too large to adequately cover here. For the purposes of this dissertation, the most important form of the definition to keep in mind is that a functor $F : \mathbb{C} \to \mathbb{D}$ is *left adjoint* to $G : \mathbb{D} \to \mathbb{C}$, notated $F \dashv G$, if and only if

$$\forall AB.\ (FA \Rightarrow_{\mathbb{D}} B) \cong (A \Rightarrow_{\mathbb{C}} GB),$$

that is, if there is some natural isomorphism matching morphisms $FA \to B$ in the category $\mathbb{D}$ with morphisms $A \to GB$ in $\mathbb{C}$. If F is left adjoint to G, we also say, symmetrically, that G is *right adjoint* to F.

One example familiar to functional programmers is *currying*,

$$(A \times B \Rightarrow C) \cong (A \Rightarrow (B \Rightarrow C)),$$

which corresponds to the adjunction

$$(- \times B) \dashv (B \Rightarrow -).$$

1.4.2 Monoidal categories

Recall that a *monoid* is a set S equipped with an associative binary operation

$$\diamond : S \times S \to S$$

and a distinguished element $\varepsilon : S$ which is an identity for $\diamond$. (See, for example, Yorgey [2012] for a discussion of monoids in the context of Haskell.) A *monoidal category* is the appropriate "categorification" of the concept of a monoid, *i.e.* with the set S replaced by a category, the binary operation by a bifunctor, and the equational laws by natural isomorphisms.

Definition 1.4.3. Formally, a *monoidal category* is a category $\mathbb{C}$ equipped with

- a bifunctor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$;
- a distinguished object $I \in \mathbb{C}$;
- a natural isomorphism $\alpha : \forall ABC.\ (A \otimes B) \otimes C \cong A \otimes (B \otimes C)$; and
- natural isomorphisms $\lambda : \forall A.\ I \otimes A \cong A$ and $\rho : \forall A.\ A \otimes I \cong A$.

α, λ, and ρ must additionally satisfy some coherence axioms, which ensure that parallel isomorphisms constructed from α, λ, and ρ are always equal; for details, see Mac Lane [1998, §VII.2].

We often write $(\mathbb{C}, \otimes, I)$ when we wish to emphasize the choice of a monoidal functor and identity object for a monoidal category $\mathbb{C}$.

Note that $\otimes$ is not just a function on objects, but a *bifunctor*, so there must also be a way to combine morphisms $f : A_1 \longrightarrow A_2$ and $g : B_1 \longrightarrow B_2$ into a morphism $f \otimes g : A_1 \otimes B_1 \longrightarrow A_2 \otimes B_2$ which respects identities and composition.

Note also that a category can be monoidal in more than one way. In fact, as we will see in Chapter 3, the category of species is monoidal in at least six different ways! Another example is **Set**, which has both Cartesian product and disjoint union as monoidal structures. More generally, categories with products (together with a terminal object) and/or coproducts (together with an initial object) are always monoidal.

There are many variants on the basic theme of monoidal categories; a few of the most important, for the purposes of this dissertation, are given here:

Definition 1.4.4. A monoidal category is *symmetric* if there is additionally a natural isomorphism $\gamma : \forall AB.\ A \otimes B \cong B \otimes A$ such that $\gamma_{AB} \circ \gamma_{BA} = id$ (along with some coherence conditions; see Mac Lane [1998, §VII.7]).

For example, **Set** is symmetric monoidal under both Cartesian product and disjoint union. As an example of a non-symmetric monoidal category, consider the functor category $\mathbb{C} \to \mathbb{C}$, with the monoid given by composition of functors.

Definition 1.4.5. A monoidal category $(\mathbb{C}, \otimes, I)$ is *closed* if there some bifunctor $[-,-] : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$ such that there is a natural isomorphism

$$\forall ABC.\ (A \otimes B \Rightarrow C) \cong (A \Rightarrow [B, C]), \tag{1.4.1}$$

that is, $- \otimes B$ is left adjoint to $[B, -]$. The object $[B, C]$ is called an *exponential* object, and can also be notated by C^B or by $B \Rightarrow C$. The bifunctor $[-,-]$ is also called an *internal Hom functor*.

Remark. The notation $[B, C]$ for the exponential of B and C is common, and used in the definition above for clarity. However, in the remainder of this dissertation, we will use the alternate notation $B \Rightarrow C$ instead. That is, the natural isomorphism (1.4.1) will be written instead as

$$\forall ABC.\ (A \otimes B \Rightarrow C) \cong (A \Rightarrow (B \Rightarrow C)).$$

This certainly does have the potential to introduce some ambiguity (although the ambiguity is only apparent, since it can always be resolved by type inference). However, it emphasizes the fact that exponential objects "act like" morphisms, and moreover it plays to the intuition of Haskell programmers, since Haskell makes no notational distinction between top-level functions and first-class functions passed as arguments or returned as results.

(**Set**, $\times$) is closed: $B \Rightarrow C$ is defined as the set of functions from B to C, and the required isomorphism is currying. Categories, like **Set**, which are closed with respect to the categorical product are called *Cartesian closed*. Intuitively, Cartesian closed categories are those which can "internalize" arrows, with objects that "act like" sets of morphisms. Put another way, Cartesian closed categories are those with "first-class morphisms". Functional programmers are familiar with this idea: in a language with first-class functions, the class of functions (morphisms) between two given types (objects) is itself a type (object).

Definition 1.4.6. A *strict* monoidal category is one in which α, λ, and ρ are *equalities* rather than natural isomorphisms.

It is often remarked that every monoidal category is equivalent to some strict one (for example, using the theory of cliques [Joyal and Street, 1991], explained in §2.1.4), which is used to justify the pretense that every monoidal category is strict; however, proving this requires the axiom of choice.

1.4.3 Ends and coends

Intuitively, ends and coends can be thought of as abstract categorical formulations of the concepts of *parametricity* and *modularity*, respectively. Both play an important role in this dissertation.

Coends Given a functor $T : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{D}$, a *coend* over T, denoted $\exists C.\, T(C, C)$,[6] is an object of $\mathbb{D}$ with morphisms $\omega_X : T(X, X) \to \exists C.\, T(C, C)$ for every X, such that the diagram

$$\begin{array}{ccc} & T(X', X) & \\ {\scriptstyle T(f,1)}\swarrow & & \searrow{\scriptstyle T(1,f)} \\ T(X, X) & & T(X', X') \\ {\scriptstyle \omega_X}\searrow & & \swarrow{\scriptstyle \omega_{X'}} \\ & \exists C.\, T(C, C) & \end{array}$$

commutes for all $X, X' : \mathbb{C}$ and $f : X \to X'$. (This square represents *dinaturality* of ω.) Additionally, a coend must satisfy an appropriate universal property guaranteeing its uniqueness up to isomorphism (see [Mac Lane, 1998, §IX.5–6]).

[6] Traditionally, coends are notated as $\int^C T(C, C)$, and ends as $\int_C T(C, C)$ (for example, this is the notation used by Mac Lane [1998]). However, the link to calculus is somewhat obscure [Fetisov, 2011] and not very helpful for building intuition. Moreover, using the traditional notation, it is hard to keep ends and coends straight. On the other hand, as I will show, $\exists C.\, T(C, C)$ and $\forall C.\, T(C, C)$ are deeply appropriate notations for coends and ends, respectively; they are easier to keep straight; and they help computer scientists and logicians build on existing intuition. I am fairly certain I have seen this notation used before, but cannot remember where; pointers are appreciated.

Since there must be morphisms $\omega_X : T(X, X) \to \exists C.\ T(C, C)$ for every C, one's first try might be to implement the coend as an indexed coproduct, $\biguplus_{C \in \mathbb{C}} T(C, C)$. Then the ω_X are just injections. This is a good start, but does not (in general) satisfy the commutative diagram shown above.

In the specific case when the objects of $\mathbb{D}$ can be thought of as sets or types with "elements", we can "force" the commutative diagram to hold by taking a *quotient* of the indexed coproduct. Elements of the indexed coproduct look like pairs (C, t) where $C \in \mathbb{C}$ and $t \in T(C, C)$. The idea behind the quotient is that we want to consider two pairs (C, t) and (C', t') equivalent if they are related by the functoriality of T. In particular, for each arrow $f : C \to C'$ in $\mathbb{C}$ and each $x \in T(C', C)$, we set $(C, T(f, 1)\ x) \sim (C', T(1, f)\ x)$. That is, we will consider (C, t) and (C', t') interchangeable as long as we have some way to map from C to C', and the associated values t and t' are related by the same mapping. (More generally, even if the objects of $\mathbb{D}$ are not sets, the same thing can be accomplished using *coequalizers*.)

Intuitively, the functor T can be thought of as an interface; (C, t) can then be thought of as a module with representation type C and implementation t. The morphisms ω_X thus package up a concrete representation type and implementation into a module, and the dinaturality condition ensures that one cannot directly observe the concrete representation type, but only distinguish values up to behavioral equivalence. Indeed, in Haskell, the coend of T is given by the type `exists c. T c c` [Kmett, 2008]—or rather, by an isomorphic encoding such as

```
data Coend t where
  Coend :: t c c → Coend t
```

since `exists` is not actually valid Haskell syntax. T is required to be a functor from $\mathbb{C}^{\mathrm{op}} \times \mathbb{C}$ since the representation type may occur both co- and contravariantly in the interface.

Coends preserve colimits; for example, in **Set**,

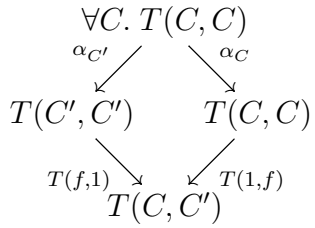
$$(\exists A.\ F\ A + G\ A) \cong (\exists A.\ F\ A) + (\exists A.\ G\ A).$$

(This is one place where using integral notation actually helps with intuition—but only for coends.)

Remark. $\exists L_1, L_2.\ \ldots$ is used as an abbrevation for a coend over the product category $\mathfrak{L} \times \mathfrak{L}$. (Given suitable assumptions it is also equivalent to an iterated coend; see Mac Lane [1998, §IX.8].)

Ends An *end* of a functor $T : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{D}$, notated $\forall C.\ T(C, C)$, is the categorical dual of a coend. That is, an end is an object of $\mathbb{D}$ together with morphisms α_X :

$\forall C.\ T(C,C) \to T(X,X)$ such that



commutes for all $C, C' : \mathbb{C}$ and $f : C \to C'$, together with an appropriate universal property. The end $\forall C.\ T(C,C)$ can thus be "instantiated" at any type X by the morphism α_X, and the dinaturality of α ensures that these instantiations all "behave uniformly". It should come as no surprise that ends in Haskell are given by universal quantification, that is, $\forall c.\mathsf{T}\ c\ c$ [Kmett, 2008].

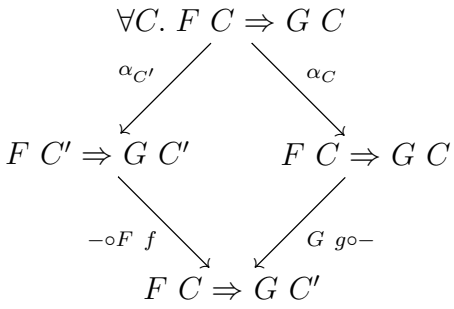
In fact, there is an even deeper connection between ends and Haskell's $\forall$ notation. It is well-known in the Haskell community that polymorphic—*i.e.* universally quantified—functions somehow correspond to natural transformations, via parametricity; this correspondence can be made formally precise as follows. Given two functors $F, G : \mathbb{C} \to \mathbb{D}$, consider the bifunctor

$$F- \Rightarrow G- : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Set},$$

which sends objects $X, Y \in C$ to the set of morphisms $F\ X \Rightarrow_{\mathbb{D}} G\ Y$ and acts on morphisms $f : X' \to X$ and $g : Y \to Y'$ by

$$(F\ f \Rightarrow G\ g)\ h = G\ g \circ h \circ F\ f.$$

Then an end $\forall C.\ F\ C \Rightarrow G\ C$ is a set with projections $\alpha_X : (\forall C.\ F\ C \Rightarrow G\ C) \to (F\ X \Rightarrow G\ X)$ such that



commutes. Reading off the edges of this diagram, we have $\alpha_{C'} \circ F\ f = Gg \circ \alpha_C$—precisely the definition of naturality for α. Thus an end over $F- \Rightarrow G-$ is precisely a natural transformation, that is,

$$(\forall C.\ F\ C \Rightarrow G\ C) \cong (F \xrightarrow{\bullet} G).$$

This is sometimes called the *naturality formula* [Cáccamo and Winskel, 2001], and formally justifies using the notation $\forall C.\ F\ C \Rightarrow G\ C$ for natural transformations between F and G, just as in Haskell. (See also Cheng and Willerton [2014a,b].)

Dually to coends, which preserve colimits, ends preserve limits. For example, in **Set**,

$$(\forall A.\ F\ A \times G\ A) \cong (\forall A.\ F\ A) \times (\forall A.\ G\ A).$$

1.4.4 The Yoneda lemma

Given a functor $F : \mathbb{C} \to \mathbf{Set}$, the Yoneda lemma states that for every $A \in \mathbb{C}$,

$$F\ A \cong (\forall B.\ (A \Rightarrow B) \Rightarrow F\ B),$$

that is, the set $F\ A$ is isomorphic to the set of natural transformations from $A \Rightarrow -$ to F. Haskell programmers may enjoy trying to implement a function of type $\mathsf{Functor}\ f \Rightarrow f\ a \to (\forall b.(a \to b) \to f\ b)$ and its inverse—doing so successfully will give some intuition into the nature of the lemma and why it is true.

The functor $j : \mathbb{C}^{\mathrm{op}} \to (\mathbb{C} \Rightarrow \mathbf{Set})$ defined on objects by $j(A) := (A \Rightarrow -)$ is known as the *Yoneda embedding*. As a corollary of the Yoneda lemma, j is full and faithful.

For further reading and intuition about the Yoneda lemma, see, for example, Baez [1999], Piponi [2006], or Kmett [2011].

1.4.5 Groupoids

A *groupoid* is a category in which all morphisms are invertible, that is, for each morphism f there is another morphism f^{-1} for which $f \circ f^{-1} = id$ and $f^{-1} \circ f = id$. Groupoids play a prominent role in both HoTT and in the theory of species and related theories [Byrne, 2006, Kock, 2012].

Example. Any group can be thought of as a groupoid with a single element, just as a monoid can be thought of as a one-object category. Conversely, groupoids can be thought of as "groups with types", where elements can only be composed if their types match (in contrast to a group, where any two elements can always be composed).

Example. There is a groupoid whose objects are natural numbers, and whose morphisms $m \longrightarrow n$ are the invertible $m \times n$ matrices over some field, with composition given by matrix multiplication. (Hence there are no morphisms when $m \neq n$, since only square matrices are invertible.)

The next two examples will play important roles in the remainder of the dissertation, so they merit the status of formal definitions.

Definition 1.4.7. B is the groupoid whose objects are finite sets and whose morphisms are bijections between finite sets.

Definition 1.4.8. **L** is the groupoid whose objects are finite sets equipped with a linear order, and whose morphisms are order-preserving bijections.

Note that there is exactly one order-preserving bijection between two linear orders of the same size, so **L** is rather impoverished compared with **B**. Nonetheless, there is a close relationship between them, which will be explored more in Chapters 2 and 3.

It is worth pointing out that **B** is an instance of a more general phenomenon:

Definition 1.4.9. Any category $\mathbb{C}$ gives rise to a groupoid $\mathbb{C}^*$, called the *core* of $\mathbb{C}$, whose objects are the objects of $\mathbb{C}$ and whose morphisms are the isomorphisms in $\mathbb{C}$.

Checking that $\mathbb{C}^*$ is indeed a groupoid is left as an easy exercise. Note in particular that $\mathbf{B} = \mathbf{FinSet}^*$.

Definition 1.4.10. The *symmetric groupoid* **P** is defined as the groupoid whose objects are natural numbers and whose morphisms $m \longrightarrow n$ are bijections $[m] \xrightarrow{\sim} [n]$.

Definition 1.4.11. The type of permutations on a set S, that is, bijections from S to itself, is denoted $S!$.

Note that the set of morphisms $m \longrightarrow n$ in **P** is empty unless $m = n$, and morphisms $n \longrightarrow n$ are permutations $[n]!$.

P is called the *symmetric groupoid* since it is isomorphic to an infinite coproduct $\coprod_{n \geqslant 0} \mathcal{S}_n$, where $\mathcal{S}_n$ denotes the symmetric *group* of all permutations on n elements, considered as a one-object groupoid. In other words, **P** consists of a collection of non-interacting "islands", one for each natural number, as illustrated in Figure 1.2. In particular, this means that any functor $F : \mathbf{P} \to \mathbb{C}$ is equivalent to a collection of functors $\prod_{n \geqslant 0} \mathcal{S}_n \to \mathbb{C}$, one for each natural number. Each functor $\mathcal{S}_n \to \mathbb{C}$ is entirely independent of the others, since there are no morphisms between different $\mathcal{S}_n$ to introduce constraints.

There is a close relationship between **B** and **P**. In the presence of the axiom of choice, they are equivalent; intuitively, **P** is what we get by noting that any two sets of the same size are isomorphic, so we might as well just forget about the elements of finite sets and work directly with their sizes. However, if the axiom of choice is rejected, the details become much more subtle; this is addressed in §2.3.

B and **P** do not have products or coproducts. For example, to see that **B** does not have coproducts, let $A, B \in \mathbf{B}$ be arbitrary finite sets, and suppose they have some coproduct $A+B$. By definition this comes with a diagram $A \xrightarrow{\iota_1} A + B \xleftarrow{\iota_2} B$ in **B**. Since morphisms in **B** are bijections, this would imply that A and B are in bijection, but since A and B were arbitrary finite sets, this is absurd. A similar argument applies in the case of products. More generally, any category with all products or coproducts is necessarily *connected*, *i.e.* has some zig-zag sequence of arrows connecting any two objects, and this is clearly not true of **B**.

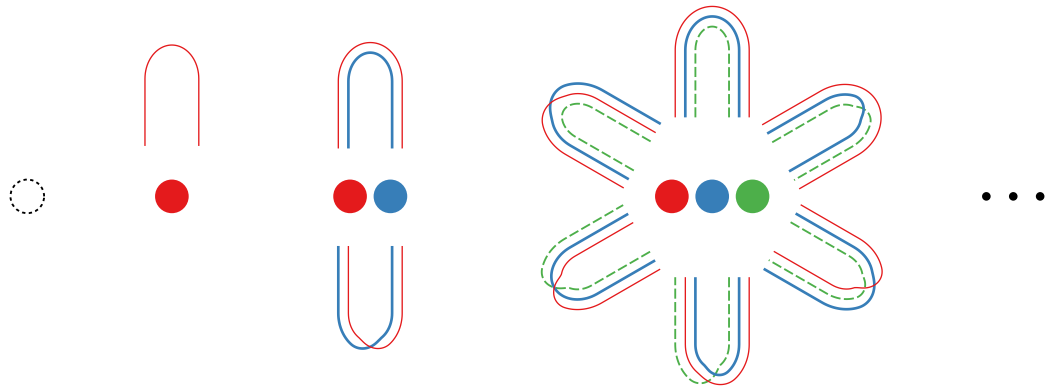


Figure 1.2: The groupoid **P**

B does, however, have monoidal structures given by Cartesian product and disjoint union of finite sets, even though these are not a categorical product or coproduct. In particular, two bijections $\sigma_1 : S_1 \xrightarrow{\sim} T_1$ and $\sigma_2 : S_2 \xrightarrow{\sim} T_2$ naturally give rise to a bijection $(S_1 \times S_2) \xrightarrow{\sim} (T_1 \times T_2)$, which sends (s_1, s_2) to $(\sigma_1(s_1), \sigma_2(s_2))$, as well as a bijection $(S_1 \uplus S_2) \xrightarrow{\sim} (T_1 \uplus T_2)$ which sends $\mathsf{inl}\ s_1$ to $\mathsf{inl}(\sigma_1(s_1))$ and $\mathsf{inr}\ s_2$ to $\mathsf{inr}(\sigma_2(s_2))$. In fact, something more general is true:

Proposition 1.4.12. *Any monoid* $(\otimes, 1)$ *on a category* $\mathbb{C}$ *restricts to a monoid* $(\otimes^*, 1)$ *on the groupoid* $\mathbb{C}^*$.

Proof. There is a forgetful functor $U : \mathbb{C}^* \to \mathbb{C}$ which is the identity on both objects and morphisms. Given $X, Y \in \mathbb{C}^*$, we may define

$$X \otimes^* Y = UX \otimes UY;$$

this may be considered as an object of $\mathbb{C}^*$ since $\mathbb{C}$ and $\mathbb{C}^*$ have the same objects. Given morphisms σ and τ of $\mathbb{C}^*$, we also define

$$\sigma \otimes^* \tau = U\sigma \otimes U\tau,$$

and note that the result must be an isomorphism in $\mathbb{C}$—hence a morphism in $\mathbb{C}^*$—since all functors (here, U and $\otimes$ in particular) preserve isomorphisms.

The fact that 1 is an identity for $\otimes^*$, associativity, and the coherence conditions all follow readily from the definitions. ∎

Chapter 2

Equality and Finiteness

Before delving into combinatorial species proper, we must first tackle some foundational issues—in particular, how equality and finiteness are handled in a constructive setting. As we will see, these topics require much more care in a constructive setting than in a classical one, but the extra care pays off in the form of deeper insight and even (in the case of finiteness) practical implementations.

We have already glimpsed some of the complexity surrounding equality in Chapter 1. Indeed, equality is the central focus of HoTT, and we saw that HoTT allows us to talk about many different notions of sameness.

This chapter highlights several other places where equality emerges as the key issue at stake. §2.1 begins by discussing the status of the *axiom of choice* (AC), which is frequently used in practice but inadmissible in a constructive setting. One of the main reasons that AC is used frequently in the context of category theory in particular has to do with the difference between equality and isomorphism. Several approaches to doing without AC are outlined, culminating in explaining (§2.2, §2.4) why it is unnecessary when formulating category theory inside of HoTT.

Interwoven with the story of equality and the axiom of choice is a story about *finiteness* (§2.3, §2.4). In a classical setting, the notion of a *finite* set is relatively uncomplicated. In a constructive setting, however, it becomes much more subtle. One must consider what counts as *constructive evidence* of finiteness, and how such evidence may be used. Finiteness turns out to play an important role in the theory of species, which are *labelled* by finite sets.

The key contributions of this chapter are

- a synthesis and presentation of many topics relevant to equality and finiteness (the axiom of choice, equivalence of categories, anafunctors, cliques, and some relevant results in HoTT) in a way accessible to functional programmers;
- a development of the theory of *cardinal-finite sets* in HoTT;
- development of HoTT-based analogues to the categories **B**, **P**, and **L**.

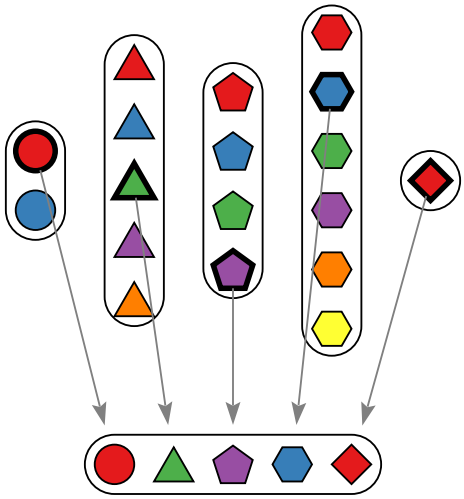


Figure 2.1: The axiom of choice

2.1 The axiom of choice (and how to avoid it)

The (in)famous *axiom of choice* (hereafter, AC) plays a central role in much of modern mathematics. In a constructive setting, however, it is problematic (§2.1.2, §2.1.3). Much effort has been expended attempting to avoid it [Makkai, 1995, 1996, 1998, Voevodsky]; in a sense, this can be seen as one of the goals of the univalent foundations program. In §2.2 and §2.4 we will see how HoTT indeed provides an excellent AC-free foundation for the mathematics we want to do. First, however, we give an introduction to AC and related issues in set theory.

2.1.1 The axiom of choice and constructive mathematics

The axiom of choice can be formulated in a number of equivalent ways. Perhaps the most well-known is

> The Cartesian product of any collection of non-empty sets is non-empty. (AC)

Given a family of sets $\{X_i \mid i \in I\}$, an element of their Cartesian product is some I-indexed tuple $\{x_i \mid i \in I\}$ where $x_i \in X_i$ for each i. Such a tuple can be thought of as a function (called a *choice function*) which picks out some particular x_i from each X_i. This can be visualized (for a particularly small and decidedly finite case) as shown in Figure 2.1.

Note that AC is *independent* of the usual set theory foundations (the so-called *Zermelo-Fraenkel axioms*, or ZF), in the sense that it is consistent to add either AC or its negation to ZF. It is somewhat controversial since it has some (seemingly)

strange consequences, *e.g.* the Banach-Tarski paradox [Wagon, 1993]. However, most mathematicians have come to accept it, and work (in principle) within ZF extended with AC, known as ZFC.

Consider how to express AC in type theory. First, we assume we have some type I which indexes the collection of sets; that is, there will be one set for each value of type I. Given some type A, we can define a subset of the values of type A using a *predicate*, that is, a function $P : A \to \mathcal{U}$. For some particular $a : A$, applying P to a yields a type, which can be thought of as the type of evidence that a is in the subset P; a is in the subset if and only if $P(a)$ is inhabited. An I-indexed collection of subsets of A can then be expressed as a function $C : I \to A \to \mathcal{U}$. In particular, $C(i, a)$ is the type of evidence that a is in the subset indexed by i. (Note that we could also make A into a family of types indexed by I, that is, $A : I \to \star$, which makes the encoding more expressive but doesn't ultimately affect the subsequent discussion.)

A set is nonempty if it has at least one element, so the fact that all the sets in C are nonempty can be modeled by a dependent function which yields an element of A for each index, along with a proof that it is contained in the corresponding subset:

$$(i : I) \to (a : A) \times C(i, a).$$

An element of the Cartesian product of C can be expressed as a function $I \to A$ that picks out an element for each I (the choice function), together with a proof that the chosen elements are in the appropriate sets:

$$(g : I \to A) \times ((i : I) \to C(i, g(i))).$$

Putting these together, apparently the axiom of choice can be modelled by the type

$$((i : I) \to (a : A) \times C(i, a)) \to (g : I \to A) \times ((i : I) \to C(i, g(i))).$$

Converting to Π and Σ notation and squinting actually gives some good insight into what is going on:

$$\left(\prod_{i:I} \sum_{a:A} C(i, a)\right) \to \left(\sum_{g:I\to A} \prod_{i:I} C(i, g(i))\right)$$

Essentially, this says that we can "turn a (dependent) product of sums into a (dependent) sum of products". This sounds a lot like distributivity, and indeed, the strange thing is that this is simply *true*: implementing a function of this type is a simple exercise! The intuitive idea can be grasped by implementing a non-dependent analogue, say, a Haskell function of type $(i \to (a, c)) \to (i \to a, i \to c)$. This is quite easy to implement, and the dependent version is essentially no harder; only the types get more complicated, not the implementation. So what's going on here? Why is AC so controversial if it is simply *true* in type theory?

The problem, it turns out, is that we've modelled the axiom of choice improperly, and it all boils down to how "non-empty" is defined. When a mathematician says "S is non-empty", they typically don't actually mean "... and here is an element of S to prove it". Instead, they literally mean "it is *not the case* that S is empty", that is, assuming S is empty leads to a contradiction. (Actually, there is something yet more subtle going on, to be discussed below, but this is a good first approximation.) In classical logic, these viewpoints are equivalent; in constructive logic, however, they are very different! In constructive logic, knowing that it is a contradiction for S to be empty does not actually help you find an element of S. We modelled the statement "this is a collection of non-empty sets" essentially by saying "here is an element in each set", but in constructive logic that is a much *stronger* statement than simply saying that each set is not empty.

From this point of view, we can see why the "axiom of choice" in the previous section was easy to implement: it had to produce a function choosing a bunch of elements, but it was given a bunch of elements to start. All it had to do was shuffle them around a bit. The "real" AC, on the other hand, has a much harder job: it is told some sets are non-empty, but without any actual elements being mentioned, and it then has to manufacture a bunch of elements out of thin air. In the context of constructive logic, this is deeply impossible: it turns out that the axiom of choice implies the law of excluded middle [Diaconescu, 1975, Goodman and Myhill, 1978], [Univalent Foundations Program, 2013, Theorem 10.1.14]! Working as we are in a type theory based on intuitionistic logic, we must therefore reject the axiom of choice.

Remark. It is worth noting that within HoTT, the notion of a "non-empty" set can be defined in a more nuanced way. The best way to model what classical mathematicians mean when they say "S is non-empty" is probably not with a negation, but instead with the *propositional truncation* of the statement that S contains an element [Univalent Foundations Program, 2013, Chapter 3]. This more faithfully mirrors the way mathematicians use it, for example, in reasoning such as "S is non-empty, so let $s \in S$ ...". Non-emptiness does in fact imply an inhabitant, but such an inhabitant can only be used to prove propositions.

Unfortunately, traditional category theory (founded in set theory) makes frequent—though hidden—use of the axiom of choice. The next sections explain the places where it occurs and some approaches to doing without it.

2.1.2 Unique isomorphism and generalized "the"

In category theory, one is typically interested in specifying objects only *up to (unique) isomorphism*. In fact, definitions which make use of actual *equality* on objects are sometimes referred to (half-jokingly) as *evil*. More positively, the *principle of equivalence* states that properties of mathematical structures should be invariant under isomorphism. This principle leads naturally to speaking of "the" object having some

property, when in fact there may be many objects with the given property but all such objects are uniquely isomorphic; this cannot cause confusion if the principle of equivalence is in effect.

Beneath this seemingly innocuous use of "the" (often referred to as *generalized "the"*), however, lurks the axiom of choice! In particular, one often wishes to define functors whose action on objects is defined only up to unique isomorphism, with no way to make a canonical choice of output object. In order to define such a functor one must resort to the axiom of choice to arbitrarily choose particular outputs. This seems like a fairly "benign" use of AC: if we have a collection of equivalence classes, where the elements in each class are all uniquely isomorphic, then using AC to pick one representative from each really "does not matter" in the sense that we cannot tell the difference between different choices (as long as we refrain from evil). Unfortunately, even such "benign" use of AC still poses a problem for computation.

2.1.3 AC and equivalence of categories

As hinted in §1.4.1, a particular example of the need for AC relates to equivalence of categories. The underlying issue is exactly that described in the previous section: namely, the need for functors defined only up to unique isomorphism.

Recall, from §1.4.1, the definition of equivalence of categories:

Definition 1.4.2. An *equivalence* between $\mathbb{C}$ and $\mathbb{D}$ is given by functors $\mathbb{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathbb{D}$ which are inverse up to natural isomorphism, that is, $1_{\mathbb{C}} \cong GF$ and $FG \cong 1_{\mathbb{D}}$.

In set theory, a function is a bijection—that is, an isomorphism of sets—if and only if it is both injective and surjective. By analogy, one might wonder what properties a functor $F : \mathbb{C} \to \mathbb{D}$ must have in order to be one half of an equivalence. This leads to the following definition:

Definition 2.1.1. $\mathbb{C}$ is *protoequivalent* to $\mathbb{D}$ if there is a functor $F : \mathbb{C} \to \mathbb{D}$ which is full and faithful (*i.e.* a bijection on each hom-set) as well as *essentially surjective*, that is, for every object $D \in \mathbb{D}$ there exists some object $C \in \mathbb{C}$ such that $F(C) \cong D$.

Intuitively, this says that F "embeds" an entire copy of $\mathbb{C}$ into $\mathbb{D}$ (the "full and faithful" part of the definition), and that every object of D which is not directly in the image of F is isomorphic to one that is. So every object of $\mathbb{D}$ is "included" in the image of $\mathbb{C}$, at least up to isomorphism (which is supposed to be all that matters).

So, are equivalence and protoequivalence the same thing? In one direction, it is not too hard to show that every equivalence is a protoequivalence: if F and G are inverse up to natural isomorphism, then they must be fully faithful and essentially surjective. It would be nice if the converse were also true: in that case, in order to prove two categories equivalent, it would suffice to construct a single functor F from one to the other, and show that F has the requisite properties. This often ends up

being more convenient than explicitly constructing two functors and showing they are inverse. However, it turns out that the converse is provable *only* if one accepts the axiom of choice![1] To get an intuitive sense for why this is, suppose $F : \mathbb{C} \to \mathbb{D}$ is fully faithful and essentially surjective. To construct an equivalence between $\mathbb{C}$ and $\mathbb{D}$ requires defining a functor $G : \mathbb{D} \to \mathbb{C}$ which is inverse to F (up to natural isomorphism). However, to define G we must give its action on each object $D \in \mathbb{D}$, that is, we must exhibit a function $\mathrm{Ob}\,\mathbb{D} \to \mathrm{Ob}\,\mathbb{C}$. We know that for each $D \in \mathbb{D}$ there *exists* some object $C \in \mathbb{C}$ with $F\ C \cong D$. That is,

$$\{\{C \in \mathbb{C} \mid F\ C \cong D\} \mid D \in \mathbb{D}\}$$

is a collection of nonempty sets. However, in a non-constructive logic, knowing these sets are nonempty does not actually give us any objects. Instead, we must use the axiom of choice, which yields a choice function $\mathrm{Ob}\,\mathbb{D} \to \mathrm{Ob}\,\mathbb{C}$, and this function can serve as the object mapping of the functor G.

Remark. It should be noted that without AC, protoequivalence is actually not even an equivalence relation on categories. To fix this, one must pass to the notion of a *weak equivalence* of categories, which consists of a *span* of protoequivalences [nLab, 2014d].

So AC is required to prove that every protoequivalence is an equivalence. In fact, the association goes deeper yet: it turns out that the statement

$$\text{every protoequivalence is an equivalence} \tag{AP}$$

(let's call this the "Axiom of Protoequivalence", or AP) not only requires AC, but is *equivalent* to it, in the sense that AC is derivable given AP as an axiom [nLab, 2014a]!

On purely intuitive grounds, however, it still feels like AP ought to be true. The particular choice of functor $G : \mathbb{D} \to \mathbb{C}$ doesn't matter, since it makes no difference up to isomorphism. One is therefore left in the awkward position of having two logically equivalent statements which it seems ought to be respectively affirmed and rejected.

Obviously this is not a tenable state of affairs; there are (at least) four options for resolving the situation.

1. If one is feeling particularly rational, one can simply say, "Since AC and AP are equivalent and I reject AC, I must therefore reject AP as well; my *feelings* about it are irrelevant."

This is a perfectly sensible and workable approach. It's important to highlight, therefore, that the "problem" is in some sense more a *philosophical* problem than

[1] At this point I should note that "protoequivalence" is not standard terminology, and now it should be clear why: there is no need for a distinct term if one accepts the axiom of choice.

a *technical* one. One can perfectly well adopt the above solution and continue to do category theory; it just may not be the "nicest" (a philosophical rather than technical notion!) way to do it.

There are also, however, several more creative options:

2. In a classical setting, one can avoid AC and affirm (an analogue of) AP by generalizing the notion of functor to that of *anafunctor* [Makkai, 1996]. Essentially, an anafunctor is a functor "defined only up to unique isomorphism". It turns out that the appropriate analogue of AP, where "functor" has been replaced by "anafunctor", is indeed true—and neither requires nor implies AC. Anafunctors act like functors in a sufficiently strong sense that one can simply do category theory using anafunctors in place of functors. However, one also has to replace natural transformations with "ananatural transformations", and so on, and it quickly gets rather fiddly. Anafunctors are defined and discussed in more detail in §2.1.5.

3. In a constructive setting, a witness of essential surjectivity is necessarily a function which gives an *actual witness* $C \in \mathbb{C}$, along with a proof that $F\ C \cong D$, for each $D \in \mathbb{D}$. In other words, a constructive witness of essential surjectivity is already a "choice function", and an inverse functor G can be defined directly, with no need to invoke AC and no need for anafunctors. So in constructive logic, AP is simply true. However, this version of "essential surjectivity" is rather strong, in that it forces you to make choices you might prefer not to make: for each $D \in \mathbb{D}$ there might be many isomorphic $C \in \mathbb{C}$ to choose from, with no canonical choice, and it is annoying (again, a philosophical rather than technical consideration!) to be forced to choose one.

4. Instead of generalizing functors, a more direct solution is to *generalize the notion of equality*. After all, what really seems to be at the heart of all these problems is differing notions of equality (*i.e.* equality of sets, isomorphism, equivalence ...). Of course, this is precisely what is done in HoTT.[2] It turns out that if one builds up suitable notions of category theory on top of HoTT instead of set theory, then AP is true, without the need for AC, and even with a *weaker* version of essential surjectivity that corresponds more closely to essential surjectivity in classical logic, using propositional truncation to encode the classical notion of existence. This is discussed in more detail in §2.2.

Ultimately, this last option using HoTT is the best. However, to fully appreciate it, it is helpful to first explore the notion of anafunctors, and the closely related notion of cliques.

[2] As a historical note, it seems that the original work on anafunctors is part of the same intellectual thread that led to the development of HoTT: see `http://byorgey.wordpress.com/2014/05/13/unique-isomorphism-and-generalized-the/#comment-13123`.

2.1.4 Cliques

As a preface to anafunctors, we begin with a brief outline of the theory of *cliques*, which are a formal way of representing the informal notion of an "equivalence class of uniquely isomorphic objects". Cliques were introduced by Joyal and Street [1991] for the specialized purpose of relating strict and non-strict monoidal categories. Makkai [1996] later noted that cliques are a special case of anafunctors; the precise relationship will be explained in §2.1.5.

The theory of cliques (and of anafunctors) amounts to a way of doing (set-theoretic) category theory without using the axiom of choice. However, building category theory directly in homotopy type theory (§2.2), instead of set theory, also obviates the need for the axiom of choice, but without the extra complication of anafunctors. This subsection and the next, therefore, are not strictly prerequisite to the remainder of this dissertation, but they help build intuition for the success of homotopy type theory, explained in §2.4.

Definition 2.1.2. A *clique* (I, A, u) in a category $\mathbb{C}$ is given by

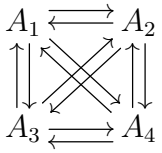
- a *non-empty* collection of objects $A = \{A_i \mid i \in I\}$, indexed by some collection I, and
- a collection of morphisms $u = \{\ A_i \xrightarrow{u_{ij}} A_j \ \mid i, j \in I\}$,

such that for all $i, j, k \in I$,

- $u_{ii} = id_{A_i}$, and
- $u_{ij}\ ;\ u_{jk} = u_{ik}$.

Remark. There are two things worth pointing out about this definition. First, the same object may occur multiple times in the collection A—that is, multiple different values of I may index the same object of $\mathbb{C}$. Second, the last two conditions together imply $u_{ij} = u_{ji}^{-1}$, since $u_{ij}\ ;\ u_{ji} = u_{ii} = id$.

A clique can thus be visualized as a graph-theoretic clique in a directed graph, with a unique morphism between any two objects:



Equivalently, a clique may be visualized as a clique in an *undirected* graph, with each edge representing an isomorphism. That is, a clique represents a collection of objects in $\mathbb{C}$ which are all isomorphic, with a single chosen isomorphism between each pair of objects.

Definition 2.1.3. A *morphism* between two cliques (I, A, u) and (J, B, v) is given by a collection of arrows

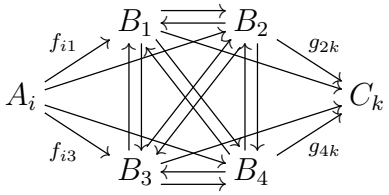
$$\{\ A_i \xrightarrow{f_{ij}} B_j \ \mid i \in I, j \in J\}$$

such that

$$\begin{array}{ccc} A_i & \xrightarrow{f_{ij}} & B_j \\ {\scriptstyle u_{ik}}\downarrow & & \downarrow{\scriptstyle v_{jl}} \\ A_k & \xrightarrow[f_{kl}]{} & B_l \end{array}$$

commutes for all $i, k \in I$ and $j, l \in J$. In other words, a morphism of cliques maps an entire class of isomorphic objects to another class—in particular, mapping each representative of the first class to each representative of the second—in a way that preserves the isomorphisms.

As one would expect, the class of cliques and clique morphisms in a category $\mathbb{C}$ itself forms a category, which we call $\operatorname{clq}\mathbb{C}$. It is easy to imagine what the identity morphism of cliques must be—the one that maps each A_i to A_j via u_{ij}. However, composition of clique morphisms is more subtle. Suppose we have three cliques with morphisms $(I, A, u) \xrightarrow{f} (J, B, v) \xrightarrow{g} (K, C, w)$. We must define a collection of morphisms $A_i \xrightarrow{h_{ik}} C_k$. For any given A_i and C_k, we have morphisms from A_i to each of the B_j, and from each of the B_j to C_k, with a representative example shown below.



If we pick a fixed $j \in J$, for each $i \in I$ and $k \in K$ we can define $h_{ik} = f_{ij} \,;\, g_{jk}$. Moreover, the resulting h_{ik} are independent of the choice of j, since everything in sight commutes. Specifically,

$$\begin{array}{ll} & f_{ij} \,;\, g_{jk} \\ = & \qquad \{ \quad v_{jl} \,;\, v_{lj} = v_{jj} = id \quad \} \\ & f_{ij} \,;\, v_{jl} \,;\, v_{lj} \,;\, g_{jk} \\ = & \qquad \{ \quad f, g \text{ are clique morphisms} \quad \} \\ & u_{ii} \,;\, f_{il} \,;\, g_{lk} \,;\, w_{kk} \\ = & \qquad \{ \quad u_{ii} = id;\ w_{kk} = id \quad \} \\ & f_{il} \,;\, g_{lk}. \end{array}$$

Since J is non-empty, it must contain some element j which we may arbitrarily use to define the h_{ik}.

Remark. If defining the theory of cliques within HoTT instead of set theory, this can be done in an even more principled way: the fact that J is non-empty should be modeled by its propositional truncation, $\|J\|$. This means that *in order* to be able to use the particular value of J hidden inside the truncation, we *must* show that the h_{ik} thus defined are independent of the choice of j.

The idea now is to replace functors $\mathbb{C} \to \mathbb{D}$ with functors $\mathbb{C} \to \operatorname{clq} \mathbb{D}$, which map objects of $\mathbb{C}$ to entire equivalence classes of objects in $\mathbb{D}$, instead of arbitrarily picking some object from each equivalence class. This gets rid of the need for AC in defining such functors. However, it is somewhat cumbersome to replace $\mathbb{D}$ by $\operatorname{clq} \mathbb{D}$ in this way. To make it tenable, one could imagine defining a new notion of "clique functor" $F : \mathbb{C} \overset{\text{clq}}{\to} \mathbb{D}$ given by a regular functor $\mathbb{C} \to \operatorname{clq} \mathbb{D}$, and showing that these clique functors act like functors in suitable ways. For example, it is easy to see that any regular functor $\mathbb{C} \to \mathbb{D}$ can be made into a trivial functor $\mathbb{C} \to \operatorname{clq} \mathbb{D}$, by sending each $C \in \mathbb{C}$ to the singleton clique containing only $F(C)$. One can also show that clique functors can be composed, have a suitable notion of natural transformations between them, and so on[3]. In fact, it turns out that this is precisely the theory of *anafunctors*.

2.1.5 Anafunctors

As an intuition for anafunctors it is helpful to keep in mind the equivalent concept of functors $\mathbb{C} \to \operatorname{clq} \mathbb{D}$—both represent functors whose values are specified only up to unique isomorphsim. Such functors represent a many-to-many relationship between objects of $\mathbb{C}$ and objects of $\mathbb{D}$. Normal functors, as with any function, may of course map multiple objects of $\mathbb{C}$ to the same object in $\mathbb{D}$. The novel aspect is the ability to have a single object of $\mathbb{C}$ correspond to multiple objects of $\mathbb{D}$. The key idea is to add a class of "specifications" which mediate the relationship between objects in the source and target categories, in exactly the same way that a "junction table" must be added to support a many-to-many relationship in a database schema. This is illustrated in Figure 2.2. On the left is a many-to-many relation between a set of shapes and a set of numbers. On the right, this relation has been mediated by a "junction table" containing a set of "specifications"—in this case, each specification is simply a pair of a shape and a number—together with two mappings (one-to-many relations) from the specifications to both of the original sets, such that a specification maps to a shape s and number n if and only if s and n were originally related.

Definition 2.1.4 (Makkai [1996]). An *anafunctor* $F : \mathbb{C} \to \mathbb{D}$ is defined as follows.

- There is a class S of *specifications*.

[3] In fact, $\operatorname{clq} -$ turns out to be a (2-)monad, and the category of clique functors is its Kleisli category [nLab, 2014b].

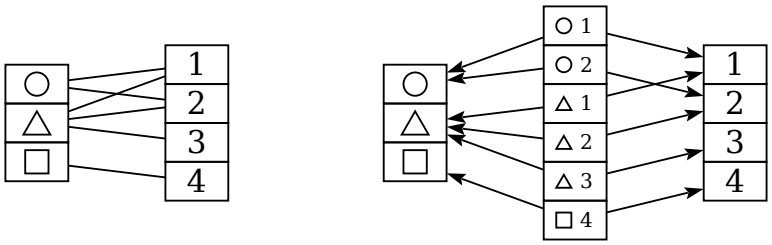


Figure 2.2: Representing a many-to-many relationship via a junction table

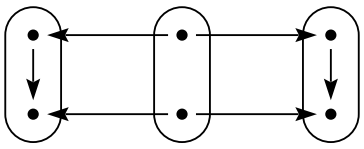
- There are two functions $\mathrm{Ob}\,\mathbb{C} \xleftarrow{\overleftarrow{F}} S \xrightarrow{\overrightarrow{F}} \mathrm{Ob}\,\mathbb{D}$ mapping specifications to objects of $\mathbb{C}$ and $\mathbb{D}$.

S, $\overleftarrow{F}$, and $\overrightarrow{F}$ together define a many-to-many relationship between objects of $\mathbb{C}$ and objects of $\mathbb{D}$. $D \in \mathbb{D}$ is called a *specified value of F at C* if there is some specification $s \in S$ such that $\overleftarrow{F}(s) = C$ and $\overrightarrow{F}(s) = D$, in which case we write $F_s(C) = D$. Moreover, D is *a value of F at C* (not necessarily a *specified* one) if there is some s for which $D \cong F_s(C)$.

The idea now is to impose additional conditions which ensure that F acts like a regular functor $\mathbb{C} \to \mathbb{D}$.

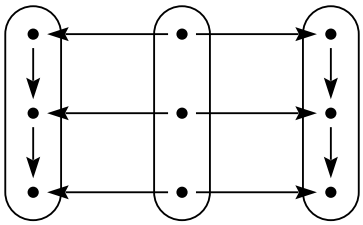
- Functors are defined on all objects; so we require each object of $\mathbb{C}$ to have at least one specification s which corresponds to it—that is, $\overleftarrow{F}$ must be surjective.

- Functors transport morphisms as well as objects. For each $s, t \in S$ (the middle of the below diagram) and each $f : \overleftarrow{F}(s) \to \overleftarrow{F}(t)$ in $\mathbb{C}$ (the left-hand side below), there must be a morphism $F_{s,t}(f) : \overrightarrow{F}(s) \to \overrightarrow{F}(t)$ in $\mathbb{D}$ (the right-hand side):



- Functors preserve identities: for each $s \in S$ we should have $F_{s,s}(id_{\overleftarrow{F}(s)}) = id_{\overrightarrow{F}(s)}$.

- Finally, functors preserve composition: for all $s, t, u \in S$ (in the middle below), $f : \overleftarrow{F}(s) \to \overleftarrow{F}(t)$, and $g : \overleftarrow{F}(t) \to \overleftarrow{F}(u)$ (the left side below), it must be the case that $F_{s,u}(f\,;g) = F_{s,t}(f)\,;F_{t,u}(g)$:



Remark. Our initial intuition was that an anafunctor should map objects of $\mathbb{C}$ to equivalence classes of objects in $\mathbb{D}$. This may not be immediately apparent from the definition, but is in fact the case. In particular, the identity morphism id_C maps to isomorphisms between specified values of C; that is, under the action of an anafunctor, an object C together with its identity morphism "blow up" into a clique. To see this, let $s, t \in S$ be two different specifications corresponding to C, that is, $\overleftarrow{F}(s) = \overleftarrow{F}(t) = C$. Then by preservation of composition and identities, we have

$$F_{s,t}(id_C)\,;F_{t,s}(id_C) = F_{s,s}(id_C\,;id_C) = F_{s,s}(id_C) = id_{\overrightarrow{F}(s)},$$

so $F_{s,t}(id_C)$ and $F_{t,s}(id_C)$ constitute an isomorphism between $F_s(C)$ and $F_t(C)$.

Remark. It is not hard to show that cliques in $\mathbb{D}$ are precisely anafunctors from $\mathbf{1}$ to $\mathbb{D}$. In fact, more is true: the class of functors $\mathbb{C} \to \operatorname{clq}\mathbb{D}$ is naturally isomorphic to the class of anafunctors $\mathbb{C} \to \mathbb{D}$ (for the proof, see Makkai [1996, pp. 31–34]).

There is an alternative, equivalent definition of anafunctors, which is somewhat less intuitive but usually more convenient to work with.

Definition 2.1.5. An anafunctor $F : \mathbb{C} \to \mathbb{D}$ is a category $\mathbb{S}$ together with a span of *functors* $\mathbb{C} \xleftarrow{\overleftarrow{F}} \mathbb{S} \xrightarrow{\overrightarrow{F}} \mathbb{D}$ where $\overleftarrow{F}$ is fully faithful and (strictly) surjective on objects.

Remark. In this definition, $\overleftarrow{F}$ must be *strictly* (as opposed to *essentially*) surjective on objects, that is, for every $C \in \mathbb{C}$ there is some $S \in \mathbb{S}$ such that $\overleftarrow{F}(S) = C$, rather than only requiring $\overleftarrow{F}(S) \cong C$. Given this strict surjectivity on objects, it is equivalent to require $\overleftarrow{F}$ to be full, as in the definition above, or to be (strictly) surjective on the class of all morphisms.

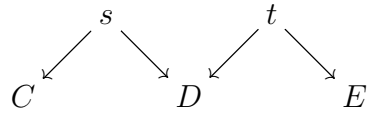
We are punning on notation a bit here: in the original definition of anafunctor, S is a set and $\overleftarrow{F}$ and $\overrightarrow{F}$ are functions on objects, whereas in this more abstract definition $\mathbb{S}$ is a category and $\overleftarrow{F}$ and $\overrightarrow{F}$ are functors. Of course, the two are closely related: given a span of functors $\mathbb{C} \xleftarrow{\overleftarrow{F}} \mathbb{S} \xrightarrow{\overrightarrow{F}} \mathbb{D}$, we may simply take the objects of $\mathbb{S}$ as the class of specifications S, and the actions of the functors $\overleftarrow{F}$ and $\overrightarrow{F}$ on

objects as the functions from specifications to objects of $\mathbb{C}$ and $\mathbb{D}$. Conversely, given a class of specifications S and functions $\overleftarrow{F}$ and $\overrightarrow{F}$, we may construct the category $\mathbb{S}$ with $\mathrm{Ob}\,\mathbb{S} = S$ and with morphisms $\overleftarrow{F}(s) \to \overleftarrow{F}(t)$ in $\mathbb{C}$ acting as morphisms $s \to t$ in $\mathbb{S}$. From $\mathbb{S}$ to $\mathbb{C}$, we construct the functor given by $\overleftarrow{F}$ on objects and the identity on morphisms, and the other functor maps $f : s \to t$ in $\mathbb{S}$ to $F_{s,t}(f) : \overrightarrow{F}(s) \to \overrightarrow{F}(t)$ in $\mathbb{D}$.

Every functor $F : \mathbb{C} \to \mathbb{D}$ can be trivially turned into an anafunctor

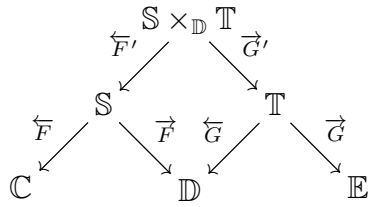
$$\mathbb{C} \xleftarrow{Id} \mathbb{C} \xrightarrow{F} \mathbb{D}\,.$$

Anafunctors also compose. Given compatible anafunctors $F : \mathbb{C} \xleftarrow{\overleftarrow{F}} S \xrightarrow{\overrightarrow{F}} \mathbb{D}$ and $G : \mathbb{D} \xleftarrow{\overleftarrow{G}} T \xrightarrow{\overrightarrow{G}} \mathbb{E}$, consider the action of their composite on objects: each object of $\mathbb{C}$ may map to multiple objects of $\mathbb{E}$, via objects of $\mathbb{D}$. Each such mapping corresponds to a zig-zag path



In order to *specify* such a path it suffices to give the pair (s, t), which determines C, D, and E. Note, however, that not every pair in $S \times T$ corresponds to a valid path, but only those which agree on the middle object $D \in \mathbb{D}$. Thus, we may take $\{(s,t) \mid s \in S, t \in T, \overrightarrow{F}(s) = \overleftarrow{G}(t)\}$ as the set of specifications for the composite $F;G$, with $\overleftarrow{F\,;G}(s,t) = \overleftarrow{F}(s)$ and $\overrightarrow{F\,;G}(s,t) = \overrightarrow{G}(t)$. On morphisms, $(F\,;G)_{(s,t),(u,v)}(f) = G_{t,v}(F_{s,u}(f))$. One can check that this satisfies the anafunctor laws.

The same thing can also be defined at a higher level in terms of spans:



Cat is complete, and in particular has pullbacks, so we may construct a new anafunctor from $\mathbb{C}$ to $\mathbb{E}$ by taking a pullback of $\overrightarrow{F}$ and $\overleftarrow{G}$ and then composing appropriately, as illustrated in the diagram.

One can go on to define ananatural transformations between anafunctors, and show that together these constitute a 2-category **AnaCat** which is analogous to the usual 2-category of (small) categories, functors, and natural transformations; in particular, there is a fully faithful embedding of **Cat** into **AnaCat**, which moreover

is an equivalence if AC holds. See Makkai [1996] for details.

To work in category theory based on set theory and classical logic, while avoiding AC, one is therefore justified in "mixing and matching" functors and anafunctors as convenient, but discussing them all as if they were regular functors (except when defining a particular anafunctor). Such usage can be formalized by turning everything into an anafunctor, and translating functor operations and properties into corresponding operations and properties of anafunctors. However, this is tediously complex (imagine if an introductory category theory textbook followed up the definition of categories with the definition of anafunctors!) and, as we will see, ultimately unnecessary. By founding category theory on HoTT instead of set theory, we can avoid the axiom of choice without incurring such complexity overhead. In a sense, HoTT takes all the added complexity of anafunctors and moves it into the background theory, so that "normal" functors secrectly become anafunctors.

2.2 Category theory in HoTT

Category theory works much better when founded in HoTT instead of set theory. Primarily, this is because in set theory the only notion of equality (extensional equality of sets) is too impoverished—one really wants to work up to *isomorphism* rather than literal equality, and the mismatch between isomorphism and strict equality introduces all sorts of difficulties and extra work. For example, many concepts have subtly different "strict" and/or "weak" variants, having to do with the sort of equality used in the definition. In contrast, via the univalence axiom, HoTT has a very rich—yet coherent—notion of equality that is able to encompass isomorphism in categories.

This section lays out a few relevant definitions along with some intuition and commentary. A fuller treatment may be found in Chapter 9 of the HoTT book [2013]. Generally, the term "h-widget" is used to refer to widgets as defined in HoTT, to distinguish from widgets as defined in set theory. There is nothing fundamentally new in this section, but it is valuable to collect and synthesize the particularly relevant bits of information which are otherwise scattered throughout the HoTT book.

We begin with the definition of a *precategory*.

Definition 2.2.1. A *precategory* $\mathcal{C}$ consists of

- A type $\mathcal{C}_0 : \mathcal{U}$ of objects (we often write simply $c : \mathcal{C}$ instead of $c : \mathcal{C}_0$);
- a function $- \Rightarrow - : \mathcal{C} \to \mathcal{C} \to \mathsf{Set}$ associating a *set* (0-type) of morphisms to each pair of objects (we often write $X \Rightarrow_{\mathcal{C}} Y$ to indicate the precategory being referenced, especially when multiple precategories are under consideration);
- a function $\mathsf{id} : (X : \mathcal{C}) \to (X \Rightarrow X)$ associating an identity morphism to each object;
- a function $- ; - : (X, Y, Z : \mathcal{C}) \to (X \Rightarrow Y) \to (Y \Rightarrow Z) \to (X \Rightarrow Z)$; and

- proofs of the identity and associativity laws.

Remark. Note how well the idea of types fits the definition: in the usual set-theoretic definition of a category, one must resort to awkward constructions like saying that composition is a partial function, with $f\,;g$ being defined only when $\mathrm{tgt}(f) = \mathrm{src}(g)$. Here, the same idea is expressed simply as the type of the composition operator.

The restriction that $X \Rightarrow Y$ is a *set*, *i.e.* a 0-type (rather than an arbitrary type) is important: otherwise one runs into problems with coherence of the identity and associativity laws, and extra laws become necessary. Down this path lie n-categories or even $(\infty, 1)$-categories; but to model traditional (1-)categories, it suffices for $X \Rightarrow Y$ to be a 0-type. In particular, this means that the identity and associativity laws, being equalities between elements of a 0-type, are themselves (-1)-types, *i.e.* mere propositions.

One might wonder why the term *precategory* is used for something that seems to be a direct port of the definition of a category from set theory into HoTT. The reason is that the usual formal definition of categories as expressed in set theory is incomplete: categories in fact come equipped with an extra *social* convention regarding their use—namely, "don't be evil", *i.e.* don't violate the principle of equivalence. In HoTT, we can formally encode this social convention as an axiom, which makes categories much nicer to work with in practice (after all, the social convention is not arbitrary, but encodes what category theorists have found to be a particularly nice way to do category theory).

Definition 2.2.2. An *isomorphism* in $\mathcal{C}$ is a morphism $f : X \Rightarrow Y$ together with a morphism $g : Y \Rightarrow X$ such that $f\,;g = \mathsf{id}_X$ and $g\,;f = \mathsf{id}_Y$. We write $X \cong Y$ for the type of isomorphisms between X and Y.

Remark. Note the distinction between $X \cong Y$, the type of isomorphisms between X and Y as objects in the precategory $\mathcal{C}$, and $X \simeq Y$, the type of equivalences between the types X and Y. The latter consists of a pair of inverse functions; the former of a pair of inverse *morphisms*. Morphisms, of course, need not be functions, and moreover, objects need not be types.

It is immediate, by path induction and the fact that id_X is an isomorphism, that equality implies isomorphism: we call this $\mathsf{idtoiso} : (X = Y) \to (X \cong Y)$. However, the other direction is not automatic; in particular, it does not follow from univalence, due to the distinction between $X \cong Y$ and $X \simeq Y$. However, requiring the other direction as an axiom is what allows us to formalize the principle of equivalence: isomorphic objects in a category should be truly interchangeable.

Definition 2.2.3. An *h-category* is a precategory $\mathcal{C}$ together with the additional univalence-like axiom that for all $X, Y : \mathcal{C}$,

$$(X \cong Y) \simeq (X = Y).$$

We write $\mathsf{isotoid} : (X \cong Y) \to (X = Y)$ for the left-to-right direction of the equivalence.

An h-groupoid is an h-category where every morphism is an isomorphism. The following example will play an important role later.

Definition 2.2.4. Any 1-type T gives rise to an h-groupoid $\mathcal{G}(T)$ where the objects are values $a : T$ and morphisms are equalities $a \Rightarrow b :\equiv (a = b)$, that is, morphisms from a to b are paths $p : a = b$.

Proof. The fact that T is a 1-type means that $a \Rightarrow b$ is a 0-type for $a, b : T$, as required. Identity morphisms, composition, the identity laws, associativity, and the fact that every morphism is an isomorphism all follow directly from properties of propositional equality. Since isomorphisms are already paths, $\mathsf{isotoid}$ is just the identity. ∎

Another important example of an h-category is an analogue to the usual category **Set** of sets and functions.

Definition 2.2.5 (HoTT book, 9.1.5, 9.1.7). $\mathcal{S}$ denotes the h-category of sets, that is, the category whose objects are 0-types, *i.e.* sets, and whose morphisms are functions $A \to B$.

Proof. This category is defined in the HoTT book in examples 9.1.5 and 9.1.7, and explored extensively in Chapter 10. However, the proof given in Example 9.1.7 leaves out some details, and it is worth spelling out the construction here.

Of course, identity morphisms are given by the identity function, and morphism composition by function composition, so the identity and associativity laws are satisfied. The definition also satisfies the requirement that the type of morphisms is a set, since $A \to B$ is a set whenever B is.

Finally, suppose $A \cong B$, that is, there are functions $f : A \to B$ and $g : B \to A$ such that $f \circ g = id_B$ and $g \circ f = id_A$. It is not *a priori* obvious that this is the same as an equivalence $A \simeq B$—indeed, it turns out to be so only because A and B are sets. Technically, $(A \cong B)$ constitutes a *quasi-inverse* between A and B, that is, $(A \cong B) \simeq (f : A \to B) \times \mathsf{qinv}(f)$, where $\mathsf{qinv}(f) :\equiv (g : B \to A) \times (f \circ g = id_B) \times (g \circ f = id_A)$. On the other hand, $(A \simeq B) \simeq (f : A \to B) \times \mathsf{isequiv}(f)$. The precise definition of $\mathsf{isequiv}(f)$ can be found in Chapter 4 of the HoTT book; for the present purpose, it suffices to say that although $\mathsf{qinv}(f)$ and $\mathsf{isequiv}(f)$ are *logically* equivalent (that is, each implies the other), $\mathsf{isequiv}(f)$ is always a mere proposition but in general $\mathsf{qinv}(f)$ may not be. However, in the specific case that A and B are sets, $\mathsf{qinv}(f)$ is indeed a mere proposition: by Lemma 4.1.4 in the HoTT book, if $\mathsf{qinv}(f)$ is inhabited then it is equivalent to $(x : A) \to (x = x)$, which is a mere proposition by function extensionality and the fact that A is a set. Therefore $\mathsf{qinv}(f) \simeq \mathsf{isequiv}(f)$, since logically equivalent mere propositions are equivalent, and we have $(A \cong B) \simeq (A \simeq B) \simeq (A = B)$ by univalence. ∎

The definitions of h-functors and h-natural transformations are straightforward ports of their usual definitions in set theory.

Definition 2.2.6. An h-functor F between (pre)categories $\mathcal{C}$ and $\mathcal{D}$ is a pair of functions

- $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$
- $F_1 : (X, Y : \mathcal{C}) \to (X \Rightarrow_{\mathcal{C}} Y) \to (F_0(X) \Rightarrow_{\mathcal{D}} F_0(Y))$

together with proofs of the functor laws,

- $(X : \mathcal{C}) \to (F_1(\mathsf{id}_X) = \mathsf{id}_{F_0(X)})$, and
- $(X, Y, Z : \mathcal{C}) \to (f : X \Rightarrow_{\mathcal{C}} Y) \to (g : Y \Rightarrow_{\mathcal{C}} Z) \to (F_1(f\,;g) = F_1(f)\,;F_1(g))$.

As is standard, we often write $F\ X$ and $F\ f$ instead of $F_0(X)$ and $F_1(f)$.

Definition 2.2.7. An h-natural transformation γ between h-functors $F, G : \mathcal{C} \to \mathcal{D}$ is a family of morphisms

- $\gamma_X : F\ X \Rightarrow_{\mathcal{D}} G\ X$

for each $X : \mathcal{C}$, satisfying

- $(X, Y : \mathcal{C}) \to (f : X \Rightarrow_{\mathcal{C}} Y) \to (\gamma_X\,; Gf = Ff\,; \gamma_Y)$.

It may not be readily apparent from the definitions, but as claimed earlier, this turns out to be a much nicer framework in which to carry out category theory. An extended example is given in §2.4. For now we describe two smaller (but also relevant) examples.

2.2.1 Monoidal categories in HoTT

The first example is the theory of *monoidal categories*. Recall that a monoidal category $\mathbb{C}$ is one with a bifunctor $\otimes : \mathbb{C}^2 \to \mathbb{C}$, an identity object $1 \in \mathbb{C}$, and natural isomorphisms α, λ, and ρ expressing the associativity and identity laws (along with some extra coherence laws). In set theory, there is also a notion of a *strict* monoidal category, where associativity and the identity laws hold up to *equality* rather than just isomorphism. In HoTT-based category theory, however, functors between h-categories—as opposed to precategories—are naturally isomorphic if and only if they are equal (HoTT book, Theorem 9.2.5). Thus, there is no difference between strict and non-strict monoidal h-categories.

2.2.2 Coends in HoTT

The second example is the notion of a *coend*. Recall that a coend over a functor $T : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{D}$ is an object of $\mathbb{D}$, denoted $\exists C.\ T(C, C)$, together with a family of morphisms $\omega_X : T(X, X) \to \exists C.\ T(C, C)$ for each $X \in \mathbb{C}$, such that

$$\begin{array}{ccccc} & & T(X', X) & & \\ & \overset{T(f,1)}{\swarrow} & & \overset{T(1,f)}{\searrow} & \\ T(X, X) & & & & T(X', X') \\ & \underset{\omega_X}{\searrow} & & \underset{\omega_{X'}}{\swarrow} & \\ & & \exists C.\ T(C, C) & & \end{array} \tag{2.2.1}$$

commutes for all $X, X' : \mathbb{C}$ and $f : X \to X'$. In set theory, recall that $\biguplus_C T(C, C)$, together with the obvious family of injections $\omega_C\ t = (C, t)$, comes close to being the right implementation of $\exists C.\ T(C, C)$, but fails to satisfy (2.2.1): in particular, the outputs of ω_X and $\omega_{X'}$ are never equal when $X \neq X'$, precisely because $\biguplus_C$ denotes a *disjoint* union. Instead, we must quotient this disjoint union by the equivalence relation induced by (2.2.1).

In HoTT, given some categories $\mathcal{C}$ and $\mathcal{D}$ and a functor $T : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$, we can directly encode this quotient as a higher inductive type $\exists T$. We first introduce a data constructor

$$\langle\!\langle -, - \rangle\!\rangle : (X : \mathcal{C}) \to T(X, X) \to \exists T.$$

So far this is equivalent to the Σ-type $\sum_C T(C, C)$, which corresponds to the disjoint union $\biguplus_C T(C, C)$. However, we also introduce a path constructor with type

$$(X, X' : \mathcal{C}) \to (t : T(X', X)) \to (f : X \Rightarrow X') \to \langle\!\langle X, T(f, 1)\ t \rangle\!\rangle = \langle\!\langle X', T(1, f)\ t \rangle\!\rangle$$

which ensures that the commutative diagram (2.2.1) is satisfied.

It is already convenient to be able to work directly with a data type representing a coend. The special case where $\mathcal{C}$ is a groupoid is even more convenient. In a groupoid, any morphism $f : X \Rightarrow X'$ is automatically an isomorphism, $f : X \cong X'$, and hence there is a path $\mathsf{isotoid}\ f : X = X'$. Moreover, one can show that

$$(\mathsf{isotoid}\ f)_*(T(f, 1)\ t) = T(1, f)\ t$$

($(\mathsf{isotoid}\ f)_*$ applies f covariantly and f^{-1} contravariantly), and therefore the above path constructor comes for free! In other words, when $\mathcal{C}$ is a groupoid and $T : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$, the coend type $\exists T$ defined above is equivalent to the simple Σ-type $\sum_C T(C, C)$—that is, the extra higher path constructor is entirely redundant. The equalities which were missing in set theory are supplied automatically by HoTT's richer system of equality.

2.3 Finiteness in set theory

Finally, we can assemble the foregoing material on anafunctors and category theory in HoTT into a coherent story about *finiteness*, first using set-theoretic foundations, and then using HoTT. The material in this section and the next (other than the lemmas and theorems cited from the HoTT book) is novel.

Recall that $\mathbf{B}$ denotes the groupoid of finite sets and bijections, and $\mathbf{P}$ the groupoid of natural numbers and permutations. In classical category theory, $\mathbf{P}$ is a *skeleton* of $\mathbf{B}$—roughly, we may think of it as the result of replacing each equivalence class of isomorphic objects in $\mathbf{B}$ with a single object. In this case, we identify each equivalence class of isomorphic finite sets with a natural number *size*—size being the one property of sets which is invariant under isomorphism. The relationship between $\mathbf{B}$ and $\mathbf{P}$ is central to the concept of finiteness: passing from $\mathbf{B}$ to $\mathbf{P}$ corresponds to taking the *size* of finite sets, and passing from $\mathbf{P}$ to $\mathbf{B}$ corresponds to constructing canonical finite sets of a given size. The study of $\mathbf{B}$ and $\mathbf{P}$ is also critical for the theory of species; as we will shortly see in Chapter 3, traditional species are defined as functors $\mathbf{B} \to \mathbf{Set}$.

It is a simple result in classical category theory that every category is equivalent to its skeletons. This equivalence allows one to pass freely back and forth between functors $\mathbf{B} \to \mathbf{Set}$ and functors $\mathbf{P} \to \mathbf{Set}$, and this is often implicitly exploited in the literature on species. However, we are interested in the *computational* content of this equivalence, and it is here that we run into trouble. After the foregoing discussion of cliques and anafunctors, the idea of quotienting out by equivalence classes of isomorphic objects ought to make us squeamish—and, indeed, the proof that $\mathbf{B}$ and $\mathbf{P}$ are equivalent requires AC.

In more detail, it is easy to define a functor $[-] : \mathbf{P} \to \mathbf{B}$ which sends the natural number n to the finite set $[n]$ and preserves morphisms; defining an inverse functor $\#- : \mathbf{B} \to \mathbf{P}$, however, is more problematic. We can send each set S to its size $\#S$, but we must send each bijection $S \xrightarrow{\sim} T$ to a permutation $[\#S] \xrightarrow{\sim} [\#T]$, and there is no obvious way to pick one. For example, suppose $S = \{\text{cat}, \text{dog}, \text{moose}\}$ and $T = \{\bigcirc, \triangle, \square\}$. Given a bijection matching each animal with its favorite shape[4], it must be sent to a permutation on $\{0, 1, 2\}$—but to which permutation should it be sent? Knowing that the size of $\{\text{cat}, \text{dog}, \text{moose}\}$ is 3 does not tell us anything about how to match up animals with $\{0, 1, 2\}$.

Abstractly, $[-] : \mathbf{P} \to \mathbf{B}$ is fully faithful and essentially surjective (every finite set is in bijection with $[n]$ for some n); this yields an equivalence of categories, and hence an inverse functor $\#- : \mathbf{B} \to \mathbf{P}$, only in the presence of AC. More concretely, we can use AC to choose an arbitrary bijection $\varphi_S : S \xrightarrow{\sim} [\#S]$ for each finite set S, somehow matching up S with the canonical set of size $\#S$. Given $\alpha : S \xrightarrow{\sim} T$ we can then construct

$$[\#S] \xrightarrow{\varphi_S^{-1}} S \xrightarrow{\alpha} T \xrightarrow{\varphi_T} [\#T] \,.$$

[4]The details are left as an exercise for the reader.

This use of AC is "benign" in the sense that all choices yield equivalent functors; this construction using AC thus in some sense yields a well-defined functor but has no computational interpretation.

We can avoid the use of AC by constructing an *anafunctor* $\#- : \mathbf{B} \to \mathbf{P}$ instead of a functor. In particular, as the class of specifications $S_\#$, we choose the class of sets paired with bijections to canonical finite sets of the appropriate size,

$$\sum_{T \in \mathbf{B}} (T \xrightarrow{\sim} [\#T]).$$

The function $\overleftarrow{\#} : S_\# \to \operatorname{Ob} \mathbf{B}$ simply forgets the chosen bijection, that is, $\overleftarrow{\#}\,(T, \varphi) = T$, and $\overrightarrow{\#} : S_\# \to \operatorname{Ob} \mathbf{P}$ sends finite sets to their size, $\overrightarrow{\#}\,(T, \varphi) = \#T$. Note that both $\overleftarrow{\#}$ and $\overrightarrow{\#}$ ignore φ, which is instead needed to define the action of $\#$ on morphisms. In particular, given $\alpha : S \xrightarrow{\sim} T$ in $\mathbf{B}$, we define $\#_{(S,\varphi_S),(T,\varphi_T)}(\alpha) = \varphi_S^{-1} ; \alpha ; \varphi_T$, which can be visualized as

$$\begin{array}{ccc} S & \xleftarrow{\varphi_S^{-1}} & [\#S] \\ {\scriptstyle\alpha}\downarrow & & \vdots\,{\scriptstyle\#\alpha} \\ T & \xrightarrow[\varphi_T]{} & [\#T] \end{array} .$$

Proof that $\#$ preserves identities and composition is given by the following diagrams:

$$\begin{array}{ccc} S & \xleftarrow{\varphi_S^{-1}} & [\#S] \\ {\scriptstyle id}\downarrow & & \vdots\,{\scriptstyle\#id} \\ S & \xrightarrow[\varphi_S]{} & [\#S] \end{array} \qquad \begin{array}{ccc} S & \xleftarrow{\varphi_S^{-1}} & [\#S] \\ {\scriptstyle\alpha}\downarrow & & \vdots\,{\scriptstyle\#\alpha} \\ T & \underset{\varphi_T^{-1}}{\overset{\varphi_T}{\rightleftarrows}} & [\#T] \\ {\scriptstyle\beta}\downarrow & & \vdots\,{\scriptstyle\#\beta} \\ U & \xrightarrow[\varphi_U]{} & [\#U] \end{array} = \begin{array}{ccc} S & \xleftarrow{\varphi_S^{-1}} & [\#S] \\ {\scriptstyle\alpha;\beta}\downarrow & & \vdots\,{\scriptstyle\#(\alpha;\beta)} \\ U & \xrightarrow[\varphi_U]{} & [\#U] \end{array}$$

The left-hand diagram represents the definition of $\#id$, in which φ_S and its inverse cancel, resulting in the identity. The center diagram shows the result of composing $\#\alpha$ and $\#\beta$; because φ_T cancels with φ_T^{-1} it is the same as the definition of $\#(\alpha ; \beta)$ (the right-hand diagram).

As a side note, it is worth mentioning an alternate way around the use of AC in this particular case, using the theory of *hereditarily finite* sets.

Definition 2.3.1. A *hereditarily finite* set is a finite set, all of whose elements are hereditarily finite.

This definition gets off the ground since the empty set is vacuously hereditarily finite. As is usual in set theory, this definition is interpreted inductively, so there cannot be any infinitely descending membership chains. Hereditarily finite sets are thus identi-

fied with finitely-branching, finite-depth trees (with no inherent order given to sibling nodes).

Now consider the groupoid $\mathbf{H}$ obtained by replacing "finite" with "hereditarily finite" in the definition of $\mathbf{B}$. That is, the elements of $\mathbf{H}$ are hereditarily finite sets, and the morphisms are bijections. This is no great loss, since given some finite set we are not particularly interested in the intensional properties of its elements, but only in its extensional properties (how many elements it has, which elements are equal to other elements, and so on).

Unlike the class of all sets, however, the class of all hereditarily finite sets (normally written V_ω) has a well-ordering. For example, we can compare two hereditarily finite sets by first inductively sorting their elements, and then performing a lexicographic comparison between the two ordered sequences of elements. This means that every hereditarily finite set has an induced ordering on its elements, since the elements are themselves hereditarily finite. In other words, picking a well-ordering of V_ω is like making a "global" choice of orderings, assigning a canonical bijection $S \xrightarrow{\sim} [\#S]$ for every hereditarily finite set S.

However, this construction is somewhat arbitrary, and has no natural counterpart in type theory, or indeed in a structural set theory. The concept of hereditary finiteness only makes sense in a material set theory such as ZF. To determine the canonical ordering on, say, $\{\mathrm{dog}, \mathrm{cat}, \mathrm{moose}\}$, we need to know the precise identity of the set used to encode each animal—but knowing their precise encoding as sets violates the principle of equivalence, since there may be many possible encodings with the right properties.

2.4 Finiteness in HoTT

We now turn to developing counterparts to the groupoids $\mathbf{P}$ and $\mathbf{B}$ in type theory. §2.4.1 presents some necessary lemmas and defines $\mathcal{P}$ as a type-theoretic analogue to $\mathbf{P}$. §2.4.2 then presents the theory of cardinal-finiteness in HoTT and uses it to define $\mathcal{B}$, a type-theoretic analogue to $\mathbf{B}$. This leads to an interesting tangent exploring "manifestly finite" sets and their relation to linear orders in §2.4.3; finally, §2.4.4 ties things together by considering the equivalence of $\mathcal{P}$ and $\mathcal{B}$, in particular showing how using HoTT as a foundation allows us to avoid the axiom of choice.

2.4.1 Preliminaries

Lemma 2.4.1. *Equivalence preserves set-ness, that is, if A and B are sets, then so is $A \simeq B$.*

Proof. $(A \simeq B) \simeq ((f : A \to B) \times \mathsf{isequiv}(f))$, where $\mathsf{isequiv}(f)$ is a mere proposition expressing the fact that f is an equivalence (*i.e.* has a suitable inverse). This is a set since $\mathsf{isequiv}(f)$ is a mere proposition (and hence a set), $A \to B$ is a set whenever B is, and $\times$ takes sets to sets [HoTT book, Lemma 3.3.4, Examples 3.1.5 and 3.1.6]. ∎

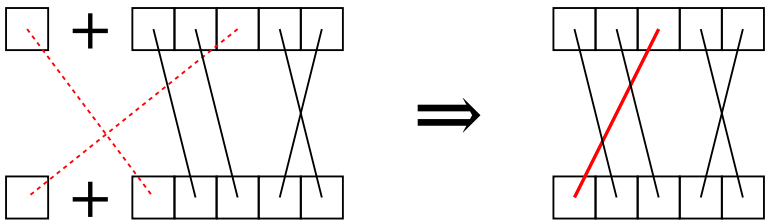


Figure 2.3: Eliminating $\top$ from both sides of an equivalence

Corollary 2.4.2. *If A and B are sets, then so is $A = B$.*

Proof. Immediate from univalence and Lemma 2.4.1. ∎

Lemma 2.4.3. *For all $n_1, n_2 : \mathbb{N}$, if $\mathsf{Fin}\ n_1 \simeq \mathsf{Fin}\ n_2$ then $n_1 = n_2$.*

Proof. The proof is by double induction on n_1 and n_2.

- If both n_1 and n_2 are zero, the result is immediate.
- The case when one is zero and the other a successor is impossible. In particular, taking the equivalence in the appropriate direction gives a function $\mathsf{Fin}\ (\mathsf{S}\ldots) \to \mathsf{Fin}\ \mathsf{O}$, which can be used to produce an element of $\mathsf{Fin}\ \mathsf{O} = \bot$, from which anything follows.
- In the case when both are a successor, we have $\mathsf{Fin}\ (\mathsf{S}\ n_1') \simeq \mathsf{Fin}\ (\mathsf{S}\ n_2')$, which is equivalent to $\top + \mathsf{Fin}\ n_1' \simeq \top + \mathsf{Fin}\ n_2'$. If we can conclude that $\mathsf{Fin}\ n_1' \simeq \mathsf{Fin}\ n_2'$, the inductive hypothesis then yields $n_1' = n_2'$, from which $\mathsf{S}\ n_1' = \mathsf{S}\ n_2'$ follows immediately. The implication $(\top + \mathsf{Fin}\ n_1' \simeq \top + \mathsf{Fin}\ n_2') \to (\mathsf{Fin}\ n_1' \simeq \mathsf{Fin}\ n_2')$ is true, but not quite as straightforward to show as one might think! In particular, an equivalence $(\top + \mathsf{Fin}\ n_1' \simeq \top + \mathsf{Fin}\ n_2')$ may not match the $\top$ values with each other. As illustrated in Figure 2.3, given $e : (\top + \mathsf{Fin}\ n_1' \simeq \top + \mathsf{Fin}\ n_2')$, it suffices to define $e'(e^{-1}\ \star) = e\ \star$, with the rest of $e' : \mathsf{Fin}\ n_1' \simeq \mathsf{Fin}\ n_2'$ defined as a restriction of e. This construction corresponds more generally to the *Gordon complementary bijection principle* [Gordon, 1983], whereby a bijection $A_1 \xrightarrow{\sim} B_1$ can be constructively "subtracted" from a bijection $(A_0 + A_1) \xrightarrow{\sim} (B_0 + B_1)$, yielding a bijection $A_0 \xrightarrow{\sim} B_0$. (Unfortunately, I do not currently know of a good way to encode a proof of the fully general bijection principle in a constructive logic.) ∎

Remark. It seems somewhat strange that the above proof has so much computational content—the required manipulations of equivalences are quite nontrivial—when the end goal is to prove a mere proposition. I do not know whether there is a simpler proof.

Constructing a type-theoretic counterpart to **P** is now straightforward.

Definition 2.4.4. $\mathcal{P}$ is the h-groupoid where

- the objects are values of type $\mathbb{N}$, and
- the morphisms $m \Rightarrow n$ are equivalences of type $\mathsf{Fin}\ m \simeq \mathsf{Fin}\ n$.

It is easy to check that this satisfies the axioms for an h-category, the salient points being that $\mathsf{Fin}\ m \simeq \mathsf{Fin}\ n$ is a set by Lemma 2.4.1 and that $\mathsf{isotoid}$ follows from Lemma 2.4.3.

2.4.2 Cardinal-finiteness

Developing a counterpart to **B** is more subtle. The first order of business is to decide how to port the concept of a "finite set". Generally, "a set with property X" ports to type theory as "a type paired with constructive evidence of property X" (or perhaps "a 0-type paired with evidence of X", depending how seriously we want to take the word *set*); so what is constructive evidence of finiteness? This is not *a priori* clear, and indeed, there are several possible answers [nLab, 2013]. However, the discussion of §2.3, where bijections $S \xrightarrow{\sim} [\#S]$ played a prominent role, suggests that we adopt the simplest option, *cardinal-finiteness*.

Definition 2.4.5. A set A is *cardinal-finite* iff there exists some $n \in \mathbb{N}$ and a bijection $A \xrightarrow{\sim} [n]$; n is called the size or cardinality of A.

Our first try at encoding this in type theory is

$$\mathcal{U}_{\mathrm{Fin}} :\equiv (A : \mathcal{U}) \times (n : \mathbb{N}) \times (A \simeq \mathsf{Fin}\ n).$$

We would like to build a groupoid having such finite types as objects, and equivalences between them as morphisms. Recall that, given some 1-type A, the groupoid $\mathcal{G}(A)$ has values $(a : A)$ as its objects and paths $a = b$ as its morphisms. For this to be applicable, we must check that $\mathcal{U}_{\mathrm{Fin}}$ is a 1-type. In fact, it turns out that it is a 0-type, *i.e.* a set—but this won't do, because the resulting groupoid is therefore *discrete*, with at most one morphism between each pair of objects. **B**, of course, has $n!$ distinct morphisms between any two sets of size n. Intuitively, the problem is that paths between objects in $\mathcal{G}(\mathcal{U}_{\mathrm{Fin}})$ involve not just the types in question but also the evidence of their finiteness, so that a path between two finite types requires them to be not just equivalent as types, but also "finite in the same way".

The situation can be pictured as shown in Figure 2.4. The elements of types A_1 and A_2 are shown on the sides; the evidence of their finiteness is represented by bijections between their elements and the elements of $\mathsf{Fin}\ n$, shown along the bottom. The catch is that the diagram necessarily contains only triangles: corresponding elements of A_1 and A_2 must correspond to the same element of $\mathsf{Fin}\ n$ on the bottom row. Therefore,

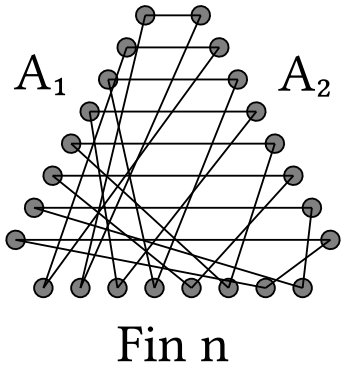


Figure 2.4: A path between inhabitants of $\mathcal{U}_{\mathrm{Fin}}$ contains only triangles

there are only two degrees of freedom. Once the evidence of finiteness is determined for A_1 and A_2, there is only one valid correspondence between them—but there ought to be $n!$ such correspondences.

Proposition 2.4.6. *$\mathcal{U}_{\mathrm{Fin}}$ is a set, that is, for any $X, Y : \mathcal{U}_{\mathrm{Fin}}$, if $p_1, p_2 : X = Y$ then $p_1 = p_2$.*

Proof (sketch). A path $(A_1, n_1, e_1) = (A_2, n_2, e_2)$ is equivalent to $(p : A_1 = A_2) \times (q : n_1 = n_2) \times (q_*(p_*(e_1)) = e_2)$. The transport of e_1 by p is given by the composition $e_1 \circ (\mathsf{ua}^{-1}(p))^{-1}$, but this essentially means that p is uniquely determined by e_1 and e_2. SDG

The underlying problem is that $\mathcal{U}_{\mathrm{Fin}}$ does not actually do a very good job at encoding what classical mathematicians usually mean by "finite set". Saying that a set A is finite with size n does not typically imply there is some specific, chosen bijection $A \xrightarrow{\sim} [n]$, but merely that A *can be* put in bijection with $[n]$, with no mention of a specific bijection. This is justified by the fact that, up to isomorphism, any bijection $A \xrightarrow{\sim} [n]$ is just as good as any other.

This suggests a better encoding of finiteness in type theory.

Definition 2.4.7. The type of finite sets is given by

$$\mathcal{U}_{\|\mathrm{Fin}\|} :\equiv (A : \mathcal{U}) \times \mathsf{isFinite}(A),$$

where

$$\mathsf{isFinite}(A) :\equiv \|(n : \mathbb{N}) \times (A \simeq \mathsf{Fin}\ n)\|\,.$$

Here we make use of propositional truncation to encode the fact that there *merely exists* some size n and an equivalence between A and $\mathsf{Fin}\ n$, but without exposing a precise choice. The finiteness evidence is now irrelevant to paths in $\mathcal{U}_{\|\mathrm{Fin}\|}$, since there is always a path between any two elements of a truncated type.

In an abuse of notation, we will often write $A : \mathcal{U}_{\|\mathrm{Fin}\|}$ instead of $(A, f) : \mathcal{U}_{\|\mathrm{Fin}\|}$ where $f : \mathsf{isFinite}(A)$. We first record a few properties of $\mathcal{U}_{\|\mathrm{Fin}\|}$.

Proposition 2.4.8. *$\mathcal{U}_{\|\mathrm{Fin}\|}$ is a 1-type.*

Proof. We first show that if $(A, f) : \mathcal{U}_{\|\mathrm{Fin}\|}$, then A is a set. Being a set is a mere proposition, so we may use the equivalence $A \simeq \mathsf{Fin}\ n$ hidden inside f. But $\mathsf{Fin}\ n$ is a set, and equivalence preserves set-ness (Lemma 2.4.1), so A is a set as well.

Finally, since $\mathsf{isFinite}(A)$ is a mere proposition, paths between $\mathcal{U}_{\|\mathrm{Fin}\|}$ values are characterized by paths between their underlying types. Since those types must be sets, *i.e.* 0-types, $\mathcal{U}_{\|\mathrm{Fin}\|}$ is consequently a 1-type. ∎

Proposition 2.4.9. *For any type A,*

$$\|(n : \mathbb{N}) \times (A \simeq \mathsf{Fin}\ n)\| \simeq (n : \mathbb{N}) \times \|A \simeq \mathsf{Fin}\ n\|\,.$$

This says that the size n of a finite type may be freely moved in and out of the propositional truncation. Practically, this means we may freely refer to the size of a finite type without worrying about how it is being used (in contrast, the value of the equivalence $A \simeq \mathsf{Fin}\ n$ may only be used in constructing mere propositions). The proof hinges on the fact that $(n : \mathbb{N}) \times \|A \simeq \mathsf{Fin}\ n\|$ is a mere proposition; intuitively, if a type is finite at all, there is only one possible size it can have, so putting n inside the truncation does not really hide anything.

Proof. We must exhibit a pair of inverse functions between the given types. A function from right to left is given by

$$f(n, |e|) = |(n, e)|\,,$$

where pattern matching on $|e| : \|A \simeq \mathsf{Fin}\ n\|$ is shorthand for an application of the recursion principle for propositional truncation. Recall that this recursion principle only applies in the case that the result is a mere proposition; in this case, the result is itself a propositional truncation, which is a mere proposition by construction.

In the other direction, define

$$g(|(n, e)|) = (n, |e|),$$

which is clearly inverse to f. It remains only to show that the implicit use of recursion for propositional truncation is justified, *i.e.* that $(n : \mathbb{N}) \times \|A \simeq \mathsf{Fin}\ n\|$ is a mere proposition.

We must show that any two values $(n_1, e_1), (n_2, e_2) : (n : \mathbb{N}) \times \|A \simeq \mathsf{Fin}\ n\|$ are propositionally equal. Since e_1 and e_2 are mere propositions, it suffices to show that $n_1 = n_2$. This equality is itself a mere proposition (since $\mathbb{N}$ is a set; see §1.3.5), so we may apply the recursion principle for propositional truncation to e_1 and e_2, giving us equivalences $A \simeq \mathsf{Fin}\ n_1$ and $A \simeq \mathsf{Fin}\ n_2$ to work with. By symmetry and transitivity of equivalences, $\mathsf{Fin}\ n_1 \simeq \mathsf{Fin}\ n_2$, and thus $n_1 = n_2$ by Lemma 2.4.3. ∎

Although it is not possible to explicitly extract the equivalence with $\mathsf{Fin}\ n$ from a finite set, it can still be implicitly used for certain purposes, such as deciding the equality of any two elements.

Proposition 2.4.10. *If $(A, f) : \mathcal{U}_{\|\mathrm{Fin}\|}$, then A has decidable equality.*

Proof. Let $x, y : A$; we must show $(x = y) + \neg(x = y)$. We first show that $(x = y) + \neg(x = y)$ is a mere proposition, and then show how to use the equivalence $A \simeq \mathsf{Fin}\ n$ contained in f to construct the desired value.

Since A is a set, $x = y$ is a mere proposition; $\neg(x = y)$ is also a mere proposition since $\neg Q$ is always a mere proposition for any Q. Now let $p, q : (x = y) + \neg(x = y)$, and consider a case analysis on p and q. If one is inl and the other inr, then we can derive $\bot$, and hence $p = q$ since anything follows. If both are inl or both inr, then $p = q$ again, since $x = y$ and $\neg(x = y)$ are both mere propositions. We therefore conclude that $(x = y) + \neg(x = y)$ is itself a mere proposition.

Since we are constructing a mere proposition, we may make use of the equivalence $A \simeq \mathsf{Fin}\ n$ contained in f. In particular, $\mathsf{Fin}\ n$ has decidable equality, which we may transport along the equivalence (using univalence for convenience, although its use here is not strictly necessary) to obtain decidable equality for A. That is, computationally speaking, given $x, y : A$, one may send them across the equivalence to find their corresponding $\mathsf{Fin}\ n$ values, and then decide the equality of those $\mathsf{Fin}\ n$ values. □

Using $\mathcal{U}_{\|\mathrm{Fin}\|}$, we can now finally define a HoTT counterpart to $\mathbf{B}$.

Definition 2.4.11. $\mathcal{B}$ is defined by

$$\mathcal{B} :\equiv \mathcal{G}(\mathcal{U}_{\|\mathrm{Fin}\|}),$$

the groupoid of cardinal-finite sets and paths between them.

Remark. It is worth pointing out that with this definition of $\mathcal{B}$, we have ended up with something akin to the category of specifications $\mathbb{S}_{\#}$ used to define the anafunctor $\# : \mathbf{B} \to \mathbf{P}$ in §2.3, rather than something corresponding directly and naïvely to $\mathbf{B}$ itself. The main difference is that $\mathcal{B}$ uses a propositional truncation to "hide" the explicit choice of finiteness evidence.

2.4.3 Manifestly finite sets and linear orders

We now return to our first attempt at encoding cardinal-finiteness,

$$\mathcal{U}_{\mathrm{Fin}} :\equiv (A : \mathcal{U}) \times (n : \mathbb{N}) \times (A \simeq \mathsf{Fin}\ n).$$

Recall that $\mathcal{U}_{\mathrm{Fin}}$ turned out to be unsuitable as a basis for $\mathbf{B}$ because it has at most one path between any two elements. However, $\mathcal{U}_{\mathrm{Fin}}$ turns out to be quite interesting in its own right; instead of a counterpart to $\mathbf{B}$, it yields a counterpart to $\mathbf{L}$, the category

whose objects are finite sets *equipped with linear orders*, and whose morphisms are *order-preserving* bijections.

For ease of reference, we will call $\mathcal{U}_{\mathrm{Fin}}$ the type of *manifestly finite* sets. The claim is that manifestly finite sets are the same as linearly ordered finite sets. In one direction, the evident linear order on $\mathsf{Fin}\ n$ induces a corresponding linear order on A via transport through the equivalence $A \simeq \mathsf{Fin}\ n$. Conversely, given a linear order on a *finite* set A, we may construct an equivalence with $\mathsf{Fin}\ n$ by matching the smallest element to 0, the second smallest to 1, and so on. More formally:

Proposition 2.4.12. *Manifestly finite sets are equivalent to linear orderings of finite sets, that is,*

$$\mathcal{U}_{\mathrm{Fin}} \simeq (A : \mathcal{U}_{\|\mathrm{Fin}\|}) \times \mathsf{linOrd}(A),$$

where $\mathsf{linOrd}(A)$ *is suitably defined as the (constructive) existence of an antisymmetric, transitive, total binary relation on* A*.*

Proof. As described above, the left-to-right direction is easy: there is a canonical inhabitant of $\mathsf{linOrd}(\mathsf{Fin}\ n)$, which we can turn into an inhabitant of $\mathsf{linOrd}(A)$ via transport.

The right-to-left direction, while not hard to understand intuitively, is more subtle from a constructive point of view. The key observation is that the smallest element of A (according to the given linear order) is uniquely determined, and hence we are justified in "peeking" at the specific isomorphism contained in the propositional truncation in order to construct it (see §1.3.7). By induction on the size n, we can thus enumerate the elements of A in order to find the smallest. We then proceed to recursively construct the isomorphism corresponding to the linear order with the smallest element removed, and then to add back the smallest element, incrementing the indices of the remaining elements. □

Paths between elements of $\mathcal{U}_{\mathrm{Fin}}$ are thus necessarily order-preserving, since they correspond to paths between elements of $(A : \mathcal{U}_{\|\mathrm{Fin}\|}) \times \mathsf{linOrd}(A)$. (Note that this constitutes an alternate proof of the fact that there is at most one path between any two elements of $\mathcal{U}_{\mathrm{Fin}}$.) We can now define a counterpart to $\mathbf{L}$:

Definition 2.4.13. Let $\mathcal{L}$ denote

$$\mathcal{L} :\equiv \mathcal{G}(\mathcal{U}_{\mathrm{Fin}}),$$

the groupoid of manifestly finite—*i.e.* linearly ordered—sets, and (order-preserving) paths between them.

2.4.4 Equivalence of $\mathcal{P}$ and $\mathcal{B}$

Finally, we turn to the equivalence of $\mathcal{P}$ and $\mathcal{B}$, with a goal of defining inverse functors $[-] : \mathcal{P} \to \mathcal{B}$ and $\# : \mathcal{B} \to \mathcal{P}$. We begin with $[-]$.

Definition 2.4.14. The functor $[-] : \mathcal{P} \to \mathcal{B}$ is defined as follows; the essential idea is to send the natural number n to the canonical finite set $\mathsf{Fin}\ n$, and permutations to paths.

- On objects, $[n] :\equiv (\mathsf{Fin}\ n, |(n, id)|)$, where $id : \mathsf{Fin}\ n \simeq \mathsf{Fin}\ n$ witnesses the finiteness of $\mathsf{Fin}\ n$.
- Recall that a morphism $\psi : m \Rightarrow_{\mathcal{P}} n$ is an equivalence $\psi : \mathsf{Fin}\ m \simeq \mathsf{Fin}\ n$. Thus $\mathsf{ua}\ \psi : \mathsf{Fin}\ m = \mathsf{Fin}\ n$, and we define $[\psi] :\equiv u\ (\mathsf{ua}\ \psi) : [m] = [n]$, where u is some function witnessing the fact, mentioned immediately following Definition 2.4.7, that paths in $\mathcal{U}_{\|\mathrm{Fin}\|}$ are characterized by paths between their underlying types.

Before turning to $\# : \mathcal{B} \to \mathcal{P}$, we note the following property of $[-]$:

Lemma 2.4.15. *$[-] : \mathcal{P} \to \mathcal{B}$ is full and faithful.*

Proof. For any $m, n : \mathcal{P}$, we must exhibit an equivalence between $(m \Rightarrow_{\mathcal{P}} n) \equiv (\mathsf{Fin}\ m \simeq \mathsf{Fin}\ n)$ and $([m] \Rightarrow_{\mathcal{B}} [n]) \equiv ([m] = [n]) \simeq (\mathsf{Fin}\ m = \mathsf{Fin}\ n)$. such an equivalence is given by univalence. ∎

On the other hand, it is not at all obvious how to directly define a functor $\# : \mathcal{B} \to \mathcal{P}$. Just as with $\mathbf{B} \to \mathbf{P}$, defining its action on morphisms requires a specific choice of equivalence $A \simeq \mathsf{Fin}\ n$. The objects of $\mathcal{B}$ contain such equivalences, in the proofs of finiteness, but they are propositionally truncated; the type of functors $\mathcal{B} \to \mathcal{P}$ is decidedly not a mere proposition, so it seems the recursion principle for truncation does not apply.

However, all is not lost! We could try porting the concept of anafunctor into HoTT, but it turns out that there is a better way. Recall that in set theory, every fully faithful, essentially surjective functor is an equivalence *if and only if* the axiom of choice holds. In HoTT the situation turns out much better, thanks to the richer notion of equality and the extra axiom associated with a category.

First, there are two relevant notions of essential surjectivity (taken from the HoTT book):

Definition 2.4.16. A functor $F : \mathcal{C} \to \mathcal{D}$ between precategories $\mathcal{C}$ and $\mathcal{D}$ is *split essentially surjective* if for each object $D : \mathcal{D}$ there *constructively exists* an object $C : \mathcal{C}$ such that $F\ C \cong D$. That is,

$$\mathsf{splitEssSurj}(F) :\equiv (D : \mathcal{D}) \to (C : \mathcal{C}) \times (F\ C \cong D).$$

Definition 2.4.17. A functor $F : \mathcal{C} \to \mathcal{D}$ between precategories $\mathcal{C}$ and $\mathcal{D}$ is *essentially surjective* if for each object $D : \mathcal{D}$ there *merely exists* an object $C : \mathcal{C}$ such that $F\ C \cong D$. That is,

$$\mathsf{essSurj}(F) :\equiv (D : \mathcal{D}) \to \|(C : \mathcal{C}) \times (F\ C \cong D)\|.$$

It turns out that being split essentially surjective is a rather strong notion. In particular:

Proposition 2.4.18. *For any precategories $\mathcal{C}$ and $\mathcal{D}$ and functor $F : \mathcal{C} \to \mathcal{D}$, F is fully faithful and split essentially surjective if and only if it is an equivalence.*

Proof. See the HoTT book [2013, Lemma 9.4.5]. Intuitively, the *split* essential surjectivity gives us exactly what we need to unambiguously *construct* an inverse functor $G : \mathcal{D} \to \mathcal{C}$: the action of G on $D : \mathcal{D}$ is defined to be the C—which exists constructively—such that $F\ C \cong D$. ∎

That is, a fully faithful, essentially surjective functor is an equivalence given AC; a fully faithful, *split* essentially surjective functor is an equivalence even without AC.

Now, what about $[-] : \mathcal{P} \to \mathcal{B}$? We have the following:

Proposition 2.4.19. $[-]$ *is essentially surjective.*

Proof. Given $(S, f) : \mathcal{B}$, we must show that there merely exists some $n : \mathcal{P}$ such that $[n] \cong S$—but this is precisely the content of the isFinite proof f. ∎

On the other hand, it would seem that $[-]$ is not split essentially surjective, since that would require extracting finiteness proofs from the propositional truncation, which is not allowed in general. However:

Proposition 2.4.20 (HoTT book, Lemma 9.4.7). *If $F : \mathcal{C} \to \mathcal{D}$ is fully faithful and $\mathcal{C}$ is a category, then for any $D : \mathcal{D}$ the type $(C : \mathcal{C}) \times (F\ C \cong D)$ is a mere proposition.*

Proof (sketch). From $F\ C \cong D$ and $F\ C' \cong D$ we derive $F\ C \cong F\ C'$, and thus $C \cong C'$ (since F is fully faithful), and $C = C'$ (since $\mathcal{C}$ is a category). The transport of the isomorphism $(F\ C \cong D)$ along this derived path $C = C'$ is precisely the isomorphism $(F\ C' \cong D)$. ∎

Intuitively, for a fully faithful functor $F : \mathcal{C} \to \mathcal{D}$ out of a category $\mathcal{C}$, there is "only one way" for some object $D : \mathcal{D}$ to be isomorphic to the image of an object of $\mathcal{C}$. That is, if it is isomorphic to the image of multiple objects of $\mathcal{C}$, then those objects must in fact be equal.

This brings us to the punchline:

Corollary 2.4.21. *If $\mathcal{C}$ is a category, a fully faithful functor $F : \mathcal{C} \to \mathcal{D}$ is essentially surjective if and only if it is split essentially surjective.*

Corollary 2.4.22. *Since $[-]$ is a fully faithful and essentially surjective functor out of a category, it is in fact* split *essentially surjective and thus an equivalence. In particular, it has an inverse (up to natural isomorphism) which we call $\# : \mathcal{B} \to \mathcal{P}$, and thus*

$$\# : \mathcal{B} \cong \mathcal{P} : [-].$$

As a final remark, note that this is at root an instance of the "trick" explained at the end of §1.3.7, whereby a function $\|A\| \to B$ may be defined, even if B is not a mere proposition, as long as the value of B produced can be uniquely characterized. Computationally speaking, $\# : \mathcal{B} \to \mathcal{P}$ does precisely what we thought was not allowed—its action on morphisms works by extracting concrete equivalences out of the finiteness proofs in the objects of $\mathcal{B}$ and using them to construct the required permutation, just as in the construction of the anafunctor $\# : \mathbf{B} \to \mathbf{P}$ in §2.3. Indeed, we are not allowed to project finiteness evidence out from the propositional truncation when defining *arbitrary* functors $\mathcal{B} \to \mathcal{P}$. However, we are not interested in constructing any old functor, but rather a very specific one, namely, an inverse to $[-] : \mathcal{P} \to \mathcal{B}$—and the inverse is uniquely determined. In essence, the construction of # proceeds by first constructing a functor paired with a proof that, together with $[-]$, it forms an equivalence—altogether a mere proposition—and then projecting out the functor.

2.5 Conclusion

In this chapter we have seen that encoding category theory and the notion of finiteness in a constructive type theory is more subtle than one might expect. The difficulty with both—as often in type theory—can be traced back to the foundational notion of equality. Homotopy type theory, with its richer notion of equality, gives us exactly the framework we need in which to encode category theory without the use of the axiom of choice. Via propositional truncation, we can also constructively encode the notion of finiteness such that we can "have our cake and eat it too"—with concrete, computationally relevant isomorphisms witnessing finiteness that nonetheless do not "leak" information inappropriately. The close connection between the encodings of **B** and **L**, when viewed via HoTT, came as a surprise, and yields new insight into equipotence as well as **L**-species (discussed in §5.5). In fact, the work on homotopy type theory itself came as a fortuitous surprise—I had been grappling with some of the questions in this chapter, off and on, for several years before the publication of the HoTT book.

The next chapter explains the formal definition of species, and puts together the tools developed in this chapter into an encoding of species within homotopy type theory.

Chapter 3

Combinatorial species

The theory of *combinatorial species*, introduced by Joyal [1981], is a unified, algebraic theory of *combinatorial structures* or *shapes*. The algebraic nature of species is of particular interest in the context of data structures and will be explored in depth in this chapter. The theory can also be seen as a *categorification* of the theory of *generating functions*.

The present chapter begins in §3.1 by presenting an intuitive sense for species along with a collection of examples. §3.2 presents Joyal's formal definition of species and related definitions in set theory, along with more commentary and intuition. The same section also discusses an encoding of species within homotopy type theory (§3.2.4), and the benefits of such an encoding. As a close follow-up to the formal definition, §3.3 presents two equivalence relations on species, *isomorphism* and *equipotence*, and in particular sheds some new light on equipotence via the encoding of species in HoTT. Finally, §3.4 introduces *generating functions*, which are in some sense the point of origin for the entire theory.

3.1 Intuition and examples

In the process of generalizing the theory of generating functions, one of Joyal's great insights in formulating the theory of species was to take the notion of *labelled* structures as fundamental, and to build other notions (such as *unlabelled* structures) on top of it. Species fundamentally describe labelled objects; for example, Figure 3.1 shows two representative examples, a labelled tree and a labelled "octopus". In these examples the integers $\{0, \ldots, 7\}$ are used as labels, but in general, labels can be drawn from any set.

Why *labelled* shapes? In the tree shown in Figure 3.1, one can uniquely identify each location in the tree by a path from the root, without referencing labels at all. However, the "octopus" illustrates one reason labels are needed. The particular way it is drawn is intended to indicate that the structure has fourfold rotational symmetry, which means there would be no way to uniquely refer to any location except by label.

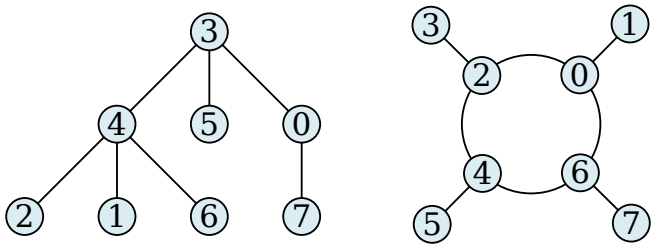


Figure 3.1: Representative labelled shapes

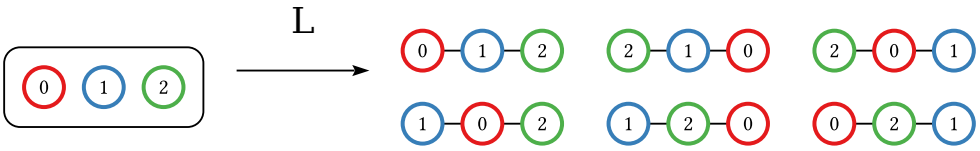


Figure 3.2: The species L of lists

More abstractly, *unlabelled* shapes can be defined as equivalence classes of labelled shapes (§3.3.2), which is nontrivial in the case of shapes with symmetry.

Besides its focus on labels, the power of the theory of species also derives from its ability to describe structures of interest *algebraically*, making them amenable to analysis with only a small set of general tools.

Example. Consider the species L of *lists*, or *linear orderings*; Figure 3.2 illustrates all the labelled list structures (containing each label exactly once) on the set of labels $[3] = \{0, 1, 2\}$. Of course, there are exactly $n!$ such list structures on any set of n labels.

The species of lists can be described by the recursive algebraic expression

$$\mathsf{L} = 1 + \mathsf{X} \cdot \mathsf{L}.$$

The meaning of this will be made precise later. For now, its intuitive meaning should be clear to anyone familiar with recursive algebraic data types in a language such as Haskell or OCaml: a labelled list (L) is empty (1) or ($+$) a single label (X) together with ($\cdot$) another labelled list (L).

Example. As another example, consider the species B of *(rooted, ordered) binary trees*. The set of all labelled binary trees on $\{0, 1, 2\}$ is shown in Figure 3.3.

Algebraically, such trees can be described by

$$\mathsf{B} = 1 + \mathsf{B} \cdot \mathsf{X} \cdot \mathsf{B}.$$

Example. The species E of *sets* describes shapes consisting simply of an unordered collection of unique labels, with no other structure imposed. There is exactly one such shape for any set of labels, as illustrated in Figure 3.4.

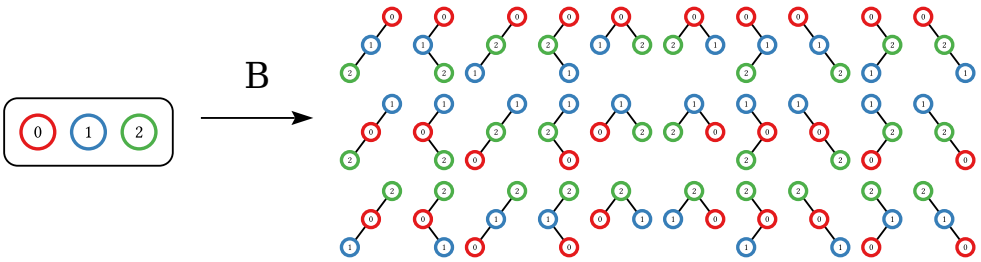


Figure 3.3: The species B of binary trees

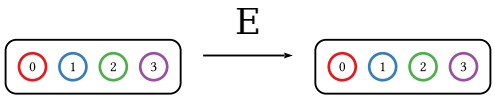


Figure 3.4: The species E of sets

Example. The species Mob of *mobiles* consists of non-empty binary trees where each node has exactly zero or two subtrees, and sibling subtrees are considered unordered. Figure 3.5 shows a single example Mob-shape, drawn in four (equivalent) ways. Algebraically,

$$\mathsf{Mob} = \mathsf{X} + \mathsf{X} \cdot (\mathsf{E}_2 \circ \mathsf{Mob}),$$

that is, a mobile is either a single label, or a label together with an unordered pair (E_2) of ($\circ$) mobiles.

Example. The species C of *cycles*, illustrated in Figure 3.6, describes shapes that consist of an ordered cycle of labels. One way to think of the species of cycles is as a quotient of the species of lists, where two lists are considered equivalent if one is a cyclic rotation of the other (see §4.6).

Example. The species S of *permutations*—*i.e.* bijective endofunctions—is illustrated in Figure 3.7. Algebraically, it can be described by

$$\mathsf{S} = \mathsf{E} \circ \mathsf{C},$$

that is, a permutation is a set of cycles.

Example. The species End of *endofunctions* consists of directed graphs corresponding to valid endofunctions on the labels—that is, where every label has exactly one outgoing edge (Figure 3.8).

Some reflection shows that endofunctions can be characterized as permutations of rooted trees,

$$\mathsf{End} = \mathsf{S} \circ T = \mathsf{E} \circ \mathsf{C} \circ T,$$

where $T = \mathsf{X} \cdot (\mathsf{E} \circ T)$. Each element which is part of a cycle serves as the root of a tree; iterating an endofunction starting from any element must eventually reach a

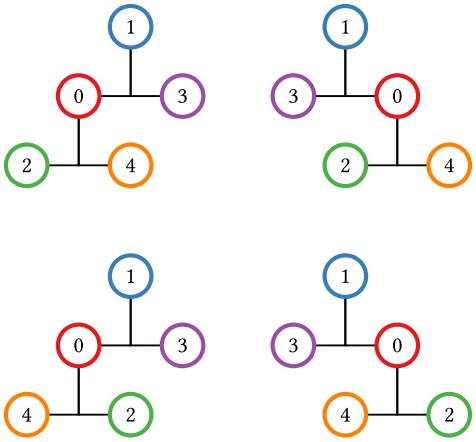


Figure 3.5: An example Mob-shape, drawn in four equivalent ways

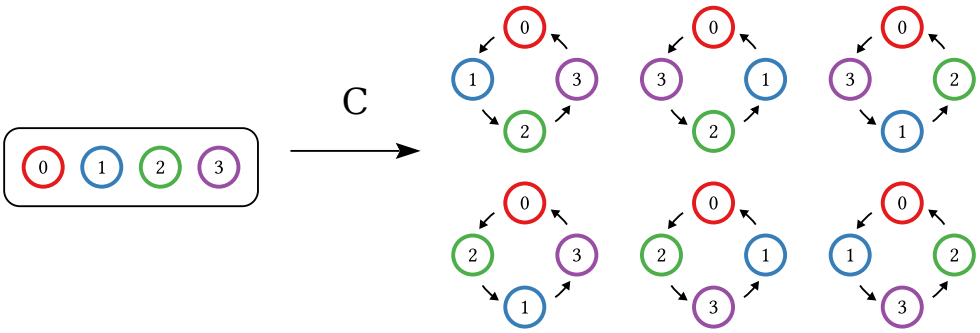


Figure 3.6: The species C of cycles

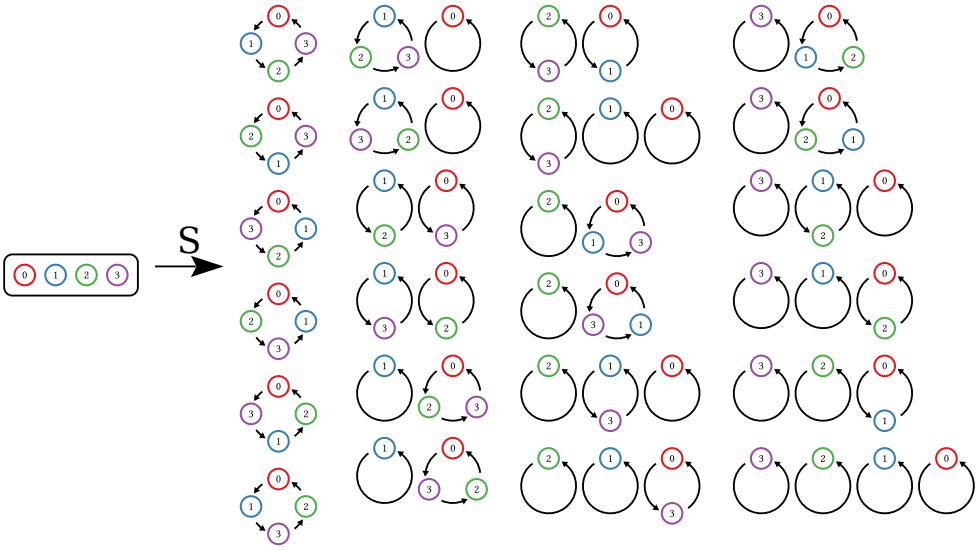


Figure 3.7: The species S of permutations

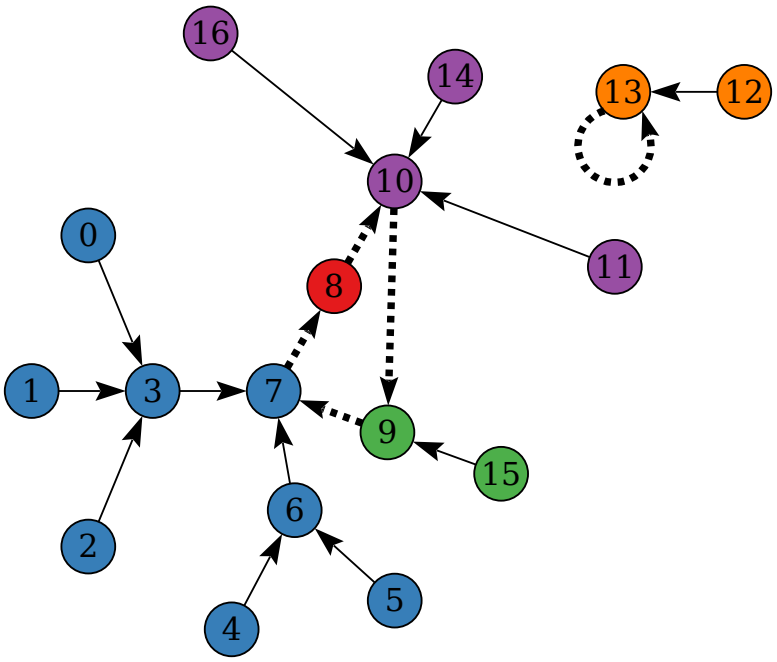


Figure 3.8: An example End-shape

cycle, so every element belongs to some tree. Figure 3.8 illustrates this by highlighting each tree in a different color. The large component contains a central cycle of four elements, each a different color, with a tree hanging off of each; the small component consists of just a single tree with a self-loop at its root.

Joyal [1981] makes use of this characterization in giving an elegant combinatorial proof of Cayley's formula, namely, that there are n^{n-2} labelled trees (in the graph-theoretic sense) of size n. One can likewise give characterizations of the species of endofunctions with various special properties, such as injections, surjections, and involutions.

In a computational context, it is important to keep in mind the distinction between *labels* and *data*, or more generally between *labelled shapes* and *(labelled) data structures*. Labels are merely names for locations where data can be stored, and (typically) have no particular computational significance beyond the ability to compare them for equality. Data structures contain data associated with each label, whereas labelled shapes have no data, only labels. Put more intuitively, species shapes are "form without content". As a concrete example, the numbers in Figure 3.1 are not data being stored in the structures, but merely labels for the locations. To talk about a data structure, one must additionally specify a mapping from labels to data; this will be made precise in Chapter 6.

Such a distinction is also important when considering the semantics of imperative languages. See, for example, Dowek [2009, §1.3.2], who decomposes states mapping variables to values into pairs of a mapping from variables to *references* (*i.e.* labels) and a mapping from references to values.

3.2 Definitions

Informally, as we have seen, a species is a family of labelled shapes. Crucially, the actual labels used shouldn't matter: for example, we should get the "same" family of binary trees no matter what labels we want to use. This intuition is made precise in the formal definition of combinatorial species as *functors*. In fact, one of the reasons Joyal's work was so groundbreaking was that it brought category theory to bear on combinatorics, showing that many specific combinatorial insights could be modeled abstractly using the language of categories.

3.2.1 Species as functors

Definition 3.2.1 (Species [Joyal, 1981]). A *species* is a functor $F : \mathbf{B} \to \mathbf{Set}$.

Recall that $\mathbf{B}$ is the groupoid of finite sets whose morphisms are bijections, and $\mathbf{Set}$ is the category of sets and (total) functions.

It is worth spelling out this definition in more detail, which will also give an opportunity to explain some intuition and terminology. Even for those who are very

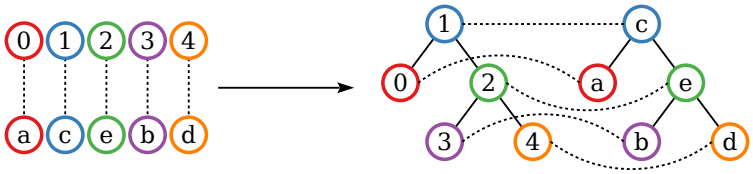


Figure 3.9: Relabelling

comfortable with category theory, it may be hard to grasp the intuition for the abstract definition right away.

Definition 3.2.2. A *species* F is a pair of mappings which

- sends any finite set L (of *labels*) to a set $F\ L$ (of *shapes*), and
- sends any bijection on finite sets $\sigma : L \xrightarrow{\sim} L'$ (a *relabelling*) to a function $F\ \sigma : F\ L \to F\ L'$ (illustrated in Figure 3.9),

satisfying the following functoriality conditions:

- $F\ id_L = id_{F\,L}$, and
- $F\ (\sigma \circ \tau) = F\ \sigma \circ F\ \tau$.

We call $F\ L$ the set of "F-shapes with labels drawn from L", or simply "F-shapes on L", or even (when L is clear from context) just "F-shapes".[1] $F\ \sigma$ is called the "transport of σ along F", or sometimes the "relabelling of F-shapes by σ".

The functoriality of a species F means that the actual labels used don't matter; the resulting family of shapes is independent of the particular labels used. We might say that species are *parametric* in label sets of a given size. In particular, F's action on all label sets of size n is determined by its action on any particular such set: if $|L_1| = |L_2|$ and we know $F\ L_1$, we can determine $F\ L_2$ by lifting an arbitrary bijection between L_1 and L_2. More formally, although Definitions 3.2.1 and 3.2.2 say only that a species F sends a bijection $\sigma : L \xrightarrow{\sim} L'$ to a *function* $F\ \sigma : F\ L \to F\ L'$, the functoriality of F guarantees that $F\ \sigma$ is a bijection as well. In particular, $(F\ \sigma)^{-1} = F\ (\sigma^{-1})$, since $F\ \sigma \circ F\ (\sigma^{-1}) = F\ (\sigma \circ \sigma^{-1}) = F\ id = id$, and similarly $F\ (\sigma^{-1}) \circ F\ \sigma = id$. Thus, *up to isomorphism*, a functor F must do the same thing for any two label sets of the same size.

We may therefore take the finite set of natural numbers $[n] = \{0, \ldots, n-1\}$ as *the* canonical label set of size n, and write $F\ n$ (instead of $F\ [n]$) for the set of F-shapes

[1] Margaret Readdy's translation of Bergeron et al. [1998] uses the word "structure" instead of "shape", but that word is likely to remind computer scientists of "data structures", which is, again, the wrong association: data structures contain *data*, whereas species shapes contain only labels. I try to consistently use the word "shape" to refer to the elements of a species, and reserve "structure" for the labelled data structures to be introduced in Chapter 6.

built from this set. In fact, since $\mathbf{B}$ and $\mathbf{P}$ are equivalent, we may formally take the definition of a species to be a functor $\mathbf{P} \to \mathbf{Set}$ (or an anafunctor, if we wish to avoid AC; see §2.3), which amounts to the same thing.

Remark. Typically, the sets of shapes $F\ L$ are required to be *finite*, that is, species are defined as functors $\mathbf{B} \to \mathbf{FinSet}$ into the category of *finite* sets. Of course, this is important if the goal is to *count* things! However, nothing in the present work hinges on this restriction, so it is simpler to drop it.

It should be noted, however, that requiring finiteness in this way would be no great restriction: requiring each *particular* set of shapes $F\ L$ to be finite is not at all the same thing as requiring the *entire family* of shapes, $\uplus_{n \in \mathbb{N}} F\ n$, to be finite. Typically, even in the cases that programmers care about, each individual $F\ n$ is finite but the entire family is not—that is, a type may have infinitely many inhabitants but only finitely many of a given size.

Remark. In my experience, computer scientists tend to have a bit of trouble with these definitions, because their first instinct is to think of a functor $\mathbf{B} \to \mathbf{Set}$ from a *computational* point of view: *i.e.* a species $F : \mathbf{B} \to \mathbf{Set}$, given some set of labels $L \in \mathbf{B}$, *computes* some family of shapes having those labels.

However, I find this intuition unhelpful, since it places too much emphasis on analyzing the "input" set of labels, making case distinctions on the size of the set, and so on. Instead of thinking of functors $\mathbf{B} \to \mathbf{Set}$ as computational, it is better to think of them as *descriptive*. We begin with some entire family of labelled shapes, and want to classify them according to the labels that they use. A functor $\mathbf{B} \to \mathbf{Set}$ is then a convenient technical device for organizing such a classification: it describes a family of labelled shapes *indexed by* their labels.

Given this shift in emphasis, one might think it more natural to define a set of labelled shapes along with a function mapping shapes to the set of labels contained in them (indeed, down this path lie the notions of *containers* [Abbott et al., 2003a, 2004, 2005, Morris and Altenkirch, 2009] and *stuff types* [Baez and Dolan, 2000, Byrne, 2006]). Species can be seen as roughly dual to these shapes-to-labels mappings, giving the *fiber* of each label set. This is parallel to the equivalence between the functor category $\mathbf{Set}^{\mathbb{N}}$ and the slice category $\mathbf{Set}/\mathbb{N}$ (see the discussion under functor categories in §1.4.1). However, since $\mathbf{B}$ is not discrete, there is not an equivalence between $\mathbf{Set}^{\mathbf{B}}$ and $\mathbf{Set}/\mathbf{B}$; this seems to account for the fact that species and containers (and, more generally, operads and stuff types/clubs [Kelly, 2005, p. 2]) seem so closely related but without a simply expressible relationship.

Remark. Historically, Joyal's first paper [1981] defined species as endofunctors $\mathbf{B} \to \mathbf{B}$. Given a restriction to finite families of shapes, and the observation that functors preserve isomorphisms, this is essentially equivalent to $\mathbf{B} \to \mathbf{FinSet}$, which is the definition used in Joyal's second paper [1986] as well as, later, by Bergeron et al. [1998]. It can be argued, however, that this second formulation is more natural, especially

when one wishes to make the connection to functors **FinSet** $\to$ **FinSet** (or **Set** $\to$ **Set**); see Chapter 6.

3.2.2 Cardinality restriction

For any species F and natural number n, we may define

$$F_n\ L := \begin{cases} F\ L & \text{if } \#L = n \\ \varnothing & \text{otherwise} \end{cases}.$$

That is, F_n is the restriction of F to label sets of size exactly n. For example, E is the species of sets of any size; E_4 is the species of sets of size 4. This is well defined since the action of a species is determined independently on label sets of each size. More abstractly, as noted previously, **B** (and **P**) are disconnected categories, so functors out of them are equivalent to a disjoint union of individual functors out of each connected component; replacing the component functors at individual sizes will always result in another valid overall functor.

More generally, we can "kill" any subset of sizes using arbitrary predicates. For example, $F_{\leqslant n}$ is the species of F-shapes of size n or less; similarly, $F_{\geqslant n}$ is the species of F-shapes of size n or greater. We also write F_+ as a shorthand, and say "nonempty F", for $F_{\geqslant 1}$, the species F restricted to nonempty sets of labels.

3.2.3 The category of species

Recall that $\mathbb{C} \Rightarrow \mathbb{D}$ denotes the *functor category* whose objects are functors and whose morphisms are natural transformations between functors. We may thus consider the *category of species*, **Spe** := (**B** $\Rightarrow$ **Set**), where the objects are species, and morphisms between species are label-preserving mappings which commute with relabelling—that is, mappings which are entirely "structural" and do not depend on the labels in any way. For example, an in-order traversal constitutes such a mapping from the species of binary trees to the species of lists, as illustrated in Figure 3.10: computing an in-order traversal and then relabelling yields the same list as first relabelling and then doing the traversal.

It turns out that functor categories have a lot of interesting structure. For example, as we will see, **B** $\Rightarrow$ **Set** has (at least) six different monoidal structures! Much of Chapter 4 is dedicated to exploring and generalizing this structure.

3.2.4 Species in HoTT

We now turn to porting the category of species from set theory into HoTT. Recall that $\mathcal{B}$ denotes the h-groupoid with objects

$$\mathcal{U}_{\|\mathrm{Fin}\|} := (A : \mathcal{U}) \times \mathsf{isFinite}(A),$$

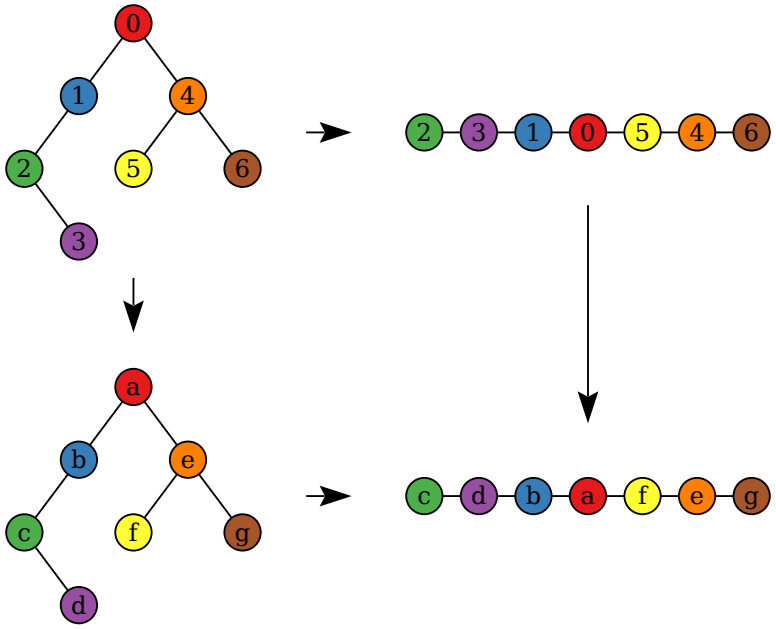


Figure 3.10: Inorder traversal is natural

where

$$\mathsf{isFinite}(A) := \|(n : \mathbb{N}) \times (A \simeq \mathsf{Fin}\ n)\|$$

and with morphisms given by paths.

Definition 3.2.3. A *constructive species*, or h-species, is an h-functor $F : \mathcal{B} \to \mathcal{S}$. We use $\mathbf{Spe} = \mathcal{B} \Rightarrow \mathcal{S}$ to refer to the h-category of constructive species, the same name as the category $\mathbf{B} \Rightarrow \mathbf{Set}$ of set-theoretic species; while technically ambiguous, this should not cause confusion since it should always be clear from the context whether we are working in set theory or in HoTT. Likewise, when working in the context of HoTT we will often simply say "species" instead of "constructive species".

The above definition corresponds directly to the definition of species in set theory. However, it is more specific than necessary. In fact, in HoTT, *any* function of type $\mathcal{B} \to \mathcal{S}$ (that is, a function from objects of $\mathcal{B}$ to objects of $\mathcal{S}$) is automatically an h-functor. Since the morphisms in $\mathcal{B}$ are just paths, functoriality corresponds to transport. Thus, as hinted in Chapter 1, within HoTT it is simply impossible to write down an invalid species. This is a strong argument for working within type theory in general and HoTT in particular: it provides exactly the right sort of type system which allows expressing only valid species.

Nevertheless, it is still not perfectly clear whether this is the right encoding of species within homotopy type theory. It cannot be directly justified by showing that $\mathbf{B} \Rightarrow \mathbf{Set}$ and $\mathcal{B} \Rightarrow \mathcal{S}$ are categorically equivalent; this does not even make sense since they live in entirely different foundational frameworks. Rather, a justification

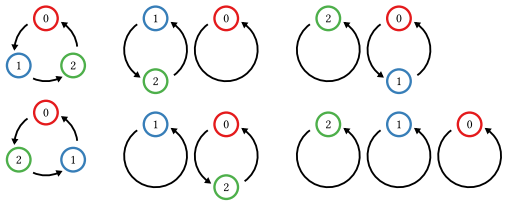


Figure 3.11: Permutations of size three

must be extensional, in the sense of showing that the two definitions have similar properties and support similar operations. In a sense, much of Chapter 4 is precisely such an extensional justification.

3.3 Isomorphism and equipotence

Just as with HoTT itself, *sameness* and related notions are also at the heart of the theory of species. In this section we explore isomorphism of species and of species shapes, as well as a coarser notion of equivalence on species known as *equipotence*.

3.3.1 Species isomorphism

An isomorphism of species is just an isomorphism in the category of species, that is, a pair of inverse natural transformations. Species isomorphism preserves all the interesting *combinatorial* properties of species; hence in the combinatorics literature everything is always done up to isomorphism. However, this is usually done in a way that glosses over the *computational* properties of the isomorphisms. Formulating species within HoTT gives us the best of both worlds: naturally isomorphic functors between h-categories are equal, and hence isomorphic species are literally identified; however, equalities (*i.e.* paths) in HoTT may still have computational content.

3.3.2 Shape isomorphism and unlabelled species

In addition to isomorphism of entire species, there is also a natural notion of isomorphism for individual species *shapes*. For example, consider the set of permutations on the labels $\{0, 1, 2\}$, shown in Figure 3.11. Notice that some of these permutations "have the same form". For example, the only difference between the two permutations shown in Figure 3.12 is their differing labels. On the other hand, the two permutations shown in Figure 3.13 are fundamentally different, in the sense that there is no way to merely *relabel* one to get the other.

We can formalize this idea as follows.

Definition 3.3.1. Given a species F and F-shapes $f_1 : F\ L_1$ and $f_2 : F\ L_2$, we say f_1 and f_2 are *equivalent up to relabelling*, or *have the same form*, and write $f_1 \approx f_2$, if

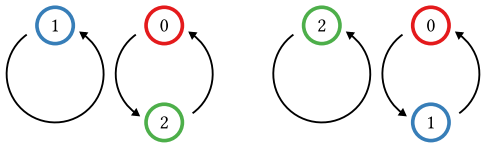


Figure 3.12: Two permutations with the same form

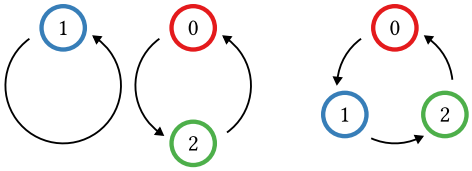


Figure 3.13: Two permutations with different forms

there is some bijection $\sigma : L_1 \xrightarrow{\sim} L_2$ such that $F\ \sigma\ f_1 = f_2$. If we wish to emphasize the particular bijection relating f_1 and f_2 we may write $f_1 \approx_\sigma f_2$.

Thus, the two labelled shapes shown in Figure 3.12 are related by $\approx$, whereas those shown in Figure 3.13 are not.

Definition 3.3.2. Given a species F, denote by $\mathrm{sh}(F)$ the groupoid whose objects are F-shapes—that is, finite sets L together with an element of $F\ L$—and whose morphisms are given by the $\approx$ relation.

Proof. We need to show this is a well-defined groupoid, *i.e.* that $\approx$ is an equivalence relation. The $\approx$ relation is reflexive, yielding identity morphisms, since any shape is related to itself by the identity bijection. If $f \approx g \approx h$ then $f \approx h$ by composing the underlying bijections. Finally, $f \approx g$ implies $g \approx f$ since the underlying bijections are invertible. SDG

Given these preliminary definitions, we can now define what we mean by a *form*, or *unlabelled shape*.

Definition 3.3.3. An *F-form* is an equivalence class under $\approx$, that is, a connected component of the groupoid $\mathrm{sh}(F)$.

In other words, an F-form is a maximal class of labelled F-shapes which are all interconvertible by relabelling, that is, a maximal clique. As defined, such classes are rather large, as they include labellings by *all possible* sets of labels! Typically, we consider only a single label set of each size, such as $\mathsf{Fin}\ n$. For example, Figure 3.14 shows all the S-forms of size four, using two different representations: on the right

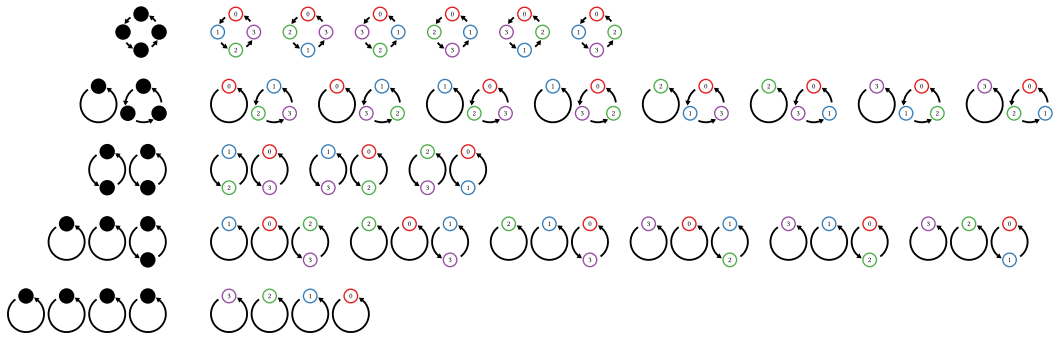


Figure 3.14: S-forms of size 4

are the literal equivalence classes of permutations on Fin 4 which are equivalent up to relabelling. On the left are schematic representations of each form, drawn by replacing labels with indistinguishable dots. Note that the schematic representations, while convenient, can break down in more complex situations, so it is important to also keep in mind the underlying definition in terms of equivalence classes.

Remark. What are here called *forms* are more often called *types* in the species literature; but using that term would lead to unnecessary confusion in the present context.

3.3.3 Equipotence

It turns out that there is another useful equivalence relation on species which is *weaker* (*i.e.* coarser) than isomorphism/equality, known as *equipotence*.

Definition 3.3.4. An *equipotence* between species F and G, denoted $F \overset{\#}{=} G$,[2] is defined as an "unnatural" isomorphism between F and G—that is, two families of functions $\varphi_L : F\ L \to G\ L$ and $\psi_L : G\ L \to F\ L$ such that $\varphi_L \circ \psi_L = \psi_L \circ \varphi_L = id$ for every finite set L. Note in particular there is no requirement that φ or ψ be natural.

We can see that an equipotence preserves the *number* of shapes of each size, since φ and ψ constitute a bijection, for each label set L, between the set of F-shapes $F\ L$ and the set of G-shapes $G\ L$. Isomorphic species are of course equipotent, where the equipotence also happens to be natural. It may be initially surprising, however, that the converse is false: there exist equipotent species which are not isomorphic. Put another way, having the same number of structures of each size is not enough to ensure isomorphism.

One good example is the species L of lists and the species S of permutations. As is well-known, there are the same number of linear orderings of n labels as there are permutations of n labels (namely, $n!$). In fact, this is so well-known that mathematicians

[2] In the species literature, equipotence is usually denoted $F \equiv G$, but we are already using that symbol to denote judgmental equality.

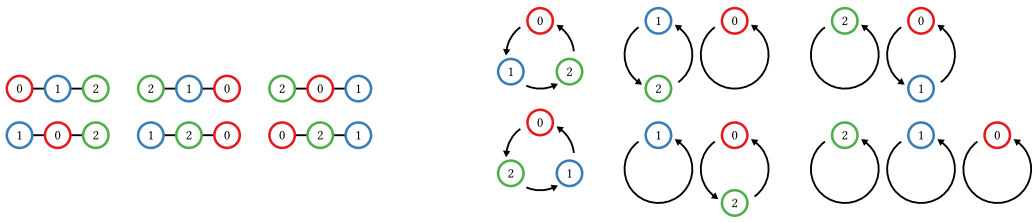


Figure 3.15: Lists and permutations on three labels

routinely conflate the two, referring to an ordered list as a "permutation". Figure 3.15 shows the six lists and six permutations on three labels.

However, L and S are not isomorphic. The intuitive way to see this is to note that although there is only a single list form of any given size, for $n \geqslant 2$ there are multiple permutation forms. Every permutation, *i.e.* bijective endofunction, can be decomposed into a set of cycles, and a relabelling can only map between permutations with the same number of cycles of the same sizes. There is thus one S-form corresponding to each integer partition of n (Figure 3.14 shows the five permutation forms of size 4, corresponding to $4 = 3+1 = 2+2 = 2+1+1 = 1+1+1+1$).

More formally, suppose there were some *natural* isomorphism witnessed by $\varphi : \mathsf{L} \xrightarrow{\bullet} \mathsf{S}$ and $\psi : \mathsf{S} \xrightarrow{\bullet} \mathsf{L}$. In particular, for any $\sigma : K \xrightarrow{\sim} K$ we would then have

$$\begin{array}{ccc} \mathsf{L}\ K & \xrightarrow{\varphi_K} & \mathsf{S}\ K \\ {\scriptstyle \mathsf{L}\ \sigma}\downarrow & & \downarrow{\scriptstyle \mathsf{S}\ \sigma} \\ \mathsf{L}\ K & \xrightarrow[\varphi_K]{} & \mathsf{S}\ K \end{array}$$

and similarly for ψ_K in the opposite direction. This says that any two K-labelled lists related by the relabelling σ correspond to permutations which are also related by σ. However, as we have seen, *any* two lists are related by some relabelling, and thus (since φ and ψ constitute a bijection) any two permutations would have to be related by some relabelling as well, but this is false.

This argument shows that there cannot exist a natural isomorphism between L and S. However, the claim is that they are nonetheless equipotent. Again, this fact is very well known, but it is still instructive to work out the details of a formal proof.

The first and most obvious "proof" is to send the permutation $\sigma : (\mathsf{Fin}\ n)!$ to the list whose ith element is $\sigma(i)$, and vice versa. Note, however, that this is not really a proof, since it only gives us a specific bijection $\mathsf{L}\ (\mathsf{Fin}\ n) \xrightarrow{\sim} \mathsf{S}\ (\mathsf{Fin}\ n)$, rather than a family of bijections $\mathsf{L}\ K \xrightarrow{\sim} \mathsf{S}\ K$. We will return to this point shortly.

The second proof, known as the *fundamental transform*, is more elegant from a combinatorial point of view. For more details, see Cartier and Foata [1969], Knuth [1973], or Bergeron et al. [1998, p. 22]. We first describe the mapping from permutations on $\mathsf{Fin}\ n$ to lists on $\mathsf{Fin}\ n$: given a permutation, order its cycles in decreasing order

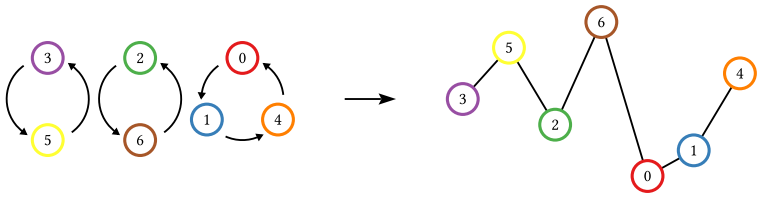


Figure 3.16: The fundamental transform

of their smallest element, and then transcribe each cycle as a list beginning with the smallest element. Figure 3.16 shows an example where the permutation (35)(26)(014) (whose cycles have minimum elements 3, 2, and 0 respectively) is sent to the list 3526014, which for emphasis is drawn with the height of each node corresponding to the size of its label. To invert the transformation, partition a list into segments with each record minimum beginning a new segment, and turn each such segment into a cycle. For example, in the list 3526014, the elements 3, 2, and 0 are the ones which are smaller than all the elements to their left, so each one marks off the beginning of a new cycle.

The way the fundamental transform is presented also makes it clear how to generalize from $\mathsf{Fin}\ n$ to other finite sets of labels L: all we require is a linear order on L, in order to find the minimum label in a given cycle and sort the cycles by minimum element, and to determine the successive record minima in a list. Looking back at the first, "obvious" proof, which sends σ to the list whose ith element is $\sigma(i)$, we can see that it also can be generalized to work for any finite set L equipped with a linear order. In particular, being equipped with a linear order is equivalent to being equipped with a bijection to $\mathsf{Fin}\ n$, as explained in §2.4.3.

Intuitively, then, the reason that these two families of bijections are not natural is that they do not work *uniformly* for all sets of labels, but require some extra structure. Any finite label set can be given a linear order, but the precise choice of linear order determines how the bijections work.

Considering this from the viewpoint of HoTT yields additional insight. A family of functions like φ_K would typically correspond in HoTT to a function of type

$$\varphi : (K : \mathcal{U}_{\|\mathrm{Fin}\|}) \to \mathsf{L}\ K \to \mathsf{S}\ K.$$

It is certainly possible to implement a function with the above type (for example, one which sends each list to the cyclic permutation with elements in the same order), but as we have seen, it is not possible to implement one which is invertible. Writing an invertible such function also requires a linear ordering on the type K. We could, of course, simply take a linear order as an extra argument,

$$\varphi : (K : \mathcal{U}_{\|\mathrm{Fin}\|}) \to \mathsf{LinOrd}\ K \to \mathsf{L}\ K \to \mathsf{S}\ K.$$

Alternatively, recall that K contains evidence of its finiteness in the form of an equivalence $K \simeq \mathsf{Fin}\, n$. This equivalence induces a linear ordering on K, corresponding to the natural linear ordering $0 < 1 < 2 < \ldots$ on $\mathsf{Fin}\ n$. In other words, each finite set K already comes equipped with a linear ordering! However, recall that the finiteness evidence is sealed inside a propositional truncation, so we cannot use it in implementing a function of type $(K : \mathcal{U}_{\|\mathrm{Fin}\|}) \to \mathsf{L}\ L \to \mathsf{S}\ K$. If we could, the resulting function would indeed *not* be natural, and it is instructive to see why. A path $K = K$ corresponds to a permutation on K, but *does not have to update the finiteness evidence in conjunction* with the permutation. Thinking of the finiteness evidence as giving a linear order on K, another way to say this is that permutations $K = K$ need not be order-preserving. Naturality is not satisfied, therefore, since applying the fundamental transform directly may give results completely incompatible with those obtained by applying a non-order-preserving permutation followed by the fundamental transform.

Bergeron et al. [1998, p. 22] note that the fundamental transform *is* in fact compatible with *order-preserving* bijections. If we consider functors $\mathbf{L} \to \mathbf{Set}$, where $\mathbf{L}$ is the groupoid of finite sets equipped with linear orders, along with order-preserving bijections, then the fundamental transform is indeed a natural isomorphism between L and S. Such functors are called $\mathbf{L}$-species, and are discussed further in §5.5.

Back in $\mathcal{U}_{\|\mathrm{Fin}\|}$, however, in order to use the linear order associated to each finite set K, we must produce a mere proposition. We cannot directly produce an equivalence—but we certainly can produce the propositional truncation of one. In particular we can encode the fundamental transform as a function of type

$$\chi : (K : \mathcal{U}_{\|\mathrm{Fin}\|}) \to \|\mathsf{L}\ K \simeq \mathsf{S}\ K\| \,.$$

This is precisely the right way to encode equipotence in HoTT. For suppose we know that $\mathsf{L}\ K$ is finite of size n, that is, we have an inhabitant of the type $(n : \mathbb{N}) \times \|\mathsf{L}\ K \simeq \mathsf{Fin}\ n\|$. Then we can conclude that $\mathsf{S}\ K$ has the same size: since we want to produce the mere proposition $\|\mathsf{S}\ K \simeq \mathsf{Fin}\ n\|$, we are allowed to use the equivalence $\mathsf{L}\ K \simeq \mathsf{Fin}\ n$ as well as the equivalence $\mathsf{L}\ K \simeq \mathsf{S}\ K$ produced by χ_K; composing them and injecting back into a truncation yields the desired result. On the other hand, we cannot use the results of χ to actually compute a correspondence between elements of $\mathsf{L}\ K$ and $\mathsf{S}\ K$.

One might expect that there are other ways to obtain an equipotence. That is, the correspondence between L and S is not a natural isomorphism because it additionally requires a linear order structure on the labels; might there be other equipotences which require other sorts of structure on the labels?

I conjecture that a linear order is as strong as one could ever want; that is, for any species which are provably equipotent, there exists a proof making use of a linear order on the set of labels.

Conjecture 3.3.5. *The type of natural isomorphisms with access to a linear order*

is logically equivalent to the type of equipotences. That is, for all species F and G,

$$((L : \mathcal{U}_{\mathrm{Fin}}) \to (F\ L \simeq G\ L)) \leftrightarrow ((L : \mathcal{U}_{\|\mathrm{Fin}\|}) \to \|F\ L \simeq G\ L\|).$$

Note that on the left-hand side, $F\ L$ and $G\ L$ are not well-typed as written, but are used as shorthands for the application of F and G to ιL, where $\iota : \mathcal{U}_{\mathrm{Fin}} \to \mathcal{U}_{\|\mathrm{Fin}\|}$ is the evident injection.

Proof (sketch). I describe here a plan of attack, *i.e.* an outline of a possible proof, although as explained below, I expect that completing the proof will require a considerable amount of effort.

($\to$) This direction is certainly true and quite easy to show. We are given a function $f : (L : \mathcal{U}_{\mathrm{Fin}}) \to (F\ L \simeq G\ L)$ and some $L : \mathcal{U}_{\|\mathrm{Fin}\|}$, and must produce $\|F\ L \simeq G\ L\|$. Since we are producing a mere proposition we may unwrap the finiteness evidence in L to turn it into a $\mathcal{U}_{\mathrm{Fin}}$, pass it to f, and then wrap the result in a propositional truncation. Intuitively, this direction is true since every natural isomorphism is also an equipotence.

($\leftarrow$) This is the more interesting direction. We are given a function $f : (L : \mathcal{U}_{\|\mathrm{Fin}\|}) \to \|F\ L \simeq G\ L\|$ and some $L : \mathcal{U}_{\mathrm{Fin}}$, *i.e.* a finite set equipped with a linear order. We must produce an equivalence $F\ L \simeq G\ L$. We can easily turn L into a $\mathcal{U}_{\|\mathrm{Fin}\|}$ by applying a propositional truncation; passing this to f results in some $s : \|F\ L \simeq G\ L\|$.

The trick is now to uniquely characterize the particular equivalence $F\ L \simeq G\ L$ we wish to produce, which we can do by producing linear orderings on the $(F\ L)$-shapes and $(G\ L)$-shapes, and matching them in order. We have the linear ordering on L to help, but the task still seems impossible without some sort of knowledge about F and G. Fortunately, it is possible to deeply characterize species based on their extensional behavior. In particular, every species can be uniquely decomposed as a sum of *molecular* species [Bergeron et al., 1998, §2.6], where each molecular species is of the form X^n/H for some natural number n and some subgroup $H \subseteq \mathcal{S}_n$ of the symmetric group on n elements. That is, molecular species are lists of a particular length quotiented by some symmetries: we let H act on X^n-shapes by permuting their elements, and consider equivalence classes of X^n-shapes corresponding to orbits under H. (For a fuller discussion of such quotient species, see §4.6.) The study and classification of, molecular and atomic species takes up an entire section of Bergeron et al. [1998], and porting all of the definitions and theorems there to HoTT would be a formidable undertaking, though I expect it would yield considerable insight. Such an undertaking is left to future work.

In any case, an equivalence $F\ L \simeq M_1\ L + M_2\ L + M_3\ L + \ldots$ should yield a canonical ordering on the classes of F-shapes resulting from each M_i: all the

M_1 shapes come first, followed by the M_2 shapes, and so on. It remains to show that we can put a linear ordering on the F shapes generated by each M_i.

Recall that each M_i is of the form X^n/H. We can thus use the linear order on L to put an ordering on M_i L as follows. First, in the case that $H = 1$, *i.e.* the trivial group, we can order all the $n!$ labelled X^n shapes using a lexicographic order (or some other appropriate order derived from the order on L). If H is nontrivial, then the orbits of X^n under the action of H are themselves the M_i-shapes, and we can extend the ordering on the X^n shapes to orbits thereof, for example, by ordering the orbits according to the smallest X^n-shape contained in each.

Even if we succeed in uniquely characterizing some equivalence, note that the equivalence we thus characterize may not be the same as the s obtained as the output of the function f. We must construct the final equivalence "from scratch", somehow using the fact that we know *some* equivalence exists to construct the one we have characterized. It is not entirely clear how to do this. One idea might be to construct a permutation on G L which, when composed with the equivalence given by f, produces the desired equivalence. However, this is admittedly the sketchiest part of the proof. ∎

3.4 Generating functions

Generating functions are a well-known tool in combinatorics, used to manipulate sequences of interest by representing them as the coefficients of certain formal power series. As Wilf says, "A generating function is a clothesline on which we hang up a sequence of numbers for display" [Wilf, 1990]. Generating functions are important to discuss here since they are in some sense the point of departure for the entire theory of species: although species can be understood independently, from a historical point of view species were explicitly developed in order to generalize (specifically, to *categorify*) the theory of generating functions. As such, generating functions yield important insights into the theory of species.

There are many types of generating functions; we will consider two in particular: *ordinary* generating functions (ogfs), and *exponential* generating functions (egfs). Ordinary generating functions are of the form

$$\sum_{n \geqslant 0} a_n x^n$$

and represent the sequence $a_0, a_1, a_2, \ldots$. For example, the ogf $x + 2x^2 + 3x^3 + \ldots$ represents the sequence $0, 1, 2, 3, \ldots$. Exponential generating functions are of the form

$$\sum_{n \geqslant 0} a_n \frac{x^n}{n!}.$$

For example, the egf $1/(1-x) = 1 + x + x^2 + x^3 + \cdots = 1 + x + \frac{2x^2}{2} + \frac{6x^3}{6} + \ldots$ represents the sequence $1, 1, 2, 6, 24, \ldots$.

This would be unremarkable if it were just a *notation* for sequences, but it is much more. The crucial point is that natural *algebraic* operations on generating functions correspond to natural *combinatorial* operations on the sequences they represent (or, more to the point, on the combinatorial objects the sequences are counting). This theme will be explored throughout the chapter: as each combinatorial operation on species is introduced, its corresponding algebraic operation on generating functions will also be discussed.

To each species F we associate two generating functions[3], an egf $F(x)$ and an ogf $\widetilde{F}(x)$, defined as follows.

Definition 3.4.1. The egf $F(x)$ associated to a species F is defined by

$$F(x) = \sum_{n \geqslant 0} f_n \frac{x^n}{n!},$$

where $f_n = \#(F\ n)$ is the number of labelled F-shapes of size n.

Example. There are $n!$ labelled L-shapes (that is, linear orders) on n labels, so

$$\mathsf{L}(x) = \sum_{n \geqslant 0} n! \frac{x^n}{n!} = \sum_{n \geqslant 0} x^n = \frac{1}{1-x}.$$

Note that this is a *formal* power series, and in particular we do not worry about issues of convergence.

Definition 3.4.2. The ogf $\widetilde{F}(x)$ associated to a species F is defined by

$$\widetilde{F}(x) = \sum_{n \geqslant 0} \widetilde{f}_n x^n,$$

where $\widetilde{f}_n = \#(F\ n/\!\approx)$ is the number of distinct *F-forms* (that is, equivalence classes of F-shapes under relabelling) of size n.

Example. There is only one list form of each size, so

$$\widetilde{\mathsf{L}}(x) = \sum_{n \geqslant 0} x^n = \frac{1}{1-x}$$

as well. Species for which $F(x) = \widetilde{F}(x)$ are called *regular* and are discussed in more detail in §4.6. For an example of a non-regular species, the reader is invited to work out the egf and ogf for the species C of cycles.

[3]There are more, *e.g.* the cycle index series and asymmetry index series [Bergeron et al., 1998], but they are outside the scope of this dissertation.

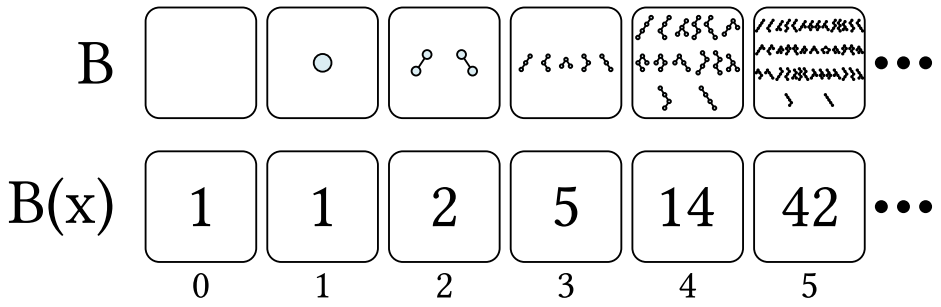


Figure 3.17: Correspondence between species and generating functions

One can see that the mapping from species to generating functions discards information, compressing an entire set of shapes or forms into a single number (Figure 3.17). Once one has defined the notion of species, it is not hard to come up with the notion of generating functions as a sort of "structured summary" of species.

Historically, however, generating functions came first. As Joyal makes explicit in the introduction to his seminal paper *Une Théorie Combinatoire des Séries Formelles* [1981]—in fact, it is even made explicit in the title of the paper itself—the main motivation for inventing species was to generalize the theory of generating functions, putting it on firmer combinatorial and categorical ground. The theory of generating functions itself was already well-developed, but no one had yet tried to view it through a categorical lens.

The general idea is to "blow everything up", replacing natural numbers by sets; addition by disjoint union; product by pairing; and so on. In a way, one can see this process as "imbuing everything with constructive significance"; this is one argument for the naturalness of developing the theory of species within a constructive type theory.

3.5 Conclusion

In this chapter we have seen the definition of species, both in set theory and type theory, and related definitions such as isomorphism and equipotence of species and generating functions. We have seen that defining species within homotopy type theory has some benefits: for example, it becomes impossible to write down invalid species within the type theory, and homotopy type theory sheds new light on some of the fundamental equivalence relations on species. However, up to this point everything has been "low-level", in the sense of working directly with the definition of species. In the next chapter we will see how to build a higher-level algebraic framework on top of species, and how this also gives us a framework for generalizing species to other categories.

Chapter 4

Generalized species and species operations

The definition of species, in either set theory or type theory, is straightforward: species are objects in a certain functor category. However, it is not the functors themselves which are fundamentally interesting, but the structure of the functor category. As we will see in this chapter, the functor category $\mathbf{B} \Rightarrow \mathbf{Set}$ has at least six different monoidal structures, corresponding to combinatorially sensible operations on species.

This also opens up the possibility of considering other functor categories with similar monoidal structures, instead of remaining tied to $\mathbf{B} \Rightarrow \mathbf{Set}$, which is too specific and restrictive. In particular, we will consider arbitrary functor categories in place of traditional species, determining the properties necessary to support each species operation. First of all, this allows us to justify $\mathcal{B} \Rightarrow \mathcal{S}$ as an analogue of species in HoTT. Even within the realm of pure mathematics, however, there are extensions to the basic theory of species (*e.g.* multisort species, weighted species, $\mathbf{L}$-species, vector species, ...) which require generalizing from $\mathbf{B} \Rightarrow \mathbf{Set}$ to other functor categories.

Sections §4.1–§4.5 examine species operations—in particular, the six monoidal structures referred to above, along with differentiation—in the context of general functor categories $\mathfrak{L} \Rightarrow \mathfrak{S}$ (where $\mathfrak{L}$ and $\mathfrak{S}$ are arbitrary categories), in order to identify precisely what properties of $\mathfrak{L}$ and $\mathfrak{S}$ are necessary to define each operation. That is, starting "from scratch", we will build up a generic notion of species that supports the operations we are interested in. In the process, we get a much clearer picture of where the operations "come from". In particular, $\mathbf{B}$ and $\mathbf{Set}$ enjoy special properties as categories (for example, $\mathbf{Set}$ is cartesian closed, has all limits and colimits, and so on), and it is enlightening to see precisely which of these properties are required in which situations. Although more general versions of specific operations have been defined previously [Kelly, 2005, Fiore et al., 2008, Lack and Street, 2014], I am not aware of any previous systematic generalization similar to this work. In particular, the general categorical treatments of arithmetic product (§4.2.2) and multisort species (§5.4) are new.

Along the way, we will explore particular instantiations of the general framework. Each instantiation arises from considering particular categories in place of $\mathbf{B}$ and $\mathbf{Set}$. To keep these functor categories straight, we will use the word "species" for $\mathbf{B} \Rightarrow \mathbf{Set}$, and "generalized species" (or, more specifically, "$(\mathfrak{L} \Rightarrow \mathfrak{S})$-species")[1] for some abstract $\mathfrak{L} \Rightarrow \mathfrak{S}$. Each section begins by defining a particular species operation in $\mathbf{B} \Rightarrow \mathbf{Set}$, then generalizes it to arbitrary functor categories $\mathfrak{L} \Rightarrow \mathfrak{S}$, and exhibits examples in other functor categories.

The chapter concludes with some comments on the relationship between symmetry and species operations (§4.6) and on *eliminators* for species (§4.7, which become important when considering species as a basis for data structures, as in Chapter 6.

4.1 Lifted monoids: sum and Cartesian product

Two of the simplest operations on species are *sum* and *Cartesian product*. These operations are structurally analogous: the only difference is that species sum arises from coproducts in $\mathbf{Set}$ (disjoint union), whereas the Cartesian product of species arises from products in $\mathbf{Set}$. We first define and give examples of these operations in the context of $\mathbf{B} \Rightarrow \mathbf{Set}$ and then generalize to other functor categories.

4.1.1 Species sum

The *sum* of two species is given by their disjoint union: an $(F + G)$-shape is either an F-shape *or* a G-shape, together with a tag to distinguish them.

Definition 4.1.1. Given $F, G : \mathbf{B} \to \mathbf{Set}$, their sum $F + G : \mathbf{B} \to \mathbf{Set}$ is defined on objects by

$$(F + G)\ L := F\ L \uplus G\ L,$$

where $\uplus$ denotes disjoint union (coproduct) of sets, and on morphisms by

$$(F + G)\ \sigma := F\ \sigma \uplus G\ \sigma,$$

where $\uplus$ is considered as a bifunctor in the evident way: $(f \uplus g)\ (\mathsf{inl}\ x) := \mathsf{inl}\ (f\ x)$ and $(f \uplus g)\ (\mathsf{inr}\ y) := \mathsf{inr}\ (g\ y)$.

It remains to prove that the $F + G$ defined above is actually functorial.

Proof. The functoriality of $F + G$ follows from that of F, G, and $\uplus$:

$$(F + G)\ id = F\ id \uplus G\ id = id \uplus id = id,$$

and

[1] Not to be confused with the generalized species of Fiore et al. [2008], who define "(A, B)-species" as functors from $\mathbf{B}A$ (a generalization of $\mathbf{B}$) to $\hat{B}$, the category of presheaves $B^{\mathrm{op}} \to \mathbf{Set}$ over B.

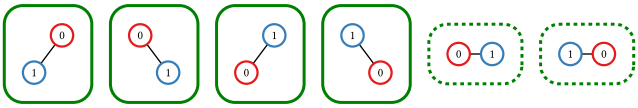


Figure 4.1: $(\mathsf{B} + \mathsf{L})\ 2$

$$
\begin{aligned}
& (F + G)(f \circ g) \\
= & \qquad \{ \quad + \text{ definition} \quad \} \\
& F\ (f \circ g) \uplus G\ (f \circ g) \\
= & \qquad \{ \quad F, G \text{ functors} \quad \} \\
& (F\ f \circ F\ g) \uplus (G\ f \circ G\ g) \\
= & \qquad \{ \quad \uplus \text{ bifunctor} \quad \} \\
& (F\ f \uplus G\ f) \circ (F\ g \uplus G\ g) \\
= & \qquad \{ \quad + \text{ definition} \quad \} \\
& (F + G)\ f \circ (F + G)\ g.
\end{aligned}
$$

∎

Remark. More abstractly, when defining a functor with a groupoid as its domain (such as $F + G$ above), it suffices to specify only its action on objects, using an arbitrary expression composed of (co- and contravariant) functors. For example, $(F + G)\ L = F\ L \uplus G\ L$ is defined in terms of the functors F, G, and $\uplus$. In that case the action of the functor on morphisms can be derived automatically by induction on the structure of the expression, simply substituting the morphism in place of covariant occurrences of the object, and the morphism's inverse in place of contravariant occurrences. In fact, in HoTT, this is simply transport; that is, given an h-groupoid B and a (pre)category C, any function $B_0 \to C_0$ extends to a functor $B \to C$.

By the same token, to define a functor with an arbitrary category (not necessarily a groupoid) as its domain, it suffices to define its action on an object using an expression containing only covariant occurrences of the object.

Example. $\mathsf{B} + \mathsf{L}$ is the species of shapes which are *either* binary trees or lists (Figure 4.1).

Example. As another example, consider $\mathsf{B} + \mathsf{B}$. It is important to bear in mind that $+$ yields a *disjoint* or "tagged" union; so $\mathsf{B} + \mathsf{B}$ consists of *two* copies of every binary tree (Figure 4.2), and in particular it is distinct from B.

Species sum corresponds to the sum of generating functions: we have

$$(F + G)(x) = F(x) + G(x) \quad \text{and} \quad \widetilde{(F + G)}(x) = \widetilde{F}(x) + \widetilde{G}(x).$$

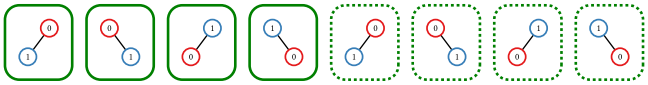


Figure 4.2: $(\mathsf{B} + \mathsf{B})\ 2$

This is because the sum of two generating functions is computed by summing corresponding coefficients,

$$\left(\sum_{n \geqslant 0} a_n x^n\right) + \left(\sum_{n \geqslant 0} b_n x^n\right) = \sum_{n \geqslant 0} (a_n + b_n) x^n$$

(and likewise for egfs), and since species sum is given by disjoint union, the number of $(F + G)$-shapes and -forms of a given size is the sum of the number of F- and G-shapes (respectively -forms) of that size.

There is also a primitive species which is an identity element for species sum.

Definition 4.1.2. The *zero* or *empty* species, $\mathsf{0}$, is the unique species with no shapes whatsoever. That is, on objects, $\mathsf{0}\ L := \varnothing$, and on morphisms $\mathsf{0}$ sends every σ to the unique function $\varnothing \to \varnothing$.

We evidently have

$$\mathsf{0}(x) = \widetilde{\mathsf{0}}(x) = 0 + 0x + 0x^2 + \cdots = 0.$$

Proposition 4.1.3. $(+, \mathsf{0})$ *is a symmetric monoid on* $\mathbf{B} \Rightarrow \mathbf{Set}$.

Proof. First, we must show that $+$ is a bifunctor. By definition it sends two functors to a functor, but this is only its action on the objects of **Spe**. We must also specify its action on morphisms, that is, natural transformations between species, and we must show that it preserves identity natural transformations and (vertical) composition of natural transformations.

In this case it's enough simply to unfold definitions and follow the types. Given species F, F', G, and G' and natural transformations $\phi : F \xrightarrow{\bullet} F'$ and $\psi : G \xrightarrow{\bullet} G'$, we should have $\phi + \psi : F + G \xrightarrow{\bullet} F' + G'$. The component of $\phi + \psi$ at some $L \in \mathbf{B}$ should thus be a morphism in **Set** of type $F\ L \uplus G\ L \to F'\ L \uplus G'\ L$; the only thing that fits the bill is $\phi_L \uplus \psi_L$.

This nicely fits with the "elementwise" definition of $+$ on species: $(F + G)\ L = F\ L \uplus G\ L$, and likewise $(\phi + \psi)_L = \phi_L \uplus \psi_L$. The action of $+$ on natural transformations thus reduces to the elementwise action of $\uplus$ on their components. From this it follows that

- $\phi + \psi$ is natural (because ϕ and ψ are), and
- $+$ preserves identity and composition (because $\uplus$ does).

Finally, we note that + inherits the symmetry of $\uplus$. ∎

Stepping back a bit, we can see that this monoidal structure on species arises straightforwardly from the corresponding monoidal structure on sets: the sum of two functors is defined as the pointwise coproduct of their outputs, and likewise $\mathsf{0}$, the identity for the sum of species, is defined as the functor which pointwise returns $\varnothing$, the identity for the coproduct of sets. More generally, any monoidal structure on a category $\mathfrak{S}$ lifts to a corresponding monoidal structure on a functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$ (this construction is spelled out in §4.1.3). This leads us naturally to consider another species operation which arises in the same way, but based on a different monoid.

4.1.2 Cartesian/Hadamard product

The definition of species sum involves *coproducts* in **Set**. Of course, **Set** also has *products*, given by $S \times T = \{(s, t) \mid s \in S, t \in T\}$, with any one-element set as the identity. We may suppose there is some canonical choice of one-element set, $\{\star\}$; since there is exactly one bijection between any two singleton sets, we do not even need the axiom of choice to implicitly make use of them. (In type theory, there is by definition a canonical singleton type $\top$.)

Definition 4.1.4. The *Cartesian* or *Hadamard product* of species is defined on objects by $(F \times G)\ L = F\ L \times G\ L$.

This is the "obvious" definition of product for species, though as we will see it is not the most natural one from a combinatorial point of view. Nonetheless, it is the simplest to define and is thus worth studying first. The action of $(F \times G)$ on morphisms, functoriality, *etc.* are omitted; the details are exactly parallel to the definition of species sum, and are presented much more generally in §4.1.3 and Chapter A.

An $(F \times G)$-shape is both an F-shape *and* a G-shape, on *the same set of labels*. There are several ways to think about this situation, as illustrated in Figure 4.3. One can think of two distinct shapes, with labels duplicated between them; this is the most literal interpretation of the definition. One can also think of the labels as *pointers* for locations in a shared memory. Finally, one can think of the shapes themselves as being superimposed. This last view highlights the fact that $\times$ is symmetric, but only up to isomorphism, since at root it still consists of an *ordered* pair of shapes.

In parallel with sum, we can see that

$$(F \times G)(x) = F(x) \times G(x) \quad \text{and} \quad \widetilde{(F \times G)}(x) = \widetilde{F}(x) \times \widetilde{G}(x),$$

where

$$\left(\sum_{n \geqslant 0} a_n x^n\right) \times \left(\sum_{n \geqslant 0} b_n x^n\right) = \sum_{n \geqslant 0} (a_n b_n) x^n$$

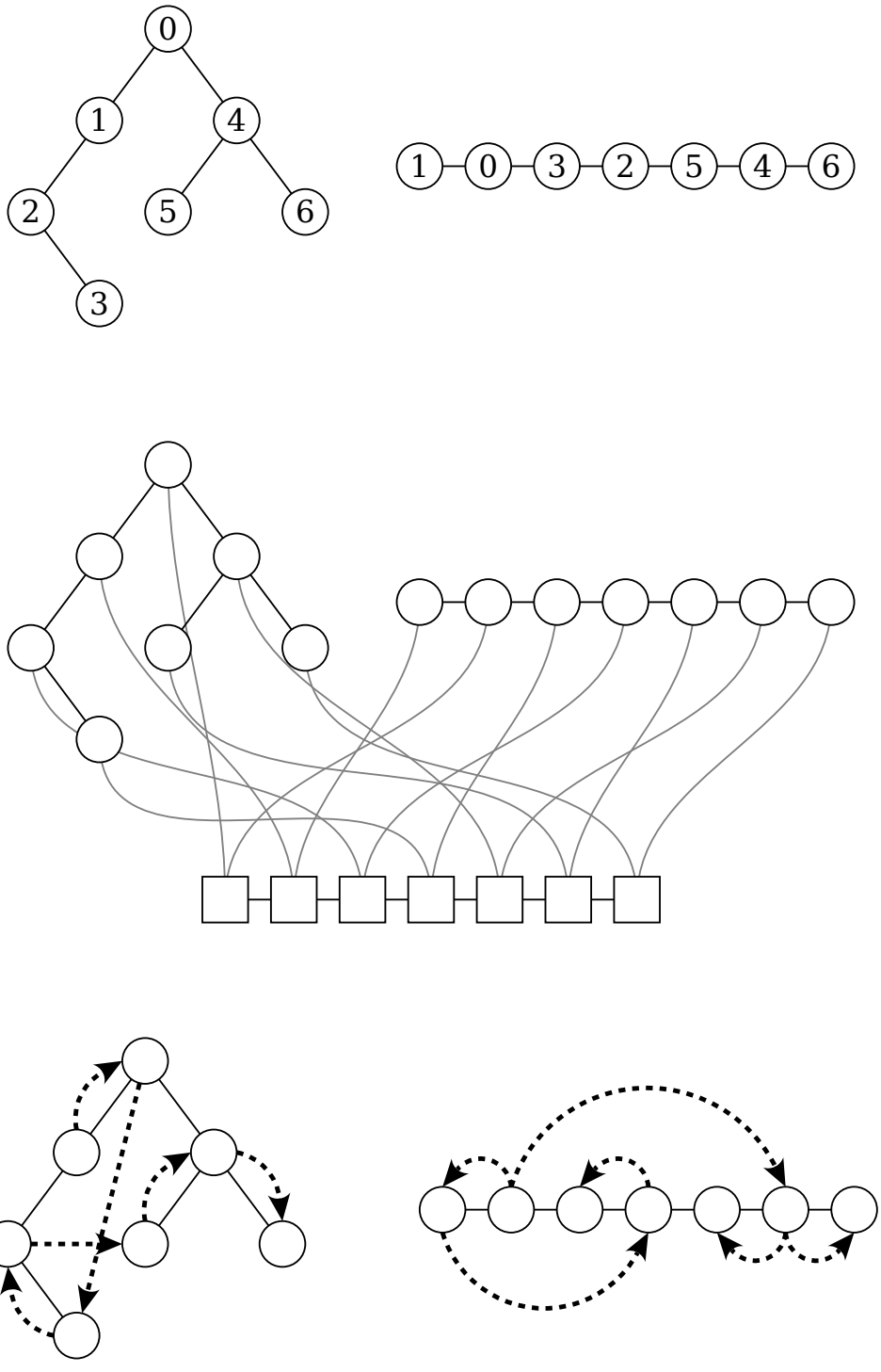


Figure 4.3: Four views on the Cartesian product $\mathsf{B} \times \mathsf{L}$

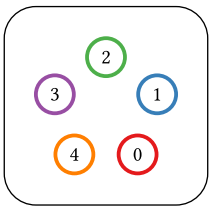


Figure 4.4: The unique E 5 shape

and

$$\left(\sum_{n\geqslant 0} a_n \frac{x^n}{n!}\right) \times \left(\sum_{n\geqslant 0} b_n \frac{x^n}{n!}\right) = \sum_{n\geqslant 0} (a_n b_n) \frac{x^n}{n!}$$

denote the elementwise or *Hadamard* product of two generating functions. This is not a particularly natural operation on generating functions (although it is easy to compute); in particular it is not what one usually thinks of as *the* product of generating functions. As we will see in §4.2, there is a different combinatorial operation that corresponds to the usual product of generating functions.

There is also a species, usually called E, which is an identity element for Cartesian product. Considering that we should have $(\mathsf{E} \times G)\ L = \mathsf{E}\ L \times G\ L \simeq G\ L$, the proper definition of E becomes clear:

Definition 4.1.5. The species of *sets*, E, is defined as the constant functor yielding $\{\star\}$, that is, $\mathsf{E}\ L = \{\star\}$.

The ogf for E is given by

$$\widetilde{\mathsf{E}}(x) = 1 + x + x^2 + \cdots = \frac{1}{1-x},$$

and the egf by

$$\mathsf{E}(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = e^x.$$

The notation E was probably chosen as an abbreviation of the French *ensemble* (set), but it is also a clever pun on the fact that $\mathsf{E}(x) = e^x$.

Remark. E is called the *species of sets* since there is exactly one shape on any set of labels, which can be thought of as the set of labels itself, with no additional structure. In fact, since all one-element sets are isomorphic, we may define $\mathsf{E}\ L = \{L\}$ (Figure 4.4).

Proposition 4.1.6. $(\times, \mathsf{E})$ *is a symmetric monoid on* **Spe**.

Proof. The proof is omitted, since it is almost exactly the same as the proof for $(+, 0)$; the only difference is the substitution of Cartesian product of sets for disjoint union. ∎

4.1.3 Lifting monoids

Both these constructions generalize readily. In fact, any monoidal structure on a category $\mathfrak{S}$ can be lifted to one on $\mathfrak{L} \Rightarrow \mathfrak{S}$ where everything is done elementwise. The basic idea is exactly the same as the standard Haskell type class instance

$$\begin{aligned}&\textbf{instance } \mathsf{Monoid}\ a \Rightarrow \mathsf{Monoid}\ (e \to a) \textbf{ where}\\ &\quad \varepsilon \quad\ = \lambda\_ \to \varepsilon\\ &\quad f \diamond g = \lambda a \to f\ a \diamond g\ a\end{aligned}$$

but quite a bit more general.

Proposition 4.1.7. *Any (strict) monoid $(\otimes, I)$ on $\mathfrak{S}$ lifts to a monoid, denoted $(\otimes^{\mathfrak{L}}, I^{\mathfrak{L}})$, on the functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$. In particular, $(F \otimes^{\mathfrak{L}} G)\ L = F\ L \otimes G\ L$, and $I^{\mathfrak{L}}$ is Δ_I, the functor which is constantly I. Moreover, this lifting preserves products, coproducts, symmetry, and distributivity.*

In fact, non-strict monoids lift as well; a yet more general version of this proposition, along with a detailed proof, will be given later. First, however, we consider some examples.

Example. Lifting coproducts in **Set** to $\mathbf{B} \Rightarrow \mathbf{Set}$ yields the $(+, \mathsf{0})$ structure on species, and likewise lifting products yields $(\times, \mathsf{E})$. According to Proposition 4.1.7, since $(\uplus, \varnothing)$ is a categorical coproduct on **Set**, $(+, 0)$ is likewise a categorical coproduct on the category $\mathbf{B} \Rightarrow \mathbf{Set}$ of species, and similarly $(\times, \mathsf{E})$ is a categorical product.

Example. Take $\mathfrak{L} = \mathbf{1}$ (the trivial category with one object and one morphism). In this case, functors in $\mathbf{1} \Rightarrow \mathfrak{S}$ are just objects of $\mathfrak{S}$, and a lifted monoidal operation is isomorphic to the unlifted one.

Example. Take $\mathfrak{L} = |\mathbf{2}|$, the discrete category with two objects. Then a functor $F : |\mathbf{2}| \to \mathfrak{S}$ is just a pair of objects in $\mathfrak{S}$. For example, if $\mathfrak{S} = \mathbf{Set}$, a functor $|\mathbf{2}| \to \mathbf{Set}$ is a pair of sets. In this case, taking the lifted sum $F + G$ of two functors $F, G : |\mathbf{2}| \to \mathbf{Set}$ corresponds to summing the pairs elementwise, that is, $(S_1, T_1) + (S_2, T_2) = (S_1 \uplus S_2, T_1 \uplus T_2)$.

Recall that when X is a discrete category, the functor category $X \Rightarrow \mathbf{Set}$ is equivalent to the slice category $\mathbf{Set}/X$. This gives another way to think of a functor $|\mathbf{2}| \to \mathbf{Set}$, namely, as a single set of elements S together with a function $S \to |\mathbf{2}|$ which "tags" each element with one of two tags ("left" or "right", 0 or 1, *etc.*). From this point of view, a lifted sum can be thought of as a tag-preserving disjoint union.

Example. As an example in a similar vein, consider $\mathfrak{L} = \mathbb{N}$, the discrete category with natural numbers as objects. Functors $\mathbb{N} \to \mathfrak{S}$ are countably infinite sequences of objects $[S_0, S_1, S_2, \ldots]$. One way to think of this is as a collection of $\mathfrak{S}$-objects, one for each natural number *size*. For example, when $\mathfrak{S} = \mathbf{Set}$, a functor $\mathbb{N} \to \mathbf{Set}$

is a sequence of sets $[S_0, S_1, S_2, \dots]$, where S_i can be thought of as some collection of objects of size i. (This "size" intuition is actually fairly arbitrary at this point—the objects of $\mathbb{N}$ are in some sense just an arbitrary countably infinite set of labels, and there is no particular reason they should represent "sizes". However, the "size" intuition carries over well to species.)

Again, $(\mathbb{N} \Rightarrow \mathbf{Set}) \cong \mathbf{Set}/\mathbb{N}$, so functors $\mathbb{N} \to \mathbf{Set}$ can also be thought of as a single set S along with a function $S \to \mathbb{N}$ which gives the size of each element.

Lifting a monoidal operation to countable sequences of objects performs a "zip", applying the monoidal operation between matching positions in the two lists:

$$[S_1, S_2, S_3, \dots] \oplus [T_1, T_2, T_3, \dots] = [S_1 \oplus T_1, S_2 \oplus T_2, S_3 \oplus T_3, \dots]$$

Example. All the previous examples have used a discrete category in place of $\mathfrak{L}$; it is instructive to see an example with nontrivial morphisms involved. As the simplest nontrivial example, consider $\mathfrak{L} = \mathbf{2}$, the category with two objects 0 and 1 and a single non-identity morphism $0 \longrightarrow 1$. A functor $\mathbf{2} \to \mathfrak{S}$ is not just a pair of objects (as with $\mathfrak{L} = |\mathbf{2}|$) but a pair of objects with a morphism between them:

$$S_0 \xrightarrow{f} S_1.$$

Combining two such functors with a lifted monoidal operation combines not just the objects but also the morphisms:

$$(S_0 \xrightarrow{f} S_1) \oplus (T_0 \xrightarrow{g} T_1) = (S_0 \oplus T_0) \xrightarrow{f \oplus g} (S_1 \oplus T_1)$$

This is possible since the monoidal operation $\oplus$ is, by definition, required to be a bifunctor.

Example. In $\mathcal{S}$, the coproduct of two types A and B is given by their sum, $A + B$, with the void type $\bot$ serving as the identity. We may thus lift this coproduct structure to the functor category $\mathcal{B} \Rightarrow \mathcal{S}$—or indeed to any $\mathfrak{L} \Rightarrow \mathcal{S}$, since no requirements are imposed on the domain category.

Example. Similarly, categorical products in $\mathcal{S}$ are given by product types $A \times B$, with the unit type $\top$ as the identity. This then lifts to products on $\mathcal{B} \Rightarrow \mathcal{S}$ (or, again, any $\mathfrak{L} \Rightarrow \mathcal{S}$) which serve as an analogue of Cartesian product of species.

A more detailed proof of the fact that any monoid on a category $\mathfrak{S}$ lifts to a corresponding monoid on $\mathfrak{L} \Rightarrow \mathfrak{S}$ can be found in Chapter A.

4.1.4 Internal Hom for Cartesian product

Recall that a *Cartesian closed* category is one which is closed with respect to Cartesian product, that is, there exists some bifunctor $B \Rightarrow C$ such that

$$\forall ABC.\ (A \times B \Rightarrow C) \cong (A \Rightarrow (B \Rightarrow C)).$$

Such categories allow morphisms to be "internalized", that is, represented as objects.

Proposition 4.1.8. Spe *is Cartesian closed.*

If $\mathfrak{L}$ is locally small and $\mathfrak{S}$ is complete and Cartesian closed, then $\mathfrak{L} \Rightarrow \mathfrak{S}$ is also complete and Cartesian closed [Shulman]. In particular, the exponential of $F, G : \mathfrak{L} \to \mathfrak{S}$ is given by

$$G^F\ L = \forall K \in \mathfrak{L}.\ \prod_{\mathfrak{L}(L,K)} G(K)^{F(K)}.$$

For example, **B**, **P**, $\mathcal{B}$, and $\mathcal{P}$ are all locally small, and **Set** and $\mathcal{S}$ are complete and Cartesian closed, so $\mathbf{B} \Rightarrow \mathbf{Set}$, $\mathbf{P} \Rightarrow \mathbf{Set}$, $\mathcal{B} \Rightarrow \mathcal{S}$, and $\mathcal{P} \Rightarrow \mathcal{S}$ are all complete and Cartesian closed as well.

Let's unpack this result a bit in the specific case of $\mathcal{P} \Rightarrow \mathcal{S}$. By a dual argument to the one given in §2.2.2, ends in $\mathcal{S}$ over the groupoid $\mathcal{P}$ are given by Π-types, *i.e.* universal quantification; hence, we have

$$\begin{aligned}(H^G)\ n &= \forall m \in \mathcal{P}.\ \prod_{\mathcal{P}(m,n)} (H\ n)^{G\ n} \\ &= (m : \mathbb{N}) \to (\mathsf{Fin}\ m \simeq \mathsf{Fin}\ n) \to G\ n \to H\ n \\ &\simeq (\mathsf{Fin}\ n \simeq \mathsf{Fin}\ n) \to G\ n \to H\ n\end{aligned}$$

where the final isomorphism follows since $(\mathsf{Fin}\ m \simeq \mathsf{Fin}\ n)$ is only inhabited when $m = n$.

Being Cartesian closed means there is an adjunction $- \times G \dashv -^G$ between products and exponentials, which yields a natural isomorphism

$$(F \times G \Rightarrow_{\mathcal{S}^{\mathcal{P}}} H) \simeq (F \Rightarrow_{\mathcal{S}^{\mathcal{P}}} H^G).$$

Expanding morphisms of the functor category $\mathcal{P} \Rightarrow \mathcal{S}$ as natural transformations, and expanding the definition of H^G derived above, this yields

$$((n : \mathbb{N}) \to (F \times G)\ n \to H\ n) \simeq ((n : \mathbb{N}) \to F\ n \to (\mathsf{Fin}\ n \simeq \mathsf{Fin}\ n) \to G\ n \to H\ n)\,.$$

Intuitively, this says that a size-polymorphic function taking a Cartesian product shape $F \times G$ and yielding an H-shape of the same size is isomorphic to a size-polymorphic function taking a triple of an F-shape, a G-shape, *and a permutation on* $\mathsf{Fin}\ n$, and yielding an H-shape. The point is that an $(F \times G)$-shape consists not just of separate F- and G-shapes, but those shapes get to "interact": in particular we need a permutation to tell us how the labels on the separate F- and G-shapes line up. An $(F \times G)$-shape encodes this information implicitly, by the fact that the two shapes share the exact same set of labels.

Practically speaking, this result tells us how to express an eliminator for $(F \times G)$-shapes. That is, to be able to eliminate $(F \times G)$-shapes, it suffices to be able to

eliminate F- and G-shapes individually, with an extra permutation supplied as an argument. Eliminators for species shapes are treated more generally and systematically in §4.7.

On the surface, the fact that **Spe** is Cartesian closed only allows us to internalize *species morphisms* as species, but not to interpret functions between data types. **Spe** being Cartesian closed does mean that the simply typed lambda calculus can be interpreted internally to **Spe**; but it is not yet clear to me what this would mean on an intuitive level.

4.2 Partitional product and Day convolution

There is another notion of product for species, the *partitional* or *Cauchy* product. It it is the partitional product, rather than Cartesian product, which corresponds to the product of generating functions and which gives rise to the usual notion of product on algebraic data types. For these reasons, partitional product is often (both in this thesis and in species literature generally) simply referred to as "product", without any modifier.

There is also another lesser-known product, *arithmetic product* [Maia and Méndez, 2008], which can be thought of as a symmetric form of composition. These two products arise in an analogous way, via a categorical construction known as *Day convolution*.

In this section, we explore each operation in turn, and then give a general account of their common abstraction.

4.2.1 Partitional/Cauchy product

The partitional product $F \cdot G$ of two species F and G consists of paired F- and G-shapes, as with the Cartesian product, but with the labels *partitioned* between the two shapes instead of replicated (Figure 4.5). The divided box with rounded corners used in Figure 4.5 will often be used to schematically indicate a partitional product.

Definition 4.2.1. The *partitional* or *Cauchy product* of two species F and G is the functor defined on objects by

$$(F \cdot G)\ L = \biguplus_{L_F, L_G \vdash L} F\ L_F \times G\ L_G$$

where $\biguplus$ denotes an indexed coproduct (*i.e.* disjoint union) of sets, and $L_F, L_G \vdash L$ indicates that L_F and L_G constitute a partition of L, (*i.e.* $L_F \cup L_G = L$ and $L_F \cap L_G = \varnothing$); note that L_F and L_G may be empty. In words, an $(F \cdot G)$-shape with labels taken from L consists of some partition of L into two disjoint subsets, with an F-shape on one subset and a G-shape on the other.

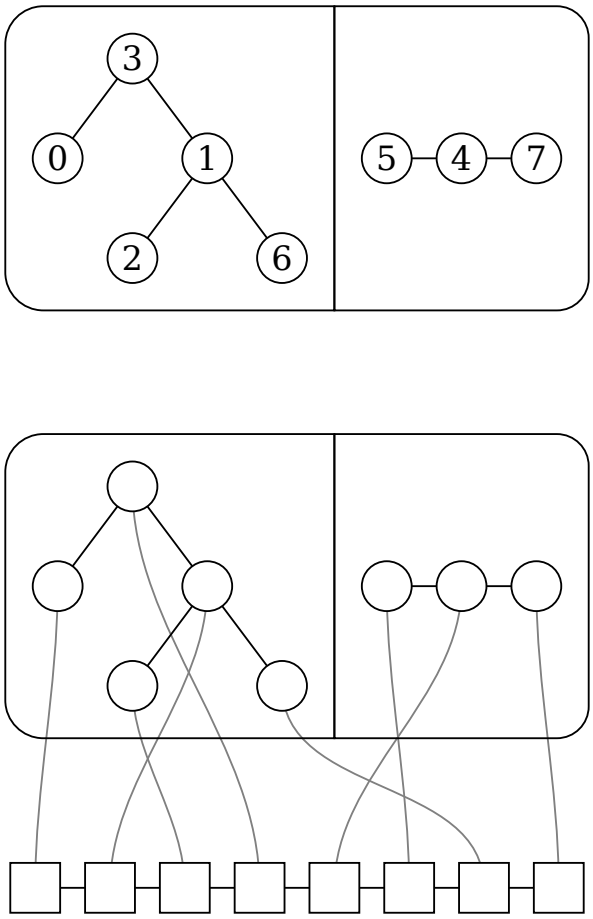


Figure 4.5: Two views on the partitional species product $\mathsf{B} \cdot \mathsf{L}$

On morphisms, $(F \cdot G)\ \sigma$ is the function

$$(L_F, L_G, x, y) \mapsto (\sigma\ L_F, \sigma\ L_G, F\ (\sigma|_{L_F})\ x, G\ (\sigma|_{L_G})\ y),$$

where $L_F, L_G \vdash L$ and $x \in F\ L_F$ and $y \in G\ L_G$. That is, σ acts independently on the two subsets of L.

To compute the ogf of a product species $F \cdot G$, consider the product of ogfs

$$\widetilde{F}(x)\widetilde{G}(x) = \left(\sum_{n\geqslant 0} f_n x^n\right)\left(\sum_{n\geqslant 0} g_n x^n\right) = \sum_{n\geqslant 0}\left(\sum_{0\leqslant k\leqslant n} f_k g_{n-k}\right) x^n.$$

Note that the inner sum $\sum_{0\leqslant k\leqslant n} f_k g_{n-k}$ is indeed the number of $(F \cdot G)$-forms of size n: such forms necessarily consist of an F-form of size k paired with a G-form of size $n-k$. Hence

$$\widetilde{(F \cdot G)}(x) = \widetilde{F}(x)\widetilde{G}(x).$$

The computation of the egf of a product species is similar.

$$\begin{aligned} F(x)G(x) &= \left(\sum_{n\geqslant 0} f_n \frac{x^n}{n!}\right)\left(\sum_{n\geqslant 0} g_n \frac{x^n}{n!}\right) \\ &= \sum_{n\geqslant 0}\left(\sum_{0\leqslant k\leqslant n} \frac{f_k}{k!}\frac{g_{n-k}}{(n-k)!}\right) x^n \\ &= \sum_{n\geqslant 0}\left(\sum_{0\leqslant k\leqslant n} \binom{n}{k} f_k g_{n-k}\right) \frac{x^n}{n!}. \end{aligned}$$

Again, we verify that the inner sum $\sum_{0\leqslant k\leqslant n} \binom{n}{k} f_k g_{n-k}$ is the number of labelled $(F \cdot G)$-shapes of size n: for each $0 \leqslant k \leqslant n$, there are $\binom{n}{k}$ ways to partition the n labels into two subsets of size k and $n-k$, and then there are f_k ways to make an F-shape on the first subset, and g_{n-k} ways to make a G-shape on the second. We therefore have

$$(F \cdot G)(x) = F(x)G(x)$$

as well.

The identity for partitional product should evidently be some species $\mathsf{1}$ such that

$$(\mathsf{1} \cdot G)\ L = \left(\biguplus_{L_F, L_G \vdash L} \mathsf{1}\ L_F \times G\ L_G\right) \simeq G\ L.$$

The only way for this isomorphism to hold naturally in L is if $\mathsf{1}\ \varnothing = \{\star\}$ (yielding a summand of $G\ L$ when $\varnothing, L \vdash L$) and $\mathsf{1}\ L_F = \varnothing$ for all other L_F (cancelling all the other summands).

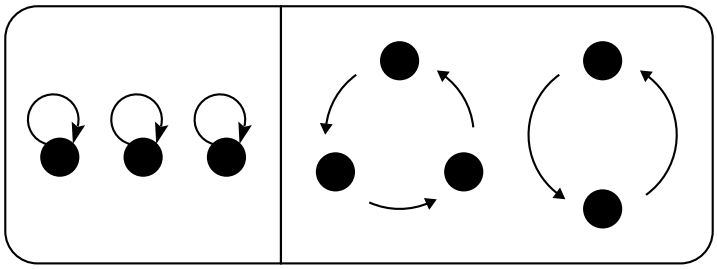


Figure 4.6: Permutation = fixpoints · derangement

Definition 4.2.2. The *unit species*, $\mathsf{1}$, is defined by

$$\mathsf{1}\ L = \begin{cases} \{\star\} & L = \varnothing \\ \varnothing & \text{otherwise.} \end{cases}$$

Remark. Recall that one should not think of $\mathsf{1}$ as doing case analysis. Rather, a more intuitive way to think of it is "there is a single $\mathsf{1}$-shape, and it has no labels"; that is, the unit species denotes a sort of "trivial" or "leaf" structure containing no labels. Intuitively, it corresponds to a Haskell type like

```
data Unit a = Unit
```

The generating functions for $\mathsf{1}$ are given by

$$\mathsf{1}(x) = \widetilde{\mathsf{1}}(x) = 1.$$

Example. The following example is due to Joyal [1981]. Recall that S denotes the species of permutations. Consider the species Der of *derangements*, that is, permutations which have no fixed points. It is not possible, in general, to directly express species using a "filter" operation, as in, "all F-shapes satisfying predicate P". However, it is possible to get a handle on Der in a more constructive manner by noting that every permutation can be canonically decomposed as a set of fixed points paired with a derangement on the rest of the elements (Figure 4.6). That is, algebraically,

$$\mathsf{S} = \mathsf{E} \cdot \mathsf{Der}. \tag{4.2.1}$$

This does not directly give us an expression for Der, since there is no notion of multiplicative inverse for species[2]. However, this is still a useful characterization of derangements. For example, since the mapping from species to egfs is a homomorphism

[2]Multiplicative inverses can in fact be defined for suitable *virtual* species [Bergeron et al., 1998, Chapter 3]. However, virtual species are beyond the scope of this dissertation.

with respect to product, (4.2.1) becomes

$$\frac{1}{1-x} = e^x \cdot \mathsf{Der}(x).$$

We can solve to obtain the egf $\mathsf{Der}(x) = e^{-x}/(1-x)$, even though we cannot make direct combinatorial sense out of $\mathsf{Der} = \mathsf{S}/\mathsf{E}$.

Proposition 4.2.3. $(\mathbf{Spe}, \cdot, 1)$ *is symmetric monoidal.*

Proof. We constructed 1 so as to be an identity for partitional product. Associativity and symmetry of partitional product are not hard to prove and are left as an exercise for the reader. ∎

In fact, $(\mathbf{Spe}, \cdot, 1)$ is closed as well, but a discussion of the internal Hom functor corresponding to partitional product must be postponed to §4.5.5, after discussing species differentiation.

4.2.2 Arithmetic/rectangular product

There is another, more recently discovered monoidal structure on species known as *arithmetic product* [Maia and Méndez, 2008]. The arithmetic product of the species F and G, written $F \boxtimes G$, can intuitively be thought of as an "F-assembly of cloned G-shapes", that is, an F-shape containing multiple copies of a *single* G-shape. Unlike the usual notion of composition (§4.3), where the F-shape is allowed to contain many different G-shapes, this notion is symmetric: an F-assembly of cloned G-shapes is isomorphic to a G-assembly of cloned F-shapes. Another intuitive way to think of the arithmetic product, which points out the symmetry more clearly, is to think of a rectangular grid of labels, together with an F-shape labelled by the rows of the grid, and a G-shape labelled by the columns. Figure 4.7 illustrates these intuitions with the arithmetic product $\mathsf{B} \boxtimes \mathsf{L}$.

A more formal definition requires the notion of a *rectangle* on a set [Maia and Méndez, 2008, Baddeley et al., 2004], which plays a role similar to that of set partition in the definition of partitional product. (So arithmetic product can also be called *rectangular product*.) In particular, whereas a binary partition of a set L is a decomposition of L into a sum, a rectangle on L can be thought of as a decomposition into a product. The basic idea is to partition L in two different ways, and require the partitions to act like the "rows" and "columns" of a rectangular matrix, which have the defining characteristic that every pair of a row and a column have a single point of intersection.

Definition 4.2.4 (Maia and Méndez [2008]). A *rectangle* on a set L is a pair (π, τ) of families of subsets of L, such that

- $\pi \vdash L$ and $\tau \vdash L$, and

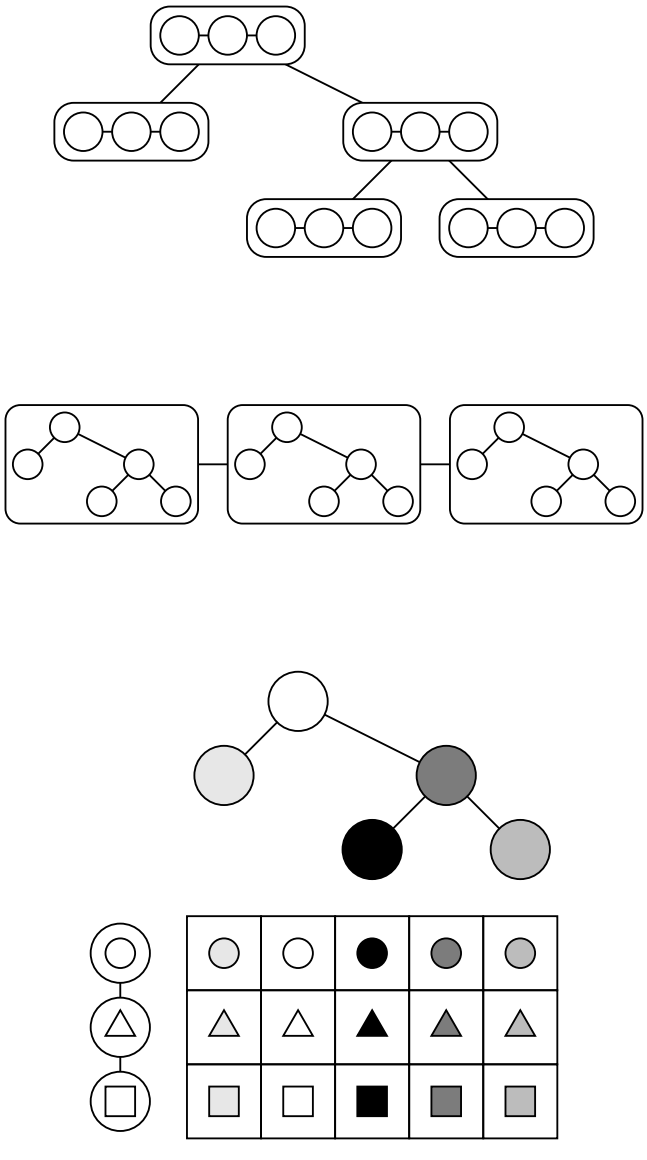


Figure 4.7: Three views on the arithmetic product $\mathsf{B} \boxtimes \mathsf{L}$

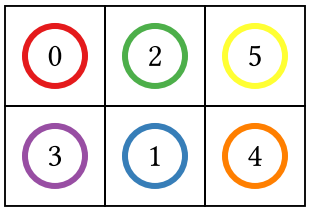


Figure 4.8: A Mat-shape of size 6

- $|X \cap Y| = 1$, for all $X \in \pi$, $Y \in \tau$.

Here, $\pi \vdash L$ denotes that π is a partition of L into any number of nonempty parts, that is, the elements of π are nonempty, pairwise disjoint, and have L as their union. We write $\pi, \tau \Vdash L$ to denote that (π, τ) constitute a rectangle on L, and call π and τ the *sides* of the rectangle.

We can now formally define arithmetic product as follows:

Definition 4.2.5. The *arithmetic product* $F \boxtimes G$ of two species F and G is the species defined on objects by

$$(F \boxtimes G)\ L = \biguplus_{L_F, L_G \Vdash L} F\ L_F \times G\ L_G.$$

$(F \boxtimes G)$ lifts bijections $\sigma : L \xrightarrow{\sim} L'$ to functions $(F \boxtimes G)\ L \to (F \boxtimes G)\ L'$ as follows:

$$(F \boxtimes G)\ \sigma\ (L_F, L_G, f, g) = (\mathcal{P}(\sigma)\ L_F, \mathcal{P}(\sigma)\ L_G, F\ \mathcal{P}(\sigma)\ f, G\ \mathcal{P}(\sigma)\ g),$$

where $\mathcal{P}(\sigma) : \mathcal{P}(L) \xrightarrow{\sim} \mathcal{P}(L')$ denotes the functorial lifting of σ to a bijection between subsets of L and L'.

Remark. The similarity of this definition to the definition of partitional product should be apparent: the only real difference is that rectangles $(L_F, L_G \Vdash L)$ have been substituted for partitions $(L_F, L_G \vdash L)$.

Example. $\mathsf{Mat} = \mathsf{L} \boxtimes \mathsf{L}$ is the species of (two-dimensional) *matrices*. Mat-shapes consist simply of labels arranged in a rectangular grid (Figure 4.8).

Example. $\mathsf{Rect} = \mathsf{E} \boxtimes \mathsf{E}$ is the species of *rectangles*. One way to think of rectangles is as equivalence classes of matrices up to reordering of the rows and columns. Each label has no fixed "position"; the only invariants on any given label are the sets of other

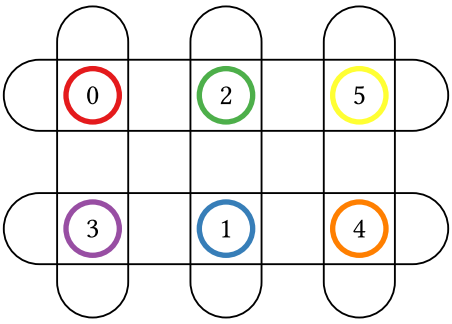


Figure 4.9: A Rect-shape of size 6

labels which are in the same row or column. Figure 4.9 shows an illustration; each rounded outline represents a *set* of labels. One can also take the species of rectangles as primitive and define arithmetic product in terms of it.

Example. Just as topological cylinders and tori may be obtained by gluing the edges of a square, species corresponding to cylinders or tori may be obtained by starting with the species of 2D matrices and "gluing" along one or both edges by turning lists L into cycles C. In particular, $\mathsf{Cyl} = \mathsf{L} \boxtimes \mathsf{C}$ is the species of (oriented) *cylinders*, and $\mathsf{Tor} = \mathsf{C} \boxtimes \mathsf{C}$ is the species of (oriented) *tori*.

Although species corresponding to Klein bottles and real projective planes (which arise from gluing the edges of a square with one or both pairs of edges given a half-twist before gluing, respectively) certainly exist, it does not seem they can be constructed using $\boxtimes$, since in those cases the actions of the symmetric group along the two axes are not independent.

The ogf for $F \boxtimes G$ is given by

$$\widetilde{F}(x) \boxtimes \widetilde{G}(x) = \left(\sum_{n \geqslant 0} f_n x^n\right) \boxtimes \left(\sum_{n \geqslant 0} g_n x^n\right) = \sum_{n \geqslant 0} \left(\sum_{d \mid n} f_d g_{n/d}\right) x^n,$$

since an $(F \boxtimes G)$-form of size n consists of a pair of an F-form and a G-form, whose sizes have a product of n.

Likewise, the egf is

$$\sum_{n \geqslant 0} \left(\sum_{d \mid n} \left\{ {n \atop d} \right\} f_d g_{n/d}\right) \frac{x^n}{n!},$$

where

$$\left\{ {n \atop d} \right\} = \frac{n!}{d!(n/d)!}$$

denotes the number of $d \times (n/d)$ rectangles on a set of size n.

An identity element for arithmetic product should be some species X such that

$$(\mathsf{X} \boxtimes G)\ L = \left(\biguplus_{L_{\mathsf{X}}, L_G \Vdash L} \mathsf{X}\ L_{\mathsf{X}} \times G\ L_G \right) \cong G\ L.$$

Thus we want $\mathsf{X}\ L_{\mathsf{X}} = \{\star\}$ when $L_G = L$, and $\mathsf{X}\ L_{\mathsf{X}} = \varnothing$ otherwise. Consider $L_{\mathsf{X}}, L \Vdash L$. Of course, L does not have the right type to be one side of a rectangle on itself, but it is isomorphic to the set of all singleton subsets of itself, which does. The definition of a rectangle now requires every element of L_{X} to have a nontrivial intersection with every singleton subset of L (such intersections will automatically have size 1). Therefore L_{X} has only one element, namely, L itself, and is isomorphic to $\{\star\}$. Intuitively, $\{\star\}, L \Vdash L$ corresponds to the fact that we can always make a $1 \times n$ rectangle on any set of size n, that is, any number n can be "factored" as $1 \times n$.

This leads to the following definition:

Definition 4.2.6. The *singleton species*, X, is defined by

$$\mathsf{X}\ L = \begin{cases} \{\star\} & |L| = 1 \\ \varnothing & \text{otherwise.} \end{cases}$$

Remark. Like the unit species $\mathsf{1}$, the singleton species X denotes a sort of "leaf" structure; however, instead of being a trivial leaf structure with no labels, it contains a single label, that is, it marks the spot where a single piece of data can go. Intuitively, it corresponds to the Haskell data type

data $\mathsf{X}\ a = \mathsf{X}\ a$

One can see that the egf and ogf for X are

$$\mathsf{X}(x) = \widetilde{\mathsf{X}}(x) = x.$$

Species corresponding to a wide variety of standard data structures can be defined using X.

Example. The species of *ordered pairs* is given by $\mathsf{X} \cdot \mathsf{X}$. Since there is only an X-shape on a single label, and product partitions the labels, there are only $(\mathsf{X} \cdot \mathsf{X})$-shapes on label sets of cardinality 2, and there are two such shapes, one for each ordering of the two labels (Figure 4.10).

More generally, $\mathsf{X}^n = \underbrace{\mathsf{X} \cdot \cdots \cdot \mathsf{X}}_{n}$ is the species of *ordered n-tuples*; there are exactly $n!$ many (X^n)-structures on n labels and none on label sets of any other size.

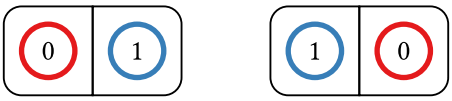


Figure 4.10: $(\mathsf{X} \cdot \mathsf{X})$-shapes

Example. Recall that L denotes the species of lists, *i.e.* linear orderings. Besides the interpretation of recursion, to be explored in §5.4.1, we have now seen all the necessary pieces to understand the algebraic definition of L:

$$\mathsf{L} = 1 + \mathsf{X} \cdot \mathsf{L}.$$

That is, a list structure is either the trivial structure on zero labels, or a single label paired with a list structure on the remainder of the labels. We also have $\mathsf{L} = 1 + \mathsf{X} + \mathsf{X}^2 + \mathsf{X}^3 + \dots$.

Example. Similarly, recall that the species B of *binary trees* is given by

$$\mathsf{B} = 1 + \mathsf{B} \cdot \mathsf{X} \cdot \mathsf{B}.$$

Example. The species $\mathsf{X} \cdot \mathsf{E}$ is variously known as the species of *pointed sets* (which may be denoted $\mathsf{E}^\bullet$) or the species of *elements* (denoted ε). $(\mathsf{X} \cdot \mathsf{E})$-structures consist of a single distinguished label paired with an unstructured collection of any number of remaining labels. There are thus n such structures on each label set of cardinality n, one for each label.

The two different names result from the fact that we may "care about" the labels in an E-structure or not—that is, when considering data structures built on top of species, E may correspond either to a bag data structure, or instead to a "sink" where we throw labels to which we do not wish to associate any data. This makes no difference from a purely combinatorial point of view, but makes a difference when considering labelled structures (Chapter 6).

4.2.3 Day convolution

Just as sum and Cartesian product were seen to arise from the same construction applied to different monoids, both partitional and arithmetic product arise from *Day convolution*, applied to different monoidal structures on $\mathbf{B}$.

It is worth first briefly mentioning the definition of an *enriched category*, which is needed here and also in §4.3. Enriched categories are a generalization of categories where the *set* of morphisms between two objects is replaced by an *object* of some other category.

Definition 4.2.7. Given some monoidal category $(\mathbb{D}, \otimes, I)$, a *category enriched over* $\mathbb{D}$ consists of

- a collection of objects O;
- for every pair of objects $A, B \in O$, a corresponding object of $\mathbb{D}$, which we notate $A \Rightarrow B$;
- for each object $A \in O$, a morphism $I \to (A \Rightarrow A)$ in $\mathbb{D}$, which "picks out" the identity morphism for each object;
- for every three objects $A, B, C \in O$, a morphism $\circ_{A,B,C} : (B \Rightarrow C) \otimes (A \Rightarrow B) \to (A \Rightarrow C)$ representing composition.

Of course, identity and composition have to satisfy the usual laws, expressed via commutative diagrams. Note we are technically overloading the $\Rightarrow$ notation, but it is natural to extend it from denoting hom sets to denoting hom objects in general.

Enriched categories and categories are notionally distinct, but we often conflate them. In particular, any category can be seen as an enriched category over **Set**, and we also often say that a category $\mathbb{C}$ *is enriched over* $\mathbb{D}$ if there exists some functor $- \Rightarrow - : \mathbb{C}^{\text{op}} \times \mathbb{C} \to \mathbb{D}$ satisfying the above criteria.

We can now give the definition of Day convolution. The essential idea, first described by Day [1970], is to construct a monoidal structure on a functor category $[\mathfrak{L}^{\text{op}}, \mathfrak{S}]$ based primarily on a monoidal structure on the *domain* category $\mathfrak{L}$. In particular, Day convolution requires

- a monoidal structure $\oplus$ on the domain $\mathfrak{L}$;
- that $\mathfrak{L}$ be enriched over $\mathfrak{S}$, so hom sets of $\mathfrak{L}$ can be seen as objects in $\mathfrak{S}$;
- a symmetric monoidal structure $\otimes$ on the codomain $\mathfrak{S}$ (satisfying an additional technical requirement, to be explained below); and
- that $\mathfrak{S}$ be cocomplete, and in particular have coends over $\mathfrak{L}$.

In addition, $\otimes$ must preserve colimits in each of its arguments. That is, $- \otimes B$ and $A \otimes -$ must both preserve colimits for any A and B. It is sufficient (though not necessary) that $\otimes$ is a left adjoint. For example, the product bifunctor in **Set** is left adjoint (via currying), and thus preserves colimits—the distributive law $(X \times (Y + Z) \cong X \times Y + X \times Z$ is a well-known example. On the other hand, the coproduct bifunctor in **Set** does not preserve colimits; it is not the case, for example, that $X + (Y + Z) \cong (X + Y) + (X + Z)$. The important point to note is that Day convolution can be instantiated using *any* monoidal structure on the source category $\mathfrak{L}$, but requires a very particular sort of monoidal structure on the target category $\mathfrak{S}$.

Definition 4.2.8. Given the above conditions, the Day convolution product of $F, G : \mathfrak{L}^{\mathrm{op}} \Rightarrow \mathfrak{S}$ is given by the coend

$$(F \circledast G)\ L = \exists L_F, L_G.\ F\ L_F \otimes G\ L_G \otimes (L \Rightarrow_{\mathfrak{L}} L_F \oplus L_G).$$

Remark. Since groupoids are self-dual, we may ignore the $-^{\mathrm{op}}$ in the common case that $\mathfrak{L}$ is a groupoid. Note that $F\ L_F$ and $G\ L_G$ are objects in $\mathfrak{S}$, and $(L \Rightarrow_{\mathfrak{L}} L_F \oplus L_G)$ is a hom set in $\mathfrak{L}$, viewed as an object in $\mathfrak{S}$ as well.

This operation is associative, and has as a unit $j(I)$, where I is the unit for $\oplus$ and $j : \mathfrak{L} \to (\mathfrak{L}^{\mathrm{op}} \Rightarrow \mathfrak{S})$ is the co-Yoneda embedding, that is, $j(L) = (- \Rightarrow_{\mathfrak{L}} L)$. See Kelly [2005] for proof.

Example. Let's begin by looking at the traditional setting of $\mathfrak{L} = \mathbf{B}$ and $\mathfrak{S} = \mathbf{Set}$. As noted in §1.4.5, $\mathbf{B}$ has a monoidal structure given by disjoint union of finite sets. $\mathbf{B}$ is indeed enriched over $\mathbf{Set}$, which is also cocomplete and has an appropriate symmetric monoidal structure given by Cartesian product.

Specializing the definition to this case, we obtain

$$(F \cdot G)\ L = \exists L_F, L_G.\ F\ L_F \times G\ L_G \times (L \xrightarrow{\sim} L_F \uplus L_G).$$

We can simplify this further by characterizing the coend more explicitly. Let

$$R := \biguplus_{L_F, L_G}\ F\ L_F \times G\ L_G \times (L \xrightarrow{\sim} L_F \uplus L_G).$$

Elements of R look like quintuples (L_F, L_G, f, g, i), where $f \in F\ L_F$, $g \in G\ L_G$, and $i : L \xrightarrow{\sim} L_F \uplus L_G$ witnesses a partition of L into two subsets. Then, as we have seen, the coend can be expressed as a quotient $R\,/\sim$, where every pair of bijections $(\sigma_F : L_F \xrightarrow{\sim} L'_F, \sigma_G : L_G \xrightarrow{\sim} L'_G)$ induces an equivalence of the form

$$(L_F, L_G, f, g, i) \sim (L'_F,\ L'_G,\ F\ \sigma_F\ f,\ G\ \sigma_G\ g,\ i\,;(\sigma_F \uplus \sigma_G)).$$

That is, $f \in F\ L_F$ is sent to $F\ \sigma_F\ f$ (the relabelling of f by σ_F); $g \in G\ L_G$ is sent to $G\ \sigma_G\ g$; and $i : L \xrightarrow{\sim} L_F \uplus L_G$ is sent to

$$L \xrightarrow[i]{\sim} L_F \uplus L_G \xrightarrow[\sigma_F \uplus \sigma_G]{\sim} L'_F \uplus L'_G\ .$$

When are two elements of R *inequivalent*, that is, when can we be certain two elements of R are not related by a pair of relabellings? Two elements $(L_{F1}, L_{G2}, f_1, g_1, i_1)$ and $(L_{F2}, L_{G2}, f_2, g_2, i_2)$ of R are unrelated if and only if

- f_1 and f_2 have different forms, that is, they are unrelated by any relabelling, or
- g_1 and g_2 have different forms, or

- L_{F1} and L_{G1} "sit inside" L differently than L_{F2} and L_{G2} in L_2, that is, $i_1^{-1}(L_{F1}) \neq i_2^{-1}(L_{F2})$.

(Note they are also unrelated if there is no bijection $L_{F1} \xrightarrow{\sim} L_{F2}$ or no bijection $L_{G1} \xrightarrow{\sim} L_{G2}$, but in those cases one of the first two bullets above would hold as well.) The first two bullets are immediate; the third follows since a pair of relabellings can permute the elements of L_F and L_G arbitrarily, or replace L_F and L_G with any other sets of the same size, but relabelling alone can never change which elements of L correspond to L_F and which to L_G, since that is preserved by composition with a coproduct bijection $\sigma_F \uplus \sigma_G$.

Therefore, all the equivalence classes of $R \,/\sim$ can be represented canonically by a partition of L into two disjoint subsets, along with a choice of F and G structures, giving rise to the earlier definition:

$$(F \cdot G)\ L = \biguplus_{L_F, L_G \vdash L} F\ L_F \times G\ L_G. \tag{4.2.2}$$

Also, in this case, the identity element is $j(I) = j(\varnothing) = \mathbf{B}(-, \varnothing)$, that is, the species which takes as input a label set L and constructs the set of bijections between L and the empty set. Clearly there is exactly one such bijection when $L = \varnothing$, and none otherwise: as expected, this is the species $\mathsf{1}$ defined in the previous section.

Example. Although $\mathbf{B}$ and $\mathbf{P}$ are equivalent, it is still instructive to work out the general definition in the case of $\mathbf{P}$, particularly because, as we have seen, proving $\mathbf{B} \cong \mathbf{P}$ requires the axiom of choice.

We find that $\mathbf{P}$ has not just one but *many* monoidal structures corresponding to disjoint union. The action of such a monoid on objects of $\mathbf{P}$ is clear: the natural numbers m and n are sent to their sum $m + n$. For the action on morphisms, we are given $\sigma : (\mathsf{Fin}\ m)!$ and $\tau : (\mathsf{Fin}\ n)!$ and have to produce some $(\mathsf{Fin}\ (m+n))!$. However, there are many valid ways to do this. One class of examples arises from considering bijections

$$\varphi : \mathsf{Fin}\ m \uplus \mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ (m+n),$$

which specify how to embed $\{0, \ldots, m-1\}$ and $\{0, \ldots, n-1\}$ into $\{0, \ldots, m+n-1\}$. Given such a φ, we may construct

$$\Omega(\varphi)(\sigma, \tau) := \mathsf{Fin}\ (m+n) \xrightarrow{\varphi^{-1}}_{\sim} \mathsf{Fin}\ m \uplus \mathsf{Fin}\ n \xrightarrow{\sigma \uplus \tau}_{\sim} \mathsf{Fin}\ m \uplus \mathsf{Fin}\ n \xrightarrow{\varphi}_{\sim} \mathsf{Fin}\ (m+n),$$

as illustrated in Figure 4.11. (Note that conjugating by φ is essential for the functoriality of the result; picking some other bijection in place of, say, φ^{-1}, would result in a permutation that sent $\sigma = \tau = id$ to something other than the identity.)

Remark. Although it is not essential to what follows, we note that the Ω defined above, which sends bijections $\varphi : \mathsf{Fin}\ m \uplus \mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ (m+n)$ to functorial maps

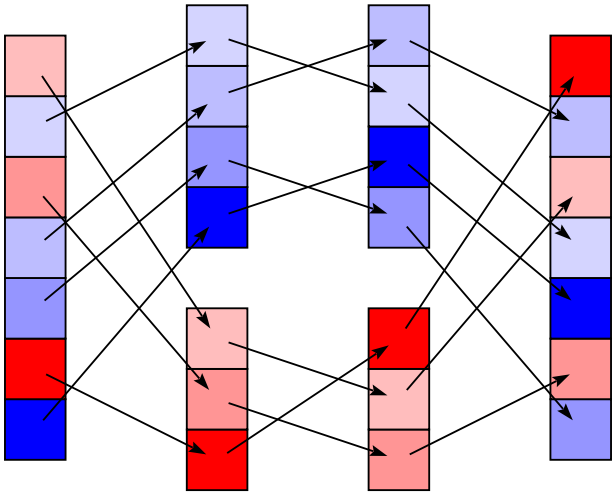


Figure 4.11: $\mathsf{Fin}\ (m+n) \xrightarrow{\sim} \mathsf{Fin}\ m \uplus \mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ m \uplus \mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ (m+n)$

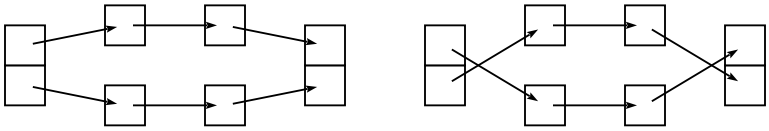


Figure 4.12: Distinct choices of φ that result in identical permutations f

$(\mathsf{Fin}\ m)! \to (\mathsf{Fin}\ n)! \to (\mathsf{Fin}\ (m+n))!$, is neither injective nor surjective. It is not injective since, for example, with $m = n = 1$, there are two distinct inhabitants of $\mathsf{Fin}\ 2 \xrightarrow{\sim} \mathsf{Fin}\ 1 + \mathsf{Fin}\ 1$, but both give rise to the same function $(\mathsf{Fin}\ 1)! \to (\mathsf{Fin}\ 1)! \to (\mathsf{Fin}\ 2)!$ (Figure 4.12), namely, the one which constantly returns the identity permutation (which, indeed, is the only such function which is functorial).

Neither is Ω surjective. Consider the case where $m = n = 2$, and the function $f : (\mathsf{Fin}\ 2)! \to (\mathsf{Fin}\ 2)! \to (\mathsf{Fin}\ 4)!$ given by the table:

	id	(12)
id	id	$(12)(34)$
(12)	(12)	(34)

It is not hard to verify that f is functorial; for example, $f\ id\ (12)\ ;\ f\ (12)\ id = (12)(34)\ ;\ (12) = (34) = f\ (12)\ (12)$. However, we will show that f cannot be of the form $f\ \sigma\ \tau = \varphi \circ (\sigma \uplus \tau) \circ \varphi^{-1}$ for any φ.

For a permutation ψ, denote by $\mathrm{Fix}(\psi) = \{x \mid \psi(x) = x\}$ the set of fixed points of ψ, and by $\mathrm{fix}(\psi) = \#\,\mathrm{Fix}(\psi)$ the number of fixed points. Note that $\mathrm{fix}(\sigma \uplus \tau) = \mathrm{fix}(\sigma) + \mathrm{fix}(\tau)$, since if some value $\mathsf{inl}\ s$ is fixed by $\sigma \uplus \tau$, then s must be fixed by σ, and conversely (and similarly for inr and τ). We also note the following lemma:

Lemma 4.2.9. fix *is invariant under conjugation; that is, for any permutations* ψ *and* φ *we have* $\mathrm{fix}(\psi) = \mathrm{fix}(\varphi \circ \psi \circ \varphi^{-1})$.

Proof. Calculate as follows:

$$
\begin{array}{ll}
& \psi(x) = x \\
\leftrightarrow & \qquad \{ \quad \text{apply } \psi^{-1} \text{ to both sides} \quad \} \\
& x = \psi^{-1}(x) \\
\leftrightarrow & \qquad \{ \quad \text{apply } \varphi \circ \psi \circ \varphi^{-1} \circ \varphi \text{ to both sides} \quad \} \\
& \varphi(\psi(\varphi^{-1}(\varphi(s)))) = \varphi(\psi(\varphi^{-1}(\varphi(\psi^{-1}(s))))) \\
\leftrightarrow & \qquad \{ \quad \text{simplify} \quad \} \\
& (\varphi \circ \psi \circ \varphi^{-1})(\varphi(s)) = \varphi(s).
\end{array}
$$

Thus φ is a bijection between the fixed points of ψ and those of $\varphi \circ \psi \circ \varphi^{-1}$. ∎

If $f\ \sigma\ \tau$ is of the form $\varphi \circ (\sigma \uplus \tau) \circ \varphi^{-1}$, we therefore have

$$\mathrm{fix}(f\ \sigma\ \tau) = \mathrm{fix}(\varphi \circ (\sigma \uplus \tau) \circ \varphi^{-1}) = \mathrm{fix}(\sigma \uplus \tau) = \mathrm{fix}(\sigma) + \mathrm{fix}(\tau).$$

However, the f exhibited in the table above does not satisfy this equality: in particular,

$$\mathrm{fix}(f\ id\ (12)) = \mathrm{fix}((12)(34)) = 0 \neq 2 = \mathrm{fix}(id) + \mathrm{fix}((12)).$$

We may now instantiate the definition of Day convolution (for some particular choice of monoid in $\mathbf{P}$), obtaining

$$(F \cdot G)\ n = \exists n_F, n_G.\ F\ n_F \times G\ n_G \times (\mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ (n_F + n_G)).$$

Again, letting $R := \biguplus_{n_F, n_G} F_{n_F} \times G_{n_G} \times (\mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ (n_F + n_G))$, the coend is equivalent to $R\,/\sim$, where

$$(n_F, n_G, f, g, i) \sim (n_F, n_G,\ F\ \sigma_F\ f,\ G\ \sigma_G\ g,\ i\,;(\sigma_F +_\varphi \sigma_G))$$

for any $\sigma_F : (\mathsf{Fin}\ n_F)!$ and $\sigma_G : (\mathsf{Fin}\ n_G)!$. Note that the meaning of $\sigma_F + \sigma_G$ depends on the particular monoid we have chosen, which fixes an interpretation of $\mathsf{Fin}\ (m+n)$ as representing a disjoint union.

Unlike in the case of $\mathbf{B} \Rightarrow \mathbf{Set}$, we cannot really simplify this any further. In particular, it is *not* equivalent to

$$\biguplus_{n_F + n_G = n} F\ n_F \times G\ n_G,$$

since that does not take into account the different ways the overall set of labels could be distributed between F and G—that is, it throws away the information contained in the bijection $\mathsf{Fin}\ n \xrightarrow{\sim} \mathsf{Fin}\ (n_F + n_G)$. The reason we could "get rid of" the bijection

in (4.2.2) is that the bijection is secretly encoded in the partition $L_F, L_G \vdash L$. In contrast, $n_F + n_G = n$ says nothing about the relationship of the actual labels, but only about the sizes of the label sets.

Example. There is another monoidal structure on $\mathbf{B}$ corresponding to the Cartesian product of sets. If we instantiate the framework of Day convolution with this product-like monoidal structure—but keep everything else the same, in particular continuing to use products on **Set**—we obtain the arithmetic product.

That is,

$$(F \boxtimes G)\ L = \exists L_F, L_G.\ F\ L_F \times G\ L_G \times (L \xrightarrow{\sim} L_F \times L_G).$$

By a similar argument to the one used above, this is equivalent to

$$\biguplus_{L_F, L_G \vdash L} F\ L_F \times G L_G.$$

In this case we also have $j(I) = j(\{\star\}) = \mathbf{B}(-, \{\star\})$, the species which constructs all bijections between the label set and $\{\star\}$. There is only one such bijection whenever the label set is of size 1 and none otherwise, so this is equivalent to the species X, as expected.

Example. We now verify that $\mathcal{B}$ and $\mathcal{S}$ have the right properties, so that partitional and arithmetic product are well defined on $(\mathcal{B} \Rightarrow \mathcal{S})$-species.

- As with $\mathbf{B}$, there are monoidal structures on $\mathcal{B}$ corresponding to the coproduct and product of types. Note that when combining two finite types, their finiteness evidence must be somehow combined to create evidence for the finiteness of their product or coproduct. For example, given equivalences $A \simeq \mathsf{Fin}\ m$ and $B \simeq \mathsf{Fin}\ n$, one must create an equivalence $A + B \simeq \mathsf{Fin}\ (m + n)$ (in the case of coproduct) or $A \times B \simeq \mathsf{Fin}\ (mn)$ (in the case of product). In the first case, this can be accomplished by combining the given equivalences with an equivalence $\mathsf{Fin}\ m + \mathsf{Fin}\ n \simeq \mathsf{Fin}\ (m + n)$, which can be implemented, say, by matching the elements of $\mathsf{Fin}\ m$ with the first m elements of $\mathsf{Fin}\ (m + n)$, and the elements of $\mathsf{Fin}\ n$ with the remaining n elements. Likewise, $A \times B \simeq \mathsf{Fin}\ (mn)$ can be implemented via an equivalence $\mathsf{Fin}\ m \times \mathsf{Fin}\ n \simeq \mathsf{Fin}\ (mn)$, *e.g.* the one which sends (i, j) to $in + j$. Fundamentally, there are many ways to implement such equivalences, but since everything is wrapped in a propositional truncation it does not ultimately matter, as long as there is *some* way to implement them.

- $\mathcal{B}$ can indeed be seen as enriched over $\mathcal{S}$, since morphisms in $\mathcal{B}$ are paths, which are equivalent to paths between the underlying sets, and because by Corollary 2.4.2, $A = B$ is a set when A and B are.

- We have already seen that there is a symmetric monoidal structure on $\mathcal{S}$ given by the product of types, which does preserve colimits.

- Finally, $\mathcal{S}$ does have coends over $\mathcal{B}$. In fact, since $\mathcal{B}$ is a groupoid, recall from §2.2.2 that coends are just Σ-types.

Given $F, G : \mathcal{B} \Rightarrow \mathcal{S}$, and picking the monoid corresponding to coproduct on $\mathcal{B}$, we can instantiate the definition of Day convolution to get

$$(F \cdot G)\ L = \sum_{L_F, L_G} F\ L_F \times G\ L_G \times (L = L_F + L_G).$$

That is, a value of type $(F \cdot G)\ L$ consists of a choice of finite types L_F and L_G, an F-shape and a G-shape, labelled by L_F and L_G respectively, and a path between L and $L_F + L_G$.

4.3 Composition

We have already seen that arithmetic product can be thought of as a restricted sort of composition, where an F-structure contains G-structures all of the same shape (or vice versa). More generally, there is an unrestricted version of composition, where $(F \circ G)$-shapes consist of F-shapes containing *arbitrary* G-shapes. That is, intuitively, to create an $(F \circ G)$-shape over a given set of labels L, we first *partition* L into subsets; create a G-shape over each subset; then create an F-shape over the resulting set of G-shapes.

4.3.1 Definition and examples

We begin with the formal definition of Bergeron et al. [1998]:

$$(F \circ G)\ L = \sum_{\pi \vdash L} F\ \pi \times \prod_{p \in \pi} G\ p \tag{4.3.1}$$

(there are some subtle issues with this definition, to be discussed shortly, but it will suffice to consider the general idea and some examples). Figure 4.13 shows an abstract representation of the definition, in which labels are shown as dots, and shapes are represented abstractly as arcs drawn across edges leading to the labels contained in the shape, identified by the name of a species. So the diagram illustrates how an $(F \circ G)$-shape consists abstractly of a top-level F-shape with subordinate G-shapes.

Example. Figure 4.14 shows a concrete example of a $(\mathsf{B} \circ \mathsf{L}_+)$-shape, a binary tree containing nonempty lists.

Example. As an example, we may define the species Par of *set partitions*, illustrated in Figure 4.15, by

$$\mathsf{Par} = \mathsf{E} \circ \mathsf{E}_+.$$

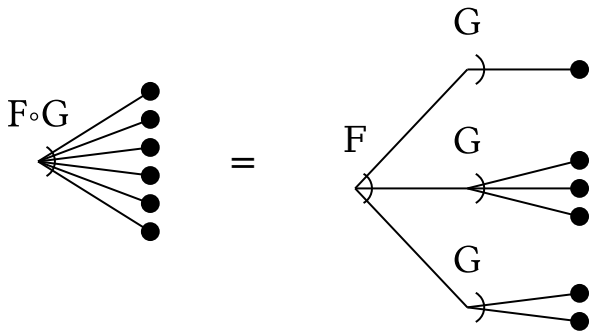


Figure 4.13: Generic species composition

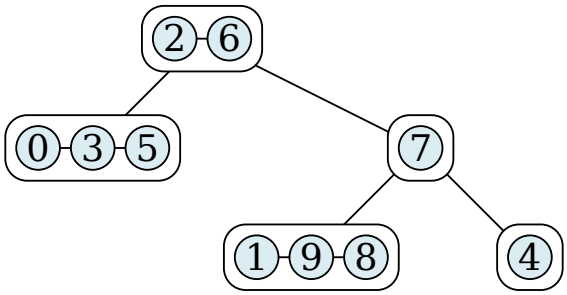


Figure 4.14: An example $(\mathsf{B} \circ \mathsf{L}_+)$-shape

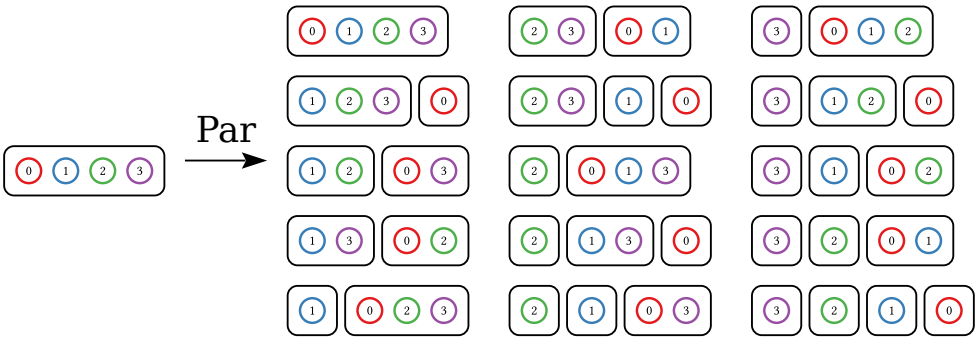


Figure 4.15: The species Par of partitions

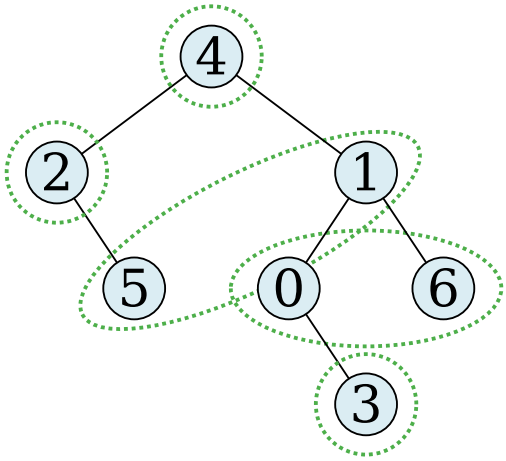


Figure 4.16: An example $(\mathsf{B} \times \mathsf{Par})$-shape

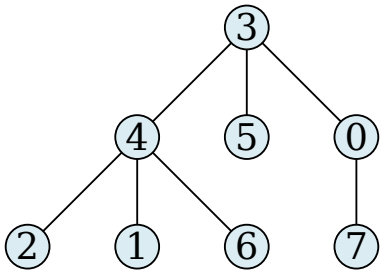


Figure 4.17: An example R-shape

That is, a set partition is a set of *non-empty* sets. Similarly, the species S of permutations is given by $\mathsf{S} = \mathsf{E} \circ \mathsf{C}$, a set of *cycles*.

Given the species Par, we may define the species $\mathsf{B} \times \mathsf{Par}$ of *partitioned trees*. Structures of this species are labeled binary tree shapes with a superimposed partitioning of the labels (as illustrated in Figure 4.16), and can be used to model trees containing data elements with decidable equality; the partition indicates equivalence classes of elements.

Example. The species R of nonempty n-ary ("rose") trees, with data stored at internal nodes, may be defined by the recursive species equation

$$\mathsf{R} = \mathsf{X} \cdot (\mathsf{L} \circ \mathsf{R}).$$

An example R-shape is shown in Figure 4.17. Note the use of L means the children of each node are linearly ordered. Using E in place of L yields a more graph-theoretic notion of a rooted tree, with no structure imposed on the neighbors of a particular node.

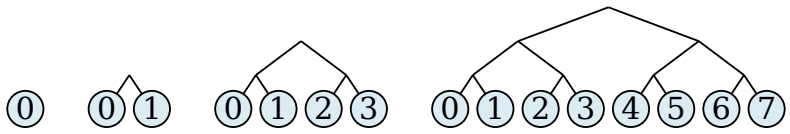


Figure 4.18: Example P-shapes

$\mathsf{Plan} = \mathsf{X} \cdot (\mathsf{C} \circ \mathsf{R})$ is the species of *planar embeddings* of rooted trees, where the top-level subtrees of the root are ordered cyclically. Each node other than the root, on the other hand, still has a linear order on its children, fixed by the distinguished edge from the node towards the root.

Example. The species P of *perfect trees*, with data stored in the leaves, may be defined by

$$\mathsf{P} = \mathsf{X} + (\mathsf{P} \circ \mathsf{X}^2).$$

That is, a perfect tree is either a single node, or a perfect tree containing *pairs*. Functional programmers will recognize this as a *non-regular* or *nested* recursive type; it corresponds to the Haskell type:

$$\textbf{data}\ \mathsf{P}\ a = \mathsf{Leaf}\ a \mid \mathsf{Branch}\ (\mathsf{P}\ (a, a))$$

Figure 4.18 illustrates some example P-shapes.

In addition to being the identity for $\boxtimes$, X is the (two-sided) identity for $\circ$ as well. We have

$$(\mathsf{X} \circ G)\ L = \sum_{\pi \vdash L} \mathsf{X}\ \pi \times \prod_{p \in \pi} G\ p,$$

in which $\mathsf{X}\ \pi$ is $\varnothing$ (cancelling the summands in which it occurs) except in the case where π is the singleton partition $\{L\}$, in which case the summand is isomorphic to $G\ L$. On the other side,

$$(F \circ \mathsf{X})\ L = \sum_{\pi \vdash L} F\ \pi \times \prod_{p \in \pi} \mathsf{X}\ p;$$

the only way to get a product in which none of the $\mathsf{X}\ p$ are $\varnothing$ is when $\pi \cong L$ is the complete partition of L into singleton subsets, in which case we again have something isomorphic to $F\ L$.

As for generating functions, the mapping from species to egfs is indeed homomorphic with respect to composition:

$$(F \circ G)(x) = F(G(x)).$$

A direct combinatorial proof can be given, making use of *Faà di Bruno's formula* [Johnson, 2002].

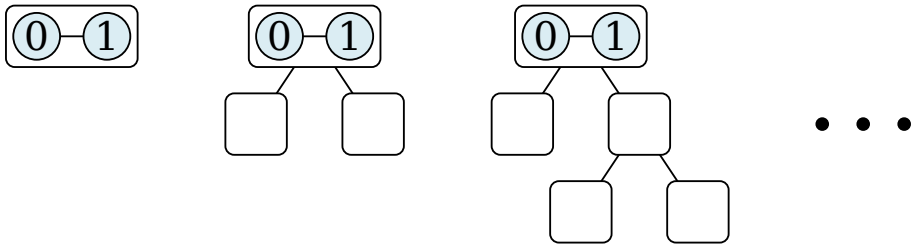


Figure 4.19: An infinite family of $(\mathsf{B} \circ \mathsf{L})$-shapes of size 2

On the other hand, in general,

$$\widetilde{(F \circ G)}(x) \neq \widetilde{F}(\widetilde{G}(x)).$$

Bergeron et al. [1998, Exercise 1.4.3] pose the specific counterexample of $\widetilde{\mathsf{S}}(x) \neq \widetilde{\mathsf{E}}(\widetilde{\mathsf{C}}(x))$, which is not hard to show (hint: $\widetilde{\mathsf{S}}(x) = \prod_{k \geqslant 1} \frac{1}{1-x^k}$ and $\widetilde{\mathsf{C}}(x) = x + x^2 + x^3 + \cdots = \frac{x}{1-x}$). A more intuitive explanation of the failure of ogfs to be homomorphic with respect to composition—along with a characterization of the situations when homomorphism does hold—is left to future work. In any case, to compute ogfs for composed species, one may turn to *cycle index series*, which can be seen as a generalization of both egfs and ogfs, and which retain more information than either; see Bergeron et al. [1998, §1.2, §1.4] for details.

As hinted previously, the formal definition of composition given in (4.3.1) requires additional qualification; in particular, it requires delicate treatment with regard to partitions and infinite families of shapes. To see the issue, let B and L be the species of binary trees and lists. Consider the species $\mathsf{B} \circ \mathsf{L}$, whose shapes should consist of binary trees containing lists at their nodes. Intuitively, this gives rise to infinite families of shapes such as those illustrated in Figure 4.19, which are all of size 2.

There are several possible reactions to Figure 4.19, depending on the exact setting in which we are working.

- If we are working in $(\mathbf{B} \Rightarrow \mathbf{Set})$, where the set of shapes on a given set of labels may be infinite, then this should be allowed, and is exactly the meaning that composition ought to have in this case. Note, however, that this means we would need to allow $\pi \vdash L$ to include "partitions" with arbitrary numbers of *empty* parts (to correspond to the empty lists). Typically, the notation $\pi \vdash L$ denotes partitions into *nonempty* parts.

- On the other hand, if we are working in $(\mathbf{B} \Rightarrow \mathbf{FinSet})$, as is more traditional in a combinatorial setting, this *must not* be allowed. One possibility would be to simply insist that $\pi \vdash L$ in (4.3.1) excludes partitions with empty parts, as is usual. But as we have just seen, this does not generalize nicely to $(\mathbf{B} \Rightarrow \mathbf{Set})$, and in any case it would still be a bit strange, since, for example, $\mathsf{B} \circ \mathsf{L}$ and

$\mathsf{B} \circ \mathsf{L}_+$ would "silently" end up being the same. It is better to front the issue by simply insisting that $F \circ G$ is only defined when $G\ \varnothing = \varnothing$.

We can reformulate the definition of composition in a better way which naturally allows for empty parts and which also makes for a clearer path to a generalized notion of composition (to be discussed in the next section). In fact, Joyal [1981, p. 11] already mentions this as an alternative definition. The idea is to use another finite set P, representing parts of a partition, and a function $\chi : L \to P$ which assigns each $l \in L$ to some $p \in P$. The fibers of χ, *i.e.* the sets $\chi^{-1}(p)$ for $p \in P$, thus constitute a partition of L. Note, however, that this naturally allows for empty parts, since χ may not be surjective. We then say that an $(F \circ G)$-shape on the labels L consists of some set P, a partition function $\chi : L \to P$, an F-shape on P, and G-shapes on the fibers of χ. However, we must also quotient out by bijections between P and other finite sets; the precise identity of P should not matter. We can thus define $(F \circ G)$ using a coend:

$$(F \circ G)\ L = \exists P \in \mathbf{B}.\ \sum_{\chi : L \to P} F\ P \times \prod_{p \in P} G\ (\chi^{-1}\ p). \tag{4.3.2}$$

In the case that $G\ \varnothing = \varnothing$ is required, only surjective χ will result in shapes, so the coend reduces to the original definition (4.3.1), using the notation $\pi \vdash L$ with its usual meaning of a partition into nonempty parts.

4.3.2 Generalized composition

We first show how to carry out the definition of composition in $\mathbf{B} \Rightarrow \mathbf{Set}$ even more abstractly, then discuss how it may be generalized to other functor categories $\mathfrak{L} \Rightarrow \mathfrak{S}$. Street [2012] gives the following abstract definition of composition:

$$(F \circ G)\ L = \exists K.\ F\ K \times G^{\#K}\ L, \tag{4.3.3}$$

where $G^n = \underbrace{G \cdot \cdots \cdot G}_{n}$ is the n-fold partitional product of G. Intuitively, this corresponds to a top-level F-shape on labels drawn from the "internal" label set K, paired with $\#K$-many G-shapes, with the labels from L partitioned among all the G-shapes. The coend abstracts over K, ensuring that the precise choice of "internal" labels does not matter up to isomorphism.

Remark. Note how this corresponds to the second definition of composition given in (4.3.2). In particular, binary partitional product allows for the labels to be partitioned into the empty subset and the entire subset, so an iterated partitional product corresponds to partitions which contain an arbitrary number of empty parts.

However, this definition is somewhat puzzling from a constructive point of view, since it would seem that $G^{\#K}$ retains no information about which element of K

corresponds to which G-shape in the product. The problem boils down, again, to the use of the axiom of choice. For each finite set K we may choose some ordering $K \xrightarrow{\sim} [\#K]$; this ordering then dictates how to match up the elements of K with the G-shapes in the product $G^{\#K}$. More formally, given a species G we can define the anafunctor $G^- : \mathbf{B} \to \mathbf{Spe}$ which sends each finite set K to the clique of $(\#K)$-ary products of G, with the morphisms in the clique corresponding to permutations (since **Spe** is symmetric monoidal with respect to partitional product). This then becomes a regular functor in the presence of the axiom of choice.

In the particular case of $\mathbf{B} \Rightarrow \mathbf{Set}$, we can also avoid the axiom of choice by using a more explicit construction (again due to Street[3]). For a finite set K and category $\mathbb{C}$, recall that we may represent a K-indexed tuple of objects of $\mathbb{C}$ by a functor $K \to \mathbb{C}$ (where K is considered as a discrete category). It's important to note that this "K-tuple" has no inherent ordering (unless K itself has one)—it simply assigns an object of $\mathbb{C}$ to each element of K. Denote by $\Delta_K : \mathbb{C} \to \mathbb{C}^K$ the diagonal functor which sends an object $C \in \mathbb{C}$ to the K-tuple containing only copies of C.

Consider $\mathbb{C} = \mathbf{FinSet}$. Given any discrete category K, the diagonal functor $\Delta_K : \mathbf{FinSet} \to \mathbf{FinSet}^K$ has both a left and right adjoint, which we call Σ_K and Π_K:

$$\Sigma_K \dashv \Delta_K \dashv \Pi_K.$$

In particular, $\Sigma_K : \mathbf{FinSet}^K \to \mathbf{FinSet}$ constructs K-indexed coproducts, and Π_K constructs indexed products. (In the special case $K = |\mathbf{2}|$, $\Sigma_{|\mathbf{2}|}$ and $\Pi_{|\mathbf{2}|}$ resolve to the familiar notions of disjoint union and Cartesian product of finite sets, respectively.) One can see this by considering the expansion of the adjoint relations as natural isomorphisms between hom-sets. For example, in the case of Π_K, we have

$$(\Delta_K\ A \to T) \cong (A \to \Pi_K\ T)$$

where $A \in \mathbf{FinSet}$ and $T \in \mathbf{FinSet}^K$. Essentially this expands to something like

$$(A \to T_1) \times \cdots \times (A \to T_n) \cong (A \to \Pi_K\ T),$$

and it is easy to see that in order for the isomorphism to hold, we should have $\Pi_K\ T = T_1 \times \cdots \times T_N$. (In general, of course, K need not have some associated indexing $1 \ldots n$, but the same argument can be generalized.) We often omit the subscripts, writing simply Σ and Π when K is clear from the context.

Now consider $\mathbb{C} = \mathbf{B}$. Δ_K does not have adjoints in $\mathbf{B}$; in fact, categorical products and coproducts can be exactly characterized as adjoints to $\Delta_{|\mathbf{2}|}$, and we have already seen that $\mathbf{B}$ does not have categorical products or coproducts. However, we can take $\Pi_K, \Sigma_K : \mathbf{FinSet}^K \to \mathbf{FinSet}$ and restrict them to functors $\mathbf{B}^K \to \mathbf{B}$. This is well-defined since **FinSet** and $\mathbf{B}$ have the same objects, and Π_K and Σ_K produce only isomorphisms when applied to isomorphisms. For example, if $\alpha : A \xrightarrow{\sim} A'$,

[3]Personal communication, 6 March 2014.

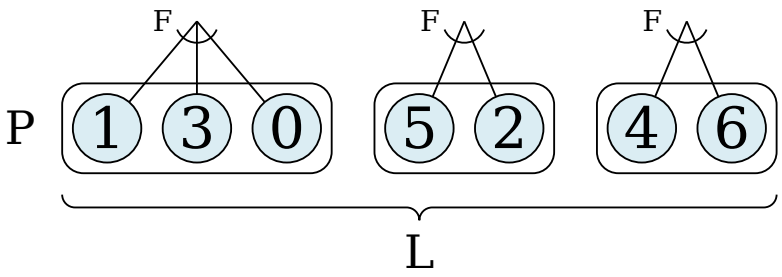


Figure 4.20: Indexed species product

$\beta : B \xrightarrow{\sim} B'$, and $\gamma : C \xrightarrow{\sim} C'$, then $\Pi_{|\mathbf{3}|}(\alpha, \beta, \gamma)$ is the isomorphism $\alpha \times \beta \times \gamma : A \times B \times C \xrightarrow{\sim} A' \times B' \times C'$.

We can now define a general notion of indexed species product. For a species $F : \mathbf{B} \Rightarrow \mathbf{Set}$ and $K \in \mathbf{B}$ a finite set, $F^K : \mathbf{B} \Rightarrow \mathbf{Set}$ represents the $\#K$-fold partitional product of F, indexed by the elements of K (see Figure 4.20):

$$F^K\ L = \exists (P : \mathbf{B}^K).\ \mathbf{B}(\Sigma P, L) \times \Pi(F \circ P).$$

Note that K is regarded as a discrete category, so $P \in \mathbf{B}^K$ is a K-indexed collection of finite sets. $\mathbf{B}(\Sigma P, L)$, a bijection between the coproduct of P and L, witnesses the fact that P represents a partition of L; the coend means there is only one shape per fundamentally distinct partition. The composite $F \circ P = \ K \xrightarrow{P} \mathbf{B} \xrightarrow{F} \mathbf{Set}$ is a K-indexed collection of F-structures, one on each finite set of labels in P; the Π constructs their product.

It is important to note that this is functorial in K: the action on a morphism $\sigma : K \xrightarrow{\sim} K'$ is to appropriately compose σ with P.

The composition $F \circ G$ can now be defined by

$$(F \circ G)\ L = \exists K.\ F\ K \times G^K\ L.$$

This is identical to the definition given in (4.3.3), except that $G^{\#K}$ has been replaced by G^K, which explicitly records a mapping from elements of K to G-shapes.

This explicit construction relies on a number of specific properties of $\mathbf{B}$ and $\mathbf{Set}$, and it is unclear how it should generalize to other functor categories. Fortunately, in the particular case of $\mathcal{B} \Rightarrow \mathcal{S}$, in HoTT, this more complex construction is not necessary. The anafunctor $G^- : \mathbf{B} \to \mathbf{Spe}$ discussed earlier corresponds in HoTT to a regular functor $G^- : \mathcal{B} \to (\mathcal{B} \Rightarrow \mathcal{S})$: in a symmetric monoidal category, the $(\#K)$-ary tensor product of G is unique up to isomorphism, which in an h-category corresponds to actual equality.

More generally, if we focus on the high-level definition

$$(F \circ G)\ L = \exists K.\ F\ K \times G^K\ L,$$

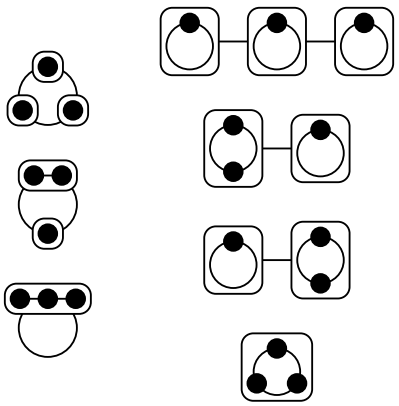


Figure 4.21: $(\mathsf{C} \circ \mathsf{L})$- and $(\mathsf{L} \circ \mathsf{C})$-forms of size 3

leaving the definition of G^K abstract, we can enumerate the properties required of a general functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$ to accommodate it: for starters, $\mathfrak{S}$ must have coends over $\mathfrak{L}$, and $(\mathfrak{S}, \times)$ must be monoidal. We can also say that, whatever the definition of G^K, it will involve partitional product—so we must add in all the requirements for that operation, enumerated in §4.2.3. In fact, this already covers the requirements of $\mathfrak{S}$ having coends and a monoid $\times$, so any functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$ which supports partitional product already supports composition as well.

For a formal proof that composition is associative, see Kelly [2005, pp. 5–6], although some reflection on the intuitive idea of composition should be enough to convince informally: for example, a tree which contains cycles-of-lists is the same thing as a tree-of-cycles containing lists.

Unlike the other monoidal structures on **Spe** (sum and Cartesian, arithmetic, and partitional product), composition is not symmetric. For example, as illustrated in Figure 4.21, there are different numbers of $(\mathsf{C} \circ \mathsf{L})$-forms and $(\mathsf{L} \circ \mathsf{C})$-forms of size 3, and hence $\mathsf{C} \circ \mathsf{L} \ncong \mathsf{L} \circ \mathsf{C}$.

Proposition 4.3.1. $(\mathbf{Spe}, \circ, \mathsf{X})$ *is monoidal.*

Proof. We have already seen that $\circ$ is associative and that X is an identity for composition. For formal proofs in a more generalized setting see, again, Kelly [2005]. ∎

Like associativity, the right-distributivity laws

$$(F + G) \circ H \cong (F \circ H) + (G \circ H)$$
$$(F \cdot G) \circ H \cong (F \circ H) \cdot (G \circ H)$$

are easy to grasp on an intuitive level. Their formal proofs are not too difficult; the second specifically requires an isomorphism $G^{K_1+K_2} \cong G^{K_1} \cdot G^{K_2}$, which ought to hold no matter what the definition of G^K. The reader may also enjoy discovering why the corresponding left-distributivity laws are false (although they do correspond to species *morphisms* rather than isomorphisms).

4.3.3 Internal Hom for composition

We have seen that $(\mathbf{Spe}, \circ, \mathsf{X})$ is monoidal; in this section we show that it is monoidal closed. Indeed, we can compute as follows (essentially the same computation also appears in Kelly [2005, p. 7], though in a more general form):

$$
\begin{array}{ll}
& F \circ G \Rightarrow_{\mathbf{Spe}} H \\
\cong & \qquad \{ \quad \text{natural transformations are ends} \quad \} \\
& \forall L.\ (F \circ G)\ L \Rightarrow_{\mathbf{Set}} H\ L \\
\cong & \qquad \{ \quad \text{definition of } \circ \quad \} \\
& \forall L.\ (\exists K.\ F\ K \times G^K\ L) \Rightarrow_{\mathbf{Set}} H\ L \\
\cong & \qquad \{ \quad (- \Rightarrow_{\mathbf{Set}} H\ L) \text{ turns colimits into limits} \quad \} \\
& \forall L.\ \forall K.\ (F\ K \times G^K\ L) \Rightarrow_{\mathbf{Set}} H\ L \\
\cong & \qquad \{ \quad \text{currying} \quad \} \\
& \forall L.\ \forall K.\ F\ K \Rightarrow_{\mathbf{Set}} (G^K\ L \Rightarrow_{\mathbf{Set}} H\ L) \\
\cong & \qquad \{ \quad (F\ K \Rightarrow_{\mathbf{Set}} -) \text{ preserves limits} \quad \} \\
& \forall K.\ F\ K \Rightarrow_{\mathbf{Set}} \forall L.\ (G^K\ L \Rightarrow_{\mathbf{Set}} H\ L) \\
\cong & \qquad \{ \quad \text{natural transformations are ends} \quad \} \\
& \forall K.\ F\ K \Rightarrow_{\mathbf{Set}} (G^K \Rightarrow_{\mathbf{Spe}} H) \\
\cong & \qquad \{ \quad \text{natural transformations are ends} \quad \} \\
& F \Rightarrow_{\mathbf{Spe}} (G^- \Rightarrow_{\mathbf{Spe}} H)
\end{array}
$$

Thus we have the adjunction

$$(F \circ G \Rightarrow_{\mathbf{Spe}} H) \cong (F \Rightarrow_{\mathbf{Spe}} (G \Rightarrow_{\circ} H)),$$

where

$$(G \Rightarrow_{\circ} H) := (G^- \Rightarrow_{\mathbf{Spe}} H)$$

is the species whose K-labelled shapes are species morphisms from G^K to H. An illustrated example is shown in Figure 4.22: a species morphism from a binary tree of cycles to a rose tree is equivalent to a species morphism that takes the underlying tree shape on the label set K and produces another species morphism, which itself expects a K-indexed partitional product of cycles and produces a rose tree. One can see how the composition is decomposed into its constituent parts, with a new label type K introduced to mediate the relationship between them.

4.4 Functor composition

There is a more direct variant of composition, known as *functor composition* [Bergeron et al., 1998, Décoste et al., 1992]. When species are defined as endofunctors $\mathbf{B} \to \mathbf{B}$, the functor composition $F \,\square\, G$ can literally be defined as the composition of F and

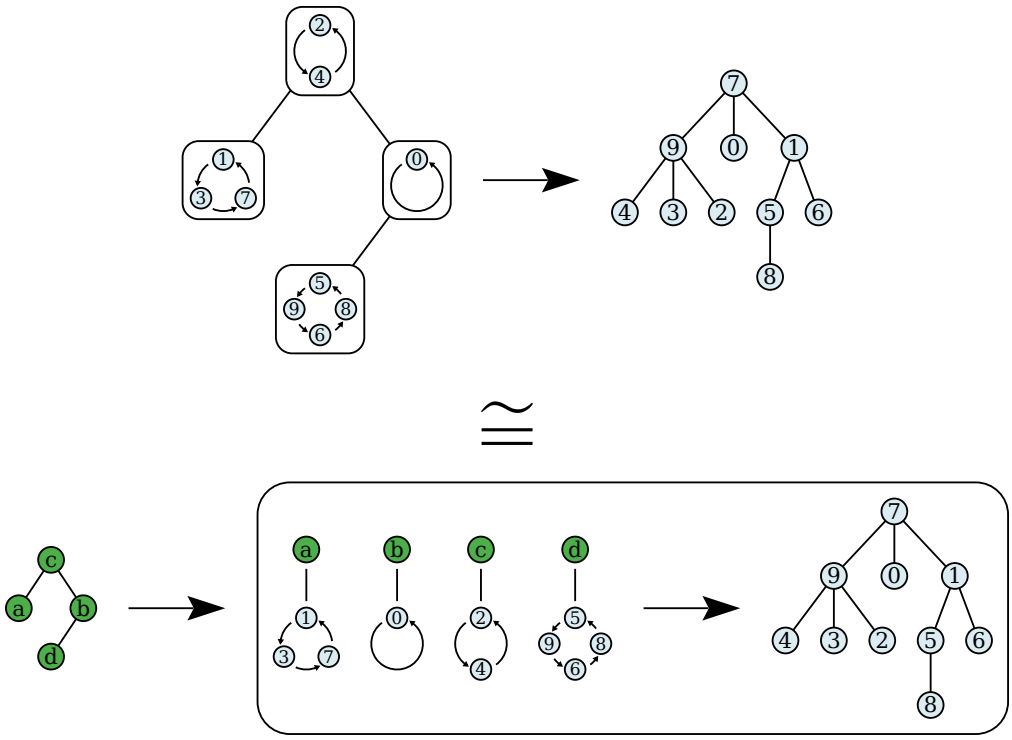


Figure 4.22: Internal Hom for composition

G as functors, that is,

$$(F \,\square\, G)\ L = F\ (G\ L).$$

However, if species are viewed as functors $\mathbf{B} \to \mathbf{Set}$ then this operation is not well-typed as stated, and indeed feels somewhat unnatural. This problem is discussed further in §6.3. For the most part, incorporating functor composition into this framework is left to future work, but it is worth describing briefly here.

An $(F \,\square\, G)$-shape on the set of labels L can intuitively be thought of as consisting of *all possible* G-shapes on the labels L, with an F-shape on this collection of G-shapes as labels. Functor composition thus has a similar relationship to partitional composition as Cartesian product has to partitional product. With partitional product, the labels are partitioned into two disjoint sets, whereas with Cartesian product the labels are shared. With partitional composition, the labels are partitioned into (any number of) subsets with a G-shape on each; with functor composition, the labels are shared among *all possible* G-shapes.

Remark. There is no standard operation which directly creates an F-shape on only *some* G-shapes, with the labels L shared among them. To accomplish this one can simply use $(F \cdot \mathsf{E}) \,\square\, G$.

Example. The species of simple, directed graphs can be described by

$$(\mathsf{E} \cdot \mathsf{E}) \,\square\, (\mathsf{X}^2 \cdot \mathsf{E}).$$

Each $(\mathsf{X}^2 \cdot \mathsf{E})$-shape applied to the same set of labels L picks out an ordered pair of distinct labels, which can be thought of as a directed edge. Taking the functor composition with $(\mathsf{E} \cdot \mathsf{E})$ thus picks out a subset of the total collection of possible directed edges.

A number of variants are also possible. For example, to allow self-loops, one can replace X^2 by $(\mathsf{X} + \mathsf{X}^2)$; to use undirected instead of directed edges, one can replace X^2 by E_2; and so on.

4.5 Differentiation

The derivative of container types is a notion already familiar to many functional programmers through the work of Huet [1997], McBride [2001, 2008] and Abbott et al. [2003b]: the derivative of a type is its type of "one-hole contexts". For example, Figure 4.23 shows a B'-shape, where B is the species of rooted binary trees; there is a "hole" in a place where a label would normally be.

This section begins by presenting the formal definition of derivative for species, along with some examples (§4.5.1). Some related notions such as up and down operators (§4.5.2) and pointing (§4.5.3) follow. The basic notion of differentiation does not generalize nicely to other functor categories, but this is rectified by a more general notion of higher derivatives, of which the usual notion of derivative is a special case

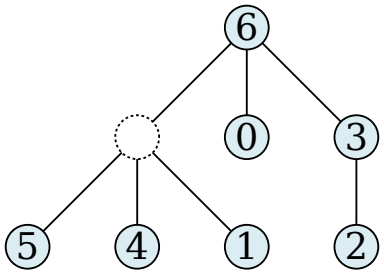


Figure 4.23: An example B'-shape

(§4.5.4). Finally, this notion of higher derivatives paves the way for discussing the internal Hom functors for partitional and arithmetic product (§4.5.5).

There is much more that can be said about differentiation [Menni, 2008, Labelle and Lamathe, 2009, Piponi, 2010b,a, Stay, 2014, McBride, 2012]; in general, there seems to remain a great deal of rich material on differentiation waiting to be explored.

4.5.1 Differentiation in $\mathbf{B} \Rightarrow \mathbf{Set}$

Formally, we create a "hole" by adjoining a new distinguished label to the existing set of labels:

Definition 4.5.1. The *derivative* F' of a species F is defined by

$$F'\ L = F\ (L \uplus \{\star\}).$$

The transport of relabellings along the derivative is defined as expected, leaving the distinguished label alone and transporting the others.

In other words, an F'-shape on the set of labels L is an F-shape on L plus one additional distinguished label. It is therefore slightly misleading to draw the distinguished extra label as an indistinct "hole", as in Figure 4.23, since, for example, taking the derivative twice results in two *different, distinguishable* holes. But thinking of "holes" is still a good intuition for most purposes.

Example. Denote by $\mathfrak{a}$ the species of *unrooted* trees, *i.e.* trees in the pure graph-theoretic sense of a collection of vertices and unoriented edges with no cycles. Also let $\mathcal{A} = \mathsf{X} \cdot (\mathsf{E} \circ \mathcal{A})$ denote the species of rooted trees (where each node can have any number of children, which are unordered). It is difficult to get a direct algebraic handle on $\mathfrak{a}$; however, we have the relation

$$\mathfrak{a}' \cong \mathsf{E} \circ \mathcal{A},$$

since an unrooted tree with a hole in it is equivalent to the set of all the subtrees connected to the hole (Figure 4.24). Note that the subtrees connected to the hole

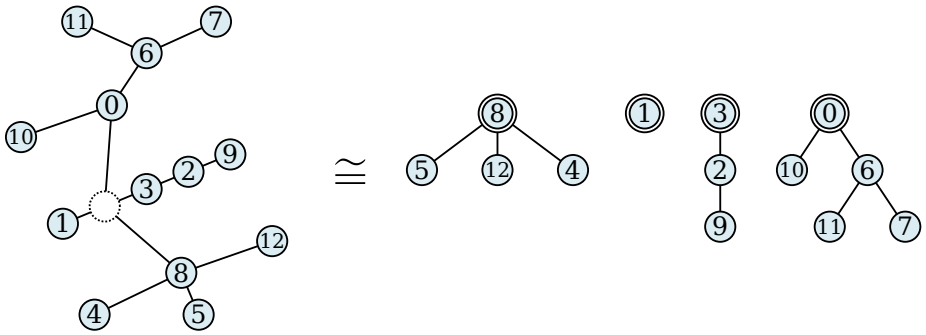


Figure 4.24: $\mathfrak{a}' \cong \mathsf{E} \circ \mathcal{A}$

become *rooted* trees; their root is distinguished by virtue of being the node adjacent to the hole.

Example. $\mathsf{C}' \cong \mathsf{L}$, as illustrated in Figure 4.25.

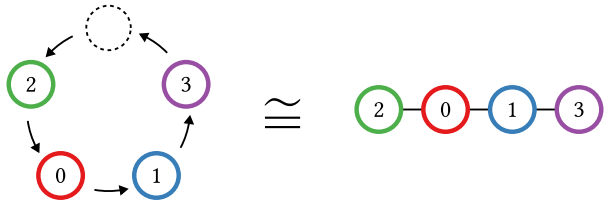


Figure 4.25: $\mathsf{C}' \cong \mathsf{L}$

Example. $\mathsf{L}' \cong \mathsf{L}^2$, as illustrated in Figure 4.26.

Example. Well-scoped terms of the (untyped) lambda calculus may be represented as shapes of the species

$$\Lambda = \varepsilon + \Lambda^2 + \Lambda'.$$

Recall that $\varepsilon = \mathsf{X} \cdot \mathsf{E}$ is the species of elements. (This example appears implicitly—without an explicit connection to species—in the work of Altenkirch et al. [2010], and earlier also in that of Altenkirch and Reus [1999] and Fiore et al. [2003].) Labels

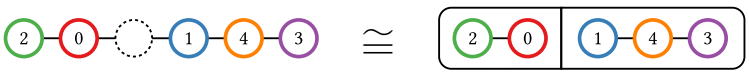


Figure 4.26: $\mathsf{L}' \cong \mathsf{L}^2$

correspond to (free) variables, that is, the elements of $\Lambda\ V$ are well-scoped lambda calculus terms with free variables taken from the set V. The above equation for Λ can thus be interpreted as saying that a lambda calculus term with free variables in V is either

- an element of V, *i.e.* a variable,
- a pair of terms (application), or
- a lambda abstraction, represented by a term with free variables taken from the set $V \uplus \{\star\}$. The new variable $\star$ of course represents the variable bound by the abstraction.

The set of *closed* terms is thus given by $\Lambda\ \varnothing$. Note that there are infinitely many terms with any given number of free variables, so this is not useful for doing *combinatorics*; as an equation of generating functions, $\Lambda(x) = \varepsilon(x) + \Lambda^2(x) + \Lambda'(x)$ has no solution. To do combinatorics with lambda terms one must also count applications and abstractions as contributing to the size, *e.g.* using a two-sort species (§5.4) such as

$$\Xi = \mathsf{X} \cdot \mathsf{E} + \mathsf{Y} \cdot \Xi^2 + \mathsf{Y} \cdot \frac{\partial}{\partial \mathsf{X}} \Xi$$

which uses labels of sort Y to mark occurrences of applications and abstractions. For a similar approach see Grygiel and Lescanne [2013], Lescanne [2013].

The operation of species differentiation obeys laws which are familiar from calculus:

$$\begin{gathered}
1' \cong 0 \\
\mathsf{X}' \cong 1 \\
\mathsf{E}' \cong \mathsf{E} \\
(F + G)' \cong F' + G' \\
(F \cdot G)' \cong F' \cdot G + F \cdot G' \\
(F \circ G)' \cong (F' \circ G) \cdot G'
\end{gathered}$$

The reader may enjoy working out *combinatorial* interpretations of these laws.

In addition, differentiation of species corresponds to differentiation of exponential generating functions, as one might hope. We have

$$\begin{aligned}\frac{d}{dx}(F(x)) &= \frac{d}{dx}\left(\sum_{n\geqslant 0} f_n \frac{x^n}{n!}\right) \\ &= \sum_{n\geqslant 1} f_n \frac{x^{n-1}}{(n-1)!} \\ &= \sum_{n\geqslant 0} f_{n+1} \frac{x^n}{n!} \\ &= \left(\frac{d}{dx}F\right)(x),\end{aligned}$$

since by definition the number of (F')-shapes of size n is indeed equal to f_{n+1}, the number of F-shapes on $n+1$ labels.

Unfortunately, once again

$$\widetilde{(F')}(x) \neq \widetilde{F}'(x)$$

in general, though a corresponding equation does hold for cycle index series, which may be used to compute the ogf for a species defined via differentiation.

4.5.2 Up and down operators

Aguiar and Mahajan [2010, §8.12] define *up* and *down operators* on species; although the import or usefulness of up and down operators is not yet clear to me, my instinct tells me that they will indeed have important roles to play, so I include a brief discussion of them here.

Definition 4.5.2. An *up operator* on a species F is a species morphism $u : F \to F'$.

Since a species morphism is a natural, label-preserving map, an up operator must essentially "add" an extra "hole" somewhere in a shape. (Of course it can also rearrange existing labels, as long as it does so in a natural way that does not depend on the identity of the labels at all.)

Example. The species E of sets has a trivial up operator which sends the unique set shape on L to the unique set shape on $L \uplus \{\star\}$ (Figure 4.27).

Example. The species L of linear orders has an up operator which adds a hole in the leftmost position (Figure 4.28). There is a similar operator which adds a hole in the rightmost position. In fact, there are many other examples (particularly since species maps are allowed to do something completely different at every size), but these are two of the most apparent.

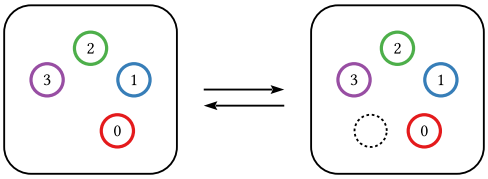


Figure 4.27: The trivial up and down operators on E

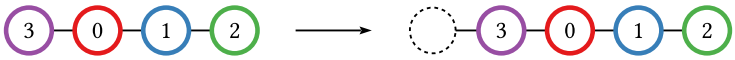


Figure 4.28: An up operator on L

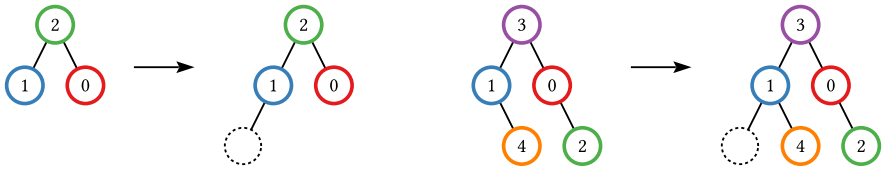


Figure 4.29: An up operator on B

Figure 4.30: A down operator on C

Example. We can similarly make an up operator for the species B of binary trees, which adds a hole as the leftmost (or rightmost) leaf (Figure 4.29).

Example. The species C of cycles, on the other hand, has no up operator. Recall that $\mathsf{C}' = \mathsf{L}$; there is no way to define a *natural* map $\varphi : \mathsf{C} \to \mathsf{L}$. As a counterexample, consider

$$\begin{array}{ccc} \mathsf{C}\ \mathbf{2} & \xrightarrow{\varphi_{\mathbf{2}}} & \mathsf{L}\ \mathbf{2} \\ {\scriptstyle \mathsf{C}\ \sigma}\downarrow & & \downarrow{\scriptstyle \mathsf{L}\ \sigma} \\ \mathsf{C}\ \mathbf{2} & \xrightarrow[\varphi_{\mathbf{2}}]{} & \mathsf{L}\ \mathbf{2} \end{array}$$

where $\mathbf{2} = \{0, 1\}$ is a two-element set, and $\sigma : \mathbf{2} \xrightarrow{\sim} \mathbf{2}$ is the permutation that swaps 0 and 1. The problem is that $C\ \sigma$ is the identity on $\mathsf{C}\ \mathbf{2}$, but $\mathsf{L}\ \sigma$ is not the identity on $\mathsf{L}\ \mathbf{2}$, so this square cannot possibly commute.

Generalizing from this example, one intuitively expects that there is no up operator whenever taking the derivative breaks some symmetry, as in the case of C. Formalizing this intuitive observation is left to future work.

Down operators are defined dually, as one would expect:

Definition 4.5.3. A *down operator* on a species F is a species morphism $d : F' \to F$.

Example. Again, E has a trivial down operator, which is the inverse of its up operator.

Example. Although we saw previously that the species C of cycles has no up operator, it has an immediately apparent down operator, namely, the natural map $\mathsf{C}' \to \mathsf{C}$ which removes the hole from a cycle, that is, which glues together the two ends of a list.

Example. The species L of linear orders also has an apparent down operator, which simply removes the hole.

Remark. Aguiar and Mahajan [2010, p. 275, Example 8.56] define a down operator on L which removes the hole *if* it is in the leftmost position, and "sends the order to 0" otherwise. However, this seems bogus. First of all, it is not clear what is meant by 0 in this context; assuming it denotes the empty list, it is not well-typed, since species morphisms must be label-preserving.

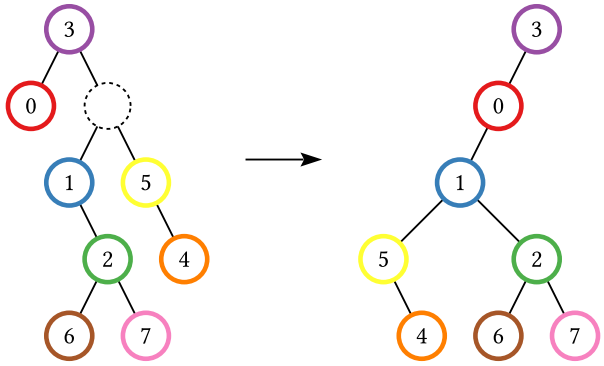


Figure 4.31: An example down operator on B, via stacking

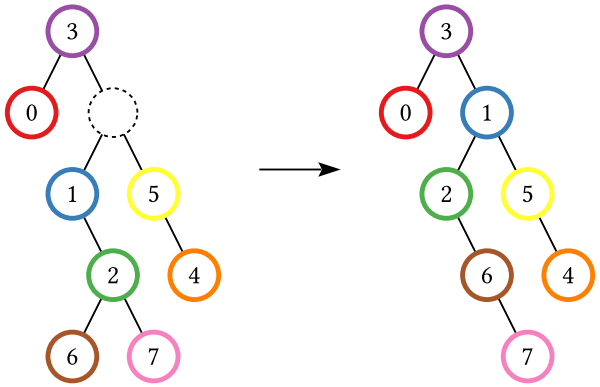


Figure 4.32: An example down operator on B, via promotion

Example. It takes a bit more imagination, but it is not too hard to come up with examples of down operators for the species B of binary trees. For example, the two subtrees beneath the hole can be "stacked", with the first subtree added as the leftmost leaf of the remaining tree, and the other subtree added as *its* leftmost leaf (Figure 4.31), or nodes could be iteratively promoted to fill the hole, say, preferring the left-hand node when one is available (Figure 4.32).

These operators are somewhat reminiscent of deletion from data structures such as binary search trees or heaps. Those algorithms rely on a linear order on the labels, and hence do not qualify as natural species morphisms. However, they do indeed qualify as down operators on the $\mathbf{L}$-species of binary search trees and heaps, respectively (see §2.4.3 and §5.5).

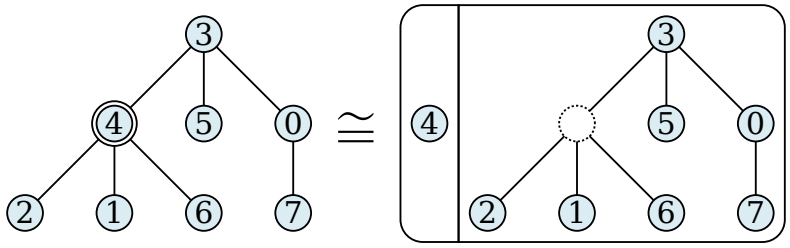


Figure 4.33: Species pointing

4.5.3 Pointing

Definition 4.5.4. The operation of *pointing* can be defined in terms of the species of elements, $\varepsilon = \mathsf{X} \cdot \mathsf{E}$, and Cartesian product:

$$F^\bullet = \varepsilon \times F.$$

As illustrated on the left-hand side of Figure 4.33, an $F^\bullet$-structure can be thought of as an F-structure with one particular distinguished element.

As is also illustrated in Figure 4.33, pointing can also be expressed in terms of differentiation,

$$F^\bullet \cong \mathsf{X} \cdot F'.$$

Similar laws hold for pointing as for differentiation; they are left for the reader to discover.

4.5.4 Higher derivatives

Aguiar and Mahajan [2010, §8.11] describe a generalization of species derivatives to "higher derivatives". The idea of higher derivatives in the context of functions of a single variable should be familiar: the usual derivative is the *first* derivative, and by iterating this operation, one obtains notions of the second, third, ... derivatives. More abstractly, we generalize from a single notion of "derivative", f', to a whole family of higher derivatives $f^{(n)}$, parameterized by a *natural number* n.

Note that taking the derivative of a polynomial reduces the degree of all its terms by one. More generally, the nth derivative reduces the degrees by n. According to the correspondence between species and generating functions, the *degrees* of terms in a generating function correspond to the *sizes* of label sets. Recall that the general principle of the passage from generating functions to species is to replace natural number sizes by finite sets of labels having those sizes. Accordingly, just as higher derivatives of generating functions are parameterized by a natural number which acts on the degree, higher derivatives of species are parameterized by a finite set which acts on the labels.

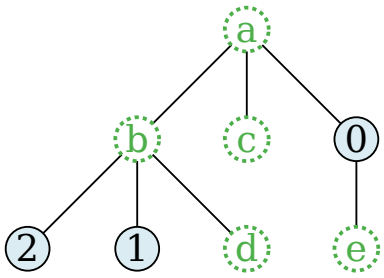


Figure 4.34: An example $\mathsf{B}^{(K)}$-shape

Definition 4.5.5. For a species F and finite set K, the K-derivative of F is defined by

$$F^{(K)}\ L = F\ (K \uplus L).$$

As should be clear from the above discussion, the exponential generating function corresponding to the K-derivative of F is

$$(F^{(K)})(x) = F^{(\#K)}(x),$$

i.e. the $(\#K)$-th derivative of F. Note that we recover the simple derivative of F by setting $K = \{\star\}$. Note also that $F^{(\varnothing)} = F$.

An $F^{(K)}$-shape with labels L is an F-shape populated by both L *and* K. The occurrences of labels from K can be thought of as "K-indexed holes", since they do not contribute to the size. For example, an "$F^{(K)}$-shape of size 3" consists of an F-shape with three labels that "count" towards the size, as well as one "hole" for each element of K. Figure 4.34 illustrates a $\mathsf{B}^{(K)}$-shape of size 3, where $K = \{a, b, c, d, e\}$.

Higher derivatives generalize easily to any functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$ where $(\mathfrak{L}, \oplus, I)$ is monoidal; we simply define

$$F^{(K)}\ L := F\ (K \oplus L).$$

4.5.5 Internal Hom for partitional and arithmetic product

As promised, we now return to consider the existence of an internal Hom functor corresponding to partitional product. We are looking for some

$$- \Rightarrow_\bullet - : \mathbf{Spe}^{\mathrm{op}} \times \mathbf{Spe} \to \mathbf{Spe}$$

for which

$$(F \cdot G \Rightarrow_{\mathbf{Spe}} H) \cong (F \Rightarrow_{\mathbf{Spe}} (G \Rightarrow_\bullet H)). \tag{4.5.1}$$

Intuitively, this is just like currying—although there are labels to contend with which make things more interesting.

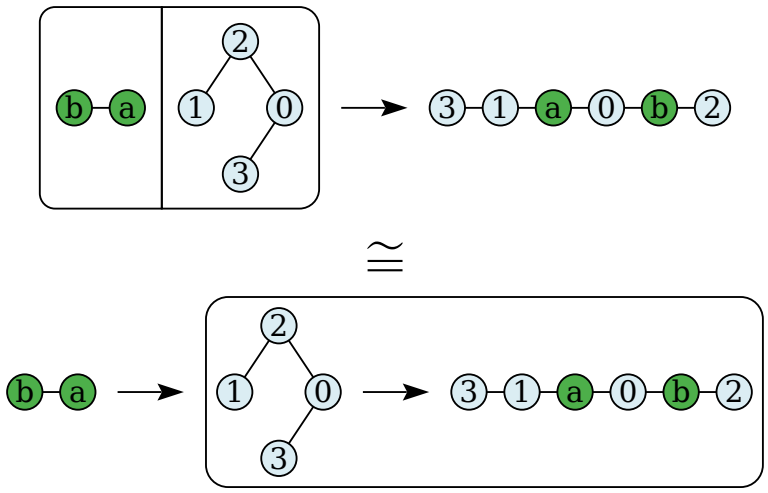


Figure 4.35: "Currying" for partitional product of species

Recall that an $(F \cdot G)$-shape on L is a partition $L_1 \uplus L_2 = L$ together with shapes from F L_1 and G L_2. Another way of saying this is that an $(F \cdot G)$-shape consists of an F-shape and a G-shape on two different sets of labels, whose disjoint union constitutes the label set for the entire product shape. Thus, a morphism out of $F \cdot G$ should be a morphism out of F, which produces another morphism that expects a G and produces an H on the disjoint union of the label sets from the F- and G-shapes.

This can be formalized using the notion of higher derivatives developed in the previous subsection. In particular, define $- \Rightarrow_\bullet -$ by

$$(G \Rightarrow_\bullet H)\ L := G \Rightarrow_{\mathbf{Spe}} H^{(L)}.$$

That is, a $(G \Rightarrow_\bullet H)$-shape with labels taken from L is a species morphism, *i.e.* a natural, label-preserving map, from G to the L-derivative of H. This definition is worth rereading a few times since it mixes levels in an initially surprising way—the *shapes* of the species $G \Rightarrow_\bullet H$ are *species morphisms* between other species. However, this should not ultimately be too surprising to anyone used to the idea of higher-order functions; it corresponds to the idea that functions can output other functions.

Thus, a $(G \Rightarrow_\bullet H)$-shape with labels from L is a natural function that takes a G-shape as input and produces an H-shape which uses the disjoint union of L and the labels from G. This is precisely what is needed to effectively curry a species morphism out of a product while properly keeping track of the labels, as illustrated in Figure 4.35. The top row of the diagram illustrates a particular instance of a species morphism from $\mathsf{L} \cdot \mathsf{B}$ to L. The bottom row shows the "curried" form, with a species morphism that sends a list to another species morphism, which in turn sends a tree to a higher derivative of a list, containing holes corresponding to the original list.

Formally, we have the adjunction (4.5.1). The same result appears in Kelly [2005]

in a slightly different guise.

This result hints at a close relationship between partitional product and higher derivatives. In particular, both are defined using the *same* monoidal structure on $\mathbf{B}$ (the one corresponding to disjoint union of finite sets), and this gives rise to the fundamental Leibniz-like law relating the two,

$$(F \cdot G)^{(L)} = \sum_{J \uplus K = L} F^{(J)} \cdot G^{(K)}.$$

Setting $L = \{\star\}$ yields the familiar product rule for differentiation,

$$(F \cdot G)' = F' \cdot G + F \cdot G',$$

since there are only two possibilities for J and K given $J \uplus K = \{\star\}$. This generalizes to functor categories other than $\mathbf{B} \Rightarrow \mathbf{Set}$: any functor category which supports a Day convolution product also has a corresponding notion of higher derivatives, and a corresponding Leibniz law.

This also suggests considering an alternate sort of higher derivative, based on the other monoidal structure on $\mathbf{B}$ (corresponding to Cartesian product of finite sets), and thus related to arithmetic product rather than partitional product. In particular, we define the *arithmetic derivative* by

$$F^{\{K\}}\ L = F\ (K \times L).$$

We have

$$(F \boxtimes G \Rightarrow_{\mathbf{Spe}} H) \cong (F \Rightarrow_{\mathbf{Spe}} (G \Rightarrow_{\boxtimes} H))$$

where

$$(G \Rightarrow_{\boxtimes} H) := (G \Rightarrow_{\mathbf{Spe}} H^{\{L\}}).$$

This is a bit harder to visualize, but works on a similar principle to higher derivative for partitional product. The problem, from a visualization point of view, is that no specific labels correspond to "holes"; an $F^{\{K\}}$-shape with labels taken from L actually has $(\#K)(\#L)$ labels, with an entire K-indexed set of labels corresponding to each element of L. Figure 4.36 illustrates the adjunction: a natural, label-preserving map from an arithmetic product $F \boxtimes G$ to some other species (here a cycle) corresponds to a nested map that takes each of F and G in turn and then produces a species on the product of their labels.

If $F(x) = \sum_{n \geqslant 0} f_n \frac{x^n}{n!}$, then

$$F^{\{K\}}(x) = \sum_{n \geqslant 0} f_{kn} \frac{x^n}{n!},$$

where $k = \#K$; I do not know whether there is a nice way to express this transformation on generating functions.

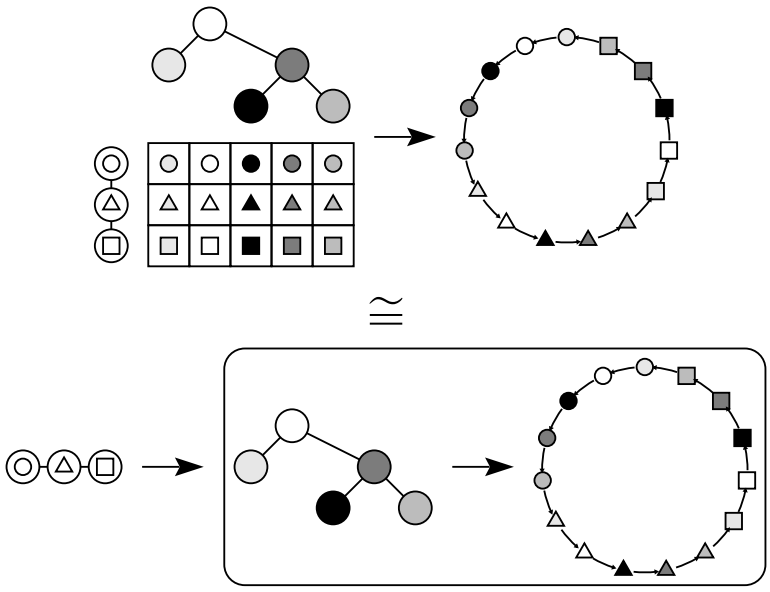


Figure 4.36: "Currying" for arithmetic product of species

4.6 Regular, molecular and atomic species

We now consider the three related notions of *regular*, *molecular*, and *atomic* species. *Regular* species, roughly speaking, are those that correspond to algebraic data types in Haskell or OCaml. A first characterization is as follows:

Definition 4.6.1. The class of *regular* species consists of the smallest class containing $\mathsf{0}$, $\mathsf{1}$, and X, and closed under (countable) sums and products.

There are a few apparent differences between regular species and algebraic data types. First, programming languages do not actually allow *infinite* sums or products. For example, the species

$$\mathsf{X}^2 + \mathsf{X}^3 + \mathsf{X}^5 + \mathsf{X}^7 + \mathsf{X}^{11} + \dots$$

of prime-length lists is a well-defined regular species, but is not expressible as, say, a data type in Haskell[4]. Second, Haskell and OCaml also allow recursive algebraic data types. However, this is not a real difference: the class of regular species is also closed under least fixed points (any implicit recursive definition of a species can in theory be unfolded into an infinite sum or product). Essentially, recursion in algebraic data types can be seen as a tool that allows *some* infinite sums and products to be encoded via finite expressions.

However, there is a more abstract characterization of regular species which does a better job of capturing their essence. We first define the *symmetries* of a structure.

[4]At least not in Haskell 2010.

Recall that $\mathcal{S}_n$ denotes the *symmetric group* of permutations on n elements under composition.

Definition 4.6.2. A permutation $\sigma \in \mathcal{S}_n$ is a *symmetry* of an F-shape $f \in F\ L$ if and only if σ fixes f, that is, $F\ \sigma\ f = f$.

Example. The C-shape in the upper left of Figure 4.37 has the cyclic permutation (01234) as a symmetry, because applying it to the labels results in the same cycle (in the upper right). On the other hand, (12) is not a symmetry; it results in the cycle on the lower left, which is not the same as the original cycle.

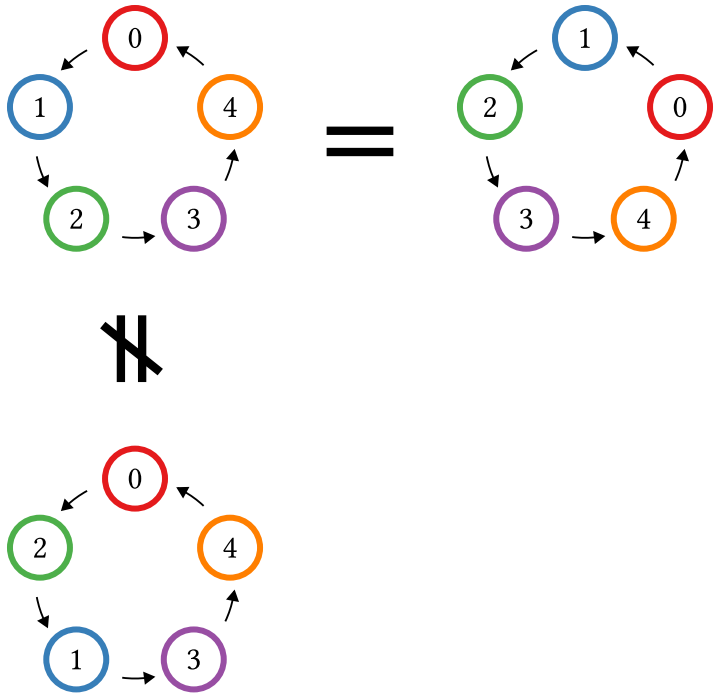


Figure 4.37: A symmetry and a non-symmetry of a C-shape

Example. An L-shape has no nontrivial symmetries: applying any permutation other than the identity to a linear order results in a different linear order.

Example. An E-shape has all possible symmetries: applying any permutation to the labels in a set results in the same set.

We can now state the more abstract definition of regular species; this definition and Definition 4.6.1 turn out to be equivalent.

Definition 4.6.3. A species F is *regular* if every F-shape has the identity permutation as its only symmetry. Such shapes are also called regular.

Example. Regular species include $\mathsf{0}$, $\mathsf{1}$, X, linear orders (L), rooted binary trees (B), and rose trees (R). Non-regular species include sets (E), cycles (C), and any sum or product of a non-regular species with any other species.

Remark. According to this definition, in addition to least fixed points, regular species are also closed under Cartesian and arithmetic product, since the Cartesian or arithmetic product of two regular shapes is also regular. That is, Cartesian and arithmetic product cannot introduce any symmetries if none are present. Given the previous definition, this means that if F and G are two species expressed entirely in terms of $\mathsf{1}$, X, sum, and partitional product, the Cartesian or arithmetic products of F and G are both themselves isomorphic to some species expressed using only those operations, without any mention of Cartesian or arithmetic product. This is certainly far from obvious.

This definition also shows why regular species can be characterized as those for which $F(x) = \widetilde{F}(x)$: with no symmetries, each F-form of size n corresponds to $n!$ distinct labelled shapes. Hence

$$F(x) = \sum_{n \geqslant 0} f_n \frac{x^n}{n!} = \sum_{n \geqslant 0} \widetilde{f}_n n! \frac{x^n}{n!} = \sum_{n \geqslant 0} \widetilde{f}_n x^n = \widetilde{F}(x).$$

That species built from $\mathsf{0}$, $\mathsf{1}$, X, sum, product, and fixed point have no symmetries is not hard to see intuitively; less obvious is the fact that up to isomorphism, every species with no symmetries can be expressed in this way. To understand why this is so, we turn to molecular species.

Definition 4.6.4. A species F is *molecular* if there is only a single F-form, that is, all F-shapes are related by relabelling.

Example. The species X^2 of ordered pairs is molecular, since any two ordered pairs are related by relabelling.

Example. On the other hand, the species L of linear orderings is not molecular, since list structures of different lengths are fundamentally non-isomorphic.

Any two shapes with different numbers of labels are unrelated by relabelling. Thus, any molecular species M necessarily has a *size*, *i.e.* some $n \in \mathbb{N}$ such that all M-shapes have size n. In other words, $M \cong M_n$ (where M_n is the restriction of M to cardinality n; see §3.2.2).

Clearly, any species of the form $F + G$ is not molecular (as long as F and G are not $\mathsf{0}$), since the set of $(F + G)$-forms consists of the disjoint union of the sets of F-forms and G-forms. It turns out that the converse is true as well:

Proposition 4.6.5 (Yeh [1985, 1986]). *The molecular species are precisely those that cannot be decomposed as the sum of two nonzero species.*

Molecular species can be characterized more deeply yet, via *quotient species*. Recall first the definition of a group action:

Definition 4.6.6. An *action* of a group G on a set S is a function

$$- \odot - : G \times S \to S$$

such that, for all $g, h \in G$ and $s \in S$,

- $id \odot s = s$ and
- $g \odot (h \odot s) = (gh) \odot s$.

Remark. Note that this is identical to the definition of a monoid action [Yorgey, 2012]; the only difference is that G has inverses, but no special laws are needed to deal with the interaction of inverses with the action.

If the action function is curried, $G \to (S \to S)$, then the laws state that a group action must be a group homomorphism from G into the symmetric group of bijections on S. Intuitively, a group action can be thought of as describing symmetries of the set S.

Definition 4.6.7. Let H be a group and F a species. H is said to *act naturally* on F if there is a family of group actions

$$\rho_L : H \times F\ L \to F\ L$$

such that the following diagram commutes for all $\sigma : L \xrightarrow{\sim} K$:

$$\begin{array}{ccc} H \times F\ L & \xrightarrow{\rho_L} & F\ L \\ {\scriptstyle id \times F\ \sigma}\downarrow & & \downarrow{\scriptstyle F\ \sigma} \\ H \times F\ K & \xrightarrow[\rho_K]{} & F\ K \end{array}$$

Example. Intuitively, H acts naturally on F if its action does not depend on the particular identity of the labels—that is, if it commutes with relabelling. For example, consider the species L_5, of linear orders on exactly 5 labels, and the cyclic group $\mathbb{Z}_5 = \{0, \ldots, 4\}$ under addition modulo 5. There is an action of $\mathbb{Z}_5$ on L_5, where $n \in \mathbb{Z}_5$ sends the list $a_0, \ldots, a_4$ to the list $a_n, \ldots, a_{n+4}$ (where the indices are taken modulo 5). In other words, $n \in \mathbb{Z}_5$ "rotates" the list n places to the left. It is clear that this is a group action (rotating a list by 0 places is indeed the identity; rotating by m and then by n is the same as rotating by $m + n$). It is also natural: the action does not depend on the identity of the labels, and is fully compatible with relabelling.

Example. More generally, by Cayley's theorem, any group of order n is isomorphic to a group of permutations, and thus has a natural action on $\mathsf{L}_n \cong \mathsf{X}^n$, given by permuting the list elements.

Example. $\mathbb{Z}_2$ has an action on L (the species of linear orders of *any* length) whereby the non-identity element acts on a linear order by reversing it.

Example. $\mathbb{Z}_2$ also acts on $\mathsf{L}_{\geqslant 2}$ by swapping the first two elements, and leaving the rest alone.

Definition 4.6.8 (Quotient species [Labelle, 1985]). Let F be a species and let H act naturally on F. Then define the *quotient species* F/H as the species which sends the finite set of labels L to the set of orbits of $F\ L$ under the action of H.

Put another way, for $f_1, f_2 \in F\ L$, define $f_1 \sim_H f_2$ if there is some $h \in H$ such that $h \odot f_1 = f_2$; this defines an equivalence relation on $F\ L$, and we define $(F/H)\ L$ to be the set of equivalence classes under this equivalence relation.

For a given $\sigma : L \xrightarrow{\sim} K$, we define $(F/H)\ \sigma : (F/H)\ L \to (F/H)\ K$ by

$$F\ \sigma\ [f] := [F\ \sigma\ f].$$

For this to be well-defined, we must show that if $f_1 \sim_H f_2$ then $F\ \sigma\ f_1 \sim_H F\ \sigma\ f_2$. This follows from the naturality of the action of H: if $f_1 \sim_H f_2$, that is, there is some $h \in H$ such that $h \odot f_1 = f_2$, then $h \odot (F\ \sigma\ f_1) = F\ \sigma\ (h \odot f_1) = F\ \sigma\ f_2$, that is, $F\ \sigma\ f_1 \sim_H F\ \sigma\ f_2$. It is also easy to see that F/H inherits the functoriality of F.

Example. Consider again the action of $\mathbb{Z}_5$ on L_5 described previously. Each orbit under the action of $\mathbb{Z}_5$ contains five elements: all the possible cyclic rotations of a given linear order. In fact, $\mathsf{L}_5/\mathbb{Z}_5$ is isomorphic to C_5, the species of size-5 cycles. Each equivalence class of five lists is sent to the unique cycle which results from "gluing" the beginning and end of each list together; conversely, each cycle is sent to the equivalence class consisting of all possible ways of cutting the cycle to obtain a list (Figure 4.38).

Example. Considering the reversing action of $\mathbb{Z}_2$ on L, shapes of the quotient $\mathsf{L}/\mathbb{Z}_2$ consist of (unordered) pairs of lists which are the reverse of each other. This can be thought of as the species of "unoriented lists" (sometimes called the species of *chains*).

Example. Considering the action of $\mathbb{Z}_2$ on $\mathsf{L}_{\geqslant 2}$ which swaps the first two elements, the quotient $\mathsf{L}_{\geqslant 2}/\mathbb{Z}_2$ is isomorphic to the species $\mathsf{E}_2 \cdot \mathsf{L}$ (Figure 4.39).

We can now state the following beautiful result:

Proposition 4.6.9 (Bergeron et al. [1998]). *Every molecular species is isomorphic to X^n/H for some natural number n and some subgroup H of the symmetric group $\mathcal{S}_n$. Moreover, X^n/G and X^n/H are isomorphic if and only if G and H are conjugate (that is, if there exists some $\varphi \in \mathcal{S}_n$ such that $G = \varphi H \varphi^{-1}$).*

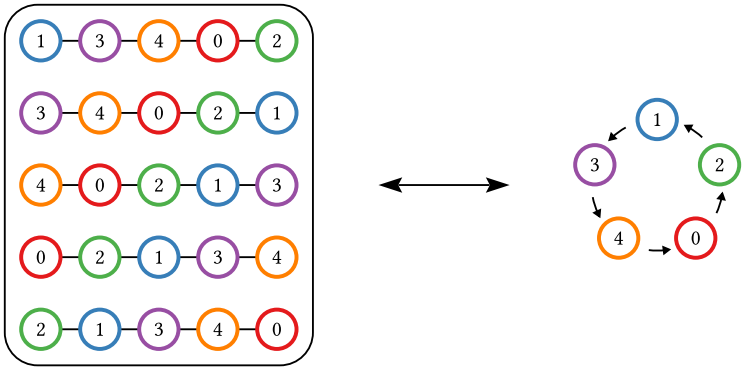


Figure 4.38: Isomorphism between $\mathsf{L}_5/\mathbb{Z}_5$ and C_5

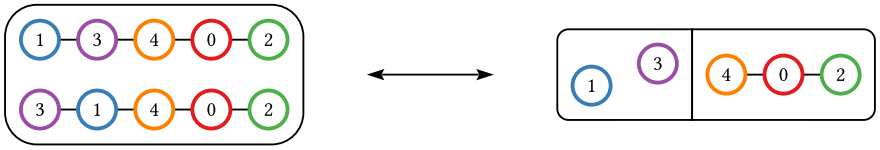


Figure 4.39: Isomorphism between $\mathsf{L}_{\geqslant 2}/\mathbb{Z}_2$ and $E_2 \cdot \mathsf{L}$

In particular, this means that, up to isomorphism, molecular species of size n are in one-to-one correspondence with conjugacy classes of subgroups of $\mathcal{S}_n$. This gives a complete classification of molecular species. For example, it is not hard to verify that there are four conjugacy classes of subgroups of $\mathcal{S}_3$, yielding the four molecular species illustrated in Figure 4.40. The leftmost is X^3, corresponding to the trivial group. The second is $\mathsf{X} \cdot \mathsf{E}_2$, corresponding to the subgroups of $\mathcal{S}_3$ containing only a single swap. The third is C_3, corresponding to $\mathbb{Z}_3$. The last is E_3, corresponding to $\mathcal{S}_3$ itself.

This can in fact be extended to a classification of all species: up to isomorphism, every species has a unique decomposition as a sum of molecular species. As a very simple example, the molecular decomposition of L is

$$\mathsf{L} = 1 + \mathsf{X} + \mathsf{X}^2 + \mathsf{X}^3 + \dots.$$

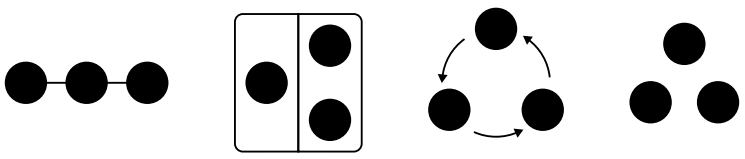


Figure 4.40: The four molecular species of size 3

Bergeron et al. [1998, p. 141] give a more complex example:

$$\mathcal{A} = \mathsf{X} + \mathsf{X}^2 + (\mathsf{X}^3 + \mathsf{X} \cdot \mathsf{E}_2) + (2\mathsf{X}^4 + \mathsf{X}^2 \cdot \mathsf{E}_2 + \mathsf{X} \cdot \mathsf{E}_3) + \ldots$$

Remark. We can now see why the two definitions of regular species given previously are equivalent. Any regular species must be isomorphic to a sum of regular molecular species; but regular molecular species must be of the form X^n. Hence, up to isomorphism, regular species are always of the form $\sum_{n \geqslant 0} a_n \mathsf{X}^n$ with $a_n \in \mathbb{N}$.

The story does not end here, however; molecular species can be decomposed yet further.

Definition 4.6.10. An *atomic* species $F \neq \mathsf{1}$ is one which is indecomposable under partitional product. That is, F is atomic if $F = G \cdot H$ implies $G = \mathsf{1}$ or $H = \mathsf{1}$.

Theorem 4.6.11 (Yeh [1985]). *Every molecular species M can be uniquely decomposed as a product of atomic species*

$$M = A_1^{n_1} \cdot A_2^{n_2} \ldots A_k^{n_k}.$$

Remark. Of course "unique" here means "unique up to isomorphism", which includes reordering of the factors.

Example. Of the four molecular species of size 3 shown in Figure 4.40, only the last two (C_3 and E_3) are atomic. The first two, X^3 and $\mathsf{X} \cdot \mathsf{E}_2$, decompose as the product of three and two atomic species, respectively.

4.7 Species eliminators

With the molecular and atomic decompositions of species under our belt, we now turn to *eliminators* for species. Generally speaking, this is not of much interest to combinatorialists, but it will play an important role in using species as a basis for data types, to be explored in the next chapter.

The idea is to characterize outgoing species morphisms. That is, for a given species F, what can one say about species morphisms $F \Rightarrow G$ for arbitrary species G?

Zero Since $\mathsf{0}\ L = \varnothing$ for all label sets L, there is only a single species morphism $\mathsf{0} \Rightarrow G$ for any species G, namely, the one which consists of the empty function at every size.

One There is only a single $\mathsf{1}$-shape, which has size zero, so a species morphism $\mathsf{1} \Rightarrow G$ is equivalent to a specified inhabitant of $G\ \varnothing$.

Singleton By a similar argument, a species morphism $\mathsf{X} \Rightarrow G$ is equivalent to a specified inhabitant of $G\ \{\star\}$.

Sum Intuitively, a species morphism $(F + G) \Rightarrow H$ should correspond to a pair of morphisms $(F \Rightarrow H) \times (G \Rightarrow H)$, characterizing the morphism in terms of the two possible cases. Indeed, this follows abstractly from the fact that contravariant Hom functors turn colimits (here, a coproduct of species) into limits (here, the product in **Set**). One may also calculate this result more directly by expanding the species morphism as an end and manipulating morphisms in **Set**.

Products Species morphisms out of various types of product (Cartesian, partitional, and arithmetic) have already been characterized in terms of the internal Hom functors for these operations (see §4.1.4 and §4.5.5):

$$\begin{aligned}((F \times G) \Rightarrow H) &\cong (F \Rightarrow H^G)\\ ((F \cdot G) \Rightarrow H) &\cong (F \Rightarrow (G \Rightarrow_\bullet H))\\ ((F \boxtimes G) \Rightarrow H) &\cong (F \Rightarrow (G \Rightarrow_\boxtimes H))\end{aligned}$$

where H^G is defined in §4.1.4, and $G \Rightarrow_\bullet H$ and $G \Rightarrow_\boxtimes H$ are defined in §4.5.5. Note that in all three cases, one may continue to recursively characterize the results in terms of an eliminator for F. It ought to be the case that the internal Hom functors can likewise be characterized in terms of an eliminator for G, although the details are not yet clear to me.

Composition Species morphisms out of compositions have also already been characterized in terms of an internal Hom functor, in §4.3.3:

$$((F \circ G) \Rightarrow H) \cong (F \Rightarrow (G \Rightarrow_\circ H)).$$

Molecular and atomic species We first consider molecular species, which by the discussion in the previous section are equivalent to X^n/H for some $H \subseteq \mathcal{S}_n$. Unfolding definitions,

$$\begin{aligned}&\mathsf{X}^n/H \Rightarrow_{\mathbf{Spe}} G\\ =\ &\{\quad \text{definition} \quad\}\\ &\forall L.\ (\mathsf{X}^n/H)\ L \Rightarrow_{\mathbf{Set}} G\ L\\ =\ &\{\quad \text{definition of } F/H \quad\}\\ &\forall L.\ (\mathsf{X}^n\ L)/{\sim_H} \Rightarrow_{\mathbf{Set}} G\ L\end{aligned}$$

where $f_1 \sim_H f_2$ iff there exists some $\sigma \in H$ such that $\mathsf{X}^n\ \sigma\ f_1 = f_2$. We now note that a function out of a set of equivalence classes can be characterized as a function out

of the underlying set which respects the equivalence relation. That is, the last line of the computation above is isomorphic to

$$(f : \forall L.\ \mathsf{X}^n\ L \Rightarrow_{\mathbf{Set}} G\ L) \times ((\sigma \in H) \to (f = f \circ (\mathsf{X}^n\ \sigma)),$$

i.e. a species morphism $\mathsf{X}^n \Rightarrow G$ paired with a proof that it respects the equivalence induced by H. Note that the same argument applies unchanged to the more general case of a quotient species F/H.

Example. Consider species morphisms $\psi : \mathsf{C}_5 \to \mathsf{B}_5$, from cycles to binary trees of size 5. (You may want to pause to think about what such morphisms could look like.)

As noted earlier, $\mathsf{C}_5 \cong \mathsf{X}^5/\mathbb{Z}_5$. So any $\psi : \mathsf{C}_5 \to \mathsf{B}_5$ is isomorphic to a species morphism $\chi : \mathsf{X}^5 \to \mathsf{B}_5$ together with a proof that $\chi = \chi \circ (\mathsf{X}^5\ \sigma)$ for all $\sigma \in \mathbb{Z}_5$. By naturality of χ, we have $\chi \circ (\mathsf{X}^5\ \sigma) = (\mathsf{B}_5\ \sigma) \circ \chi$, and hence $\chi = (\mathsf{B}_5\ \sigma) \circ \chi$. However, we now see that this was something of a trick "example": since B_5 is regular, $\mathsf{B}_5\ \sigma$ has no fixed points unless $\sigma = id$, and $\mathbb{Z}_5$ certainly contains nontrivial permutations. Therefore no such morphisms $\psi : \mathsf{C}_5 \to \mathsf{B}_5$ can exist!

For molecular species which are not atomic, of course it is possible to decompose them as a product, and characterize morphisms out of them via currying. So in some sense we are only "forced" to use the above characterization of morphisms out of quotient species in the case of atomic species.

Chapter 5

Species variants

One of the goals of the previous chapter was to determine the properties needed to define operations on "variant species", *i.e.* functors in some category $(\mathfrak{L} \Rightarrow \mathfrak{S})$. We have seen a few variants already:

- $\mathbf{B} \Rightarrow \mathbf{Set}$
- $\mathbf{B} \Rightarrow \mathbf{FinSet}$, a more traditional notion of species which is more appropriate for doing combinatorics.
- $\mathbf{P} \Rightarrow \mathbf{Set}$, species as families of shapes organized by *size* instead of by labels.
- $\mathcal{B} \Rightarrow \mathcal{S}$, species as constructed in HoTT.

There are quite a few other possible variants, some of which we explore in this chapter. First, §5.2 and §5.3 develop two novel species variants based on the idea of replacing bijections between labels with injections (or coinjections). Such species variants are conjectured to be especially useful for modelling data structures that takes memory allocation and layout into account. Multisort species (§5.4) are a standard species variant that can also be seen as functors between certain categories other than $\mathbf{B}$ and $\mathbf{Set}$. Multisort species are discussed in some detail, and also enable a discussion of recursive species and the Implicit Species Theorem (§5.4.1). Finally, some other standard species variants (such as $\mathbf{L}$-species) are mentioned in §5.5 and §5.6.

5.1 Generalized species properties

We first gather and summarize the properties needed on $\mathfrak{L}$ and $\mathfrak{S}$ to support the operations we have studied. The data are summarized in Table 5.1. At the head of each column is an operation or group of operations. In general, $\odot$-E denotes the eliminator for $\odot$; in some cases the eliminator for a given operation requires different or additional properties than the operation itself. ∂ indicates (higher) differentiation. The rows are labelled by properties, to be elaborated below.

	$+$, $\times$	$+$-E	$\times$-E	$\cdot$, $\boxtimes$, $\circ$, ∂	$\cdot$-E, $\boxtimes$-E	$\circ$-E
$\mathfrak{S}$ monoidal	✓	✓	✓	✓	✓	✓
... coproduct		✓				
... symm., pres. colimits				✓	✓	✓
... left adjoint						✓
$\mathfrak{L}$ locally small			✓			
$\mathfrak{S}$ complete, Cart. closed			✓			
$\mathfrak{L}$ monoidal				✓	✓	✓
$\mathfrak{L}$ enriched over $\mathfrak{S}$				✓	✓	✓
$\mathfrak{S}$ has coends over $\mathfrak{L}$				✓	✓	✓
$\mathfrak{L} \Rightarrow \mathfrak{S}$ enriched over self					✓	✓

Table 5.1: Properties of $(\mathfrak{L} \Rightarrow \mathfrak{S})$ needed for species operations

- All operations require $\mathfrak{S}$ to be monoidal. Some require additional properties of this monoidal structure:
  - The eliminator for $+$ assumes that it is derived from the actual coproduct in $\mathfrak{S}$.
  - All the operations built on Day convolution or something similar require the monoidal structure on $\mathfrak{S}$ to be symmetric and to preserve colimits. It suffices, but is not necessary, for the monoidal product to be a left adjoint.
  - On the other hand, the eliminator for $\circ$ really does require the monoidal product to be a left adjoint.
- The eliminator for Cartesian product, which corresponds to the Cartesian closure of $(\mathfrak{L} \Rightarrow \mathfrak{S})$, requires that $\mathfrak{L}$ be locally small and $\mathfrak{S}$ complete and Cartesian closed.
- Again, the operators defined via Day convolution and related operators require that $\mathfrak{L}$ be monoidal and enriched over $\mathfrak{S}$, and that $\mathfrak{S}$ have coends over $\mathfrak{L}$.
- Finally, eliminators for partitional and arithmetic product and for composition require $(\mathfrak{L} \Rightarrow \mathfrak{S})$ to be enriched over itself: for example, in the context of $(\mathbf{B} \Rightarrow \mathbf{Set})$ we end up treating a species morphism as itself being a species.

In each of the following sections we describe a particular species variant, identifying the categories $\mathfrak{L}$ and $\mathfrak{S}$, and verifying which of the above properties hold.

5.2 Copartial species

As a larger example, which will also recur in Chapter 6, we develop the theory of species based on injections (and their dual, coinjections, in the subsequent section).

The development will be carried out in HoTT, though it works equally well in set theory.

5.2.1 Copartial bijections

We begin by exploring the notion of a *copartial bijection*, a bijection which is allowed to be partial in the backwards direction, as illustrated in Figure 5.1. Clasically, a copartial bijection is the same as an injection; constructively, we must take care to distinguish them.

Remark. A *bijection* in HoTT is taken to be a pair of inverse functions. Recall that in general, this may not be the same as an *equivalence*, although in the specific case of sets (0-types) the notions of bijection and equivalence do coincide. The following discussion sticks to the terminology of "bijections", but the reader should bear in mind that "equivalences" could also be used with no difference.

The basic idea is to introduce a type of evidence witnessing the fact that one set (0-type) is a "subset" of another, written $A \subseteq B$.[1] Of course there is no subtyping in HoTT, so there is no literal sense in which one type can be a subset of another. However, the situation can be modelled using constructive evidence for the embedding of one type into another. In order to focus the discussion, we begin with copartial bijections between arbitrary sets, and only later restrict to finite ones.

Definition 5.2.1. A *copartial bijection* $f : A \subseteq B$ between two sets A and B is given by:

- an embedding function $f^{\rightarrow} : A \to B$ (we will often simply use f, rather than $f^{\rightarrow}$, to refer to the embedding function),
- a projection function $f^{\leftarrow} : B \to \top + A$,

together with two round-trip laws:

- $f^{\leftarrow} \circ f^{\rightarrow} = \mathsf{inr}$, and
- for all $a : A$ and $b : B$, if $f^{\leftarrow}\ b = \mathsf{inr}\ a$ then $f^{\rightarrow}\ a = b$.

That is, $A \subseteq B$ witnesses that there is a 1-1 correspondence between all the elements of A and *some* (possibly all) of the elements of B, as pictured in Figure 5.1. This concept is also known as a *prism* in the Haskell `lens` library [Kmett, b].

There is also a more elegant, though perhaps less intuitive, formulation of the round-trip laws in Definition 5.2.1.

[1] There should be no problem in generalizing copartial bijections to copartial equivalences which work over any types, using an appropriate notion of copartial adjoint equivalences. However, there is no need for such generalization in the present work, so we stick to the simpler case of 0-types.

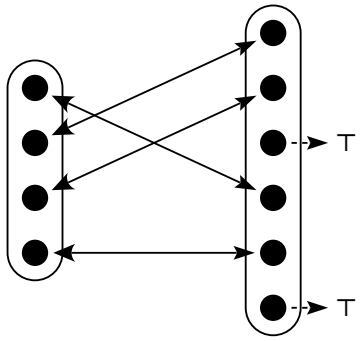


Figure 5.1: A typical copartial bijection

Proposition 5.2.2. *The round-trip laws given in Definition 5.2.1 are equivalent to*

$$\forall ab.\ (f^{\rightarrow}\ a = b) \leftrightarrow (\mathsf{inr}\ a = f^{\leftarrow}\ b). \tag{5.2.1}$$

Proof. Since the laws in question are all mere propositions, it suffices to show that they are logically equivalent; moreover, since the right-to-left direction of (5.2.1) is precisely the second round-trip law, it suffices to show that the left-to-right direction is logically equivalent to the first round-trip law. In one direction, (5.2.1) implies the first round-trip law, by setting $b = f^{\rightarrow}\ a$. Conversely, given the first round trip law,

$$\begin{array}{ll} & f^{\rightarrow}\ a = b \\ \rightarrow & \quad \{ \quad \text{apply } f^{\leftarrow} \text{ to both sides} \quad \} \\ & f^{\leftarrow}\ (f^{\rightarrow}\ a) = f^{\leftarrow}\ b \\ \leftrightarrow & \quad \{ \quad \text{first round-trip law} \quad \} \\ & \mathsf{inr}\ a = f^{\leftarrow}\ b. \end{array}$$

SDG

As an aid in discussing copartial bijections we define $\mathsf{plnv}(f)$ which together with $f : A \to B$ constitutes a copartial bijection $A \subseteq B$.

Definition 5.2.3. A *partial inverse* $\mathsf{plnv}(f)$ to $f : A \to B$ is defined so that

$$(A \subseteq B) \equiv (f : A \to B) \times \mathsf{plnv}(f),$$

that is,

$$\mathsf{plnv}(f) :\equiv (g : B \to \top + A) \times (\forall ab.\ (f\ a = b) \leftrightarrow (\mathsf{inr}\ a = g\ b)).$$

We also define some notation to make working with copartial bijections more convenient.

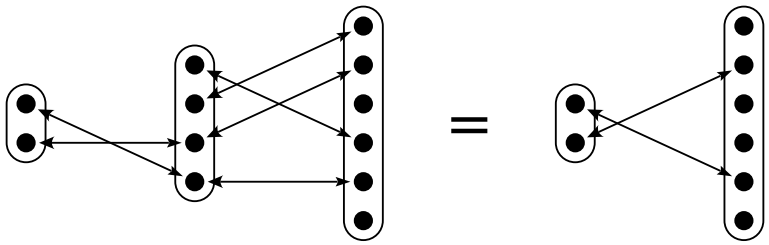


Figure 5.2: Composition of copartial bijections

Definition 5.2.4. First, for any types A and B, there is a canonical copartial bijection $A \subseteq A + B$, which we denote simply by inl; similarly, $\mathsf{inr} : B \subseteq A + B$.

For the remainder, assume there is a copartial bijection $p : K \subseteq L$.

- $p^{\rightarrow}\ K$ denotes the image of K under p, that is, the set of values in L in the range of $p^{\rightarrow}$; we often simply write $p\ K$ instead of $p^{\rightarrow}\ K$.
- $p|_K : K \xrightarrow{\sim} p\ K$ denotes the bijection between K and the image of K in L.
- When some $q : K' \subseteq K$ is understood from the context, we also write $p|_{K'}$ as an abbreviation for $(p \circ q)|_{K'}$, the bijection between K' and the image of K' in L under the composite $(p \circ q)$.
- $p_\top = \{l : L \mid l^{\leftarrow} = \mathsf{inl}\ \star\}$ denotes the "extra" values in L which are not in the image of K.
- $\tilde{p}$ denotes the canonical bijection $K + p_\top \xrightarrow{\sim} L$.

We now turn to the category structure on copartial bijections.

Proposition 5.2.5. *Copartial bijections compose, that is, there is an associative operation*

$$- \circ - : (B \subseteq C) \to (A \subseteq B) \to (A \subseteq C).$$

Proof. This can be intuitively grasped by studying a diagram such as the one shown in Figure 5.2.

More formally, we set $(g \circ f)^{\rightarrow} = g^{\rightarrow} \circ f^{\rightarrow}$ and $(g \circ f)^{\leftarrow} = f^{\leftarrow} \bullet g^{\leftarrow}$, where $- \bullet -$ denotes Kleisli composition for the $\top + -$ monad (*i.e.* $(<=<) :: (b \to \mathsf{Maybe}\ c) \to (a \to \mathsf{Maybe}\ b) \to (a \to \mathsf{Maybe}\ c)$ in Haskell). Associativity thus follows from the associativity of function composition and Kleisli composition. In the following proof we also make use of $(-)^* : (A \to \top + B) \to (\top + A \to \top + B)$, *i.e.* $(=\!\ll)$ in Haskell.

To show the required round-trip property we reason as follows.

$$
\begin{array}{lll}
& (g \circ f)^{\rightarrow}\ a = c & \\
\leftrightarrow & & \{\quad \text{definition} \quad\} \\
& (g^{\rightarrow} \circ f^{\rightarrow})\ a = c & \\
\leftrightarrow & & \{\quad \text{take } b = f^{\rightarrow}\ a \quad\} \\
& \exists b.\ f^{\rightarrow}\ a = b \wedge g^{\rightarrow}\ b = c & \\
\leftrightarrow & & \{\quad \text{round-trip laws for } f \text{ and } g \quad\} \\
& \exists b.\ \mathsf{inr}\ a = f^{\leftarrow}\ b \wedge \mathsf{inr}\ b = g^{\leftarrow}\ c & \\
\leftrightarrow & & \{\quad \text{definition of } (-)^*\text{; case analysis} \quad\} \\
& \mathsf{inr}\ a = (f^{\leftarrow})^*\ (g^{\leftarrow}\ c) & \\
\leftrightarrow & & \{\quad \text{Kleisli composition} \quad\} \\
& \mathsf{inr}\ a = (f^{\leftarrow} \bullet g^{\leftarrow})\ c & \\
\leftrightarrow & & \{\quad \text{definition} \quad\} \\
& \mathsf{inr}\ a = (g \circ f)^{\leftarrow}\ c &
\end{array}
$$

∎

Proposition 5.2.6. *Copartial bijections form an h-category, $\mathcal{S}_{\subseteq}$, with sets as objects.*

Proof. The identity morphism $id : A \subseteq A$ is given by $id^{\rightarrow} = id$ and $id^{\leftarrow} = \mathsf{inr}$. The identity laws follow from the fact that id is the identity for function composition, and inr is the identity for Kleisli composition.

$\mathcal{S}_{\subseteq}$ is thus a precategory. It remains only to show that isomorphism is equivalent to equality. An isomorphism $A \cong B$ is given by $f : A \subseteq B$ and $g : B \subseteq A$ such that $f \circ g = id = g \circ f$. Note that we have $f^{\rightarrow} : A \to B$ and $g^{\rightarrow} : B \to A$ with $f^{\rightarrow} \circ g^{\rightarrow} = (f \circ g)^{\rightarrow} = id^{\rightarrow} = id$, and likewise for $g^{\rightarrow} \circ f^{\rightarrow}$. Thus, $f^{\rightarrow}$ and $g^{\rightarrow}$ constitute a bijection $A \xrightarrow{\sim} B$; since A and B are sets, this is the same as an equivalence $A \simeq B$, and hence by univalence an equality $A = B$. ∎

Remark. Note that a bijection $f : A \xrightarrow{\sim} B$ can be made into a copartial bijection $h : A \subseteq B$ trivially by setting $h^{\rightarrow} = f$ and $h^{\leftarrow} = \mathsf{inr} \circ f^{-1}$, and moreover that this is a homomorphism with respect to composition; that is, the category of bijections embeds into the category of copartial bijections as a subcategory. We will usually not bother to note the conversion, simply using bijections as if they were copartial bijections when convenient.[2]

5.2.2 Finite copartial bijections

Finally, we turn to the theory of copartial bijections on *finite* sets. In the case of finite sets, it turns out that copartial bijections $A \subseteq B$ can be more simply characterized as injective functions $A \hookrightarrow B$. This might seem obvious, and indeed, it is straightforward

[2]In fact, using the `lens` library—and more generally, using a van Laarhoven formulation of lenses [Jaskelioff and O'Connor, 2014]—this all works out automatically: the representations of bijections (isomorphisms) and copartial bijections (prisms) are such that the former simply *are* the latter, and they naturally compose via the standard function composition operator.

in a classical setting. One direction, namely, converting a copartial bijection into an injection, is straightforward in HoTT as well (Lemma 5.2.8). However, to produce a copartial bijection from an injection, we must be able to recover the computational content of the backwards direction, and this depends on the ability to enumerate the elements of A. Recall that the computational evidence for the finiteness of A is propositionally truncated (Definition 2.4.7), so it is not *a priori* obvious that we are allowed do this. However, given a function $f : A \to B$, its partial inverse (if any exists) is uniquely determined, independent of the evidence for the finiteness of A (Lemma 5.2.9), so such evidence may be used in the construction of a partial inverse (Lemma 5.2.10).

Definition 5.2.7. The type of *injections* $A \hookrightarrow B$ is defined in HoTT analogously to the usual definition in set theory:

$$A \hookrightarrow B :\equiv (f : A \to B) \times \mathsf{isInjective}(f),$$

where

$$\mathsf{isInjective}(f) :\equiv \prod_{a_1, a_2 : A} (f\ a_1 = f\ a_2) \to (a_1 = a_2).$$

Remark. Note that $\mathsf{isInjective}(f)$ is a mere proposition when A is a set: given $i, j : \mathsf{isInjective}(f)$, for all $a_1, a_2 : A$ and $e : f\ a_1 = f\ a_2$, we have $i\ a_1\ a_2\ e = j\ a_1\ a_2\ e$ (since they are parallel paths between elements of a set) and hence $i = j$ by function extensionality.

Lemma 5.2.8. *Every copartial bijection is an injection, that is,* $(A \subseteq B) \to (A \hookrightarrow B)$.

Proof. Let $f : A \subseteq B$. Then $f^{\to} : A \to B$ is injective:

$$\begin{array}{ll}
& f^{\to}\ a_1 = f^{\to}\ a_2 \\
\to & \qquad \{ \quad \text{apply } f^{\leftarrow} \text{ to both sides} \quad \} \\
& f^{\leftarrow}\ (f^{\to}\ a_1) = f^{\leftarrow}\ (f^{\to}\ a_2) \\
\leftrightarrow & \qquad \{ \quad f \text{ is a copartial bijection} \quad \} \\
& \mathsf{inr}\ a_1 = \mathsf{inr}\ a_2 \\
\leftrightarrow & \qquad \{ \quad \mathsf{inr} \text{ is injective} \quad \} \\
& a_1 = a_2.
\end{array}$$

∎

Lemma 5.2.9. *If A and B are sets and $f : A \to B$, then $\mathsf{pInv}(f)$ is a mere proposition.*

Proof. Let $(g, p), (g', p') : \mathsf{pInv}(f)$. That is, $g, g' : B \to \top + A$, and

- $p : \forall ab.\ (f\ a = b) \leftrightarrow (\mathsf{inr}\ a = g\ b)$, and

- $p' : \forall ab.\ (f\ a = b) \leftrightarrow (\mathsf{inr}\ a = g'\ b)$.

We must show that $(g, p) = (g', p')$. To this end we first show $g = g'$. By function extensionality it suffices to show that $g\ b = g'\ b$ for arbitrary $b : B$. We proceed by case analysis on $g\ b$ and $g'\ b$:

- If $g\ b = g'\ b = \mathsf{inl}\ \star$ we are done.
- If $g\ b = \mathsf{inr}\ a$ then by p and p' we also have $g'\ b = \mathsf{inr}\ a$ and hence $g\ b = g'\ b$; a symmetric argument handles the case $g'\ b = \mathsf{inr}\ a$.

Letting $r : g = g'$ denote the equality just constructed, we complete the argument by noting that $r_*(p)$ and p' are parallel paths between elements of a set, and hence equal. ∎

Lemma 5.2.10. *If A is a finite set and B a set with decidable equality, then*

$$(A \hookrightarrow B) \to (A \subseteq B).$$

Proof. Let $f : A \to B$ be an injective function; we must construct $h : A \subseteq B$. First, we set $h^{\to} = f$. It remains to construct $\mathsf{plnv}(h^{\to})$, which is a mere proposition by Lemma 5.2.9. Thus, by the recursion principle for propositional truncation, we are justified in using the constructive evidence of A's finiteness, that is, its cardinality $n : \mathbb{N}$ and bijection $\sigma : A \simeq \mathsf{Fin}\ n$. We define $h^{\leftarrow} : B \to \top + A$ on an input $b : B$ as follows: by recursion on n, find the smallest $k : \mathsf{Fin}\ n$ such that $h^{\to}\ (\sigma^{-1}\ k) = b$. If such a k exists, yield $\mathsf{inr}\ (\sigma^{-1}\ k)$; otherwise, yield $\mathsf{inl}\ \star$.

Finally, we establish the round-trip law $\forall ab.\ (h^{\to}\ a = b) \leftrightarrow (\mathsf{inr}\ a = h^{\leftarrow}\ b)$.

($\to$) Suppose $h^{\to}\ a = b$. Then $h^{\leftarrow}\ b$ will certainly find some $k : \mathsf{Fin}\ n$ with $h^{\to}\ (\sigma^{-1}\ k) = b$, and thus $h^{\leftarrow}\ b = \mathsf{inr}\ (\sigma^{-1}\ k)$; since $h^{\to}$ is injective it must actually be the case that $\sigma^{-1}\ k = a$.

($\leftarrow$) This follows directly from the definition of $h^{\leftarrow}$. ∎

Proposition 5.2.11. *For A a finite set and B a set with decidable equality,*

$$(A \hookrightarrow B) \simeq (A \subseteq B).$$

Proof. Lemma 5.2.8 and Lemma 5.2.10 establish functions in both directions. It is easy to see that they act as the identity on the underlying $f : A \to B$ functions, and the remaining components are mere propositions by Lemma 5.2.9 and the remark following Definition 5.2.7. Thus the functions defined by Lemma 5.2.8 and Lemma 5.2.10 are inverse. ∎

Definition 5.2.12. Denote by $\mathcal{B}_\subseteq$ the h-category of finite sets and copartial bijections (*i.e.* injections). That is, objects in $\mathcal{B}_\subseteq$ are values of type $\mathcal{U}_{\|\mathrm{Fin}\|} \equiv (A : \mathcal{U}) \times \mathsf{isFinite}(A)$, and morphisms $(A, f_A) \Rightarrow_{\mathcal{B}_\subseteq} (B, f_B)$ are copartial bijections $A \subseteq B$. Showing that this is indeed an h-category is left as an easy exercise.

$\mathcal{B}_\subseteq$ also has a corresponding skeleton category, just like $\mathcal{B}$:

Definition 5.2.13. Denote by $\mathcal{P}_\hookrightarrow$ the h-category whose objects are natural numbers and whose morphisms are given by $m \Rightarrow_{\mathcal{P}_\hookrightarrow} n :\equiv \mathsf{Fin}\ m \hookrightarrow \mathsf{Fin}\ n$. The proof that this is an h-category is also left as an exercise.

Remark. $\mathcal{P}_\hookrightarrow$ has $m!\binom{n}{m}$ distinct morphisms $m \Rightarrow n$, since there are $\binom{n}{m}$ ways to choose the m distinct objects in the image of the morphism, and $m!$ ways to permute the mapping. Note this this means there are zero morphisms when $m > n$, and exactly $n!$ morphisms $n \Rightarrow n$.

Proposition 5.2.14. $\mathcal{B}_\subseteq \cong \mathcal{P}_\hookrightarrow$.

Proof. The proof is similar to the proof that $\mathcal{B}$ is equivalent to $\mathcal{P}$ (Corollary 2.4.22). We define a functor $[-]_\subseteq : \mathcal{P}_\hookrightarrow \to \mathcal{B}_\subseteq$ which sends n to $(\mathsf{Fin}\ n, |(n, id)|)$ (just like the functor $[-] : \mathcal{P} \to \mathcal{B}$ defined in Definition 2.4.14), and which sends $\iota : m \Rightarrow_{\mathcal{P}} n \equiv \mathsf{Fin}\ m \hookrightarrow \mathsf{Fin}\ n$ to the corresponding copartial bijection (Proposition 5.2.11). It is not hard to show that this functor is full, faithful, and essentially surjective, which by Proposition 2.4.18 and Corollary 2.4.21 implies that $[-]_\subseteq : \mathcal{P}_\hookrightarrow \to \mathcal{B}_\subseteq$ is one half of an equivalence. ∎

5.2.3 Copartial species

The point of all this machinery is that we can now use the category $\mathcal{B}_\subseteq$ as the category of labels for a new notion of species.

Definition 5.2.15. A *copartial species* is a functor $F : \mathcal{B}_\subseteq \to \mathcal{S}$. We denote by $\mathbf{Spe}_\subseteq = \mathcal{B}_\subseteq \Rightarrow \mathcal{S}$ the functor category of copartial species.

Remark. Since $\mathcal{B}_\subseteq \cong \mathcal{P}_\hookrightarrow$ (Proposition 5.2.14) copartial species are also equivalent to functors $\mathcal{P}_\hookrightarrow \to \mathcal{S}$.

Since the objects of $\mathcal{B}_\subseteq$ are the same as the objects of $\mathcal{B}$, the object mapping of a copartial species is similar to that of a normal species. That is, one can still think of a copartial species as mapping a finite set of labels to a set of structures "built from" those labels.

A copartial species F also has an action on morphisms: it must lift any copartial bijection $K \subseteq L$ to a function $F\ K \to F\ L$. Of course, bijections are (trivially) copartial bijections, so this includes the familiar case of "relabelling"; bijections are

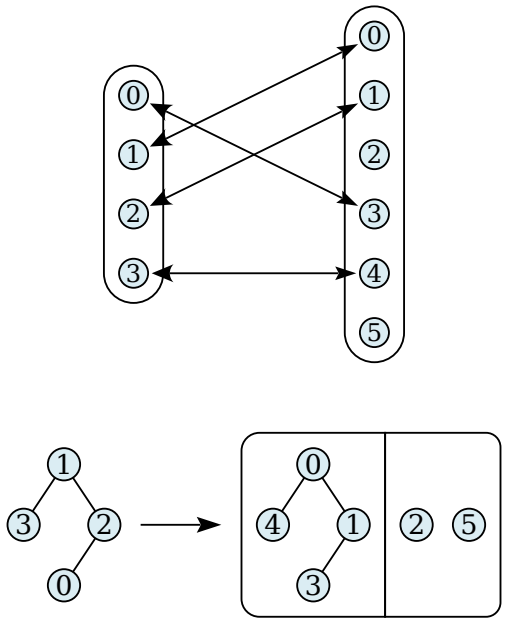


Figure 5.3: Lifting a strictly copartial bijection

isomorphisms in $\mathcal{B}_{\subseteq}$, and functors necessarily preserve isomorphisms, so bijections on labels are still sent to bijections between structures.

The case of strictly copartial bijections, that is, $K \subseteq L$ where $\#K < \#L$, is more interesting. Each structure in the set $F\,K$, with labels in K, must map to a structure in $F\,L$, given an embedding of K into L. Intuitively, this can be thought of as introducing extra labels which must be incorporated into the structure in a suitably canonical way. However, the copartial bijection $p : K \subseteq L$ affords no structure whatsoever on the extra labels (that is, those $l \in L$ for which $p^{\leftarrow}\ l = \mathsf{inl}\ \star$). So it is not acceptable, for example, to prepend the extra labels to the front of a list structure, since there is no canonical way to choose an ordering on the extra labels. The only feasible approach is to simply attach the extra labels in a *set*, as illustrated in Figure 5.3.

Moreover, note that one cannot adjoin a *new* set of labels with every lift. Performing multiple lifts would then result in multiple sets of extra labels (*e.g.* a list of such sets), but this fails to be functorial, since separately lifting two copartial bijections (resulting in a list of two extra sets) would be different than lifting their composition (resulting in only one). So the only option is to have *every* copartial species structure accompanied by a set of "extra" labels (which may be empty). Transporting along a strictly copartial bijection results in some labels being added to the set.

Intuitively, every normal species F gives rise to a copartial species $F_{\subseteq}$ which acts like the species $F \cdot \mathsf{E}$. In fact, along these lines we can formally define a fully faithful embedding of **Spe** into $\mathbf{Spe}_{\subseteq}$.

Definition 5.2.16. The functor $-_{\subseteq} : \mathbf{Spe} \to \mathbf{Spe}_{\subseteq}$ is defined as follows.

First consider the action of $-_{\subseteq}$ on objects, that is, species $F : \mathcal{B} \to \mathcal{S}$. We define $F_{\subseteq} : \mathcal{B}_{\subseteq} \to \mathcal{S}$ as the copartial species which

- sends the finite set of labels K to the set of structures $(F \cdot \mathsf{E})\ K$, and
- lifts the copartial bijection $p : K \subseteq L$ to a function $p_{\subseteq} : F_{\subseteq}\ K \to F_{\subseteq}\ L$. This function takes as input a structure of type $(F \cdot \mathsf{E})\ K$, that is, a tuple

  $$(K_1, K_2, f, \star, \sigma)$$

  where $f : F\ K_1$ is a K_1-labelled F-structure, the unit value $\star$ represents a K_2-labeled set, and $\sigma : K \xrightarrow{\sim} K_1 + K_2$ witnesses that K_1 and K_2 form a partition of the label set K. As output, $p_{\subseteq}$ yields

  $$(L_1, L_2 + p_\top, F\ (p_1|_{K_1})\ f, \star, \psi),$$

  where:

  - p_1 is the "restriction of p to K_1", that is, the composite copartial bijection

    $$p_1 :\ K_1 \xrightarrow[\mathsf{inl}]{\subseteq} K_1 + K_2 \xrightarrow[\sigma^{-1}]{\sim} K \xrightarrow[p]{\subseteq} L\ .$$

    Similarly, $p_2 : K_2 \subseteq L$.
  - $L_1 = p_1\ K_1$ is the image of K_1 under the restricted copartial bijection p_1. Similarly, $L_2 = p_2\ K_2$. Note that we "throw in the extra labels" by using the coproduct $L_2 + p_\top$ as the second set of labels.
  - Recall that $p_1|_{K_1} : K_1 \xrightarrow{\sim} p_1\ K_1$; thus $F\ (p_1|_{K_1})\ f$ denotes the relabelling of the F-structure f from K_1 to $p\ K_1 = L_1$.
  - $\psi : L \xrightarrow{\sim} L_1 + (L_2 + p_\top)$ is given by the composite

    $$L \xrightarrow[\tilde{p}]{\sim} p\ K + p_\top\ .$$

Next, consider the action of $-_{\subseteq}$ on morphisms, that is, natural transformations $\varphi : \forall L.\ F\ L \to G\ L$ where F and G are species. Define $(\varphi_{\subseteq})_L : F_{\subseteq}\ L \to G_{\subseteq}\ L$ by

$$(L_1, L_2, f, \star, \sigma) \mapsto (L_1, L_2, \varphi_{L_1}\ f, \star, \sigma).$$

For this to be natural, the following square must commute for all $F, G : \mathcal{B} \to \mathcal{S}$, all

$\varphi : \forall L.\ F\ L \to G\ L$, and all $p : K \subseteq L$:

$$\begin{array}{ccc} F_\subseteq\ K & \xrightarrow{(\varphi_\subseteq)_K} & G_\subseteq\ K \\ {\scriptstyle F_\subseteq\ p}\downarrow & & \downarrow{\scriptstyle G_\subseteq\ p} \\ F_\subseteq\ L & \xrightarrow[(\varphi_\subseteq)_L]{} & G_\subseteq\ L \end{array}$$

Consider an arbitrary element $(K_1, K_2, f, \star, \sigma)$ of the top-left corner. Note that the action of $\varphi_\subseteq$ on a five-tuple only affects the middle value, and likewise note that the action of $F_\subseteq\ p$ and $G_\subseteq\ p$ are identical on all but the middle value (that is, the middle value is the only one affected by F or G specifically). Thus, it suffices to consider only the fate of f as it travels both paths around the square. Travelling around the left and bottom sides yields

$$\varphi_{L_1}\ (F\ (p_1|_{K_1})\ f),$$

whereas the top and right sides yield

$$G\ (p_1|_{K_1})\ (\varphi_{K_1}\ f).$$

These are equal by naturality of φ.

Finally, it is easy to verify that $-_\subseteq$ itself satisfies the functor laws, since the mapping

$$(L_1, L_2, f, \star, \sigma) \mapsto (L_1, L_2, \varphi_{L_1}\ f, \star, \sigma)$$

clearly preserves identity and composition of natural transformations.

Conjecture 5.2.17. *$-_\subseteq$ is full and faithful.*

The above discussion might lead one to believe that $-_\subseteq$ must also be essentially surjective, *i.e.* that there is an equivalence of categories $\mathbf{Spe} \cong \mathbf{Spe}_\subseteq$. However, this is not the case. To see why, we consider the connection between species, copartial species, and generating functions.

According to our intuition so far, a copartial species corresponds to a regular species with a set of extra labels possibly attached. Consider, therefore, the relationship of the species F to the species $F \cdot \mathsf{E}$. An $(F \cdot \mathsf{E})$-shape of size n consists of an F-shape, of *any* size from 0 to n, paired with a (unique) E-shape on the remaining labels. $F \cdot \mathsf{E}$ thus represents a sort of "prefix sum" of F, where the collection of $(F \cdot \mathsf{E})$-shapes of size n consists of the sum of all F-shapes of sizes 0 through n. This is illustrated in Figure 5.4. In terms of generating functions, the operator $- \cdot \mathsf{E}(x) = - \cdot e^x$ indeed corresponds to a prefix sum on coefficients:

$$- \cdot e^x : (a_0 + a_1 x + a_2 x^2 + \dots) \mapsto (a_0 + (a_0 + a_1)x + (a_0 + a_1 + a_2)x^2 + \dots).$$

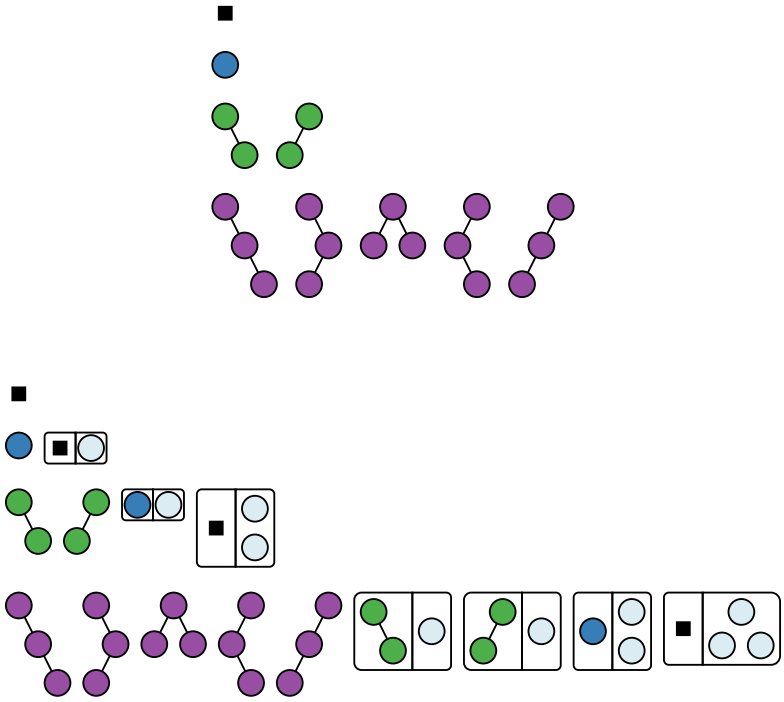


Figure 5.4: $\mathsf{B} \cdot \mathsf{E}$ (bottom) is the prefix sum of B (top)

Note that as an operator on generating functions, this has an inverse, given by

$$(b_0 + b_1 x + b_2 x^2 + \dots) \mapsto (b_0 + (b_1 - b_0)x + (b_2 - b_1)x^2 + \dots). \tag{5.2.2}$$

Consider, then, an attempted proof that $-_{\subseteq} : \mathbf{Spe} \to \mathbf{Spe}_{\subseteq}$ is essentially surjective. Given a copartial species $S \in \mathbf{Spe}_{\subseteq}$, this would require us to produce some $F \in \mathbf{Spe}$ such that $F_{\subseteq} \cong S$. If we think of $F_{\subseteq}$ as intuitively acting like $F \cdot \mathsf{E}$, we see that S should correspond to a "prefix sum" of F. Then we should ideally be able to construct F via an operation similar to (5.2.2). That is (passing to $\mathbf{P} \Rightarrow \mathbf{Set}$ and $\mathcal{P}_{\hookrightarrow} \Rightarrow \mathcal{S}$), we would define

$$F\ 0 := S\ 0$$
$$F\ (n+1) := S\ (n+1) - S\ n.$$

However, we must make sense of this subtraction. We cannot simply take a set difference (indeed, set difference makes no sense in the context of HoTT). What is needed is some sort of canonical injection $\iota : S\ n \hookrightarrow S\ (n+1)$, in which case we could make sense of $S\ (n+1) - S\ n$ as the elements of $S\ (n+1)$ not in the image of ι. In the case of species of the form $F \cdot \mathsf{E}$, there indeed exists such a canonical injection, which sends each shape in $(F \cdot \mathsf{E})\ n$ to the same shape with the extra label n adjoined to the set. The whole point, however, is that we are *trying to prove* that every $S \in \mathbf{Spe}_{\subseteq}$ is of

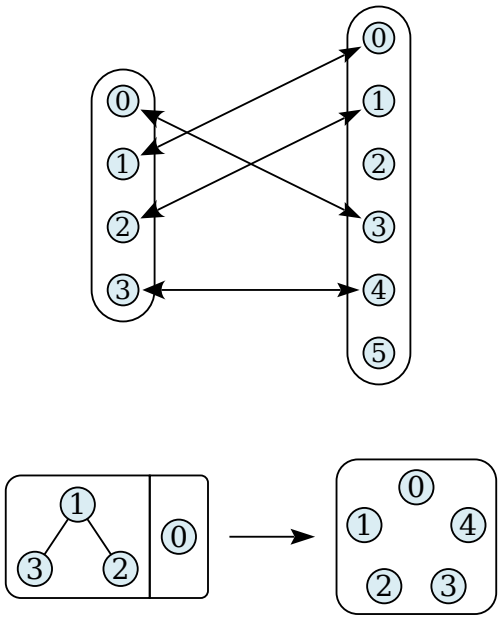


Figure 5.5: A copartial species which loses information

this form. We must therefore come up with some injection $S\ n \hookrightarrow S\ (n+1)$ without any intensional knowledge about S.

This is precisely where we get stuck. There is, of course, a canonical injection $\mathsf{Fin}\ n \hookrightarrow \mathsf{Fin}\ n+1$, but the functoriality of S only guarantees that this lifts to a *function* $S\ n \to S\ (n+1)$—functors may preserve isomorphisms, but in general, they need not preserve monomorphisms. This insight guides us to a counterexample. Consider the $(\mathcal{B}_{\subseteq} \Rightarrow \mathcal{S})$-species whose shapes *of size* 5 *or smaller* consist of a binary tree paired with a set, and *on larger sizes* simply consist of a set (Figure 5.5).

One may verify that this does, in fact, describe a valid functor $\mathcal{B}_{\subseteq} \to \mathcal{S}$. However, it does not preserve information: above size 5 the shapes all collapse, and information about smaller shapes is lost. The intuition that all copartial species shapes must come equipped with a set of labels is correct, in a sense, but there is some latitude in the way the rest of the shape is handled.

We may also, therefore, consider the subcategory $\mathcal{B}_{\subseteq} \hookrightarrow \mathcal{S}$ of *monomorphism-preserving* functors $\mathcal{B}_{\subseteq} \Rightarrow \mathcal{S}$; along the lines sketched above, we can indeed prove an equivalence between this category and **Spe**. At present, the pros and cons of considering $\mathbf{Spe}_{\subseteq}$ versus this subcategory are not clear to me.

Finally, we consider which of the properties from Table 5.1 hold for $\mathcal{B}_{\subseteq} \Rightarrow \mathcal{S}$.

- Since the target category is just $\mathcal{S}$ we automatically get all the properties required of $\mathfrak{S}$ alone (*e.g.* monoidal, complete, and Cartesian closed); $\mathcal{S}$ is also cocomplete and so has coends over $\mathcal{B}_{\subseteq}$.

- $\mathcal{B}_\subseteq$ is locally small.
- $\mathcal{B}_\subseteq$ is monoidal: though it does not have products or coproducts, it is not hard to see that it has monoidal structures corresponding to the Cartesian product and disjoint union of finite sets. Given injections $f : S_1 \hookrightarrow T_1$ and $g : S_2 \hookrightarrow T_2$ we can use them to form the evident injections $S_1 \uplus S_2 \hookrightarrow T_1 \uplus T_2$ and $S_1 \times S_2 \hookrightarrow T_1 \times T_2$.
- $\mathcal{B}_\subseteq$ is enriched over $\mathfrak{S}$, since its morphisms can be seen as injective functions.

5.3 Partial species

We now consider the dual category $\mathcal{B}_\subseteq^{\mathrm{op}}$, whose objects are finite sets and whose morphisms are partial bijections, *i.e.* coinjections, and written $K \supseteq L$. These can be thought of as partially defined functions which are both injective and surjective.

Species corresponding to $\mathcal{B}_\subseteq^{\mathrm{op}} \Rightarrow \mathcal{S}$ were studied by Schmitt [1993] (under the name "species with restriction", or "R-species") and correspond to species with a natural notion of "induced subspecies". That is, $F : \mathcal{B}_\subseteq^{\mathrm{op}} \to \mathcal{S}$ must lift morphisms of the form $K \supseteq L$ to functions $F\ K \to F\ L$. Instead of adding more labels, this operation may *delete* labels. Examples include the species of lists, where labels may simply be deleted, keeping the rest of the labels in the same order; similarly, the species of cycles; and the species of simple graphs, where the lifting operation corresponds to forming induced subgraphs.

When we consider the properties in Table 5.1, however, we find in particular that $\mathcal{B}_\subseteq^{\mathrm{op}}$ is not enriched over $\mathcal{S}$. Coinjections are not, in general, total functions, so there is no way to canonically treat morphisms in $\mathcal{B}_\subseteq^{\mathrm{op}}$ as objects of $\mathcal{S}$. Instead of $\mathcal{S}$, we actually want to consider the category $\mathcal{S}_{\rightharpoonup}$ of sets and *partial* functions. One may check that $\mathcal{S}_{\rightharpoonup}$ is monoidal, complete, and Cartesian closed, that it has coends over $\mathcal{B}_\subseteq^{\mathrm{op}}$, and that $\mathcal{B}_\subseteq^{\mathrm{op}}$ is indeed enriched over $\mathcal{S}_{\rightharpoonup}$.

5.4 Multisort species

Multisort species are a generalization of species in which the labels are classified according to multiple *sorts*. We often use X, Y, Z or X_1, X_2, ... to denote sorts. In particular, (say) Y denotes the species, analogous to X, for which there is a single shape containing a single label *of sort* Y (and none of any other sort). More generally, multisort species correspond to multivariate generating functions. See Bergeron et al. [1998, §4.2] for a precise, detailed definition. For now, an intuitive sense is sufficient; we will give a more abstract definition later.

Example. Consider

$$\mathfrak{T} = \mathsf{Y} + \mathsf{X} \cdot \mathfrak{T}^2,$$

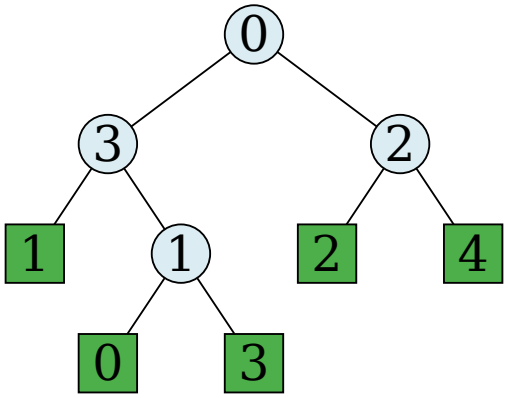


Figure 5.6: A two-sort species of binary trees

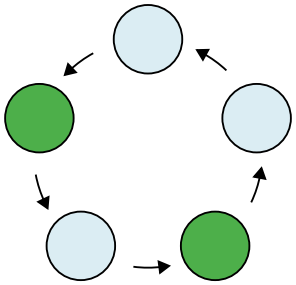


Figure 5.7: A bicolored cycle

the two-sort species of binary trees with internal nodes and leaves labelled by distinct sorts. Figure 5.6 illustrates an example shape of this species, with one label sort represented by blue circles, and the other by green squares.

Example. $\mathsf{C} \circ (\mathsf{X} + \mathsf{Y})$ is the species of *bicolored cycles*, *i.e.* cycles whose labels are colored with one of two colors (Figure 5.7).

Example. Recall the example from §4.5.1, in which open terms of the untyped lambda calculus are modeled using the species

$$\Lambda = \varepsilon + \Lambda^2 + \Lambda'.$$

However, this species does not correctly model the size of lambda calculus terms. §4.5.1 suggested instead using the multisort species

$$\Xi = \mathsf{X} \cdot \mathsf{E} + \mathsf{Y} \cdot \Xi^2 + \mathsf{Y} \cdot \frac{\partial}{\partial \mathsf{X}} \Xi.$$

An even better idea is to use a three-sort species, as in

$$\Xi = \mathsf{X} \cdot \mathsf{E} + \mathsf{Y} \cdot \Xi^2 + \mathsf{Z} \cdot \frac{\partial}{\partial \mathsf{X}} \Xi,$$

where X stands in for variables, Y for applications, and Z for lambdas.

In general, any algebraic data type may be modelled by a multisort species, with one sort corresponding to each constructor. The singletons of a given sort count occurrences of the corresponding construct and ensure that species structures of a finite size correspond to data structures that take only a finite amount of memory.

Bergeron et al. [1998] detail how to extend operations such as sum, partitional product, and composition to multisort species. For example, partitional product is straightforward: an $(F \cdot G)$-shape over a given collection of labels (of multiple sorts) corresponds to a pair of an F-shape and G-shape over a binary partition of the collection (which can also be thought of as a collection of binary partitions over each sort). Composition can also be extended to multisort species—although, as we will see, it does not follow quite as naturally from the single-sort setting. If F is an m-sort species, and $(G_1, \ldots, G_m)$ is an m-tuple of n-sort species, then $F \circ (G_1, \ldots, G_m)$ is an n-sort species whose shapes consist of a top-level F-shape with G_i-shapes substituted for each label of sort i. Of course, this presentation assumes a linear ordering on the sorts of F; more generally, if the sorts of F are indexed by some finite set S, then F can be composed with an S-indexed tuple of T-sort species, resulting in a T-sort species.

Remark. Multisort species are often notated using "multi-argument function" notation, for example,

$$\mathcal{H}(\mathsf{X}, \mathsf{Y}) = \mathsf{X} + \mathsf{Y}^2.$$

This makes clear which singleton species are being used to represent the various sorts, and makes it possible to refer to them positionally as well. This notation also meshes well with the notion of generalized composition just introduced: writing something like $\mathcal{H}(F, G)$ can be interpreted as $\mathcal{H} \circ (F, G)$ and corresponds exactly to the substitution of F and G for X and Y.

Defining multisort species and all the operations on them (such as composition) from scratch is unnecessary; they can be defined abstractly as objects in a certain functor category, and hence fit into the abstract framework developed in Chapter 3 and outlined in §5.1. Bergeron et al. [1998] acknowledge as much in Exercise 2.4.6, but only as something of an afterthought; the following development is yet more general than the intended solution to the exercise.

Let S be a finite set, thought of as a collection of names for sorts; that is, each element $s \in S$ represents a different sort. Let $\mathfrak{L}$ be a category, thought of as a category of labels (*e.g.* $\mathbf{B}$). Now consider the functor category $\mathfrak{L}^S$ (with S considered as a discrete category, as usual). Objects of $\mathfrak{L}^S$ are functors $S \to \mathfrak{L}$, that is, assignments

of an object from $\mathfrak{L}$ to each $s \in S$. Morphisms in $\mathfrak{L}^S$ are natural transformations, that is, S-indexed families of $\mathfrak{L}$-morphisms between corresponding objects of $\mathfrak{L}$. For example, in the case $\mathfrak{L} = \mathbf{B}$, objects of $\mathbf{B}^S$ are just S-tuples of finite sets, and morphisms are S-tuples of bijections between them.

Remark. Recall that $\mathbf{Set}^S \cong \mathbf{Set}/S$ (§1.4.1). It is not the case that $\mathbf{B}^S \cong \mathbf{B}/S$, since objects of $\mathbf{B}/S$ consist of a finite set L paired with a *bijection* $L \xrightarrow{\sim} S$, which only allows one label of each sort. However, it is possible to consider the category whose objects are finite sets L paired with a *function* $\chi : L \to S$, as in $\mathbf{Set}/S$, but whose morphisms $(L_1, \chi_1) \to (L_2, \chi_2)$ are *bijections* $\sigma : L_1 \xrightarrow{\sim} L_2$ such that $\chi_2 \circ \sigma = \chi_1$. In other words, the objects are finite sets with sorts assigned to their elements, and the morphisms are sort-preserving bijections. This category is indeed equivalent to $\mathbf{B}^S$.

However, this construction only works because the objects of $\mathbf{B}$ are sets; in the general case we must stick with $\mathfrak{L}^S$.

Definition 5.4.1. For a finite set S, define *S-sorted ($\mathfrak{L}$,$\mathfrak{S}$)-species* as functors

$$\mathfrak{L}^S \to \mathfrak{S}.$$

One can verify that taking $\mathfrak{L} = \mathbf{B}$, $\mathfrak{S} = \mathbf{Set}$, and $S = [k]$ for some $k \in \mathbb{N}$, we recover exactly the definition of k-sort species given by Bergeron *et al.*

The payoff is that we can now check that $\mathfrak{L}^S$ inherits the relevant properties from $\mathfrak{L}$, and thus conclude that $(\mathfrak{L}^S, \mathfrak{S})$-species inherit operations from $(\mathfrak{L}, \mathfrak{S})$-species. Simply unfolding definitions is then enough to describe the action of the operations on S-sorted species.

- $\mathfrak{L}^S$ inherits all the monoidal structure of $\mathfrak{L}$, as seen in §4.1.3.
- $\mathfrak{L}^S$ is a groupoid whenever $\mathfrak{L}$ is.
- If $\mathfrak{L}$ is enriched over $\mathfrak{S}$ and $\mathfrak{S}$ has finite products, then $\mathfrak{L}^S$ can be seen as enriched over $\mathfrak{S}$ as well: morphisms in $\mathfrak{L}^S$ are represented by S-indexed products of morphisms in $\mathfrak{L}$.
- $\mathfrak{L}^S$ is locally small whenever $\mathfrak{L}$ is.
- $\mathfrak{S}$ has coends over $\mathfrak{L}^S$ as long as it has products and coends over $\mathfrak{L}$, which we can argue as follows. Since S is discrete, everything in $\mathfrak{L}^S$ naturally decomposes into discrete S-indexed collections. For example, a morphism in $\mathfrak{L}^S$ is isomorphic to an S-indexed collection of morphisms in $\mathfrak{L}$, a functor $\mathfrak{L}^S \to \mathfrak{S}$ is isomorphic to an S-indexed product of functors $\mathfrak{L} \to \mathfrak{S}$, and so on. Note that $(\mathfrak{L}^S)^{\mathrm{op}} \times \mathfrak{L}^S \cong (\mathfrak{L}^{\mathrm{op}})^S \times \mathfrak{L}^S \cong (\mathfrak{L}^{\mathrm{op}} \times \mathfrak{L})^S$, so a functor $T : (\mathfrak{L}^S)^{\mathrm{op}} \times \mathfrak{L}^S \to \mathfrak{S}$ can also be decomposed in this way. In particular, this means that, as long as $\mathfrak{S}$ has S-indexed products, we may construct the coend $\exists L.\ T(L, L)$ componentwise, that is,
$$\exists L.\ T(L, L) := \prod_{s \in S} \exists K.\ T_s(K, K),$$

where T_s denotes the s-component of the decomposition of T. One can check that this defines a valid coend.

- By a similar argument, $\mathfrak{L}^S \Rightarrow \mathfrak{S}$ is enriched over itself as long as $\mathfrak{L} \Rightarrow \mathfrak{S}$ is enriched over itself, and $\mathfrak{S}$ (and hence $\mathfrak{L} \Rightarrow \mathfrak{S}$) has products.

We can thus instantiate the generic definitions to obtain notions of sum, Cartesian, arithmetic, and partitional product, and differentiation (along with corresponding eliminators) for S-sorted species. For example, the notion of partitional product obtained in this way is precisely the definition given above.

One operation that we do *not* obtain quite so easily is composition: the generic definition relied on a definition of the "K-fold partitional product" G^K where, in this case, $G : \mathfrak{L}^S \to \mathfrak{S}$ and $K \in \mathfrak{L}^S$. It is not *a priori* clear what should be meant by the K-fold partitional product of G where K itself is a collection of labels of different sorts. We could ignore the sorts on K, using a monoidal structure on $\mathfrak{L}$ to reduce K to an object of $\mathfrak{L}$; for example, in $\mathbf{B}^S \Rightarrow \mathbf{Set}$, given some collection of finite sets K, each corresponding to a different sort, we can simply take their coproduct. This results in a notion of composition $F \circ G$ where we simply ignore the sorts on the labels of F, replacing each with a (sorted) G-shape. This certainly yields a monoid on $\mathfrak{L}^S \Rightarrow \mathfrak{S}$, but one that does not really make use of the sorts at all.

The generalized notion of composition defined earlier, on the other hand, is not a monoid on $\mathfrak{L}^S \Rightarrow \mathfrak{S}$ (indeed, it is not a monoid at all). Instead, it has the type

$$- \circ - : (\mathfrak{L}^S \Rightarrow \mathfrak{S}) \to (\mathfrak{L}^T \Rightarrow \mathfrak{S})^S \to (\mathfrak{L}^T \Rightarrow \mathfrak{S}).$$

This seems somewhat reminiscent of a relative monad [Altenkirch et al., 2010]; exploring the connection is left to future work.

5.4.1 Recursive species

Multisort species make it possible to give a precise semantics to recursively defined species, which have been used in examples throughout this document. Given a recursive equation of the form

$$F = \ldots F \ldots,$$

we can turn the right-hand side into a two-sort species $\mathcal{H}(\mathsf{X}, \mathsf{Y})$, with Y replacing the recursive occurrences of F. For example, the recursive equation

$$\mathsf{R} \cong \mathsf{X} \cdot (\mathsf{L} \circ \mathsf{R})$$

corresponds to the two-sort species $\mathcal{H}(\mathsf{X}, \mathsf{Y}) = \mathsf{X} \cdot (\mathsf{L} \circ \mathsf{Y})$. We then define R as the least fixed point (if it exists) of $\mathcal{H}(\mathsf{X}, -)$, that is, a solution to $\mathsf{R} \cong \mathcal{H}(\mathsf{X}, \mathsf{R})$. The following theorem expresses the conditions on $\mathcal{H}$ under which such fixed point solutions exist.

Theorem 5.4.2 (Implicit Species Theorem, [Joyal, 1981, Bergeron et al., 1998]). *Let $\mathcal{H}$ be a two-sort species satisfying*

- $\mathcal{H}(0,0) \cong 0$
- $\dfrac{\partial \mathcal{H}}{\partial Y}(0,0) \cong 0$

Then there exists a species F, unique up to isomorphism, satisfying

$$F \cong \mathcal{H}(X, F),$$

with $F(0) \cong 0$.

Remark. Recall that the notation $\mathcal{H}(0,0) = \mathcal{H} \circ (0,0)$ denotes the composition of the two-sort species $\mathcal{H}$ with the pair of one-sort species $(0,0)$. These criteria are thus expressed in the form of species isomorphisms, but in this particular case they could equally well be expressed in terms of the action of $\mathcal{H}$ on empty label sets, *e.g.* $\mathcal{H}(\varnothing, \varnothing) = \varnothing$.

The proof essentially proceeds by constructing F as the infinite expansion

$$F = \mathcal{H}(X, \mathcal{H}(X, \mathcal{H}(X, \dots))).$$

The conditions on $\mathcal{H}$ ensure that this is well-defined. In particular, since $(\partial\mathcal{H}/\partial Y)$-shapes have a single hole in the place of a Y, which is the placeholder for recursive occurrences of F, $\frac{\partial \mathcal{H}}{\partial Y}(0,0) \cong 0$ means that there are no $\mathcal{H}(X,Y)$-shapes consisting solely of (some constant multiple of) a Y. Such shapes would allow a recursive step that did not "use" any X's, resulting in infinitely large shapes of size 0. For details of the proof, see Bergeron et al. [1998, §3.2]. The implicit species theorem can also be suitably generalized to systems of mutually recursive equations; see Bergeron et al. [1998] as well as Pivoteau et al. [2012].

Many common examples of recursively defined species, such as $L = 1 + X \cdot L$, or $B = 1 + X \cdot B^2$, do not actually satisfy the premises of the implicit species theorem, in particular the requirement that $\mathcal{H}(0,0) \cong 0$. In both the above cases we instead have $\mathcal{H}(0,0) \cong 1$. The Implicit Species Theorem only gives sufficient, but not necessary, conditions for well-foundedness; we would like to have a different theorem that tells us when equations such those governing L and B are well-founded. Pivoteau et al. [2012] prove quite a general theorem which is applicable to this case, and is also applicable to mutually recursive systems. Its very generality somewhat obscures the essential ideas, however, so we give a "baby" version of the theorem here.

The basic idea can be seen by considering the case of $L = 1 + X \cdot L$. Decompose L as $L = 1 + L_+$, so $L_+ = X \cdot L \cong X \cdot (1 + L_+)$. Then $\mathcal{H}(X,Y) = X \cdot (1 + Y)$ does satisfy the premises of the Implicit Species Theorem, so L_+ is well-defined, and hence so is $L = 1 + L_+$. This approach is used, implicitly and in an ad-hoc manner, by Bergeron

et al. [1998]; see, for example, Example 3.2.3 on p. 195. It also appears in a sketchy form in my Haskell Symposium paper [Yorgey, 2010].

Theorem 5.4.3 (Implicit Species Theorem II). *Let $\mathcal{H}(\mathsf{X}, \mathsf{Y})$ be a two-sort species satisfying*

$$\mathcal{H}(0, \mathsf{Y}) \cong n,$$

where $n \in \mathbb{N}$ represents the species $\underbrace{1 + \cdots + 1}_{n}$ with n shapes of size 0. Then there exists a species F, unique up to isomorphism, satisfying

$$F \cong \mathcal{H}(\mathsf{X}, F)$$

with $F(0) \cong n$.

Proof. Since $\mathcal{H}(0, \mathsf{Y}) \cong n$, there is some two-sort species $\mathcal{H}_+$ such that $\mathcal{H}$ can be uniquely decomposed as

$$\mathcal{H}(\mathsf{X}, \mathsf{Y}) \cong n + \mathcal{H}_+(\mathsf{X}, \mathsf{Y})$$

(this follows from an analogue of the molecular decomposition theorem for multisort species). Note that $\mathcal{H}_+(0, \mathsf{Y}) \cong 0$ and $\partial\mathcal{H}/\partial\mathsf{Y} = \partial\mathcal{H}_+/\partial\mathsf{Y}$.

Moreover, $\mathcal{H}(0, \mathsf{Y}) \cong n$ means that, other than the constant term n, every term of the molecular decomposition of $\mathcal{H}$ must contain a factor of X. In other words, $\mathcal{H}_+(\mathsf{X}, \mathsf{Y}) \cong \mathsf{X} \cdot \mathcal{G}(\mathsf{X}, \mathsf{Y})$ for some species $\mathcal{G}$. Thus we have $\frac{\partial\mathcal{H}}{\partial\mathsf{Y}}(\mathsf{X}, \mathsf{Y}) = \mathsf{X} \cdot \frac{\partial\mathcal{G}}{\partial\mathsf{Y}}(\mathsf{X}, \mathsf{Y})$, and in particular $\frac{\partial\mathcal{H}}{\partial\mathsf{Y}}(0, \mathsf{Y}) \cong 0$ as well.

Now define

$$\mathcal{H}_{n+}(\mathsf{X}, \mathsf{Y}) := \mathcal{H}_+(\mathsf{X}, n + \mathsf{Y}).$$

Note that

$$\frac{\partial\mathcal{H}_{n+}}{\partial\mathsf{Y}}(\mathsf{X}, \mathsf{Y}) = \frac{\partial\mathcal{H}_+}{\partial\mathsf{Y}}(\mathsf{X}, n + \mathsf{Y}) = \frac{\partial\mathcal{H}}{\partial\mathsf{Y}}(\mathsf{X}, n + \mathsf{Y})$$

(the first equality follows from the chain rule for differentiation). Thus

$$\frac{\partial\mathcal{H}_{n+}}{\partial\mathsf{Y}}(0, 0) = \frac{\partial\mathcal{H}}{\partial\mathsf{Y}}(0, n) = 0.$$

We also have

$$\mathcal{H}_{n+}(0, 0) = \mathcal{H}_+(0, n) \cong 0.$$

Thus, $\mathcal{H}_{n+}$ satisfies the criteria for the Implicit Species Theorem, and we conclude there uniquely exists a species F_+ satisfying $F_+ \cong \mathcal{H}_{n+}(\mathsf{X}, F_+)$, with $F_+(0) \cong 0$.

Finally, let $F := n + F_+$, in which case

$$\begin{aligned} F &= n + F_+ \\ &\cong n + \mathcal{H}_{n+}(\mathsf{X}, F_+) \\ &= n + \mathcal{H}_+(\mathsf{X}, n + F_+) \\ &\cong \mathcal{H}(\mathsf{X}, n + F_+) \\ &= \mathcal{H}(\mathsf{X}, F). \end{aligned}$$

So $F = n + F_+$ is a solution to $F = \mathcal{H}(\mathsf{X}, F)$. The uniqueness of F follows from the uniqueness of F_+: if F_1 and F_2 are both solutions to $F = \mathcal{H}(\mathsf{X}, F)$, then they both decompose as $F_i = n + F_i^+$, and the F_i^+ both satisfy $F_i^+ = \mathcal{H}_{n+}(\mathsf{X}, F_i^+)$; hence $F_1^+ \cong F_2^+$ but then $F_1 \cong F_2$. ∎

Remark. The condition $\mathcal{H}(\mathsf{0}, \mathsf{Y}) \cong n$, as opposed to the weaker condition $\mathcal{H}(\mathsf{0}, \mathsf{0}) \cong n$, is critical. For example, consider the implicit equation

$$Q = \mathsf{1} + \mathsf{X} + Q^2.$$

In this case $\mathcal{H}(\mathsf{X}, \mathsf{Y}) = \mathsf{1} + \mathsf{X} + \mathsf{Y}^2$ satisfies $\frac{\partial \mathcal{H}}{\partial \mathsf{Y}}(\mathsf{0}, \mathsf{0}) \cong \mathsf{0}$ and $\mathcal{H}(\mathsf{0}, \mathsf{0}) \cong \mathsf{1}$, but $\mathcal{H}(\mathsf{0}, \mathsf{Y}) \cong \mathsf{1} + \mathsf{Y}^2 \ncong n$. In fact, Q is ill-defined: by always picking the Q^2 branch and never X, and putting $\mathsf{1}$ at the leaves, one can construct infinitely many trees of size 0.

5.5 L-species

Consider the category $\mathbf{L}$ of linear orders and order-preserving bijections (discussed previously in §2.4.3). An *L-species* is defined as a functor $\mathbf{L} \to \mathbf{Set}$. The theory of $\mathbf{L}$-species is large and fascinating; for example, it allows one to solve differential equations over species, and to define a notion of intergration dual to differentiation. More practically, it allows modelling data structures with ordering constraints, such as binary search trees and heaps. Unfortunately there is not time or space to include more on the theory here. For now, we simply note that $(\mathbf{L} \Rightarrow \mathbf{Set})$ has many of the required properties:

- $\mathbf{L}$ is indeed monoidal; in fact, there are many interesting choices regarding how to combine linear orders.
- $\mathbf{L}$ is clearly a groupoid, locally small, and enriched over $\mathbf{Set}$.
- $\mathbf{Set}$ is cocomplete and hence has coends over $\mathbf{L}$.
- $(\mathbf{L} \Rightarrow \mathbf{Set})$ is enriched over itself, for reasons similar to $(\mathbf{B} \Rightarrow \mathbf{Set})$.
- Since objects in $\mathbf{L}$ are finite sets, the indexed partitional product G^K can be defined in exactly the same way as in the case of $\mathbf{B} \Rightarrow \mathbf{Set}$.

5.6 Other species variants

Many other examples of species variants appear in the literature. For example:

- **Vec**-valued species, *i.e.* functors $\mathbf{B} \to \mathbf{Vec}$, which send finite sets of labels to *vector spaces* of shapes. Joyal had these in mind from the beginning [Joyal, 1986], and they play a central role in more recent work as well (see, for example, Aguiar and Mahajan [2010]), though I do not yet have a good intuition for them.

- **Cat**-valued species, *i.e.* functors $\mathbf{B} \to \mathbf{Cat}$ which send finite sets of labels to *categories* of shapes. Intriguingly, in the case of groupoids in particular, a suitable notion of cardinality/Euler characteristic can be defined for categories [Baez and Dolan, 2000], allowing **Cat**-valued species to be seen as a categorification of generating functions with positive *rational* coefficients.

- Fiore et al. [2008] define a generalized notion of species, parameterized over arbitrary small categories A and B, as functors in the category

$$\mathbf{B}A \Rightarrow (B^{\mathrm{op}} \Rightarrow \mathbf{Set}).$$

  $\mathbf{B}A$ is a generalization of $\mathbf{B}$ such that $\mathbf{B1} \cong \mathbf{B}$; the objects are tuples of objects from A, labelled by the elements of some finite set from $\mathbf{B}$, and the morphisms permute the labelled A-objects according to some bijection on the finite set elements. They go on to show that this functor category has all the same important properties as $\mathbf{B} \Rightarrow \mathbf{Set}$.

- Bergeron et al. [1998, §2.3] define a notion of *weighted species*, where each shape is assigned a *weight* from some polynomial ring of weights $\mathbb{W}$. It seems that it should be possible to fit weighted species into this framework as well, by considering a slice category $\mathfrak{S}/A$. We can interpret objects of $\mathfrak{S}/A$ as objects of $\mathfrak{S}$ paired with a weighting; morphisms in $\mathfrak{S}/A$ are thus weight-preserving morphisms of $\mathfrak{S}$. Traditional weighted species would then correspond to functors $\mathbf{B} \Rightarrow (\mathbf{Set}/\mathbb{A})$ for some polynomial ring $\mathbb{A}$; more generally, one can add weighting to any species variant $\mathfrak{L} \Rightarrow \mathfrak{S}$ by passing to $\mathfrak{L} \Rightarrow (\mathfrak{S}/A)$ for some appropriate choice of $A \in \mathfrak{S}$. Verifying this construction, in particular the properties of A which give rise to the appropriate species operations on weighted species, is left to future work.

In each case, one can verify the required properties and automatically obtain definitions of the various operations.

Chapter 6

Labelled structures

Now that we have a foundation for describing labelled shapes, the next step is to extend them into full-blown *data structures* by adjoining mappings from labels to data. For example, Figure 6.1 illustrates an example of a labelled shape together with a mapping from the labels to data values. However, this must be done in a principled way which allows deriving properties of labelled structures from corresponding properties of labelled shapes. This chapter begins with an overview of *Kan extensions* (§6.1) and *analytic functors* (§6.2), which provide the theoretical basis for extending labelled shapes to labelled structures. It then continues to briefly discuss introduction and elimination forms for labelled structures, and outline some directions for future work.

6.1 Kan extensions

The definition of analytic functors, given in §6.2, makes central use of the notion of a (left) *Kan extension*. This section defines Kan extensions and provides some useful intuition for understanding them in this context. For more details, see Mac Lane

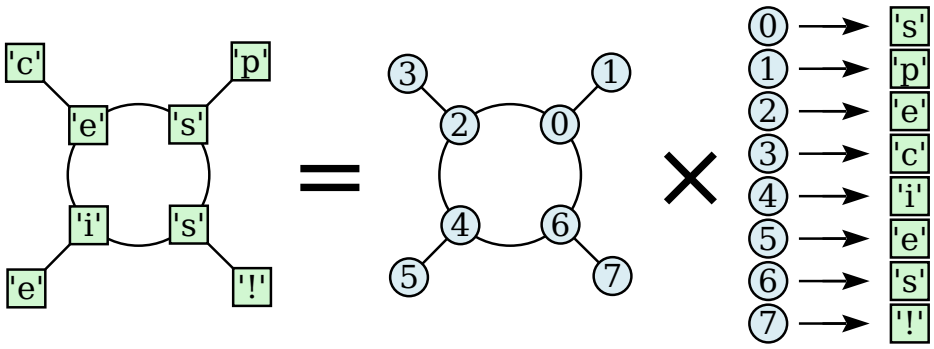


Figure 6.1: Data structure = shape + data

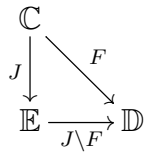
[1998, Chapter X]; for some good intuition with a computational bent, see Hinze [2012].

Definition 6.1.1. Given functors $F : \mathbb{C} \to \mathbb{D}$ and $J : \mathbb{C} \to \mathbb{E}$, the *left Kan extension of F along J* is a functor $\mathbb{E} \to \mathbb{D}$, written[1] $J\backslash F$, and characterized by the natural isomorphism

$$\forall G : \mathbb{E} \to \mathbb{D}.\ (J\backslash F \xrightarrow{\bullet} G) \cong (F \xrightarrow{\bullet} G \circ J). \tag{6.1.1}$$

(Note that the left-hand side consists of natural transformations between functors $\mathbb{E} \to \mathbb{D}$, whereas the right-hand side are between functors $\mathbb{C} \to \mathbb{D}$.) If this isomorphism exists for all F, then we may say even more succinctly that the left Kan extension functor $J\backslash -$ is left adjoint to the precomposition functor $- \circ J$.

The situation can be pictured as shown below:



Intuitively, if $J : \mathbb{C} \to \mathbb{E}$ is thought of as an "embedding" of $\mathbb{C}$ into $\mathbb{E}$, then $J\backslash F$ can be thought of as a way of "extending" the domain of F from $\mathbb{C}$ to $\mathbb{E}$ in a way compatible with J. If we substitute $J\backslash F$ for G in equation (6.1.1) and take the image of $id_{J\backslash F}$, we obtain $\eta : F \to (J\backslash F) \circ J$, a natural transformation sending F to the embedding J followed by the extension $J\backslash F$. Intuitively, the existence of η guarantees that $J\backslash F$ has to "act like" F on the image of $\mathbb{C}$ under J. Of course, $J\backslash F$ must also be defined and functorial on all of $\mathbb{E}$, not just on the image of C. These facts together justify thinking of $J\backslash F$ as an "extension" of F. Note also that substituting $G \circ J$ for F in equation (6.1.1) and taking the image of $id_{G\circ J}$ under the inverse yields $\varepsilon : J\backslash(G \circ J) \to G$, which can be thought of as a computation or reduction rule.

The above gives an abstract characterization of left Kan extensions; under suitable conditions we can explicitly construct them.

Proposition 6.1.2. *When $\mathbb{D}$ has coends over $\mathbb{C}$ and $\mathbb{D}$ has all coproducts, $J\backslash F$ can be constructed explicitly as a coend:*

$$(J\backslash F)\ E = \exists C.\ (J\ C \Rightarrow E) \cdot F\ C. \tag{6.1.2}$$

[1] $J\backslash F$ is traditionally notated $\mathrm{Lan}_J F$. Inspired by the corresponding notion in relational algebra, Roland Backhouse suggested the notation F/J for the right Kan extension of F along J, which was adopted by Hinze [2012]. This notation is a bit more perspicuous than the traditional notation $\mathrm{Ran}_J F$, especially with respect to the accompanying computation (β) and reflection (η) laws. Unfortunately, there is not quite a satisfactory parallel in the case of left Kan extensions. In relational algebra, the notations A/P and $P\backslash A$ are used for the right adjoints to pre- and post-composition, respectively; whereas we want notations for the left and right adjoints of precomposition. I nevertheless adopt the notation $J\backslash F$ for left Kan extensions, and hope this does not cause confusion.

Note that $\cdot$ denotes a copower*; in this particular case, $(J\ C \to E) \cdot F\ C$ denotes an indexed coproduct,* i.e. *the collection of pairs of a morphism in $(J\ C \Rightarrow E)$ and an object $F\ C$.*

As a Haskell type, this construction corresponds to

```
data Lan j f a where
  Lan :: (f c, j c → a) → Lan j f a
```

Remark. Giving the pair in the order $(j\ c \to a, f\ c)$ would correspond more closely to the abstract definition above; however, the given order makes some of the subsequent code a bit nicer.

The `kan-extensions` package [Kmett, a] contains a similar definition. Of course, this type is quite a bit less general than the abstract definition given above—in particular, it instantiates $\mathbb{C}$, $\mathbb{D}$, and $\mathbb{E}$ all to **Hask**. However, it is still quite useful for gaining intuition. Note that c is existentially quantified; recall that in Haskell this corresponds to a coend. Note also that the copower (which in general relates two different categories) turns into simple pairing.

We now turn to a proof of Proposition 6.1.2.

Proof. We must show $(J\backslash F \xrightarrow{\bullet} G) \cong (F \xrightarrow{\bullet} G \circ J)$.

$$
\begin{aligned}
& J\backslash F \xrightarrow{\bullet} G \\
\equiv & \qquad \{ \quad \text{definition} \quad \} \\
& (\exists C.\ (J\ C \to -) \cdot F\ C) \xrightarrow{\bullet} G \\
\cong & \qquad \{ \quad \text{natural transformations are ends} \quad \} \\
& \forall E.\ (\exists C.\ (J\ C \to E) \cdot F\ C) \to G\ E \\
\cong & \qquad \{ \quad (- \to X) \text{ turns colimits into limits} \quad \} \\
& \forall E.\ \forall C.\ ((J\ C \to E) \cdot F\ C \to G\ E) \\
\cong & \qquad \{ \quad \text{currying} \quad \} \\
& \forall E.\ \forall C.\ (J\ C \to E) \to (F\ C \to G\ E) \\
\cong & \qquad \{ \quad \text{Yoneda} \quad \} \\
& \forall C.\ F\ C \to G\ (J\ C) \\
\cong & \qquad \{ \quad \text{natural transformations are ends} \quad \} \\
& F \xrightarrow{\bullet} G \circ J
\end{aligned}
$$

∎

Instead of merely showing the *existence* of an isomorphism, the above proof can in fact be interpreted as constructing a specific one: each step has some constructive justification, and the final isomorphism is the composition of all the steps. However, instead of formally expounding this isomorphism, it is useful to carry out the construction in Haskell, using the type Lan defined above. This brings out the essential

```
lanAdjoint :: Functor g ⇒ (∀c.f c → g (j c)) → (∀a.Lan j f a → g a)
lanAdjoint h = homL (uncurry (yoneda′ h))
  where
    — Turn a forall outside an arrow into an existential to the left
    — of the arrow
    homL :: (∀a c.(f c, j c → a) → g a) → (∀a.Lan j f a → g a)
    homL h (Lan (fc, jc_a)) = h (fc, jc_a)

    — One direction of the Yoneda lemma.
    yoneda :: Functor f ⇒ f c → (∀a.(c → a) → f a)
    yoneda fc h = fmap h fc

    — A particular instantiation of yoneda. This needs to be
    — declared and given a type signature, since there are higher-
    — rank types involved which GHC is not able to infer.
    yoneda′ :: Functor g
            ⇒ (∀c.f c → g (j c)                )
            → (∀c.f c → (∀a.(j c → a) → g a))
    yoneda′ h fc = yoneda (h fc)
```

Figure 6.2: (One half of) "proof" of Proposition 6.1.2 in Haskell

components of the proof without getting too bogged down in abstraction. The code corresponding to the backwards direction of the proof is shown in Figure 6.2 (note that it requires the GADTs and RankNTypes extensions).[2] The code for the forward direction is similar, and it is the backwards direction which will be of particular use later.

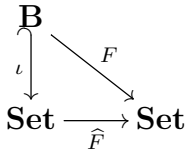
6.2 Analytic functors

We are now ready to consider Joyal's definition of *analytic functors* [Joyal, 1986]. Analytic functors, of type **Set** → **Set**, are those which arise as left Kan extensions of species. They can also intuitively be characterized as those functors **Set** → **Set** which "have a Taylor expansion". This will be made more precise in §6.2.2, although this latter definition is less immediately useful in a generalized setting.

[2] As evidenced by the `kan-extensions` package [Kmett, a], the implementation of this constructive proof in Haskell can be considerably simplified, but at the expense of obscuring the connection to the original abstract proof given above.

6.2.1 Definition and intuition

Definition 6.2.1 (Joyal). Given a species $F : \mathbf{B} \Rightarrow \mathbf{Set}$, the *analytic functor* $\widehat{F}$ corresponding to F is given by $\iota\backslash F$, the left Kan extension of F along the inclusion functor $\iota : \mathbf{B} \hookrightarrow \mathbf{Set}$. A functor $\mathbf{Set} \to \mathbf{Set}$ is *analytic* when it arises in this way from some species.



We can think of $\widehat{F}$ as the polymorphic "data type" arising from the species F. The construction in Proposition 6.1.2 tells us exactly what structures of such a data type look like:

$$\widehat{F}\ A = \exists L.\ (\iota L \to A) \times F\ L.$$

We call inhabitants of $\widehat{F}\ A$ *labelled structures*. That is, given a set A, a labelled structure of type $\widehat{F}\ A$ consists of an L-labelled F-shape together with a function (*i.e.* a morphism in **Set**) from ιL to A. The coend means that the choice of a particular label set L does not matter: any two values $f : (\iota L \to A) \times F\ L$ and $g : (\iota L' \to A) \times F\ L'$ are considered equal if there is some bijection $\sigma : L \xrightarrow{\sim} L'$ which sends f to g.

Moreover, the natural isomorphism (6.1.1) in this case becomes

$$(\widehat{F} \xrightarrow{\bullet} G) \cong (F \xrightarrow{\bullet} G\iota),$$

that is, the natural maps (*i.e.* parametrically polymorphic functions) out of $\widehat{F}$ are in one-to-one correspondence with species morphisms out of F. The isomorphism constructed in Figure 6.2 can give us some insight into the computational content of this correspondence. We identify both **B** and **Set** with **Hask**—formally dubious but close enough for intuition—and thus the inclusion functor $\iota : \mathbf{B} \to \mathbf{Set}$ becomes the identity. Let $h :: \forall c.f\ \ c \to g\ \ c$ be an arbitrary natural transformation from f to $g = g \circ \iota$, which should be thought of as a morphism between species, that is, between functors $\mathbf{B} \to \mathbf{Set}$. The *lanAdjoint* function turns such species morphisms into polymorphic functions (that is, natural transformations between $\mathbf{Set} \to \mathbf{Set}$ functors) from $\mathsf{Lan}\ \iota\ f\ \ a$ to $g\ \ a$. In particular, let $\mathsf{Lan}\ (sp, m)$ be a value of type $\mathsf{Lan}\ \iota\ f\ \ a$, containing, for some label type c, a shape $sp{:}f\ \ c$ and a mapping $m{:}\iota\ c \to a$. Then $lanAdjoint\ h\ (\mathsf{Lan}\ (sp, m))$ has type $g\ \ a$, and we can carry out the following simplication just by unfolding definitions:

$$\begin{aligned}
& lanAdjoint\ h\ (\mathsf{Lan}\ (sp, m)) \\
= & \qquad \{ \quad \text{definition of } lanAdjoint \quad \} \\
& homL\ (uncurry\ (yoneda'\ h))\ (\mathsf{Lan}\ (sp, m)) \\
= & \qquad \{ \quad \text{definition of } homL \quad \} \\
& uncurry\ (yoneda'\ h)\ (sp, m)
\end{aligned}$$

$$
\begin{array}{ll}
= & \{ \quad \text{definition of } \mathit{uncurry} \quad \} \\
\quad \mathit{yoneda}'\ h\ sp\ m & \\
= & \{ \quad \text{definition of } \mathit{yoneda}' \quad \} \\
\quad \mathit{yoneda}\ (h\ sp)\ m & \\
= & \{ \quad \text{definition of } \mathit{yoneda} \quad \} \\
\quad \mathit{fmap}\ m\ (h\ sp). &
\end{array}
$$

This can be interpreted as follows: given the species morphism h out of the species F, it is turned into a function out of the corresponding analytic functor $\widehat{F}$ by applying it to the underlying shape, and then functorially applying the associated data mapping. Note in particular that *lanAdjoint* is an *isomorphism*, which means that *every* polymorphic function out of an analytic functor arises in this way. That is, every polymorphic function out of $\widehat{F}\ A$ is "just a reshaping": it is equivalent to a process consisting of splitting a labelled structure of type $\widehat{F}\ A$ into a labelled shape and a mapping from labels to data, followed by a "reshaping"—an application of some species morphism to the shape—and concluding with re-combining the new shape with the data mapping.

Such a reshaping only has access to the labelled shape, and not to the values of type A, so it obviously cannot depend on them. However, this is not surprising, since this property is already implied by naturality. More interesting is the fact that the set of labels must be finite. This means, intuitively, that functors corresponding to infinite data structures are not analytic. It is not possible to represent all possible natural maps out of an infinite data structure by natural maps out of structures containing only a finite number of labels. This is proved more formally in §6.2.3.

Analytic functors have a close connection to the *shapely types* of Jay and Cockett [1994]. Shapely types essentially correspond to analytic functors over **L**-species; shapely types are those which decompose into a shape and a *list* of data, which can be thought of as a mapping from a linearly ordered set of labels to data.

6.2.2 Analytic functors and generating functions

Joyal [1986] showed that analytic functors can also be characterized as those which "have Taylor expansions" (in a suitable sense). Passing from **B** to **P**, suppose we have a species $F : \mathbf{P} \to \mathbf{Set}$; then the analytic functor $\widehat{F}$ is given by

$$\widehat{F}\ A = \exists (n : \mathbb{N}).\ (\iota n \to A) \times F\ n,$$

where $\iota : \mathbf{P} \to \mathbf{Set}$ in this case sends the natural number n to the set $[n]$. Note that functions $[n] \to A$ are in bijection with the n-fold product A^n, so $\widehat{F}\ A$ may equivalently be expressed as

$$\widehat{F}\ A \cong \exists (n : \mathbb{N}).\ A^n \times F\ n \cong \exists (n : \mathbb{N}).\ F\ n \times A^n.$$

The coend, in this case, is a quotient by permutations on $[n]$, which act on $F\ n \times A^n$ by permuting the elements of the n-fold product. So each value of the coend is an equivalence class of $n!$ pairs, one for each possible permutation of A^n. We may therefore suggestively (if informally) write

$$\widehat{F}\ A \approx \sum_{n:\mathbb{N}} F\ n \times \frac{A^n}{n!}$$

which very strongly resembles the exponential generating function associated to the species F,

$$F(x) = \sum_{n \geqslant 0} |F\ n| \times \frac{x^n}{n!}.$$

Of course, the resemblance is no accident! This gives another glimpse of the sense in which species (and their associated analytic functors) are said to be a categorification of such generating functions.

6.2.3 Analytic functors and finiteness

Joyal [1986] also gave yet another characterization of analytic functors, namely, those which preserve *filtered colimits*, *cofiltered limits*, and *weak pullbacks*. It is instructive to use this characterization as a lens to consider some examples of functors which are *not* analytic.

Definition 6.2.2. A *filtered* category $\mathbb{C}$ [Adámek et al., 2002] is one which "has all finite cocones", that is, for any finite collection of objects and morphisms in $\mathbb{C}$, there is some object $C \in \mathbb{C}$ with morphisms from all the objects in the collection to C, such that all the relevant triangles commute.

Equivalently, and more simply, a filtered category is one for which

- there exists at least one object;
- any two objects $C_1, C_2 \in \mathbb{C}$ have an "upper bound", that is, an object C_3 with morphisms
$$C_1 \longrightarrow C_3 \longleftarrow C_2\ ;$$
- and finally, any two parallel morphisms $C_1 \overset{f}{\underset{g}{\rightrightarrows}} C_2$ also have an "upper bound", that is, another morphism
$$C_1 \overset{f}{\underset{g}{\rightrightarrows}} C_2 \xrightarrow{h} C_3$$
such that $f\ ;h = g\ ;h$.

These binary upper bound operations on objects and morphisms may be used to inductively "build up" cocones for arbitrary diagrams in $\mathbb{C}$.

This can be seen as a "categorification" of the notion of a *directed set* (also known as a *filtered set*), a preorder in which any two elements have an upper bound. Categories can be seen as generalizations of preorders in which multiple morphisms are allowed between each pair of objects, so the above definition has to extend the idea of pairwise upper bounds to apply to parallel morphisms as well as objects; in a preorder there are no parallel morphisms so this does not come up.

Example. Any category with a terminal object is filtered: the terminal object may be taken as the upper bound of any two objects, and the unique morphism to the terminal object as the upper bound of any two parallel morphisms.

Example. The poset $(\mathbb{N}, \leqslant)$, considered as a category whose objects are natural numbers, with morphisms $m \leqslant n$, is a filtered category. The upper bound of any two objects is their maximum, and there are no parallel morphisms to consider.

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \dots$$

Note that filteredness only requires that every *finite* collection of objects have an upper bound; in particular, in this example it is not true of *infinite* collections of objects. For example, the set of all even numbers has no upper bound in $\mathbb{N}$.

Example. Consider the category $\mathbf{Fin}\mathbb{N}_{\subseteq}$ whose objects are finite subsets of $\mathbb{N}$ and whose morphisms are inclusion maps. That is, whenever $S \subseteq T$ there is a single morphism $\iota_{ST} : S \to T$ defined by $\iota_{ST}(s) = s$. Since this is a nonempty preorder, to see that $\mathbf{Fin}\mathbb{N}_{\subseteq}$ is filtered it suffices to note that any two finite sets S and T have $S \cup T$ as an upper bound.

Example. Filtered categories can also be seen as a generalization of finitely cocomplete categories, *i.e.* categories having all finite colimits. In particular, categories having all finite colimits can be characterized as those having an initial object, all binary coproducts, and all coequalizers: these are exactly parallel to the three criteria given above for filtered categories, with an extra "universal property" corresponding to each (for example, the binary coproduct of two objects is an upper bound along with a universal property).

Therefore, any (finitely) cocomplete category is automatically filtered: for example, **Set**, **Grp**, and **Vec**.

Recall that a *diagram* in $\mathbb{C}$ is a functor $\mathcal{I} \to \mathbb{C}$ from some "index category" $\mathcal{I}$, which determines the "shape" of diagrams in $\mathbb{C}$.

Definition 6.2.3. A *filtered diagram* in $\mathbb{C}$ is a functor $\mathcal{I} \to \mathbb{C}$ from a filtered index category $\mathcal{I}$. A *filtered colimit* is a colimit of a filtered diagram.

That is, a filtered diagram in $\mathbb{C}$ is a diagram that "looks like" a filtered category "sitting inside" $\mathbb{C}$. A filtered colimit is then just a normal colimit which happens to be taken over a filtered diagram.

Example. Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor on the category $\mathbb{C}$. Suppose $\mathbb{C}$ contains an initial object 0, and let ! denote the unique morphism $0 \to C$. Then consider the diagram

$$0 \xrightarrow{!} F0 \xrightarrow{F!} F^2 0 \xrightarrow{F^2!} F^3 0 \longrightarrow \dots$$

The colimit of this diagram is the least fixed point μF, and is a filtered colimit since the diagram has the filtered poset $(\mathbb{N}, \leqslant)$ as its index category.

Example. Pushouts are an example of colimits which are *not* filtered, since pushouts are colimits over a span $X \longleftarrow Z \longrightarrow Y$, which is not filtered (X and Y do not necessarily have an upper bound).

Example. Recall the filtered poset $\mathbf{Fin}\mathbb{N}_{\subseteq}$ introduced earlier, consisting of finite subsets of $\mathbb{N}$ and inclusion maps. The inclusion functor $\mathbf{Fin}\mathbb{N}_{\subseteq} \hookrightarrow \mathbf{Set}$ allows viewing $\mathbf{Fin}\mathbb{N}_{\subseteq}$ as a diagram in **Set**, and we consider the (filtered) colimit of this diagram, which must consist of some set S along with maps from all the finite subsets of $\mathbb{N}$ into S, which commute with the inclusion maps among the finite subsets of $\mathbb{N}$. In fact, it suffices to take $\mathbb{N}$ itself, together with the inclusion maps from each finite subset of $\mathbb{N}$ into $\mathbb{N}$. Intuitively, $\mathbb{N}$ arises here as the disjoint union of all finite subsets of $\mathbb{N}$, quotiented by the relationships induced by all the inclusion maps—which collapses the disjointness, resulting in a simple union of all finite subsets.

To see that this is universal, suppose we have a set X with maps $m_S : S \to X$ for each finite $S \subset \mathbb{N}$, such that the m_S all commute with inclusion maps between the finite subsets of $\mathbb{N}$. Define $\theta : \mathbb{N} \to X$ by $\theta(n) = m_{\{n\}}(n)$. We must show that the m_S all factor through θ.

$$\begin{array}{ccc} S & \longleftarrow & \{k\} \\ {\scriptstyle \iota_S}\downarrow & \searrow^{m_S} & \downarrow{\scriptstyle m_{\{k\}}} \\ \mathbb{N} & \xrightarrow[\theta]{} & X \end{array}$$

Given some $S \subset \mathbb{N}$ and some $k \in S$, we have $\theta(\iota_S(k)) = \theta(k) = m_{\{k\}}(k)$; but this is indeed equal to $m_S(k)$, since there exists an inclusion map $\{k\} \to S$, and we assumed the m_S commute with inclusion maps.

Now consider the functor $F := (-)^{\mathbb{N}} : \mathbf{Set} \to \mathbf{Set}$, which sends the set A to the set $A^{\mathbb{N}}$ of functions from $\mathbb{N}$ to A [Trimble, 2014]. The claim is that F is not analytic, and in particular that it does not preserve the filtered colimit of $\mathbf{Fin}\mathbb{N}_{\subseteq}$, discussed above. As we will see, the "problem" is that F corresponds to an *infinite* data type, *i.e.* one which can contain infinitely many A values. In particular, F corresponds to the data type of *infinite streams*: a function $\mathbb{N} \to A$ can be thought of as an infinite

stream of A values, where the value of the function at n gives the value of A located at position n in the stream.

We also consider how F acts on inclusion maps. The action of F on morphisms is given by postcomposition, so F sends the inclusion $\iota : S \hookrightarrow T$ to $\iota \circ - : S^{\mathbb{N}} \to T^{\mathbb{N}}$, which is also an inclusion map: it sends the stream $s : \mathbb{N} \to S$ to the stream $\iota \circ s : \mathbb{N} \to T$, consisting of the application of ι to every element in s. That is, $\iota \circ -$ does not actually modify any values of a stream, but simply codifies the observation that whenever $S \subseteq T$, a stream containing only values from S may also be thought of as a stream containing only values from T (which simply happens not to include any values from $T - S$).

We saw above that the colimit of $\mathbf{Fin}\mathbb{N}_{\subseteq}$, considered as a diagram in **Set**, is $\mathbb{N}$ (together with the obvious inclusion maps to $\mathbb{N}$ from each finite subset). F sends $\mathbb{N}$ to $\mathbb{N}^{\mathbb{N}}$, the type of infinite streams of natural numbers. F also sends each inclusion map $S \hookrightarrow \mathbb{N}$ to the inclusion $S^{\mathbb{N}} \hookrightarrow \mathbb{N}^{\mathbb{N}}$, which allows a stream of S values to be "upgraded" to a stream of natural numbers.

Now consider where F sends the diagram $\mathbf{Fin}\mathbb{N}_{\subseteq}$. F sends each finite set $S \subset \mathbb{N}$ to the set of infinite streams of S values, $S^{\mathbb{N}}$, and it sends each inclusion $S \hookrightarrow T$ to the inclusion $S^{\mathbb{N}} \hookrightarrow T^{\mathbb{N}}$. However, the colimit of this new diagram $F(\mathbf{Fin}\mathbb{N}_{\subseteq})$ is not $\mathbb{N}^{\mathbb{N}}$, the set of streams of natural numbers, but instead the set of *finitely supported* streams of natural numbers, that is, the set of all streams which contain only finitely many distinct elements. Thus $F\ (\operatorname{colim} \mathbf{Fin}\mathbb{N}_{\subseteq}) \ncong \operatorname{colim}(F\ \mathbf{Fin}\mathbb{N}_{\subseteq})$, and we conclude that F is not analytic since it does not preserve filtered colimits.

Another example[3] is given by the covariant power set functor $P : \mathbf{Set} \to \mathbf{Set}$, which sends each set A to its power set $P(A)$, the set of all subsets of A, and sends each function $f : A \to B$ to the function $P(f) : P(A) \to P(B)$ which gives the image of a subset of A under f. $P(\mathbb{N})$ is the set of all (finite and infinite) subsets of $\mathbb{N}$, but $\operatorname{colim} P(\mathbf{Fin}\mathbb{N}_{\subseteq})$ is the set of all *finite* subsets of $\mathbb{N}$. Note, however, that the covariant finite powerset functor $FP : \mathbf{Set} \to \mathbf{Set}$, which sends each set A to the set of all its *finite* subsets, is analytic; it corresponds to the species $\mathsf{E} \cdot \mathsf{E}$.

6.3 An attempt at generalized functor composition

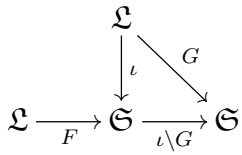
Recall from §4.4 that if species are taken to be functors $\mathbf{B} \to \mathbf{B}$, then one can define the *functor composition* $G \,\square\, F$ to be the literal composition of G and F, that is, $(G \,\square\, F)\ L = G\ (F\ L)$. However, if species are taken as functors $\mathbf{B} \to \mathbf{Set}$ it is not clear how to generalize this idea. Given the idea of analytic functors, we are ready to consider one possible idea—which unfortunately does not work.

It seems we must begin by restricting ourselves to functors $F, G : \mathbf{B} \to \mathbf{FinSet}$; we cannot use infinite sets of shapes as label sets. In that case, as observed previously,

[3] Also due to Trimble [2014].

functors $\mathbf{B} \to \mathbf{FinSet}$ are essentially equivalent to functors $\mathbf{B} \to \mathbf{B}$, so in fact defining functor composition for species $\mathbf{B} \to \mathbf{FinSet}$ is not difficult. However, it still does not give us any idea of how to generalize to $(\mathfrak{L}, \mathfrak{S})$-species.

There is another possibility: given some embedding functor $\iota : \mathfrak{L} \to \mathfrak{S}$ (with some mild restrictions), there is a monoidal structure on $\mathfrak{L} \Rightarrow \mathfrak{S}$, given by $G \bullet F = F \circ (\iota \backslash G)$, that is,



[Altenkirch et al., 2010]. Unfortunately, if we examine the special case of $\mathbf{B} \to \mathbf{FinSet}$, we can see that this monoidal structure is *not* the same as functor composition. To see the difference, suppose $G : \mathbf{B} \to \mathbf{FinSet}$ is some species all of whose shapes contain each label exactly once. Note that $\iota \backslash G$ is by definition $\widehat{G}$, the analytic functor corresponding to G. Therefore, $(\widehat{G} \circ F)$-shapes on L consist of G-structures containing $(F\ L)$-shapes as data. Note that any particular $(F\ L)$-shape can occur multiple times or none at all. In contrast, a $(G \,\square\, F)$-shape contains every possible F-shape exactly once.

6.4 Introduction and elimination forms for labelled structures

Deriving and fully working out introduction forms for analytic functors and labelled structures remains future work. However, much of the work has already been done. For example, Joyal [1986] shows that analytic functors are closed under sums, products, composition, and least fixed points; moreover, the mapping $F \mapsto \widehat{F}$ from species to their corresponding analytic functors is homomorphic with respect to these operations. For example, one has

$$\begin{aligned}
\widehat{(F + G)} &= \widehat{F} + \widehat{G} \\
\widehat{(F \cdot G)} &= \widehat{F} \times \widehat{G} \\
\widehat{(F \circ G)} &= \widehat{F} \circ \widehat{G} \\
\widehat{1} &= \Delta_1 \\
\widehat{\mathsf{X}} &= id
\end{aligned}$$

(for proof, see Joyal [1986]). This means that, for example, in order to introduce a labelled structure of shape $F \cdot G$, one simply gives a pair of a labelled F-structure and a labelled G-structure; this should come as no surprise.

Now consider how to *eliminate* labelled structures. In particular, given a labelled

structure type $\widehat{F}\ A$, consider a morphism $\widehat{F}\ A \to B$, for some arbitrary object B. We compute as follows, noting that these computations apply equally in $(\mathbf{B} \Rightarrow \mathbf{Set})$ or $(\mathcal{B} \Rightarrow \mathcal{S})$:

$$
\begin{array}{ll}
& \widehat{F}\ A \to B \\
= & \qquad \{ \quad \text{definition} \quad \} \\
& (\exists L.\ F\ L \times (\iota\ L \to A)) \to B \\
\cong & \qquad \{ \quad (- \to B) \text{ turns colimits into limits} \quad \} \\
& \forall L.\ (F\ L \times (\iota\ L \to A)) \to B \\
\cong & \qquad \{ \quad \text{currying} \quad \} \\
& \forall L.\ F\ L \to ((\iota L \to A) \to B) \\
\cong & \qquad \{ \quad \text{ends are natural transformations} \quad \} \\
& F \xrightarrow{\bullet} ((\iota - \to A) \to B)
\end{array}
$$

Note that $((\iota - \to A) \to B)$ is a functor of type $\mathbf{B} \to \mathbf{Set}$, that is, a species, which we will abbreviate B^{A^-}. Hence the above derivation amounts to saying that a function $(\widehat{F}\ A \to B)$ eliminating a labelled structure is equivalent to a species morphism $F \to B^{A^-}$. We can therefore characterize labelled structure eliminators in terms of the species eliminators described in §4.7.

6.4.1 Generalized analytic functors

In general, suppose we have a functor $F : \mathfrak{L} \to \mathfrak{S}$, as well as a functor $\iota : \mathfrak{L} \to \mathfrak{S}$. We can define

$$\widehat{F} := \exists L.\ (\iota\ L \Rightarrow A) \cdot F\ L$$

as long as

- $\mathfrak{S}$ is copowered over **Set**, *i.e.* has all coproducts, and
- $\mathfrak{S}$ has coends over $\mathfrak{L}$.

As a particular example, the definition of analytic functors ports almost unchanged into homotopy type theory: we merely replace the set-theoretic categories **B** and **Set** with the homotopy-theoretic $\mathcal{B}$ and $\mathcal{S}$, respectively, yielding

$$\widehat{F}\ A = \exists L.\ (\iota\ L \Rightarrow_{\mathcal{S}} A) \times F\ L,$$

where $\iota : \mathcal{B} \to \mathcal{S}$ is the evident injection which acts on morphisms via transport. Recalling that coends in HoTT are just Σ-types, and that morphisms in $\mathcal{S}$ are functions, we have

$$\widehat{F}\ A = \sum_{L:\mathcal{B}} (\iota\ L \to A) \times F\ L.$$

The following section gives another example, of analytic functors over partial and copartial species.

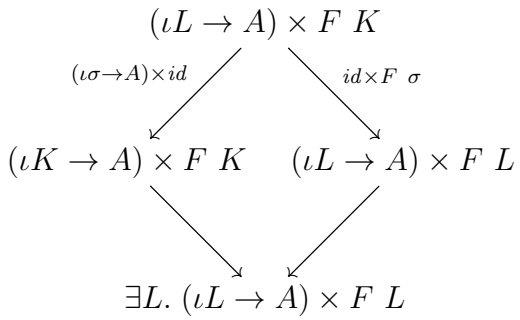


Figure 6.3: The commuting condition for analytic functors over copartial species

6.5 Analytic functors for partial and copartial species

A variant of labelled structures that one might consider is one which "takes the coend off", exposing the labels in the type:

$$\langle F \rangle_L\ A := (\iota L \to A) \times F\ L.$$

We have seen that introduction and elimination forms for labelled structures are straightforward, for example, eliminating $\widehat{(F \cdot G)}\ A \cong (\widehat{F} \times \widehat{G})\ A \cong \widehat{F}\ A\ = \widehat{G}\ A$ just amounts to eliminating a pair. However, eliminating $\langle F \rangle_L\ A$ is not so straightforward; in particular we do *not* have $\langle F \cdot G \rangle_L\ A \cong \langle F \rangle_L\ A \times \langle G \rangle_L\ A$, since the labels L need to be partitioned between the two shapes.

However, we can recover something along these lines using copartial species (§5.2); the idea is that $\langle F \rangle_L\ A$ will contain a shape on some *subset* of the labels L. One way to think of this is as having some large "shared memory" indexed by L, and a labelled structure $\langle F \rangle_L\ A$ gives us a "view" onto some (but not necessarily all) of the data in memory. Thus, it is possible to decompose a product as $\langle F \cdot G \rangle_L\ A \to \langle F \rangle_L\ A \times \langle G \rangle_L\ A$ (note this is *not* an isomorphism), breaking an $F \cdot G$ structure into two separate shapes referencing data contained in the same shared memory. Aside from allowing us to express an eliminator for such labelled products, from an operational point of view, this may also be exactly what we want when dealing with data structures whose memory allocation really matters: doing a decomposition operation should not allocate any new storage, but simply yield new shapes with labels pointing into existing storage.

More formally, consider constructing analytic functors over copartial species $\mathcal{B}_{\subseteq} \Rightarrow \mathcal{S}$. There is a natural injection $\iota : \mathcal{B}_{\subseteq} \to \mathcal{S}$, which we can use to define such analytic functors via the usual Kan extension formula, $\widehat{F} := \exists L.\ (\iota\ L \Rightarrow A) \cdot F\ L$. The commuting condition for the coend is shown in Figure 6.3. Here $\sigma : K \subseteq L$ is a copartial bijection from K into L. Since $\sigma : K \subseteq L$, the top of the diagram represents a labelled shape (of type $F\ K$) with its data "living inside" some larger shared memory

labelled by L. Following the right-hand arrow amounts to explicitly expanding the shape with the extra labels (*e.g.* throwing the extra labels into the "catch-all" set); following the left-hand arrow amounts to throwing away the extra data which is not explicitly referenced by the F-shape. According to the diagram, these must both inject into the coend in a way that renders them indistinguishable: intuitively, this means that if we have a shape with data living in a larger shared memory, we cannot observe anything about the data values not explicitly referenced by the shape.

We can also consider analytic functors over partial species in $\mathcal{B}_{\subseteq}^{\mathrm{op}} \Rightarrow \mathcal{S}_{\rightharpoonup}$, using the natural inclusion functor $\iota : \mathcal{B}_{\subseteq}^{\mathrm{op}} \to \mathcal{S}_{\rightharpoonup}$. The commutative diagram for the coend looks almost identical to the one in Figure 6.3, but is interpreted somewhat differently. In particular, note that the arrows now represent *partial* functions. In addition, $\sigma : K \supseteq L$ is a constructive witness that K is a superset of L. Thus, $(\iota L \to A) \times F\ K$ is a shape with labels taken from K along with a partial mapping from only *some* of those labels to data values. Following the right-hand arrow amounts to taking a subshape of F corresponding to the labels that do map to data; following the left-hand arrow amounts to expanding the domain of the mapping with extra labels for which the mapping is undefined.

More work is needed to flesh out the details, but it seems that this may allow us to model structures where we do not wish to store data associated with every label. For example, $\mathsf{X} \cdot \mathsf{E}$ may represent a pointed set, but it may also represent the species of "elements", where we really only care about the data associated with the label of X, and do not wish to associate data to all the labels contained in the set.

Chapter 7

Conclusion and future work

This dissertation has laid the theoretical groundwork to pursue applications of combinatorial species (and variants thereof) to algebraic data types in functional programming languages. It is too early to say with confidence what sorts of applications there might be, though the future work laid out below hints at some ideas.

Two aspects of this work have turned out to be particularly surprising and gratifying to me. The first is the host of nontrivial issues that arise when attempting to formalize species in a constructive type theory—and the way that homotopy type theory is able to neatly dispatch them all. I hope I have successfully made the case that HoTT is the right framework in which to carry out this work; in any case it is a lot of fun to see such a recent and groundbreaking theory put to productive work in a different area of mathematics (namely, combinatorics). The second gratifying aspect of this work is the way that analytic functors neatly encapsulate the idea of labelled shapes associated with a mapping from labels to data. Indeed, Jacques and I had the basic idea of associating shapes and mappings for a long time before realizing that what we were thinking of were "just" analytic functors.

Although this dissertation merely hints at practical applications, I feel that building on the foundations of this work, practical applications will not be far behind. In general, there is no shortage of future work! This dissertation has just scratched the surface of what is possible, and indeed, some of my initial conceptions of what my thesis would contain seem to actually constitute a viable five- or ten-year research program. I mention here some of the most promising avenues for continued work in this area.

- This dissertation discussed exponential and ordinary generating functions, but omitted *cycle index series*, which are a generalization of both (and which are necessary for computing, *e.g.*, ogfs of compositions). For that matter, even ogfs were not discussed much, but correspond to "unlabelled" structures, which may often be what one actually wants to work with. My gut sense is that there is quite a lot more that could be said about computational interpretation of generating functions, and that this may have very practical applications, for example, in

the enumeration and random generation of data structures.

- As mentioned in the discussion of Conjecture 3.3.5, it seems promising to translate the theory of molecular and atomic species into homotopy type theory, ideally using Coq or Agda to formalize the development. My sense is that this will shed additional light on the theory, and may have some practical applications as well.

- It seems there is something interesting that could be said about recursively defined ($\mathbf{B} \Rightarrow \mathbf{Set}$)-species, where infinite families of same-size shapes are allowed. Some of the criteria for the implicit species theorems presented in §5.4.1 work simply to prevent such infinite families, and hence are not needed in such a setting. It would be interesting to explore minimal criteria for analogues of the implicit species theorems in more generalized settings.

- Although the introduction mentions generic programming, the remainder of the dissertation has little to say on the topic, but there are certainly interesting connections to be made. In a practical vein, using generic programming, it should be possible to create tools that allow algebraic data types to be manipulated via generic "views" as species.

- It seems that there ought to be some sort of connection between species and linear logic. In general, labels are "treated linearly"; partitional product feels analogous to multiplicative conjunction, Cartesian product to additive conjunction, and species sum to additive disjunction. This suggests looking for a species operation analogous to multiplicative disjunction, although I have not been able to make sense of such an operation[1]. It seems worthwhile to investigate the possibility of a real, deeper connection between species and linear logic.

- §6.5 considered a variant of $\widehat{F}A$ which "removed the coend", exposing the labels in the type:

  $$\langle F \rangle_L \; A := (\iota L \to A) \times F \; L.$$

  It is still not clear, however, whether there is any benefit to being able to explicitly talk about the label type in this way. Coming up with a more precise story about such "exposed" labelled structures—or showing conclusively why one does not want to work with them—would be an important next step.

[1] Much like multiplicative disjunction itself.

Appendix A

Lifting monoids

This chapter contains a detailed proof showing how monoids (and many other structures of interest) can be lifted from a category $\mathfrak{S}$ to a functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$. The high-level ideas of this construction seem to be "folklore", but I have been unable to find any detailed published account, so it seemed good to include some proofs here for completeness. Unfortunately, the proof presented here is still incomplete; as future work I hope to completely understand the proof in detail.

We must first develop some technical machinery regarding functor categories. In particular, we show how to lift objects, functors, and natural transformations based on the category $\mathfrak{S}$ into related ones based on the functor category $\mathfrak{S}^{\mathfrak{L}}$.

Lemma A.1. *An object $D \in \mathbb{D}$ lifts to an object (i.e. a functor) $D^{\mathbb{C}} \in \mathbb{D}^{\mathbb{C}}$, defined as the constant functor Δ_D.*

Lemma A.2. *Any functor $F : \mathbb{D} \to \mathbb{E}$ lifts to a functor $F^{\mathbb{C}} : \mathbb{D}^{\mathbb{C}} \to \mathbb{E}^{\mathbb{C}}$ given by postcomposition with F. That is, $F^{\mathbb{C}}(G) = F \circ G = FG$, and $F^{\mathbb{C}}(\alpha) = F\alpha$.*

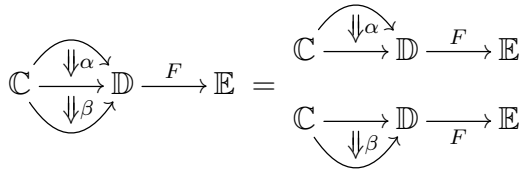
Proof. $F\alpha$ denotes the "right whiskering" of α by F,

$$\mathbb{C} \underset{H}{\overset{G}{\Downarrow \alpha}} \mathbb{D} \xrightarrow{F} \mathbb{E}.$$

$F^{\mathbb{C}}$ preserves identities since

$$\mathbb{C} \xrightarrow{G} \mathbb{D} \xrightarrow{F} \mathbb{E}$$

can be seen as both Fid_G and id_{FG}, and it preserves composition since

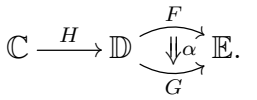


by the interchange law for horizontal and vertical composition. ∎

Natural transformations lift in the same way:

Lemma A.3. *Given functors $F, G : \mathbb{D} \to \mathbb{E}$, any natural transformation $\alpha : F \xrightarrow{\bullet} G$ lifts to a natural transformation $\alpha^{\mathbb{C}} : F^{\mathbb{C}} \xrightarrow{\bullet} G^{\mathbb{C}} : \mathbb{D}^{\mathbb{C}} \to \mathbb{E}^{\mathbb{C}}$ given by postcomposition with α. That is, the component of $\alpha^{\mathbb{C}}$ at $H : \mathbb{C} \to \mathbb{D}$ is $\alpha^{\mathbb{C}}_H = \alpha H$. Moreover, if α is an isomorphism then so is $\alpha^{\mathbb{C}}$.*

Proof. Here αH denotes the "left whiskering" of α by H,



Note that $\alpha^{\mathbb{C}}_H$ should be a morphism $F^{\mathbb{C}}H \longrightarrow G^{\mathbb{C}}H$ in $\mathbb{E}^{\mathbb{C}}$, that is, a natural transformation $FH \xrightarrow{\bullet} GH$, so αH has the right type. The naturality square for $\alpha^{\mathbb{C}}$ is

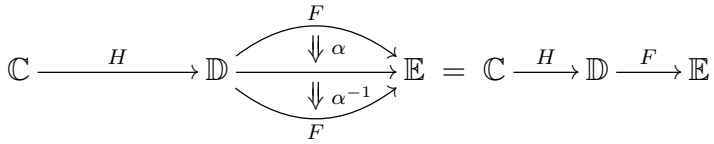
$$\begin{array}{ccc} FH & \xrightarrow{\alpha^{\mathbb{C}}_H} & GH \\ {\scriptstyle F\beta}\downarrow & & \downarrow{\scriptstyle G\beta} \\ FJ & \xrightarrow[\alpha^{\mathbb{C}}_J]{} & GJ \end{array}$$

which commutes by naturality of α: at any particular $C \in \mathbb{C}$ the above diagram reduces to:

$$\begin{array}{ccc} FHC & \xrightarrow{\alpha_{HC}} & GHC \\ {\scriptstyle F\beta_C}\downarrow & & \downarrow{\scriptstyle G\beta_C} \\ FJC & \xrightarrow[\alpha_{JC}]{} & GJC \end{array}$$

If α is an isomorphism, then $(\alpha^{-1})^{\mathbb{C}}$ is the inverse of $\alpha^{\mathbb{C}}$: for any H, $\alpha^{-1}H \cdot \alpha H =$

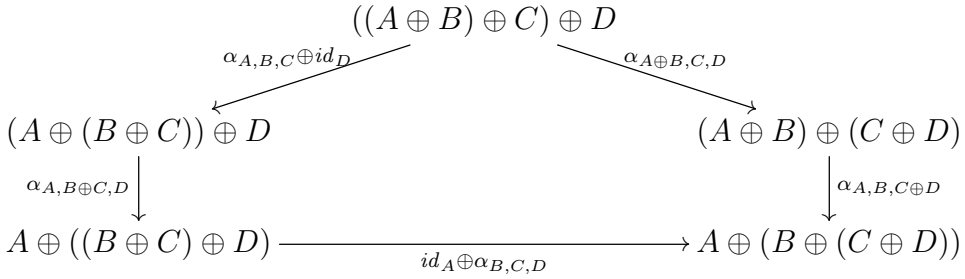
$(\alpha^{-1} \cdot \alpha)H = id_{FH}$.



SDG

Finally, we need to know that *laws*—expressed as commutative diagrams—also lift appropriately from $\mathbb{D}$ to $\mathbb{D}^{\mathbb{C}}$. For example, if we lift the functor and natural transformations defining a monoid in $\mathbb{D}$, we need to know that the resulting lifted functor and lifted natural transformations still define a valid monoid in $\mathbb{D}^{\mathbb{C}}$.

The first step is to understand how to appropriately encode laws as categorical objects. Consider a typical commutative diagram, such as the following diagram expressing the coherence of the associator for a monoidal category. The parameters to all the instances of α have been written out explicitly, to make the subsequent discussion clearer, although in common practice these would be left implicit.



There are two important points to note. The first is that any commutative diagram has a particular *shape* and can be represented as a functor from an "index category" representing the shape (in this case, a category having five objects and morphisms forming a pentagon, along with the required composites) into the category in which the diagram is supposed to live. Such a functor will pick out certain objects and morphisms of the right "shape" in the target category, and the functor laws will ensure that the target diagram commutes in the same ways as the index category. (This much should be familiar to anyone who has studied abstract limits and colimits.) The second point is that this diagram, like many such diagrams, is really supposed to hold for *all* objects A, B, C, D. So instead of thinking of this diagram as living in a category $\mathbb{C}$, where the vertices of the diagram are objects of $\mathbb{C}$ and the edges are morphisms, we can think of it as living in $\mathbb{C}^4 \Rightarrow \mathbb{C}$, where the vertices are *functors* $\mathbb{C}^4 \to \mathbb{C}$ (for example, the top vertex is the functor which takes the quadruple of objects (A, B, C, D) and sends them to the object $((A \oplus B) \oplus C) \oplus D$), and the edges are natural transformations.

All told, then, we can think of a typical diagram D in $\mathbb{C}$ as a functor $D : J \to (\mathbb{C}^A \Rightarrow \mathbb{C})$, where A is some (discrete) category recording the arity of the diagram.

Lemma A.4. *Any diagram $D : J \to (\mathbb{C}^A \Rightarrow \mathbb{C})$ in $\mathbb{C}$ lifts to a diagram $D^{\mathbb{D}} : J \to ((\mathbb{C}^{\mathbb{D}})^A \Rightarrow \mathbb{C}^{\mathbb{D}})$ in $\mathbb{C}^{\mathbb{D}}$.*

Proof. This amounts to implementing a higher-order function with the type

$$(J \to (A \to \mathbb{C}) \to \mathbb{C}) \to J \to (A \to \mathbb{D} \to \mathbb{C}) \to \mathbb{D} \to \mathbb{C}$$

which can be easily done as follows:

$$\Phi\ D\ j\ g\ d = D\ j\ (\lambda a.\ g\ a\ d).$$

Of course there are some technical conditions to check, but they all fall out easily. ∎

At this point there is a gap in the proof. To know that this lifting does the right thing, one must show that the lifted diagram defined above is "about" (*i.e.* has as its vertices and edges) the lifted versions of the vertices and edges of the original diagram. Even this is still not quite enough; to really know that the lifted diagram "says the same thing" as the unlifted diagram, we need to show not just that the vertices and edges of the lifted diagram are lifted versions of the original diagram's vertices and edges, but that these lifted vertices and edges are themselves composed of lifted versions of the components of the originals. For example, we want to ensure that the lifting of the example diagram shown above still expresses coherence of the lifted associator with respect to the lifted tensor product. It is not enough to have vertices like $(((A \oplus B) \oplus C) \oplus D)^{\mathbb{D}}$; we must show this is the same as $((A^{\mathbb{D}} \oplus^{\mathbb{D}} B^{\mathbb{D}}) \oplus^{\mathbb{D}} C^{\mathbb{D}}) \oplus^{\mathbb{D}} D^{\mathbb{D}}$, so that it says something about the lifted tensor product $\oplus^{\mathbb{D}}$.

The basic idea would be to write down a formal syntax for the functors and natural transformations that may constitute a diagram, and show that the lifting of an expression is the same as the original expression with its atomic elements each replaced by their lifting.

Assuming this result for now, we can go on to show how monoids lift into a functor category.

Theorem A.5. *Any monoidal structure $(\otimes, I, \alpha, \lambda, \rho)$ on a category $\mathfrak{S}$ lifts pointwise to a monoidal structure $(\otimes^{\mathfrak{L}}, I^{\mathfrak{L}}, \alpha^{\mathfrak{L}}, \lambda^{\mathfrak{L}}, \rho^{\mathfrak{L}})$ on the functor category $\mathfrak{L} \Rightarrow \mathfrak{S}$.*

Proof. Immediate from Propositions A.1, A.2, and A.3, and our assumed result that diagrams lift to diagrams which "say the same thing" as the original, but say it "about" lifted things. ∎

In Proposition 4.1.7 it was claimed that this lifting preserves products, coproducts, symmetry, and distributivity. We can already show that symmetry and distributivity are preserved:

Proposition A.6. *The lifting defined in Theorem A.5 preserves symmetry.*

Proof. Symmetry is given by a natural isomorphism $\forall XY.\ X \otimes Y \simeq Y \otimes X$. By our previous assumption, this lifts to a natural isomorphism $\forall FG.\ F \otimes^{\mathfrak{L}} G \simeq G \otimes^{\mathfrak{L}} F$. ▨

Proposition A.7. *The lifting defined in Theorem A.5 preserves distributivity.*

Proof. In any category with all products and coproducts, there is a natural transformation $\forall XYZ.\ X \times Y + X \times Z \to X \times (Y + Z)$, given by $\langle [\pi_1, \pi_1], [\pi_2, \pi_2] \rangle$. The category is *distributive* if this is an isomorphism. Again by our assumption about lifting, such an isomorphism lifts to another natural isomorphism

$$\forall FGH.\ (F \times^{\mathfrak{L}} G) +^{\mathfrak{L}} (F \times^{\mathfrak{L}} H) \to F \times^{\mathfrak{L}} (G +^{\mathfrak{L}} H).$$ ▨

To show that products and coproducts are preserved requires first showing that lifting preserves adjunctions.

Lemma A.8. *Let $F : \mathbb{D} \to \mathbb{E}$ and $G : \mathbb{D} \leftarrow \mathbb{E}$ be functors. If $F \dashv G$, then $F^{\mathbb{C}} \dashv G^{\mathbb{C}}$.*

Proof. Since $F \dashv G$, assume we have $\gamma_{A,B} : \mathbb{E}(FA, B) \cong \mathbb{D}(A, GB)$. To show $F^{\mathbb{C}} \dashv G^{\mathbb{C}}$, we must define a natural isomorphism $\gamma^{\mathbb{C}}_{H,J} : \mathbb{E}^{\mathbb{C}}(F \circ H, J) \cong \mathbb{D}^{\mathbb{C}}(H, G \circ J)$. Given $\phi \in \mathbb{E}^{\mathbb{C}}(FH, J)$, that is, $\phi : FH \xrightarrow{\bullet} J : \mathbb{C} \to \mathbb{E}$, and an object $C \in \mathbb{C}$, define

$$\gamma^{\mathbb{C}}_{H,J}(\phi)_C = \gamma_{HC,JC}(\phi_C).$$

Note that $\gamma_{HC,JC} : \mathbb{E}(FHC, JC) \cong \mathbb{D}(HC, GJC)$, so it sends $\phi_C : FHC \to JC$ to a morphism $HC \to GJC$, as required.

From the fact that γ is an isomorphism it thus follows directly that $\gamma^{\mathbb{C}}$ is an isomorphism as well. Naturality of $\gamma^{\mathbb{C}}$ also follows straightforwardly from naturality of γ. For a more detailed proof, see Hinze [2012, pp. 17–18]. ▨

Proposition A.9. *The lifting defined in Theorem A.5 preserves coproducts and products.*

Proof. Consider a category $\mathfrak{S}$ with coproducts, given by a bifunctor $+ : \mathfrak{S} \times \mathfrak{S} \to \mathfrak{S}$. Lifting yields a functor $+^{\mathfrak{L}} : (\mathfrak{S} \times \mathfrak{S})^{\mathfrak{L}} \to \mathfrak{S}^{\mathfrak{L}}$. Note that $(\mathfrak{S} \times \mathfrak{S})^{\mathfrak{L}} \cong \mathfrak{S}^{\mathfrak{L}} \times \mathfrak{S}^{\mathfrak{L}}$, so we may consider $+^{\mathfrak{L}}$ as a bifunctor $\mathfrak{S}^{\mathfrak{L}} \times \mathfrak{S}^{\mathfrak{L}} \to \mathfrak{S}^{\mathfrak{L}}$.

There is *a priori* no guarantee that $+^{\mathfrak{L}}$ has any special properties, but it turns out that $+^{\mathfrak{L}}$ is a coproduct on $\mathfrak{S}^{\mathfrak{L}}$, which we demonstrate as follows. The key idea is that the property of being a coproduct can be described in terms of an adjunction: in particular, $+$ is a coproduct if and only if it is left adjoint to the diagonal functor $\Delta : \mathfrak{S} \to \mathfrak{S} \times \mathfrak{S}$.[1] Since lifting preserves adjunctions (Lemma A.8), we must have $+^{\mathfrak{L}} \dashv \Delta^{\mathfrak{L}}$. But note we have $\Delta^{\mathfrak{L}} : \mathfrak{S}^{\mathfrak{L}} \to (\mathfrak{S} \times \mathfrak{S})^{\mathfrak{L}} \cong \mathfrak{S}^{\mathfrak{L}} \times \mathfrak{S}^{\mathfrak{L}}$, with $\Delta^{\mathfrak{L}}(F) = \Delta \circ F \cong (F, F)$, so in fact $\Delta^{\mathfrak{L}}$ is the diagonal functor on $\mathfrak{S}^{\mathfrak{L}}$. Hence $+^{\mathfrak{L}}$, being left adjoint to the diagonal functor, is indeed a coproduct on $\mathfrak{S}^{\mathfrak{L}}$.

Of course, this dualizes to products as well, which are characterized by being right adjoint to the diagonal functor. ▨

[1] Proving this standard fact takes a bit of work but mostly just involves unfolding definitions, and is left as a nice exercise for the interested reader.

Bibliography

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003a.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In *Typed Lambda Calculi and Applications, TLCA*, volume 2701 of *LNCS*. Springer-Verlag, 2003b.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing Polymorphic Programs with Quotient Types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, volume 3125 of *LNCS*. Springer-Verlag, 2004.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

Jiřì Adámek, Francis Borceux, Stephen Lack, and Jiřì Rosický. A classification of accessible categories. *Journal of Pure and Applied Algebra*, 175(1):7–30, 2002.

Marcelo Aguiar and Swapneel A. Mahajan. *Monoidal Functors, Species and Hopf Algebras*. CRM monograph series / Centre de recherches mathématiques, Montréal. American Mathematical Society, 2010. ISBN 9780821847763. URL `http://books.google.ca/books?id=tXnPbwAACAAJ`.

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, pages 453–468. Springer, 1999.

Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, pages 297–311. Springer, 2010.

Carlo Angiuli, Ed Morehouse, Daniel R. Licata, and Robert Harper. Homotopical patch theory. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, New York, NY, USA, 2014. ACM.

Steve Awodey. *Category theory*, volume 49. Oxford University Press, 2006.

Robert W Baddeley, Cheryl E Praeger, and Csaba Schneider. Transitive simple subgroups of wreath products in product action. *Journal of the Australian Mathematical Society*, 77(01):55–72, 2004.

John Baez. From the Yoneda lemma to categorical physics, September 1999. `http://www.lepp.cornell.edu/spr/1999-09/msg0017972.html`.

John C Baez and James Dolan. From finite sets to feynman diagrams. *arXiv preprint math/0004133*, 2000.

Michael Barr and Charles Wells. *Category theory for computing science*, volume 10. Prentice Hall New York, 1990.

François Bergeron, Gilbert Labelle, and Pierre Leroux. *Combinatorial species and tree-like structures*. Number 67 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 1998.

Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. *Preprint*, 2014.

Simon Byrne. On groupoids and stuff. Master's thesis, Macquarie University, 2006.

Mario Cáccamo and Glynn Winskel. *A higher-order calculus for categories*. Springer, 2001.

Jacques Carette and Gordon Uszkay. Species: making analytic functors practical for functional programming. Available at `http://www.cas.mcmaster.ca/~carette/species/`, 2008.

Pierre Cartier and Dominique Foata. *Problemes combinatoires de commutation et réarrangements*. Springer, 1969.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 90–104, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi:http://doi.acm.org/10.1145/581690.581698. URL `http://doi.acm.org/10.1145/581690.581698`.

Eugenia Cheng and Simon Willerton. Ends 2. YouTube, January 2014a. URL `http://youtu.be/gyc86NFT0Sw`.

Eugenia Cheng and Simon Willerton. Ends 3. YouTube, January 2014b. URL `http://youtu.be/TfSUxhCNZZ0`.

Dæv Clarke, Ralf Hinze, Johan Theodoor Jeuring, Andres Löh, and Jan de Wit. The generic Haskell user's guide. Technical Report UU-CS-2002-047, Utrecht University, 2002.

Brian Day. On closed categories of functors. In *Reports of the Midwest Category Seminar IV*, pages 1–38. Springer, 1970.

Hélène Décoste, Gilbert Labelle, and Pierre Leroux. The functorial composition of species, a forgotten operation. *Discrete mathematics*, 99(1):31–48, 1992.

Radu Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51(1):176–178, 1975.

Gilles Dowek. *Principles of Programming Languages*. Springer, 2009.

Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Random sampling from boltzmann principles. In Peter Widmayer, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo, and Matthew Hennessy, editors, *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 776–776. Springer Berlin / Heidelberg, 2002. ISBN 978-3-540-43864-9.

Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13:577–625, 2004. doi:doi:10.1017/S0963548304006315.

Anton Fetisov. Answer to "intuition for coends". MathOverflow, November 2011. URL: `http://mathoverflow.net/a/80719/856` (visited on 2014-08-13).

Marcelo Fiore, Nicola Gambino, Martin Hyland, and Glynn Winskel. The cartesian closed bicategory of generalised species of structures. *J. London Math. Soc.*, 77(1): 203–220, 2008. doi:10.1112/jlms/jdm096. URL `http://jlms.oxfordjournals.org/cgi/content/abstract/77/1/203`.

Marcelo P Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. 2003.

Philippe Flajolet and Bruno Salvy. Computer algebra libraries for combinatorial structures. *Journal of Symbolic Computation*, 20(5-6):653–671, 1995. doi:10.1006/jsco.1995.1070.

Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. Lambda-upsilon-omega: The 1989 cookbook. Technical Report 1073, Institut National de Recherche en Informatique et en Automatique, August 1989. URL `http://www.inria.fr/rrrt/rr-1073.html`. 116 pages.

Philippe Flajolet, Éric Fusy, Carine Pivoteau, et al. Boltzmann sampling of unlabeled structures. *ANALCO*, 7:201–211, 2007.

Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002. URL `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/acmmpc-calcfp.pdf`.

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi:http://doi.acm.org/10.1145/165180.165214. URL `http://doi.acm.org/10.1145/165180.165214`.

Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi:http://doi.acm.org/10.1145/1596638.1596653. URL `http://doi.acm.org/10.1145/1596638.1596653`.

Nelson Goodman and John Myhill. Choice implies excluded middle. *Mathematical Logic Quarterly*, 24(25-30):461–461, 1978.

Basil Gordon. Sieve-equivalence and explicit bijections. *Journal of Combinatorial Theory, Series A*, 34(1):90–93, 1983.

Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming*, 23(05):594–628, 2013.

Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 119–132, New York, NY, USA, 2000. ACM. ISBN 1-58113-125-9. doi:http://doi.acm.org/10.1145/325694.325709. URL `http://doi.acm.org/10.1145/325694.325709`.

Ralf Hinze. Kan extensions for program optimisation or: Art and Dan explain an old trick. In *Mathematics of Program Construction*, pages 324–362. Springer, 2012.

Ralf Hinze and Daniel WH James. Reason isomorphically! In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 85–96. ACM, 2010.

Gérard Huet. Functional pearl: The zipper. *J. Functional Programming*, 7:7–5, 1997.

Patrik Jansson and Johan Jeuring. Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles*

of programming languages, POPL '97, pages 470–482, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi:http://doi.acm.org/10.1145/263699.263763. URL `http://doi.acm.org/10.1145/263699.263763`.

Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *arXiv preprint arXiv:1402.1699*, 2014.

C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 302–316, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

Warren P Johnson. The curious history of Faà di Bruno's formula. *American Mathematical Monthly*, pages 217–234, 2002.

André Joyal. Une théorie combinatoire des Séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

André Joyal. Foncteurs analytiques et espèces de structures. In Gilbert Labelle and Pierre Leroux, editors, *Combinatoire Énumérative*, volume 1234 of *Lecture Notes in Mathematics*, pages 126–159. Springer Berlin Heidelberg, 1986. ISBN 978-3-540-17207-9. doi:10.1007/BFb0072514. URL `http://dx.doi.org/10.1007/BFb0072514`.

André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991.

G Max Kelly. On the operads of J. P. May. *Repr. Theory Appl. Categ*, 13:1–13, 2005.

Edward Kmett. The `kan-extensions` package, a. URL `http://hackage.haskell.org/package/kan-extensions`.

Edward Kmett. The `lens` package, b. URL `http://hackage.haskell.org/package/lens`.

Edward Kmett. Kan Extensions III: As Ends and Coends, May 2008. `http://comonad.com/reader/2008/kan-extension-iii/`.

Edward Kmett. Free Monads for Less (Part 2 of 3): Yoneda, June 2011. `http://comonad.com/reader/2011/free-monads-for-less-2/`.

Donald E Knuth. Sorting and Searching (The Art of Computer Programming volume 3), 1973.

Joachim Kock. Data types with symmetries and polynomial functors over groupoids. In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (Bath, 2012)*, 2012.

Gilbert Labelle and Cédric Lamathe. General combinatorial differential operators. *Séminaire Lotharingien de Combinatoire*, 61(B61Ag):24pp, 2009.

Jacques Labelle. Quelques especes sur les ensembles de petite cardinalité. *Ann. Sc. Math. Québec*, 9(1):31–58, 1985.

Stephen Lack and Ross Street. Combinatorial categorical equivalences. *arXiv preprint arXiv:1402.7151*, 2014.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8. doi:http://doi.acm.org/10.1145/604174.604179. URL `http://doi.acm.org/10.1145/604174.604179`.

F William Lawvere and Stephen Hoel Schanuel. *Conceptual mathematics: a first introduction to categories*. Cambridge University Press, 2009.

Pierre Lescanne. On counting untyped lambda terms. *Theoretical Computer Science*, 474:80–97, 2013.

Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer, 1998.

Manuel Maia and Miguel Méndez. On the arithmetic product of combinatorial species. *Discrete Mathematics*, 308(23):5407 – 5427, 2008. ISSN 0012-365X. doi:http://dx.doi.org/10.1016/j.disc.2007.09.062. URL `http://www.sciencedirect.com/science/article/pii/S0012365X07007960`.

Michael Makkai. First order logic with dependent sorts, with applications to category theory. *Preprint: http://www. math. mcgill. ca/makkai*, 1995.

Michael Makkai. Avoiding the axiom of choice in general category theory. *Journal of Pure and Applied Algebra*, 108(2):109–173, 1996.

Michael Makkai. Towards a categorical foundation of mathematics. In *Logic Colloquium'95*, pages 153–190. Springer, 1998.

Simon Marlow. Haskell 2010 Language Report, 2010.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics*, 80:73–118, 1975.

Per Martin-Löf. Constructive mathematics and computer programming. *Studies in Logic and the Foundations of Mathematics*, 104:153–175, 1982.

Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, 1984.

Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.cs.nott.ac.uk/~ctm/diff.ps.gz`, 2001.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, San Francisco, California, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi:10.1145/1328438.1328474. URL `http://portal.acm.org/citation.cfm?id=1328474`.

Conor McBride. Answer to "writing cojoin or cobind for n-dimensional grid type". StackOverflow, 2012. URL: `http://stackoverflow.com/a/13100857/305559` (visited on 2014-09-21).

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin / Heidelberg, 1991. ISBN 978-3-540-54396-1. doi:doi:10.1007/3540543961_7. URL `http://dx.doi.org/10.1007/3540543961_7`.

Matías Menni. Combinatorial functional and differential equations applied to differential posets. *Discrete Mathematics*, 308(10):1864–1888, May 2008. doi:10.1016/j.disc.2007.04.035.

Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

The nLab. Finite sets, 2013. URL `http://ncatlab.org/nlab/show/finite+set`.

The nLab. The axiom of choice, 2014a. URL `http://ncatlab.org/nlab/show/axiom+of+choice`.

The nLab. Clique, 2014b. URL `http://ncatlab.org/nlab/show/clique`.

The nLab. Equivalence of categories, 2014c. URL `http://ncatlab.org/nlab/show/equivalence+of+categories`.

The nLab. Equivalence of categories (weak equivalence), 2014d. URL `http://ncatlab.org/nlab/show/equivalence+of+categories#WeakEquivalence`.

The nLab. Principle of equivalence, 2014e. URL `http://ncatlab.org/nlab/show/principle+of+equivalence`.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.

Dan Piponi. Reverse Engineering Machines with the Yoneda Lemma, November 2006. `http://blog.sigfpe.com/2006/11/yoneda-lemma.html`.

Dan Piponi. Constraining Types with Regular Expressions, August 2010a. `http://blog.sigfpe.com/2010/08/constraining-types-with-regular.html`.

Dan Piponi. Divided Differences and the Tomography of Types, August 2010b. `http://blog.sigfpe.com/2010/08/divided-differences-and-tomography-of.html`.

Carine Pivoteau, Bruno Salvy, and Michèle Soria. Algorithms for combinatorial structures: Well-founded systems and newton iterations. *J. Comb. Theory, Ser. A*, 119 (8):1711–1773, 2012.

John C Reynolds. Types, abstraction and parametric polymorphism. 1983.

Olivier Roussel and Michele Soria. Boltzmann sampling of ordered structures. *Electronic Notes in Discrete Mathematics*, 35:305–310, 2009.

William R. Schmitt. Hopf algebras of combinatorial structures. *Canadian Journal of Mathematics*, pages 412–428, 1993.

Mike Shulman. Answer to "exponentials in functor categories". MathOverflow. URL: `http://mathoverflow.net/a/104178` (visited on 2014-08-06).

Mike Stay. Q, Jokers, and Clowns, August 2014. `http://reperiendi.wordpress.com/2014/08/05/q-jokers-and-clowns/`.

Ross Street. Monoidal categories in, and linking, geometry and algebra. *Bulletin of the Belgian Mathematical Society-Simon Stevin*, 19(5), 2012.

Todd Trimble. Answer to "Examples of functors **Set** $\to$ **Set** which are not analytic". MathOverflow, 2014. URL: `http://mathoverflow.net/a/171456` (visited on 2014-06-10).

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

Vladimir Voevodsky. Foundations/formalization of mathematics and homotopy theory. URL `http://video.ias.edu/node/68`. talk at the Instutute for Advanced Study.

Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73:231–248, January 1988. ISSN 0304-3975. doi:http://dx.doi.org/10.1016/0304-3975(90)90147-A.

Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.

Stan Wagon. *The Banach-Tarski Paradox*, volume 24. Cambridge University Press, 1993.

Stephanie Weirich. Type-safe run-time polytypic programming. 16(10):681–710, November 2006a.

Stephanie Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, September 2006b.

Herbert S. Wilf. *Generatingfunctionology*. Academic Press, 1990.

Yeong-Nan Yeh. *On the combinatorial species of Joyal*. PhD thesis, State University of New York at Buffalo, 1985.

Yeong-Nan Yeh. The calculus of virtual species and K-species. In Gilbert Labelle and Pierre Leroux, editors, *Combinatoire énumérative*, volume 1234 of *Lecture Notes in Mathematics*, pages 351–369. Springer Berlin Heidelberg, 1986. ISBN 978-3-540-17207-9. doi:10.1007/BFb0072525. URL `http://dx.doi.org/10.1007/BFb0072525`.

Brent A. Yorgey. Species and Functors and Types, Oh My! In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 147–158, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi:http://doi.acm.org/10.1145/1863523.1863542. URL `http://doi.acm.org/10.1145/1863523.1863542`.

Brent A Yorgey. Monoids: theme and variations (functional pearl). In *ACM SIGPLAN Notices*, volume 47, pages 105–116. ACM, 2012.